FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Karel Maděra

Accelerating cross-correlation with GPUs

Name of the department

Supervisor of the master thesis: Supervisor's Name
Study programme: Computer Science
Study branch: ISS

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedication.

Title: Accelerating cross-correlation with GPUs

Author: Karel Maděra

Department: Name of the department

Supervisor: Supervisor's Name, department

Abstract: Abstract.

Keywords: key words

Contents

Introduction

The field of Signal processing is present everywhere in today's world. From image processing through seismology to particle physics, the need to analyze, modify, or synthesize signals such as sound, images, and other scientific measurements is shared throughout many fields. One of the commonly used algorithms in signal processing is cross-correlation, which will be the subject of this thesis. The aim is to analyze, implement and evaluate possible methods of optimization, and parallelization of definition based cross-correlation algorithm. The implementations will then be further compared to the generally used implementation based on Fast Fourier transform.

Motivation

Cross-correlation is one of the key operations in both analog and digital signal processing. It is widely used in image analysis, pattern recognition, image segmentation, particle physics, electron tomography, and many other fields [3]. For many of these applications, the computation time of cross-correlation is often the limiting factor in the data processing pipeline. The amount of input data combined with the computational complexity make simple sequential CPU-based implementations and even more advanced parallel CPU-based implementation inadequate.

Algorithms based on the definition of cross-correlation or on Fast Fourier transform (FFT) can take advantage of the inherent high degree of data parallelism in the definition of cross-correlation or FFT respectively to utilize the high throughput and massive amounts of computational power provided by massively parallel systems in the form of Graphical processing units (GPU).

This thesis is a continuation of the thesis "Employing GPU to Process Data from Electron Microscope" [1], which uses both basic definition based cross-correlation as well as one based on FFT. This thesis aims to compare the asymptotically faster FFT based algorithm with the asymptotically slower definition based algorithm and provide an optimized implementation of the definition based algorithm which, for the input sizes used by the original thesis, should be faster than the FFT based implementation.

Objective

The objective of this thesis is to analyze the possibilities for optimization and parallelization of the definition based algorithm and provide detailed measurements and comparisons with the FFT based algorithm for range of input forms and sizes. The optimizations and parallelization of the definition based algorithm will utilize capabilities provided by the CUDA platform.

The main contributions of this thesis are:

- a family of optimized definition based implementations utilizing the CUDA platform,

- comparison of the definition based implementations with one based on Fast Fourier Transform,
- measurements of the behavior of these implementations based on input size and type.

Thesis outline

The contents of this thesis are ordered as follows:

- description of cross-correlation algorithm;
- introduction to computations utilizing GPU hardware and the CUDA platform;
- analysis of the optimizations of the definition based algorithm, focused on parallelization using CUDA platform,
- measurement of the behavior of both the optimized definition based and Fast Fourier transform based algorithm.

1. Cross-correlation

In this chapter, we define cross-correlation and describe the ways for its computation. We first define one-dimensional cross-correlation, extending it into multiple dimensions and introducing circular cross-correlation. We then describe how circular cross-correlation is used to compute cross-correlation using discrete Fourier transform. Lastly we describe the possibilities for optimization and parallelization of cross-correlation, with real-world usage examples where these optimizations can be used.

1.1 Definition

Cross-correlation, also known as sliding dot product or sliding inner-product, is a function describing similarity of two series or two functions based on their relative displacement [11]. Cross-correlation of functions $f, g : \mathbb{C} \to \mathbb{R}$, denoted as $f \star g$, is defined by the following formula:

$$(f \star g)(\tau) = \int_{-\infty}^{\infty} \overline{f(t)} g(t + \tau)\, dt,$$

where $\overline{f(t)}$ denotes the complex conjugate of $f(t)$ and τ is the displacement of the two functions f and g. In simpler words, the value $(f \star g)(\tau)$ tells us how similar the function f is to g when g is shifted by τ, with higher value representing higher similarity. Figure 1.1 shows cross-correlation of two example functions.

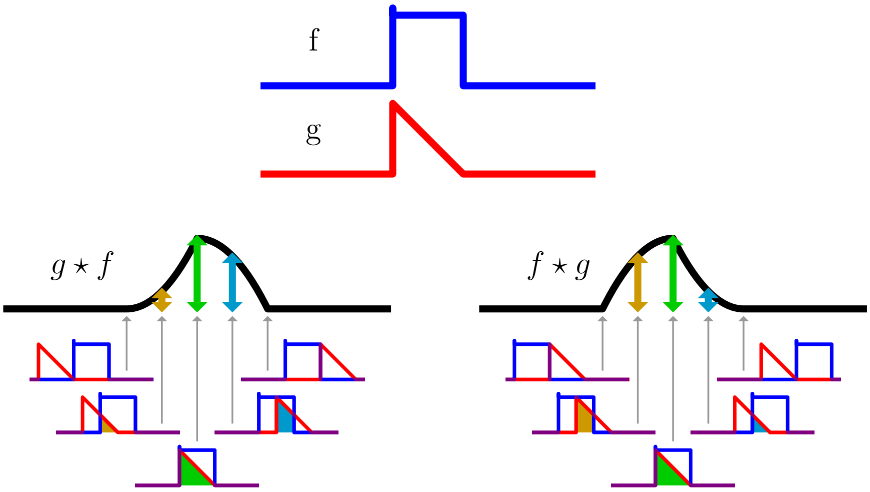


Figure 1.1: Cross-correlation of two functions. [10]

For two discrete functions, as will be used in our case, cross-correlation of functions $f, g : \mathbb{Z} \to \mathbb{R}$ is defined by the following formula:

$$(f \star g)[m] = \sum_{i=-\infty}^{\infty} \overline{f[i]} g[i + m],$$

This definition of cross-correlation can be extended for use in two dimensions, as is required, for example, in image processing. For two discrete functions $f, g : \mathbb{Z}^2 \to \mathbb{R}$, cross-correlation is defined as:

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

Even though cross-correlation is defined on the whole $\mathbb{Z}$ for one dimension and $\mathbb{Z}^2$ for two dimensions, most use cases of cross-correlation work only on finite inputs, such as image processing working on finite images. The only values we are interested in are those where the two images overlap, which limits the computation to $(w_1 + w_2 - 1) * (h_1 + h_2 - 1)$ resulting values, where w_i denotes width of the image i and h_i denotes the height of the image i.

This limits the part of the output we are interested in and leads us to the time complexity of the definition based algorithm, or *naive* algorithm as it is called in the code associated with the thesis. For each of the $(w_1 + w_2 - 1) * (h_1 + h_2 - 1)$ output values, we need to multiply the overlapping pixel values and sum all the multiplication results together. There will be at most $min(w_1, w_2) * min(h_1, h_2)$ overlapping pixels. For simplicity, let us work with two images of size $w_i * h_i$. Then the time complexity of the definition based algorithm is $((2 * w_i - 1) * (2 * h_i - 1) * (w_i * h_i))$, which gives us asymptotic complexity of $\mathcal{O}(w_i^2 * h_i^2)$.

1.2 Computation using discrete Fourier Transform

In this section, we describe an algorithm which uses discrete Fourier transform to compute cross-correlation of two finite two-dimensional series. The asymptotic complexity of this algorithm will be $\mathcal{O}(w_i * h_i * \log_2(w_i * h_i))$, where w_i is the width of each series and h_i the height of each series. This improves on the asymptotic complexity $\mathcal{O}(w_i^2 * h_i^2)$ of the definition based algorithm described in the previous section 1.1.

Discrete Fourier transform can only be used to compute a special type of cross-correlation, so called *circular* cross-correlation. For finite series $N \in \mathbb{N}\{x\}_n = x_0, x_1, ..., x_{N-1}, \{y_n\} = y_0, y_1, ..., y_{N-1}$, circular cross-correlation is defined as:

$$(x \star_N y)_m = \sum_{i=0}^{N-1} \overline{x_m} y_{(m+i) mod N},$$

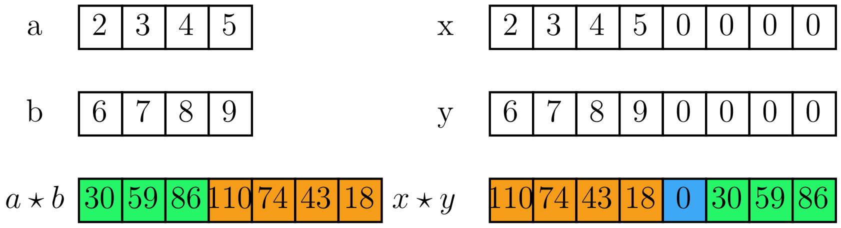
where $\overline{x_m}$ denotes complex conjugate of x_m.

Based on the Cross-Correlation Theorem [9], circular cross-correlation $(x \star_N y)_m$ can be computed using discrete Fourier Transform based on the following formula:

$$(x \star_N y)_m = \mathbb{F}^{-1}(\overline{\mathbb{F}(x)} * \mathbb{F}(y))$$

where $\mathbb{F}(x)$ and $\mathbb{F}(y)$ denote discrete Fourier Transform of series x and y respectively, $\overline{\mathbb{F}(x)}$ denotes complex conjugate of the discrete Fourier Transform, $*$ denotes element-wise multiplication of two series and $\mathbb{F}^{-1}$ denotes inverse discrete Fourier Transform.

As described by Bali [1], to compute non-circular (linear) cross-correlation of non-periodic series of size N, we pad both series with N zeros to the size *2N*, as can be see in Figure 1.2. The results of circular cross-correlation are then the results of linear cross-correlation, only circularly shifted by $N - 1$ places to the left with one additional 0 value at index N.



a	2	3	4	5					
b	6	7	8	9					
$a \star b$	30	59	86	110	74	43	18		
x	2	3	4	5	0	0	0	0	
y	6	7	8	9	0	0	0	0	
$x \star y$	110	74	43	18	0	30	59	86	

Figure 1.2: Comparison of linear and circular cross-correlation [1].

This process can be expanded into two dimensions, where the matrices are padded with N rows and N columns of zeros before being passed through 2D discrete Fourier transform. Here the circular shift of the results can be inverted by swapping the quadrants of the results while discarding row N and column N which will be filled with zeros [1], as shown by Figure 1.3.

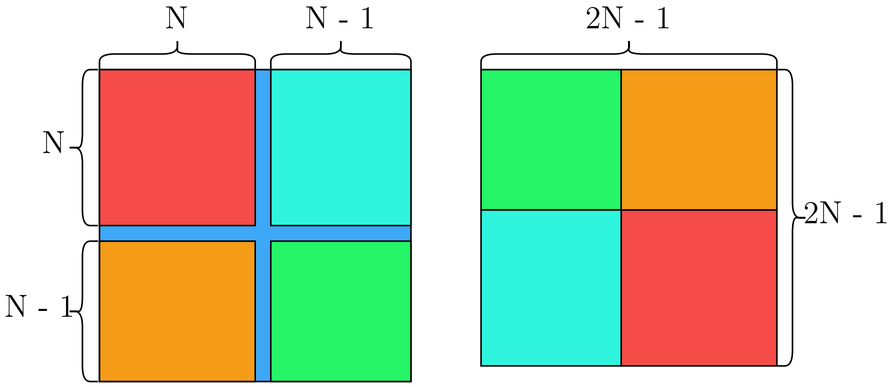


Figure 1.3: Result quadrant swap.

Based on this description, we can deduce the time complexity of the algorithm. For two matrices $a, b \in \mathbb{R}^{h \times w}$,the steps of the algorithm are:

1. Padding $a_p, b_p \in \mathbb{R}^{2h \times 2w}$ of a and b with h rows and w columns of zeros in $\mathcal{O}(h * w)$;

2. Discrete Fourier Transform $A, B \in \mathbb{C}^{2h \times 2w}$ of a_p and b_p in $\mathcal{O}(h * w * \log_2(h * w))$;

3. Element-wise multiplication, also known as Hadamard product, $C \in \mathbb{C}^{2h \times 2w}$: $C = \overline{A} \circ B$, where $\overline{A}$ denotes complex conjugate of A, in $\mathcal{O}(w_i * h_i)$;

4. Inverse Discrete Fourier Transform $c \in \mathbb{R}^{2h \times 2w}$ of C in $\mathcal{O}(h * w * \log_2(h * w))$;

5. Quadrant swap in $\mathcal{O}(h * w)$

Put together, the steps described above give us an algorithm with asymptotic time complexity of $\mathcal{O}(h * w * \log_2(h * w))$.

1.3 Definition based optimizations

In the original thesis by Bali [1], and in the field of image processing in general, 2D version of cross-correlation is mostly used to find a grayscale image or a piece of a grayscale image represented as an integer or floating point matrix in another image, also represented as such matrix. This can be done to, for example, find a displacement of certain point of interest between images taken at different times, as is done in Bali [1] and Zhang et al. [12].

This thesis will implement cross-correlation of integer and floating point matrices, which encompasses the usage in previously mentioned works. The implementations will be optimized to take advantage of different forms of cross-correlation input, such as cross-correlation of one matrix with many other matrices, different sizes of input matrices etc.

1.3.1 Data parallelism

Definition based algorithm for computing cross-correlation is highly data parallel. Not only can every element in the result matrix be computed independently, computation of each element can also be parallelized with a simple reduction of the final results.

When using the definition based algorithm, each element of the resulting matrix corresponds to an overlap of the two cross-correlated matrices. Every two overlapping elements of the two input matrices are multiplied and results of these multiplications are then summed together to get the final value for given overlap.

For each overlap, there is $h_o * w_o$ multiplications, where h_o and w_o describe the number of rows and columns which overlap. The following formula describes the total number of multiplications for all overlaps:

$$(h * (h + 1) - 1) * (w * (w + 1) - 1)$$

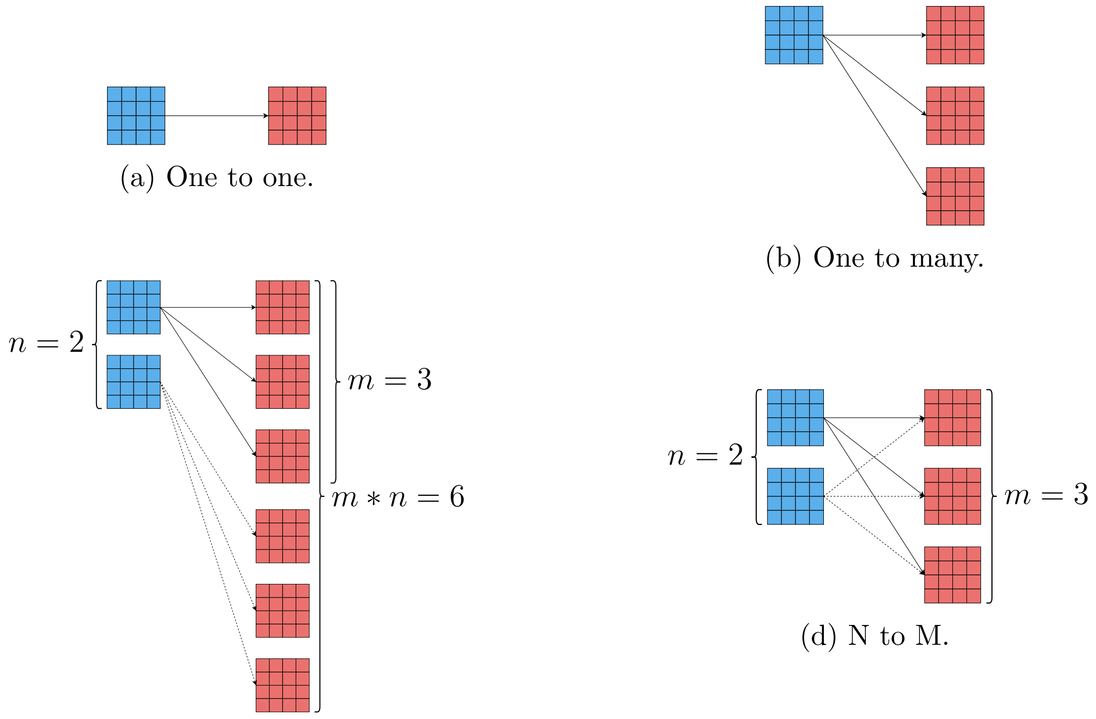
.

All these multiplications can be done independently in parallel. Afterwards, each overlap has to compute a sum of the $h_o * w_o$ results to produce the final result.

1.3.2 Forms of cross-correlation

In works using cross-correlation, there are several forms of computation which can be used for optimization such as data caching and reuse, batching, precomputing etc. The forms differ in the number of inputs and in the way cross-correlation is computed between the inputs. The four basic forms are, as shown in Figure 1.4:

1. one left input with one right input, in the rest of the thesis referred to as *one-to-one* and depicted in Figure 1.4a;
2. one left input with many right inputs, referred to as *one-to-many* and depicted in Figure 1.4b;
3. n left inputs, each cross-correlated with m **different** right inputs, referred to as *n-to-mn* and depicted in Figure 1.4c (used by Bali [1], Zhang et al. [12], Kapinchev et al. [3]);



(a) One to one.

(b) One to many.

(c) N to MN.

(d) N to M.

Figure 1.4: Forms of cross-correlation.

4. n left inputs, each cross-correlated with all m right inputs, referred to as *n-to-m* and depicted in Figure 1.4d (used by Clark et al. [2]).

While each pair of input matrices can always be computed independently, the *one-to-many*, *n-to-mn* and *n-to-m* types allow for the reuse of left input matrix data for computation with multiple right input matrices. Additionally, the *n-to-m* allows for reuse of data from the right matrix for computation with multiple left input matrices.

For the same size of input data, i.e. x left input matrices and y right input matrices, the *n-to-m* type results in $x * y$ pairs of matrices to be computed, compared to the *n-to-mn* type which results in only y pairs. The increased level of parallelism and increased arithmetic intensity allows for additional optimizations of the *n-to-m* computation type compared to the *n-to-mn* type. The *one-to-one* and *one-to-many* types are described separately as their implementations can more aggressively cache and reuse the left input matrix compared to the general *n-to-mn* or *n-to-m* implementation.

Implementations of the two simpler types *one-to-one* and *one-to-many* can be used for implementation of either *n-to-m* or *n-to-mn* by running the simpler type implementation multiple times, possibly in parallel. Inversely, any implementation of either *n-to-m* or *n-to-mn* can be used to implement the two simpler types. Another possible subtype is the computation of large number of pairs, which can be implemented by *n-to-mn* with m equal to one. The large number of pairs type is not discussed further as it does not provide any additional opportunity for optimization compared to parallel run of many *one-to-one* implementations.

1.4 Post-processing

In most use cases, cross-correlation itself is not a final output but the results are used further in further processing. It is often used to find position of a smaller signal in larger signal, for example in the field of Digital image processing for template matching, image alignment etc. In these use cases, the only information of interest is the maximum value in the result matrix.

In Digital Image correlation, we are also interested in finding the maximum, but this time with a subpixel precision. This requires us to find the maximum value and use the results in an area around it to interpolate a function [12] [1].

In the field of Seismology, cross-correlation is used for picking, ambient noise monitoring, waveform comparison and signal, event and pattern detection [8].

In optical coherence tomography, the whole result of cross-correlation is summed to compute the intensity of each pixel [3].

Although post-processing is often also a good candidate for optimization and parallelization, it is outside the scope of this thesis. Result of cross-correlation will be taken as-is and validated against preexisting cross-correlation implementations.

2. GPU

This chapter describes Graphical processing unit (GPU) hardware and its basic advantages and disadvantages when compared to the Central processing unit (CPU) in general. Next we introduce Compute Unified Device Architecture, better known by its acronym CUDA, a ”general purpose parallel computing platform and programming model” [5], which will be used in this thesis. Lastly we list several key points which need to be addressed for optimal code performance mostly in CUDA, but also applicable when working with GPUs of other vendors or when using GPUs for graphics.

2.1 Fundamentals

Graphical processing unit (GPU) is optimized for throughput of a single stream of instructions working on many streams of data, which allows GPU hardware design to make trade-offs which are not available for Central processing unit (CPU) hardware. CPUs are optimized to process a single stream of instructions working on a single stream of data as fast as possible. This requires CPU design to minimize instruction latency, which is achieved by using branch predictions, multiple levels of caching, and other such mechanisms. On the other hand, the single stream of instructions is executed many times in parallel in a GPU, which allows GPU hardware to hide high latency operations by switching to other threads instead of trying to optimize for lower latency of each instruction. The thread switching is made instantaneous by keeping the execution context, such as registers, of all threads resident at all times. Compare this with CPU context switching, where the state of all registers has to be offloaded into memory and state of another thread loaded from memory each time a CPU switches threads.

Another defining characteristic of the GPU hardware is a separate memory space. Code running on the GPU device cannot directly access the same memory as code running on the CPU. Instead, all data to be processed on the GPU has to be moved across the bus connecting the GPU to the host system, most commonly a PCIe bus. Similarly to the GPU execution units, the separate GPU device memory is optimized for high throughput at the cost of higher latency compared to the host memory. The device memory is further optimized for specific access patterns by groups of threads running on the GPU, with Nvidia GPU version of the optimization described in Section 2.3.3.

2.2 CUDA Programming model

Compute Unified Device Architecture, better known by its acronym CUDA, a ”general purpose parallel computing platform and programming model” [5], which allows simplified utilization of NVIDIA Graphics processing units (GPU) for solving complex computational problems.

CUDA distinguishes two parts of the system running two types of code. First is the *host* code running on the host part of the system. This is standard C++ program running on the CPU, accessing system memory and calling the operating system, as any other standard C++ program would. The second part is the *device* code, running on a device or on multiple devices. Each device corresponds to a single GPU [1].

Both parts of the code are programmed in the same language, CUDA C++, which is an extension to the C++ language, with some restrictions to the device code and some parts of the language only usable in the device code. One of the important things CUDA C++ introduces are *function execution space specifiers*, which are attributes added to a function declaration and which specify if the given function is part of the host code, device code or if it should be compiled both for host and device code. The available *function execution space specifiers* are:

- `__global__`, which declares the function as being a kernel, callable from host code and executed on the device,
- `__device__`, which declares the function as executed on the device, callable by another device or global function,
- `__host__`, which declares the function as executed on the host, callable from the host only.

Without any specifiers, function is compiled as part of the host code. **Kernel** is a function with the `__global__` specifier, which is callable from the host code but is executed on a device. Kernels serve as entry points which the host code uses to offload computation to the device. Kernel invocation is asynchronous, where the function call to the kernel in host code does not wait for the kernel on the device to finish but returns immediately after the kernel is submitted.

When invoking a kernel, host code specifies the number of threads which are to run the device code. Basic description of how the device code is ran on the GPU is provided in Section 2.2.1. Detailed description of the abstraction defining the behavior of the device code, called the SIMT execution model, is given in Section 2.2.6.

2.2.1 Running the device code

The device code, written in CUDA C++ as a part of *global* or *device* function, describes the behavior of a single thread running on the device. Compared to host code running on the CPU, the device code is always ran by many threads simultaneously. On the surface, the device code is very similar to the host code written for the CPU, and will most likely work correctly if written as if for the CPU. The number of threads running the device code is determined by the arguments provided to the *kernel* function. The threads are hierarchically grouped on several levels. These groups define scheduling behavior and guarantees, access to different kinds of memory and primitives for cooperation.

To maximize performance, one must structure the code and the overall algorithm according to details provided in Section 2.2.6.

[1] Since Compute Capability 8.0 Ampere, device can represent a GPU slice.

2.2.2 Thread hierarchy

Threads on the device are grouped into Cooperative Thread Arrays (CTA), also known as thread blocks. Thread blocks can be one-dimensional, two-dimensional or three-dimensional, which provides an easy way to distribute work when processing arrays, matrices or volumes. Thread blocks are further organized into one-dimensional, two-dimensional or three-dimensional grid, as can be seen in Figure 2.1. When launching a kernel, we specify thread block size and grid size, which combined together give us the number of threads executing the given kernel.

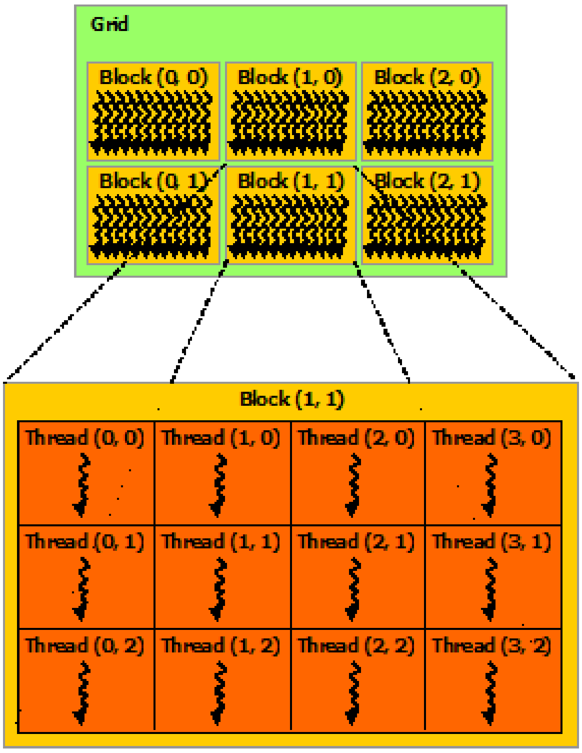


Figure 2.1: Thread grouping hierarchy [5].

Each thread is assigned an index, accessible through `threadIdx` built-in variable. Each thread can also access the index of the thread block it is part of through `blockIdx`, the block size through `blockDim` and grid size through `gridDim`. All of these variables are three dimensional vectors, with dimensions unused during kernel launch set to zero for indices and one for dimensions. Using these built-in variables, we can distribute work between threads, most often assigning each thread a part of the input to process.

Due to hardware details described in Section 2.2.6, each 32 threads of a thread block are grouped into a *warp*. Warps are used for scheduling and close cooperation of threads.

2.2.3 Thread cooperation

CUDA provides several mechanisms for thread cooperation. Threads can cooperate on the following levels of thread hierarchy, with increasing levels of speed and capability:

- grid level,
- thread block level,
- warp level.

The rest of this subsection describes the older API using intrinsic functions. The newer Cooperative Groups API, which is a superset of the older API, is described in Subsection 2.2.4.

Grid level

On grid level, the only available tools for cooperation are atomic operations on global memory. These operations can be used to perform read-modify-write on a 32-bit or 64-bit word in global memory without introducing race conditions.

Thread block level

On a thread block level, threads can use two mechanisms for cooperation:

- shared memory,
- synchronization barrier.

As per the CUDA C++ programming guide: ”the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and __syncthreads() is expected to be lightweight” [5].

Shared memory is a small memory close to the execution cores, described in more detail in the subsection 2.2.5. Each thread block receives a slice of shared memory, which is then accessible only from the threads of the given thread block. Shared memory can be used as software managed cache or to share results between threads of the thread block.

To synchronize threads of a single thread block, for example to communicate through shared memory, we use synchronization barrier `__syncthreads()`. All threads in the block must execute the call to `__syncthreads()` before any of the threads can proceed beyond the call to `__syncthreads()`. The `__syncthreads()` function also serves as memory barrier.

Warp level

Threads of the warp, or lanes as they are referred to in the documentation, can utilize intrinsic functions to exchange data without the use of shared memory and perform simple hardware accelerated operations.

For data exchange between lanes of a warp, CUDA provides several warp shuffle functions, such as __shfl_sync, __shfl_up_sync, __shfl_down_sync, __shfl_xor_sync. These differ in how they interpret the provided index, either using it directly as lane index, adding or subtracting from current lane index or executing *xor* with current lane index. The data exchange does not have to span the whole warp. Shuffle operations allow the warp to be subdivided into groups with width of a power of 2. These operations can be used for different data exchange patterns, such as the obvious shuffle up or down, data rotation across lanes, broadcast of a value from single lane etc.

Warps can perform the following types of operations:

- vote operations (__any_sync, __all_sync, __ballot_sync), which determine if any or all threads provided non-zero value, or return a mask of threads which provided non-zero value respectively;
- match operations (__match_any_sync, __match_all_sync), which return mask of threads which provided the same value, or determine if all threads provided the same value;
- reduce operations (__reduce_[*op*]_sync), which execute one of the following operations on values provided: *add*, *max*, *min*, *and*, *or*, *xor*.

The API described in this subsection forms the basis of thread cooperation in CUDA. Most of this API is available since the early versions of CUDA. Subsection 2.2.4 will describe the newer Cooperative groups API, which builds on top of and extends the API described in this subsection.

2.2.4 Cooperative groups

Cooperative Groups API, introduced with CUDA 9, is an extension to the CUDA programming model for organizing groups of communicating threads [5]. The API introduces data types representing groups of cooperating threads, be it a warp, a part of a warp, a thread block, a grid or even a multigrid[2].

The API distinguishes two types of groups. First are the *implicit groups*, which are present implicitly in each CUDA kernel. These are:

- thread block,
- grid,
- multigrid.

The API provides functions to create objects representing the implicit groups. The other type are *explicit groups*, which must be explicitly created from one of the implicit groups. The two explicit groups are:

- thread block tile,
- coalesced group.

Both of these groups represent warp or subwarp size grouping of threads. Thread block tile can be created from a thread block or from another thread block tile, representing a warp or a part of a warp of size of a power of 2. The warp level operations described in the previous subsection 2.2.3 are available as methods on this group, with mask and width arguments of the built-in functions implicitly derived from the properties of the group.

[2] Multigrid represents multiple grids each running on a separate device.

Coalesced group contains threads of a warp which are currently active, i.e. not masked.

Creating an object representing an implicit group is a collective operation, in which all threads of the group must participate. Creating the object in a conditional branch may lead to deadlocks or data corruption. It may also introduce unnecessary synchronization points, limiting concurrency. Similarly to implicit group object creation, partitioning of groups is a collective operation which must be executed by all threads of the parent group and may introduce synchronization points. It is recommended to create objects representing implicit groups and do all partitioning at the start of the kernel and pass *const* references throughout the code [5].

2.2.5 Memory hierarchy

Each CUDA device has its own DRAM memory, so called *device memory* or *VRAM*, which is separate from the host system memory and from the *device memory* of all other devices. Physically, *device memory* can be seen on most GPU boards as DRAM chips separate from the main silicon chip.

Data transferred between the host and device memory has to cross over the PCI-e bus, either explicitly by calls to *cudaMemcpy* in the host code or by mapping parts of host memory to the *device memory* address space using the *Unified Memory* system, which then handles the data transfers in the background automatically.

From the point of view of a CUDA thread, there are several types of memory available, as can be seen in Figure 2.2. For this thesis, the main types are:

- global memory,
- shared memory,
- registers,
- local memory.

Global memory is part of the device memory. It is shared by all threads of a grid, and as such any access which could lead to race condition must be synchronized using atomic operations, as described in Section 2.2.3. Global memory is allocated by the host code using `cudaMalloc` family of functions. When host code transfers data to the device using `cudaMemcpy` or any other means, global memory is the part of device memory this data will be transferred to. The pointers returned by `cudaMalloc` and possibly used in `cudaMemcpy` are then passed as arguments to the kernel. Device code can then use these to access the global memory.

Shared memory, as mentioned in the section 2.2.3, is expected to be a low-latency memory near each processor core (much like an L1 cache). The relation with L1 cache can be seen in the fact that each kernel can configure the proportion between hardware allocated to L1 cache and to Shared memory, which means these memories share the same underlying hardware. Shared memory can be allocated either dynamically by declaring an array type variable with the

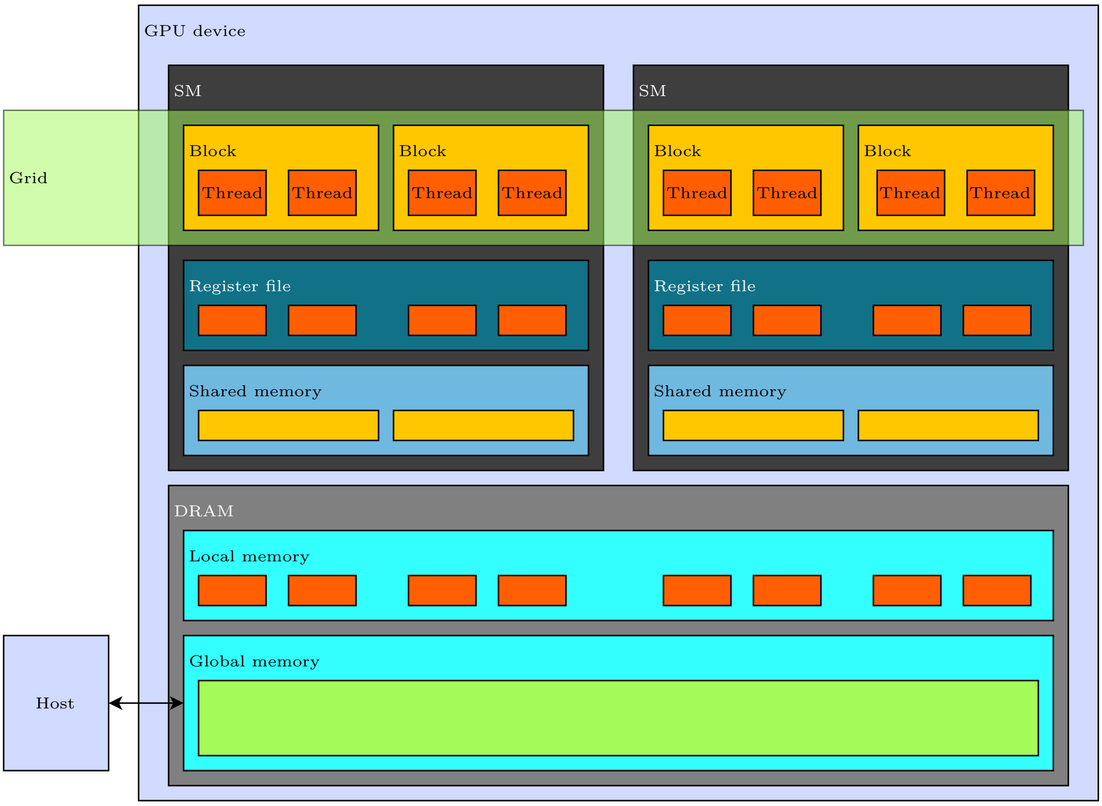


Figure 2.2: Memory types on a CUDA device.

memory space specifier `__shared__` and providing the size to be allocated during kernel launch, or statically by defining the variable with static size.

Registers are the fastest memory available for device code. Compared to CPUs, GPUs provide large amount of registers. For all recent GPU generations, the register file provides 65536 32bit registers. All variables used by the kernel code are stored in registers. If a kernel requires more registers than available, the data is *spilled* into Local memory.

Local memory is part of device memory private for each thread, allocated automatically based on the requirements of the CUDA compiler. This type of memory is used for register spilling, arrays with non-constant indexing and large structures or arrays which would consume too much register space.

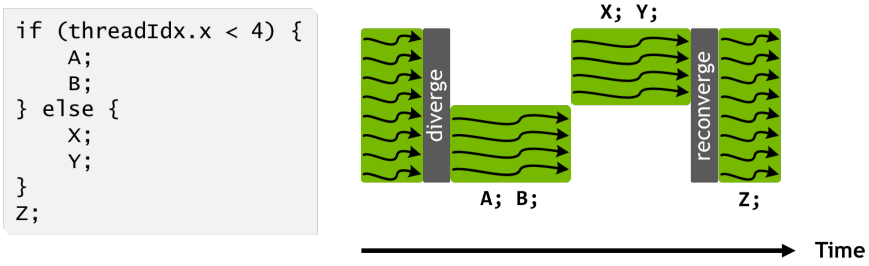
2.2.6 Hardware details

The abstraction defining the behavior of the device code is called the Single instruction, multiple threads(SIMT) execution model. In this execution model, threads on the device are split into groups of 32, called *warps*. Each warp of threads is scheduled together, starting at the same program address and executing in lockstep.[3]

If branching occurs, as can be seen in Figure 2.3, any branch that is taken by at least a single thread of a warp is executed by the whole warp, masking out any threads that did not take given branch. When masked, thread does

[3] Since Compute Capability 7.0 Volta, threads of a warp can be scheduled more independently and do not execute strictly in lockstep [4].

not execute any reads or writes, but still has to continue execution with other threads in the warp. This is most apparent in loops, where a single thread of a warp executing the loop thousand times will result in the whole warp executing the loop thousand times, even if other threads are masked and do nothing for most of the loops. This cuts the theoretical throughput by a factor of 32, as only one of the 32 threads does useful work.



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Figure 2.3: Branching in device [5].

On the other side of the spectrum, the SIMT execution model can be compared to the Single instruction, multiple data (SIMD) execution model, where the number of elements processed by a single instruction is directly exposed in the user code, compared to the SIMT model, where the user code itself describes a behavior of a single thread and the grouping of threads is abstracted by the platform.

To maximize performance, one must keep in mind the SIMT model, grouping into warps, thread divergence when branching, coalesced memory accesses etc.

NVIDIA GPUs are build around an array of *Streaming multiprocessors* (SM). SM of a GPU is similar to a core of a multicore CPU. Each SM has separate execution units, schedulers, register file, shared memory and L1 cache. An example of an SM can be seen in Figure 2.4. Each SM can have multiple schedulers, each scheduling up to one warp per cycle.

Each thread block is assigned to a single SM exclusively, and each SM can run multiple thread blocks at once. Warps of all thread blocks resident on the given SM are scheduled regardless of the thread block the warps belong to.

2.2.7 Versioning

When working with CUDA, there are two main parts of the platform which are versioned separately:

- CUDA Toolkit,
- GPU Compute Capability.

CUDA Toolkit represents the software development part of the CUDA platform, encompassing the CUDA runtime library, the *nvcc* compiler and other tools for development of the software.

GPU Compute Capability (CC) represents the features provided by the hardware. This includes the number of registers, memory sizes, set of instructions etc. In general, each consumer GPU generation corresponds to a new CC, such

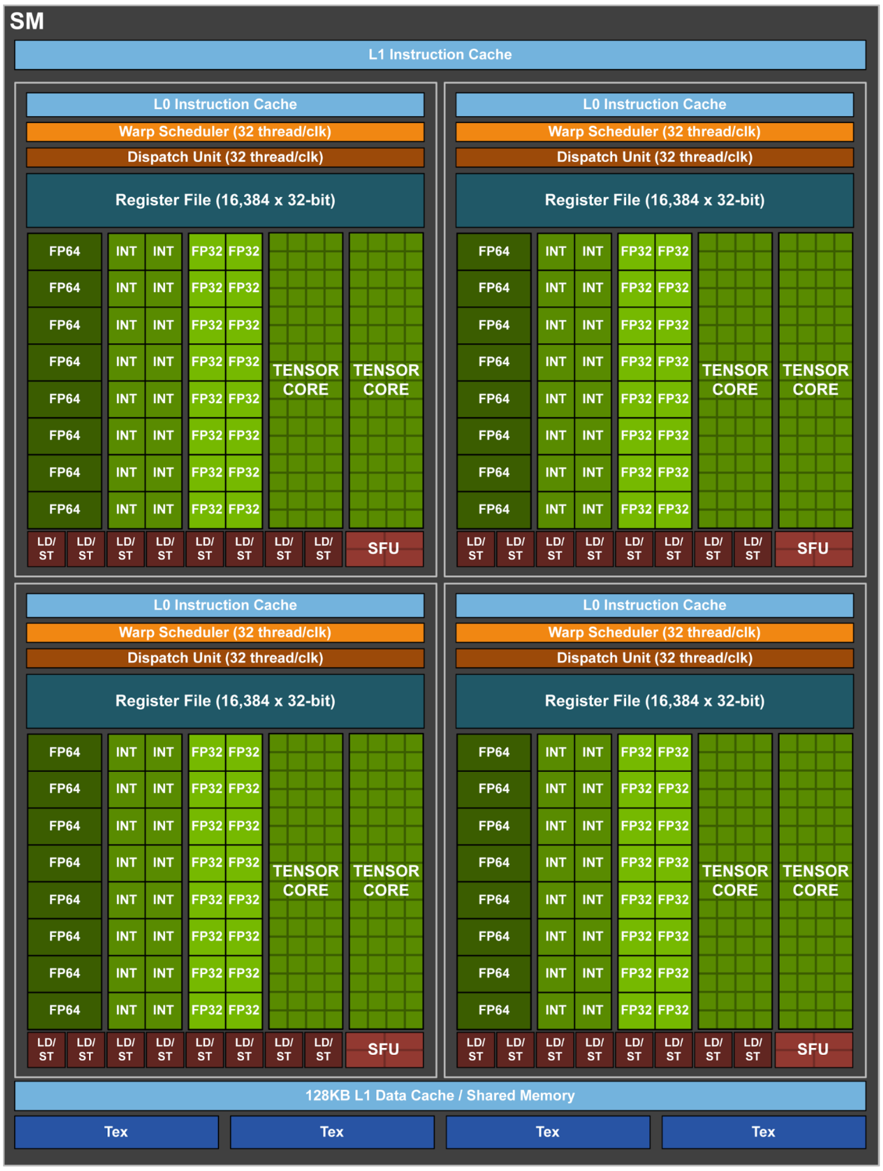

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
INT
FP32
FP32
TENSOR CORE
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex

Figure 2.4: Streaming multiprocessor [4].

as GTX 1000 cards corresponding to CC 6.0 Pascal and RTX 3000 cards corresponding to CC 8.0 Ampere. There are some exceptions, for example CC 7.0 Volta having only enterprise cards. With each release of new Compute Capability cards, there is generally accompanying CUDA Toolkit release providing access to the new features provided by the hardware.

Compute Capabilities are backwards compatible, so code created for older generation of cards can be ran on newer cards, even though it may not take advantage of new hardware features and may be inefficient on the newer cards.

2.3 Code optimizations

This section introduces basic principles for producing performant CUDA code. The observations and recommendations provided in this section are based on the principles and properties described in the previous section 2.2.

2.3.1 Occupancy

The GPU design prioritizes high instruction throughput of many concurrent threads over single thread performance at the cost of high latency of each instruction. To hide the high latency between dependent instructions, each scheduler keeps a pool of warps between which it switches, possibly on each instruction. Warps in a pool of a scheduler are called *active* warps. Each cycle, there may be multiple warps which have instructions ready to be executed. Such warps are called *eligible* warps. Each cycle, a warp scheduler can select one of the *eligible* warps as *issued* warp, issuing its instruction to be executed.

For optimal performance, we want to have enough active warps so that there is at least one eligible warp each cycle to enable the GPU to hide the high latency of each instruction. As described in Section 2.2.6, the number of warps resident on a SM depends on the number and size of thread blocks resident on a SM.

The number of thread blocks assigned to an SM is limited by three factors:

- hardware limit,
- register usage,
- shared memory usage.

The hardware limit differs, but is either 16 or 32 for all currently supported Compute Capabilities.

To enable no cost execution context switching (program counters, registers, etc.), the whole execution context for all warps is kept on the SM for the whole lifetime of each warp.

Number of registers used by all warps of all blocks which reside on the given SM must be smaller than or equal to the number of registers in the register file. For example, for SM with 65536 registers, code using 64 registers per thread and 512 threads in a block, there can only be two blocks resident on the SM, as $2 * 512 * 64 = 65536$. If the code requires just a single register more, only a single block will be resident on each SM.

The total amount of shared memory required by all blocks residing on an SM must be smaller than or equal to the size of shared memory provided by the SM.

2.3.2 Pipeline saturation

Other than occupancy, there are other possible reasons why no warp may be eligible in a given cycle. Pipeline saturation is one of such reasons. GPU hardware has several pipelines, each implementing a different part of the instruction set. As an example, for the RTX 2060 card, these include[7]:

- Load Store Unit (LSU),
- Arithmetic Logic Unit (ALU),
- Fused Multiply Add/Accumulate (FMA),
- Transcendental and Data Type Conversion Unit (XU).

Each instruction has a Compute Capability specific throughput. If this throughput is exceeded, the pipeline implementing the instruction becomes saturated and is unable to execute additional instructions. This becomes a problem when, for example, many or all warps often execute the same low throughput instruction, such as sinus, cosinus or inverse square root, which are implemented by the XU pipeline. Even for simpler operations implemented by the ALU or FMA, if all warps execute the same instruction, the pipelines may become saturated and warps which are waiting to execute more of the given instruction will not be eligible to be issued.

High LSU utilization reflects that the program may be memory bound, waiting for data from global or shared memory, or that the program executes many warp shuffle instructions, which are also implemented by the LSU pipeline. Due to this, the usage of shared memory together with warp shuffles is not advisable, as they both utilize the same pipeline and compete for resources.

2.3.3 Global memory access

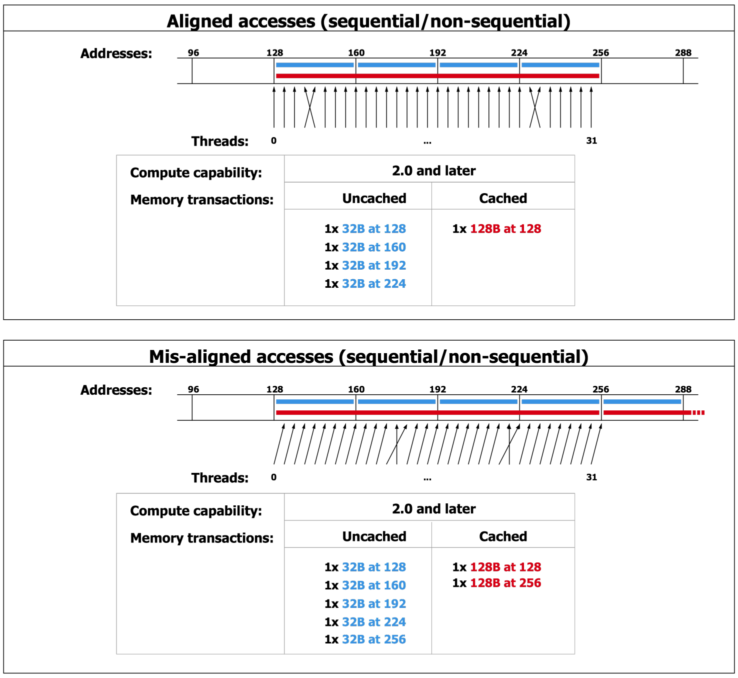
As shown in Figure 2.5, each access to global memory is grouped into 128 B naturally aligned chunks, where any chunk accessed by any of the threads of a warp has to be transferred from global memory. The maximum performance is achieved when access to memory is aligned and coalesced, i.e. all threads of a warp access elements in the same 128 B chunk which is aligned to 128 B. Any other form of access introduces overhead in a form of unnecessary data being transferred from global memory.

When accessing data larger than 32 bits, the access is split into 2 half-warp transactions for 64 bit or 4 quarter warp transactions for 128 bit values, which are then processed independently, again reading any 128 B chunk any of the accesses.

Access to global memory goes through at least one level of cache. On older architectures, global memory accesses are by default only cached in L2 cache, with L1 cache utilized only for local memory access to speed up register spills. Special instructions can be used to cache data in L1 explicitly. For Compute Capabilities newer than 5.0, compiler can generate an instruction to load read-only data, such as the two input matrices in cross-correlation, and cache it in L1 cache. L1 cache, with cache line size of 128 B, is local to each SM and shares hardware with shared memory, described in the following section. L2 cache, with cache line size of 32 B, is still on-chip but is shared by all SMs. Each 128 B memory transfer is either served by a single L1 cache access or split into 4 32 B L2 cache accesses.

2.3.4 Shared memory access

To achieve high bandwidth, shared memory is divided into 32 banks. The optimal access pattern the shared memory is designed for is for each thread of a warp to access a different bank. To enable this access pattern, successive 32bit words are mapped to successive shared memory banks. The simplest pattern is that the 32 threads of a warp access 32 consecutive 32bit items from an array in shared memory, shown in the left column of Figure 2.6. If multiple threads access different addresses mapped to the same bank, as shown in the middle column of the figure,


Aligned accesses (sequential/non-sequential)

Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned accesses (sequential/non-sequential)

Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Figure 2.5: Global memory access [5].

their accesses are serialized, the throughput of shared memory being divided by the maximum number of different addresses accessed in any of the banks. This is called a *bank conflict*. Read access of the same address by multiple threads does not lead to a bank conflict, instead leading to a broadcast of the value between the threads. Writes to the same address by threads without synchronization results in data-race and an undefined behavior, as does unsychronized read and write access.

2.3.5 General recommendations

We can summarize the information in previous subsections into few simple rules [5]:

1. Maximize parallel execution by ensuring that the workload is distributed between large enough number of threads, where each thread requires low enough number of registers and each thread block requires small enough part of shared memory so that enough thread blocks fit onto an SM;

2. Optimize memory usage by minimizing transfers from lower bandwidth memory by reusing data in hardware cache or manually moving data to shared memory. When accessing global memory, utilize coalesced accesses to minimize unnecessary data transferred. When accessing shared memory, minimize bank conflicts;

3. Optimize instruction usage by minimizing the use of low throughput instructions such as sinus, cosinus or inverse square root. When working with floating point numbers, use 32 bit numbers if precision is not crucial.

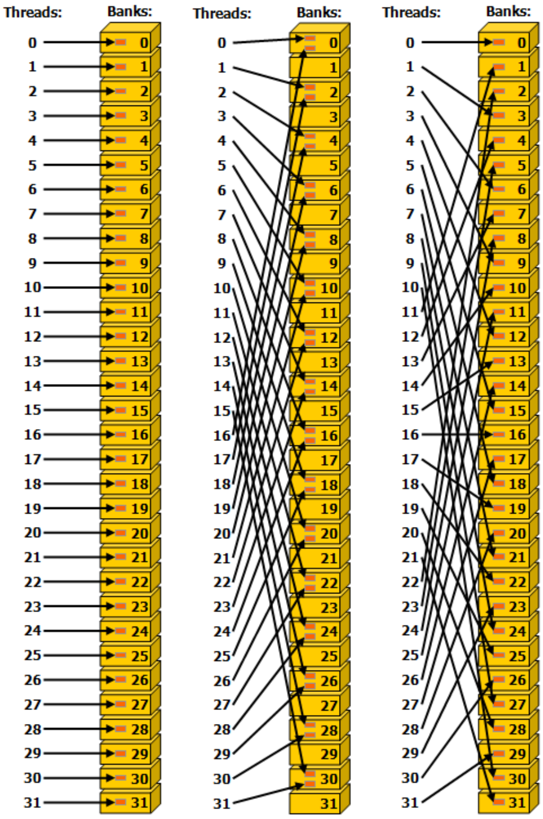


Figure 2.6: Shared memory access patterns [5].

Minimize thread divergence to ensure all threads in a warp execute useful instructions.

These rules are used in the design of optimized definition-based cross-correlation implementations described in the following chapter.

3. Implementation

In this chapter, we first give a high level overview of the possibilities for parallelization and data reuse in the implementation of definition-based cross-correlation algorithm, introduced in section 1.1. We then describe a basic implementation of the definition-based algorithm with all its problems. Next, we try to mitigate the problems of the basic implementation by introducing a simple implementation based on Warp Shuffle instructions. We continue with several optimizations of this implementation. Lastly we implement a solution to the problem of low occupancy for small inputs and go through some additional possibilities for optimization of this solution.

The definition-based algorithm has several properties which allow for parallelization, optimization through data reuse and distribution of work. Figure 3.1 depicts the output matrix with corresponding relative shift of the two input matrices, which defines the computed overlap. As described in Section 1.3, each element of the output matrix can be computed independently in parallel.

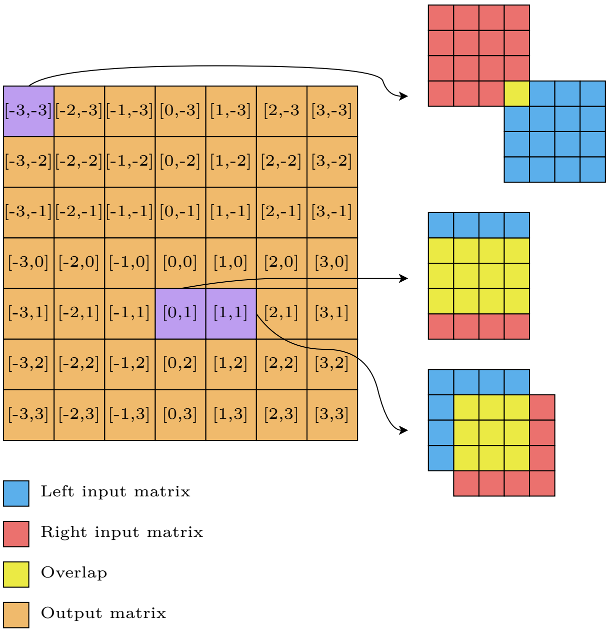


Figure 3.1: Result matrix with corresponding relative shifts.

Each overlap defines a unique set of element pairs which are to be multiplied. Each of these pairs of overlapping elements belongs to exactly one overlap of the two matrices.

3.1 Parallelization

In this section, we first highlight the independent parallel tasks present in the definition-based cross-correlation algorithm.

3.1.1 Two matrices

When we focus on the computation of cross-correlation between two matrices, called *one-to-one* in the rest of the thesis, we can reformulate the definition-based algorithm as a problem with two levels of independent parallel tasks, as shown in Figure 3.2. The top level represents the single output matrix, which with only two input matrices represents the result of the whole computation. The second level of tasks, represented by orange boxes, contains one box for each relative shift of the two input matrices, or in other words each orange box corresponding to a single element of the output matrix. Each shift defines an overlap of the two input matrices, which in turn defines a set of independent subtasks, each subtask representing an overlapping pair of elements of the two input matrices. In Figure 3.2, the subtasks representing pairs of overlapping elements are represented by yellow boxes. Every such subtask belongs to exactly one second level task, creating a tree structure. The set of subtasks which are children of the same orange box defines a submatrix in both input matrices, as shown in Figure 3.1.

When we look at the operations, every yellow box represents a single multiplication and every orange box represents a sum of the results of all of its children.

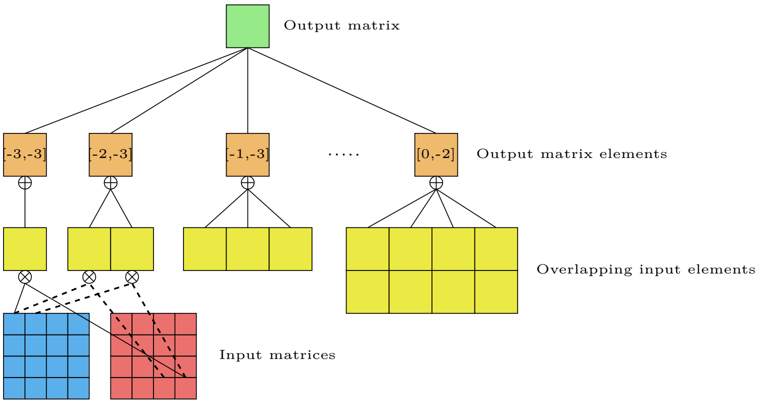


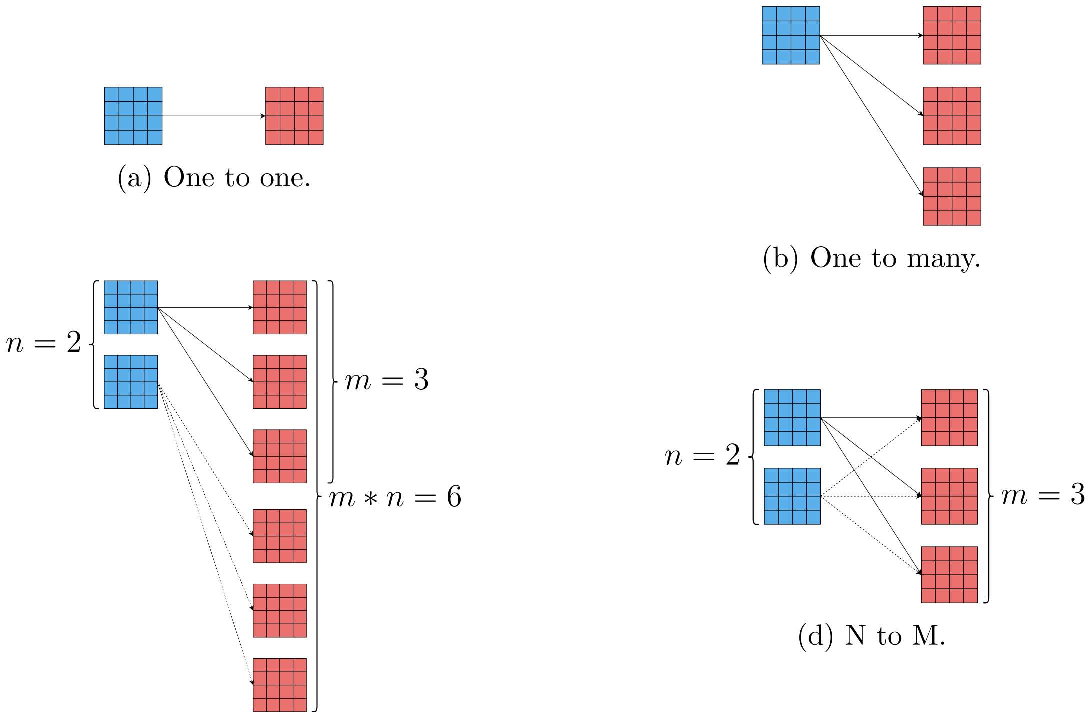
Figure 3.2: Tasks hierarchy in definition-based one-to-one cross-correlation.

The goal is to distribute the tasks between workers in such a way that we maximize parallelism, maximize data reuse and minimize the need for communication and synchronization between workers.

3.1.2 Many matrices

With more than two matrices, we add additional tasks to the top level of the task hierarchy shown in Figure 3.2, creating a forest of trees. As described in Section 1.3.2, there are several forms of cross-correlation between multiple matrices. For us, the most important of these are:

1. *one-to-many* (Figure 3.3b),
2. *n-to-mn* (Figure 3.3c),
3. *n-to-m* (Figure 3.3d).



(a) One to one.

(b) One to many.

(c) N to MN.

(d) N to M.

Figure 3.3: Forms of cross-correlation.

The *one-to-many* type together with the *one-to-one* type, shown in Figures 3.3b and 3.3a respectively, are subtypes of the *n-to-mn* type and *n-to-m*, as implementations of both of these general types can be used to compute the two simpler types. We separate the *one-to-one* and *one-to-many* types as they offer a greater possibility for caching the single left input matrix.

All of the described types can be partitioned into many *one-to-one* cross-correlations, as shown in Figure 3.4. For both *n-to-mn* and *n-to-m* types, the number of green top level tasks, corresponding to the number of result matrices, is equal to $n * m$. To reiterate, the difference between the *n-to-mn* and *n-to-m* types is that in the *n-to-mn* type, each of the n left input matrices is cross-correlated with a different set of m right input matrices, whereas in the *n-to-m* type, all n left input matrices are cross-correlated with the same m right input matrices. Based on this, the *n-to-m* type allows for greater data reuse, as each right input matrix is used multiple times compared to being used just once in the *n-to-mn* type.

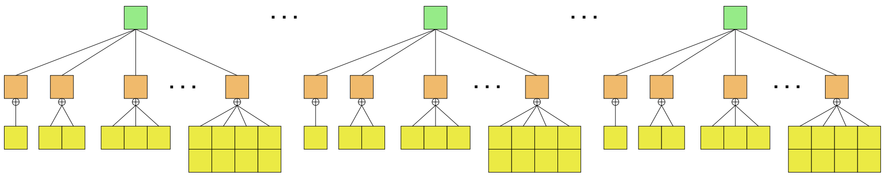


Figure 3.4: Task hierarchy of types with many matrices.

As in the case of *one-to-one* type, the meaning of the boxes is as follows:

- Each green box represents a pair of input matrices, or equivalently a single output matrix;

- Each orange box represents an element in the output matrix, or equivalently a relative shift of the two input matrices;
- Each yellow box represents a pair of overlapping elements from the two input matrices.

All boxes on a given level can be processed completely independently. Results of the orange boxes have to be written into the output matrix represented by their green box parent. Each orange box represents a different element of the parent matrix, which allows the writes corresponding to different orange box tasks to be executed without any collisions. All results of the yellow box children of an orange box have to be summed together.

As the number of tasks cannot be reduced, the only directions for optimization are parallelization and data reuse. Even through tasks can be processed independently and in parallel, many tasks can share and reuse data from other tasks. For example, orange boxes from different subtrees representing the same shift between input matrices and sharing the same left input matrix can be computed by reusing the data from the left matrix. Relationships such as this can be used to group tasks into **jobs** for workers to reuse data or pass data to other neighboring workers to reduce the required memory throughput and keep data closer to the execution units.

3.2 Data reuse

To fully utilize the GPU hardware, we cannot rely on parallelization alone, but must keep the data as close as possible to the execution units for fast access. To achieve this goal, we have to group tasks defined in the previous section 3.1 into cooperating groups, mostly for sharing input data. We call these groups **jobs**.

The two basic levels on which tasks are grouped into jobs in our implementations are:

- overlap,
- row group or column group.

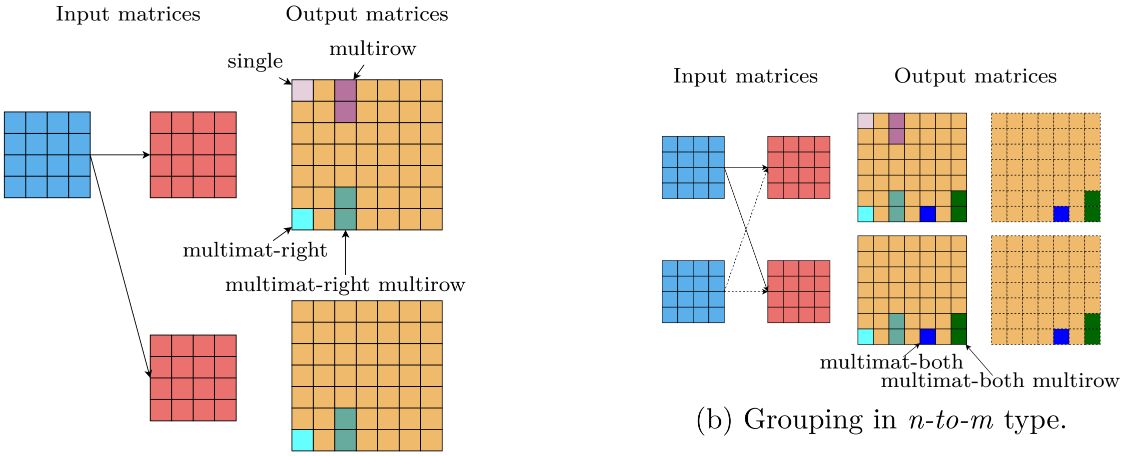
The following subsections describe groupings based on each of these units.

3.2.1 Overlap

With the basic job size of single *overlap*, each job contains a single element in the output matrix. This corresponds to a single orange box in the task hierarchy in Figure 3.4. The job then represents all tasks in the subtree with the assigned orange box as its root.

With *multimat overlap* job size, multiple overlaps (orange boxes) from different output matrices (green boxes) are grouped into a single job. To enable data reuse, overlaps representing the same shift between different matrices are grouped, as shown in Figure 3.5. There are two versions of this optimization:

- multimat-right, designed for *n-to-mn* computation type, where job contains overlaps with the same shift between single left input matrix and multiple right input matrices, shown in Figure 3.5a;



(a) Grouping in *n-to-mn* type.

(b) Grouping in *n-to-m* type.

Figure 3.5: Grouping of overlaps into jobs.

- multimat-both, designed for *n-to-m* computation type, where job contains overlaps with the same shift between multiple left input matrices and multiple right matrices, shown in Figure 3.5b.

Another way to reuse data is to compute multiple overlaps which are close to each other in a single output matrix. Their proximity in the output matrix corresponds to the shifts of the input matrices being very similar, which in turn means that most of the input data is shared between the overlaps, as shown in Figure 3.6. This grouping is also shown in Figure 3.5. Our implementation groups overlaps from multiple consecutive rows of a single column of the output matrix into a single job. The reasons for this grouping choice are further expanded in Section 3.4.5.

The *multimat* and *multirow* optimizations can be combined as shown in Figure 3.5.

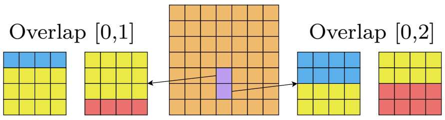


Figure 3.6: Input data shared between neighboring overlaps.

3.2.2 Row group and Column group

With overlaps as the tasks grouped into jobs, load balancing becomes a problem. The amount of work required by each job may differ massively, as illustrated by the two tasks shown by the two overlaps in Figure 3.7. This leads to problems with occupancy once jobs with small amount of work are finished.

We implement an alternative where instead of grouping overlaps into jobs, we go one step lower in the task hierarchy and group the individual tasks representing multiplication of one overlapping pair of input elements (yellow box) into jobs. This change results in the computation of a single overlap being split into several smaller jobs, improving load balancing between jobs while also increasing parallelism. One problem of this change is the required final reduce operation to consolidate the results of all jobs computing parts of the same overlap.

To enable additional optimizations described in the following sections of this chapter, we choose to group the yellow box tasks into jobs by whole rows of the overlap, with the maximum number of rows making up a job configurable through algorithm argument. This grouping is illustrated in Figure 3.7, where *Max job rows* is the argument of the algorithm. Each overlap with r overlapping rows of the two input matrices is split into $\lceil \frac{r}{max\_job\_rows} \rceil$ jobs, with each job containing at most max_job_rows consecutive rows. With this distribution, each job represents a submatrix of the overlap with the same number of columns as the original overlap. This also means that all overlaps with the same number of rows will be split into the same number of jobs.

For data reuse, one important observation is that jobs from the same overlap do not share any input data, but jobs with the same ID from neighboring overlaps in the same row of the output matrix, i.e. with the same number of rows in the overlap, will share most of the input data, as shown in Figure 3.8.

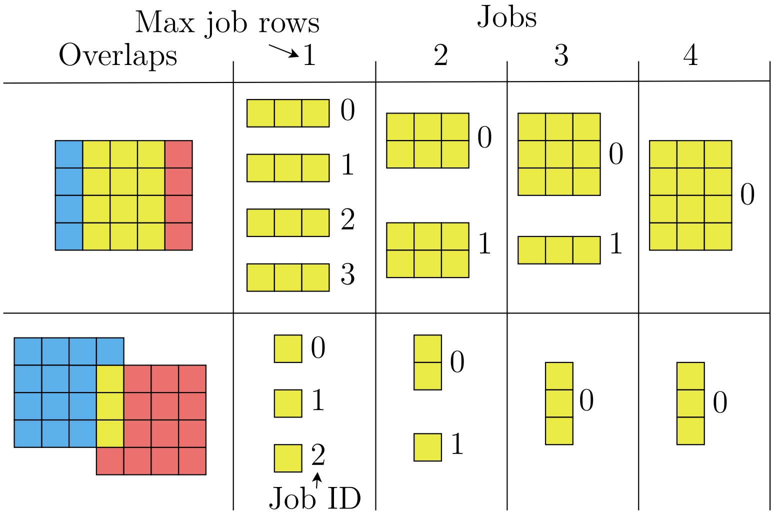


Figure 3.7: Grouping of rows of tasks into jobs.

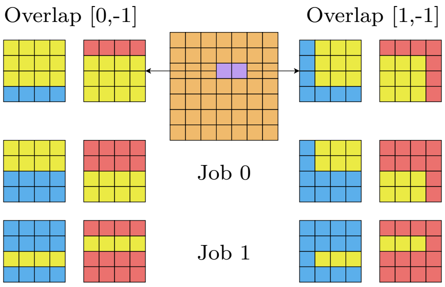


Figure 3.8: Data shared between jobs with the same ID from neighboring overlaps.

For some optimizations, it is better to group tasks by columns instead of rows. From the data reuse point of view, it is symmetrical to the grouping by rows. Again, column groups from the same shift do not share any input data, but column groups with the same ID from two neighboring overlaps in the same column of the output matrix share most of their input data.

The *multimat* optimization introduced in Section 3.2.1 can also be used with both row group and column group task grouping. The *multimrow* optimization is unfortunately incompatible with both row group and column group task grouping.

3.2.3 Workers

Jobs defined in the previous section are assigned to one level of the CUDA thread group hierarchy, described in Section 2.2.2. The CUDA thread hierarchy contains the following groups of threads:

1. thread,
2. warp,
3. thread block,
4. grid.

We call each member from the chosen thread hierarchy level, i.e. the thread, warp, thread block, or grid, a **worker**. Each worker is assigned at most one job, but due to the fixed warp size and limited size of thread blocks, we might need to start more workers than jobs and later stop the redundant workers. Based on the choice of worker size, we can utilize smaller groups in the hierarchy to compute tasks of the assigned job, and primitives provided by larger groups to exchange input data or combine results with other workers.

3.2.4 List of implementations

The different implementations of the definition-based cross-correlation utilize different job sizes, worker types and different ways of assigning jobs to workers. The chosen parameters then lead to different ways of cooperation, communication, synchronization, work distribution and load balancing. Following is the list of algorithms implemented in this thesis with their choice of job size and worker type. Each algorithm is described in more detail in the following sections of this chapter.

Algorithm	Job size	Worker type
Basic	overlap	thread
Warp shuffle	overlap multimat overlap multirow overlap multimat multirow overlap row group multimat row group	thread
Warp per shift	overlap row group	warp
Warp per shift with shared memory	overlap column group multimat column group	warp
Block per shift	overlap	thread block

Algorithms with multiple job sizes are provided in multiple implementations, each implementing different optimization for data reuse.

3.3 Basic algorithm

A basic implementation of definition based cross-correlation of two input matrices, in our naming scheme corresponding to the *one-to-one* type, utilizes two dimensional grid of two dimensional thread blocks to start one thread for each element of the output matrix. The overlap to be computed by given thread is derived from the position of the output matrix element, as shown in Figure 3.9. The thread then iterates over the overlapping parts of the two input matrices, multiplying the overlapped elements and accumulating the result which is then written to the assigned output matrix element.

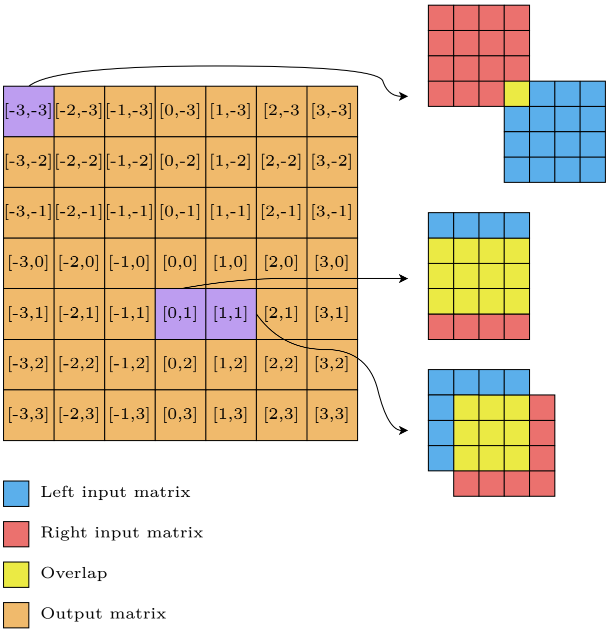


Figure 3.9: Examples of overlaps corresponding to output matrix elements.

This implementation allows for great amount of parallelism, as each element of the output matrix is computed independently. The disadvantages on the other hand are no data reuse, thread divergence, and a large difference between workloads assigned to each thread.

The problem with no data reuse is illustrate by Figure 3.10. The figure shows elements of the left input matrix (in blue) and right input matrix (in red) accessed by thread 0 and thread 1 in iterations 0 to 11. As memory accesses are done in units of 32 threads, also known as warp, when we extrapolate this example, the 32 values loaded from the left input matrix in global memory will contain 31 values read in the previous iteration by the threads of this warp. In other words, we are moving a window of 32 elements across each row of the left matrix 1 element per iteration and reading the window from global memory in each iteration. This problem is exacerbated with thread blocks with the x component of their size not multiple of warp size. In this situation, threads of a warp are assigned overlaps which differ in number of rows, and consequently in the rows of the input matrices which are to be read in the given iteration. This means that threads of a single warp access two or more different rows of the left input matrix in each iteration, leading to non-coalesced loads.

In the right input matrix, all threads of the warp will read the same value from global memory in each iteration, resulting in 32 reads of the same block of global memory when reading 32bit values.

The exact pattern of reads from global memory differs based on the overlaps processed by the threads of the warp, but will always result in repeated reads of the same block of global memory.

The problem of thread divergence is also illustrated by Figure 3.10 with iterations 3, 7 and 11. The basic implementation of looping over a two dimensional matrix with two nested for loops results in thread 0 reading an element from each input matrix, computing multiplication and accumulating the result in these iterations, while thread 1 has no data to process on this row and its inner for loop being masked due to the range condition being false. While we show only two threads in Figure 3.10, the situation is much worse as 31 threads of the 32 threads of the warp will be masked and not doing anything in the last iteration. All threads of the warp will go through both for loops based on the size of the largest overlap in the given axis processed by any thread of the warp.

Last significant problem of the Basic algorithm, largely connected with the previous problem, is that overlaps differ in size. From overlaps containing one element from each input matrix to overlap of the whole input matrices, this difference in workload size means that some threads will finish rather quickly, while small number of threads computing the largest overlaps will take much longer. For example warps of threads assigned overlaps in the first or last row of the output matrix will only read a single row of each input matrix, and finish quickly, while warps assigned overlaps in the middle of the output matrix will read most elements of each input matrix and run for much longer. This results in problems with occupancy of the GPU.

Slight modification of this algorithm to allow for an *n-to-mn* computation is implemented by the original thesis Bali [1].

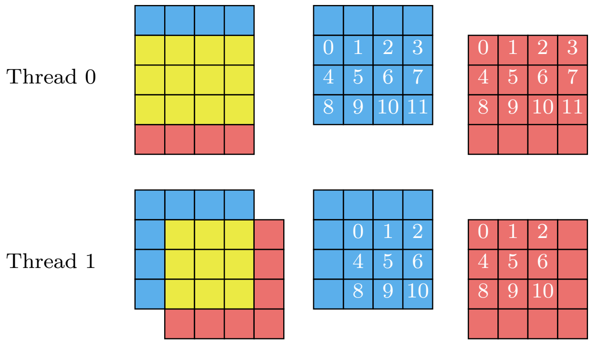


Figure 3.10: The iteration in which each input matrix element is accessed by two neighboring threads.

3.4 Warp shuffle algorithm

This section describes the implementation of definition-based cross-correlation utilizing Warp Shuffle instructions, which tries to fix the problems of the Basic algorithms described in the previous section. We first introduce a simple version of the implementation, later improving it step by step with optimizations evaluated in Section 4.2.2.

In the simplified implementation, Warp Shuffle instructions are utilized to shuffle data loaded from the left matrix and broadcast data loaded from the right matrix between threads in a warp. As shown in Section 3.2.4, each CUDA thread is assigned an overlap to compute the same way as in the Basic algorithm.

The main idea behind this algorithm can be illustrated by using Figure 3.10. The two threads computing overlaps with shifts $[0, 1]$ and $[1, 1]$ process elements of the two input matrices in the indicated iterations of a *for loop* in the code.

The element from the left matrix read by the thread processing shift $[x, y]$ in iteration i is required by thread processing shift $[x + 1, y]$ in iteration $i + 1$. This holds for any two neighboring shifts and maps exactly onto the Warp Shuffle Down function, described in Section 2.2.3. When looking at the right matrix in any given iteration, both shifts require the exact same element from the right matrix. This broadcast can be implemented using the general Warp Shuffle function with direct source lane indexing. Iterations 3, 7, and 11 are skipped by thread 1 to preserve this property.

To utilize these properties, threads of a single warp process 32 consecutive overlaps in a row of the output matrix, as shown in Figure 3.11. The x axis of *thread block size* is set to *warp size* for simplified grouping of threads into warps. The number of warps per thread block is configurable using a run-time algorithm argument. The grid size is set so that the output matrix is fully covered by threads. Any redundant threads do not write to the output matrix and during computations are handled by the bound checked reads described next.

The problem of iteration 3, 7 and 11 in Figure 3.10, in which thread 1 does not have any value to compute, can be solved in several ways. If we were programming for a CPU, we would give the two for loops implementing the two worker threads different bounds so that the second worker stops earlier. A more GPU friendly implementation needs to prevent thread divergence. This is achieved by executing the range check only once when loading the data from the left matrix into a register of the thread. If the thread is loading value outside the matrix, it loads 0 instead. This makes the result of the multiplication performed in each step 0 which is then added to the sum, effectively skipping the iteration while preventing thread divergence. This also handles any extra threads introduced due to the fixed size of thread block and overlap sizes not divisible by warp size.

3.4.1 Algorithm steps

The following description assumes *warp size* to be 32, as is the case for all currently existing Nvidia GPUs. To utilize coalesced loading from global memory, the buffer that is shuffled between threads of a warp, containing data from the left input matrix, is split into two parts of 32 items each, which together function as a single 64 item ring buffer. We call the parts *top* and *bottom*, where new data

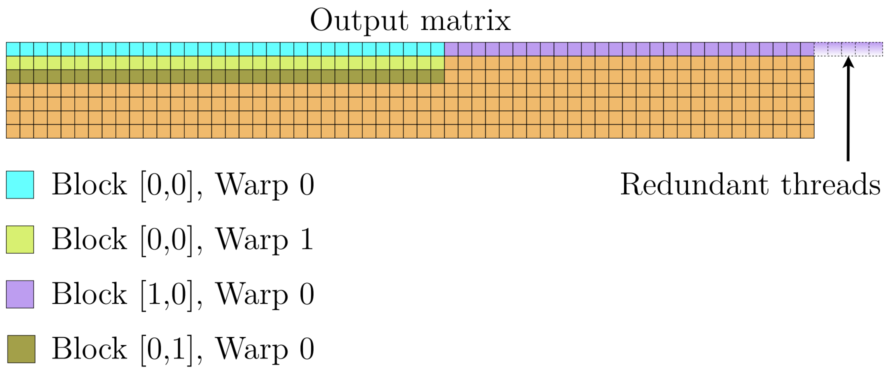


Figure 3.11: Distribution of overlaps in a 7 by 59 output matrix between threads of thread blocks and warps.

is always loaded to the top part and then shuffled towards the bottom part, as shown in Figure 3.12. As described above, any out of bounds loads are range checked and load value 0 instead using the following code:

```
template<typename T>
__device__ T load_with_bounds_check(const T* source, int idx,
    size_t size) {
  return (idx >= 0 && idx < size) ? source[idx] : 0;
}
```

When loading a buffer, bound checked load shown above is used to load 32 consecutive values (or load the value 0 if out of bounds), storing one item per thread into a register. This is implemented by the following code:

```
T thread_left_bottom = load_with_bounds_check(
  left_row,
  warp_x_left + warp.thread_rank(),
  matrix_size.x
);
```

Each thread holds single value of `thread_left_bottom`. Same is done for `thread_left_top`, creating a 64 item ring buffer distributed between threads of a warp which is shuffled using the Warp Shuffle instructions as shown in Figure 3.12.

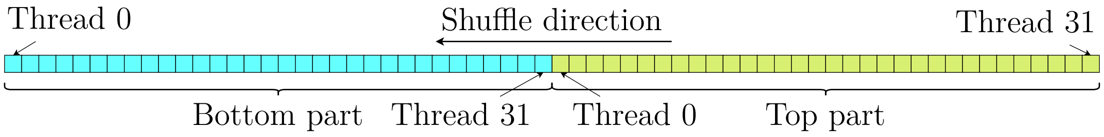


Figure 3.12: Shuffling of the left buffer.

For the data loaded from the right matrix, which we use for broadcasting, we require only a single value per thread, as in the 32 iterations of the innermost loop, we will broadcast only these 32 values, whereas we need 64 from the left matrix as the 32 values loaded into the top part will be shifted all the way to the bottom part.

The outer loops of the algorithm are illustrated by the following code:

```
// Compute bounds of the overlaps processed by warp
// Comput thread output position

T sum = 0;
/* for each overlapping row in the right input matrix */
for (size_t warp_y_right ... ) {

  // Corresponding row in the left input matrix
  int warp_y_left = warp_y_right + warp_min_shift.y;

  // First column in the left input matrix
  // corresponding to the first column in the right input matrix
  int warp_x_left = warp_x_right_start + warp_min_shift.x;

  // Preload 1 value to each thread from the input left matrix
  T thread_left_bottom = load_with_bounds_check(left_matrix, ...);

  for (size_t warp_x_right ...; warp_x_right += warp.size()) {
    int warp_x_left = warp_x_right + warp_min_shift.x;

    T thread_left_top = load_with_bounds_check(left_matrix, ...);
    T thread_right = load_with_bounds_check(right_matrix, ...);

    /* Main loop shown below, with warp.size() iterations */
  }
}

if (/* thread output not out of bounds */) {
  out_matrix[output_offset] = sum;
}
```

The 32 threads of a warp are assigned 32 consecutive overlaps, where thread i computes overlap with shift $[warp\_min\_shift.x+i, warp\_min\_shift.y]$. Thread 31 (or possibly earlier thread if there are redundant threads in the warp) is then assigned shift $[warp\_max\_shift.x, warp\_max\_shift.y]$, which in a warp without redundant threads is equal to $[warp\_min\_shift.x + 31, warp\_max\_shift.y]$, otherwise the x component is clamped to the maximum shift defined by the output matrix. The *min* and *max* shifts are then used to compute a submatrix of the right input matrix which contains elements required by any thread in the warp, shown in Figure 3.13. We call this the *warp submatrix*.

The two loops in the code above then iterate over this submatrix of the right input matrix, outer loop iterating over rows of the submatrix and the inner loop iterating in buffer load size steps over the columns of the submatrix. As described above, both buffers are loaded 32 items at the time to allow for coalesced loads.

The innermost loop, which we call the *main loop* in the code above, then does 32 (warp size) iterations illustrated by the following code:

```
for (size_t i = 0; i < warp.size(); ++i) {
```

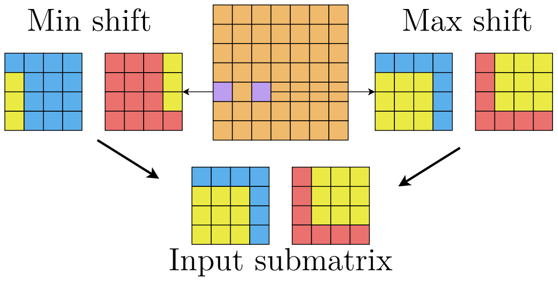


Figure 3.13: Submatrix of input data used by any thread in the warp.

```
  // Broadcast right buffer
  auto right_val = warp.shfl(thread_right, i);
  sum += thread_left_bottom * right_val;

  // Shift left buffer
  // General shuffle does module on source lane argument
  // Thread 0 needs to connect the top buffer to the bottom buffer
  thread_left_bottom = warp.shfl(
    warp.thread_rank() != 0 ? thread_left_bottom : thread_left_top,
    warp.thread_rank() + 1
  );
  thread_left_top = warp.shfl_down(thread_left_top, 1);
}
```

In iteration i we broadcast value stored in the part of the *right buffer* owned by thread i, which is then multiplied by each thread with the value from the *bottom left buffer* stored by given thread. We then shuffle the whole *left buffer* one step towards the index 0 of the bottom part of the buffer using two warp shuffle instructions. After the 32 steps, the top part of the *left buffer* is now in the bottom part of the *left buffer* and all the values from *right buffer* have been broadcast.

The next iteration of the middle loop then loads 32 values to the top part of the *left buffer* and 32 values to the *right buffer* which are then processed by the *main loop*. This is repeated until the whole row of the warp submatrix is processed, where we continue to next iteration of the outer loop and process the next row. This is repeated until the whole warp submatrix is processed.

3.4.2 Work distribution

The work distribution optimization changes job size from a whole overlap used by the simplified implementation to a row group, as described in Section 3.2.2. Each job is still assigned to a single thread, with jobs of the same ID from 32 consecutive overlaps in a row of the output matrix assigned to threads of a single warp. This enables us to again compute the warp submatrix of the right input matrix containing elements used by all threads of the warp and reuse the core computation code of the simplified implementation described in the previous section without any change by just providing different bounds to the outer and middle for loops.

As there are multiple workers computing single element of the output matrix, we need to sum all their results together. Because it is only a single write of a single value per worker, utilizing the *atomicAdd* operation on the output matrix in global memory is sufficient. It also allows us greater freedom of assigning tasks to workers across the whole grid compared to grouping workers for the given overlap into a thread block, which would be required to utilize shared memory for communication. The maximum number of rows in a task is provided as a run-time argument to the algorithm, and influences the number of workers created.

In the simplified algorithm, the output matrix was covered by a two dimensional grid of two dimensional thread blocks. The x component of the thread block size was set to warp size for easier implementation of the data shuffling, and y component (number of warps per thread block) set using a run-time argument. With work distribution, we leave the block size the same, i.e. x component fixed to warp size and y component configurable by run-time argument, but instead of the grid just covering the output matrix, it is extended in the y axis to start at least as many workers as there are jobs. We call the y axis rank ($threadIdx.y + blockIdx.y * blockDim.y$) of each thread the **worker ID**. The use of the x axis rank of the thread is unchanged from the simplified implementation. We use the combination of the x axis of the rank and *worker ID* to assign an overlap, i.e. an element of the output matrix, to the worker. We further use *worker ID* to assign a *job ID* within the assigned overlap to the worker. Thanks to the unchanged use of the x axis rank, all threads of a warp will be mapped to 32 consecutive overlaps in a row of the output matrix same as in the simplified algorithm, but also to the same job ID as they all share the same y axis rank.

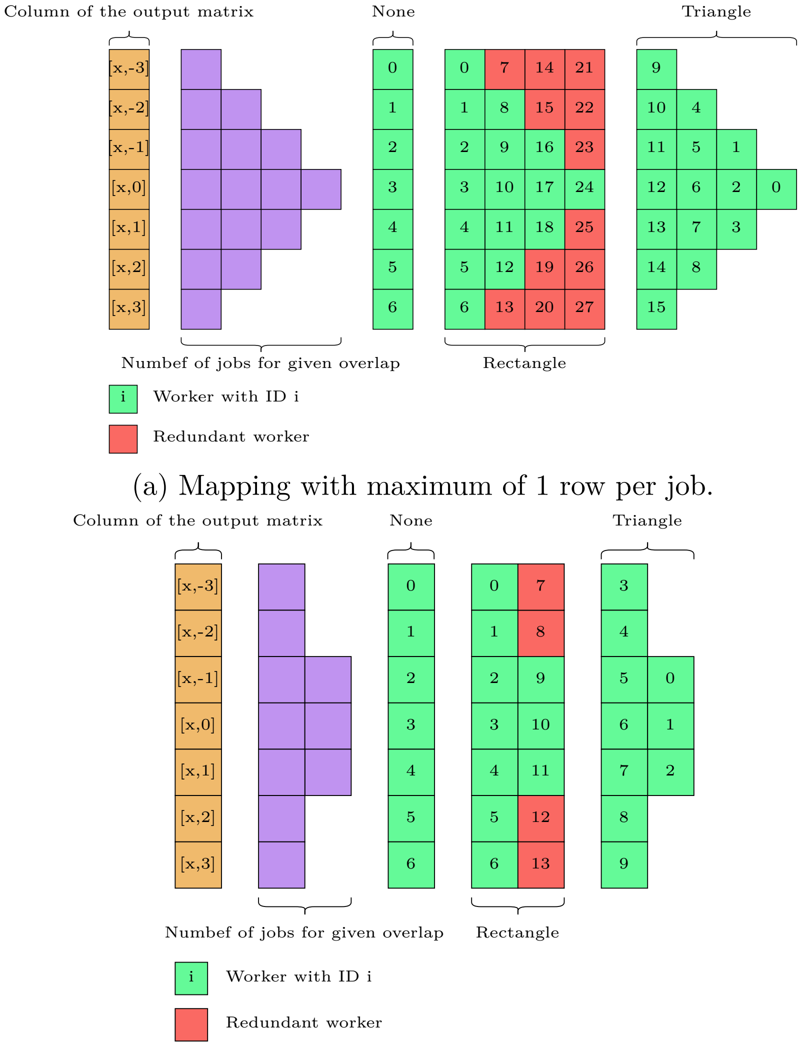
We provide several algorithms to derive the number of workers started for given total number of jobs and the mapping from worker ID to the overlap and the job ID. The provided algorithms are:

- None,
- Rectangle,
- Triangle.

The algorithms are illustrated in Figure 3.14. The purple boxes represent number of jobs for each overlap in the given row of the output matrix. As row groups are made up of rows and all overlaps in a given row of the output matrix have the same number of rows, they will also have the same number of jobs.

None distribution

The *None* distribution behaves exactly the same as simplified implementation introduced in Section 3.4.1, keeping the job size at overlap. Grid covers the whole output matrix only once and the worker ID is used to assign the row of the output matrix, as is done in the simplified implementation. This distribution is provided mainly to measure the overhead of the code changes required to implement work distribution.



(a) Mapping with maximum of 1 row per job.

(b) Mapping with maximum of 2 rows per job.

Figure 3.14: Mapping of a column of workers onto a column of the output matrix.

Rectangle distribution

The *Rectangle* distribution computes m, the maximum number of jobs any overlap will be split into. This is always the number of jobs the overlaps with the y of their shift equal to 0, for which all rows of both input matrices overlap. The output matrix is then covered by the grid m times, mapping m workers to each overlap so that they can be assigned the at most m jobs each overlap will be split into. To get the output matrix row from *worker ID*, we compute $\lfloor \frac{worker\_ID}{output\_matrix\_size.y} \rfloor$. Job ID is then computed as $worker\_ID$ modulo $output\_matrix\_size.y$. The redundant workers are stopped immediately after the work assignment step. As whole warps share the same *worker ID*, either a whole warp will be assigned work or stopped, leading to no thread divergence.

Triangle distribution

The *Triangle* distribution extends the grid to contain the exact number of rows of thread blocks required. Due to the user configured number of warps per thread block, there will most often still be some redundant workers, but always less than one row of thread blocks worth of workers.

The disadvantage of this distribution is the complex computation required to assign overlaps and job IDs to workers. This computation includes many multiplications, divisions and most importantly a low throughput square root instruction. For small job sizes the overhead of triangle distribution may be greater than any gains provided by better load balancing and increased occupancy.

To map worker ID to output matrix row and job ID, we use the following formula, which computes the worker ID i of the worker at the start of the row x of the triangle, shown in Figure 3.14a (where the triangle is rotated 90 degrees clockwise):

$$r * x^2 - (3r - t) * x + (2r - t) = i \tag{3.1}$$

where r is the algorithm argument *maximum number of rows per worker*, t is the number of workers on the top row of the triangle, x is workers row in the triangle, corresponding to worker rank, and i is worker ID. To simplify the equation, we use one-based indexing. For example in Figure 3.14a, we have $t = 1$ and $r = 1$, which results in the following values:

Row	1	2	3	4
Worker ID	0	1	4	9

For Figure 3.14b, we have $t = 3$ and $r = 2$ which results in the following values:

Row	1	2
Worker ID	0	3

Solved for x, we get the following formula:

$$x = \lfloor \frac{(3r - t) + \sqrt{(3r - t)^2 - 4r(2r - t - i)}}{2r} \rfloor \tag{3.2}$$

which is the positive solution of the quadratic equation rounded down. For any worker ID on the row x, this formula will return row x when given the worker ID as i.

To compute the variable t, the number of items on the top row of the triangle, we use the following equation:

$$t = \mathit{output\_matrix\_size.y} \mod (2 * r) \tag{3.3}$$

Lastly to compute the height of the triangle h, i.e. the number of rows of the triangle, we use the following equation:

$$h = \lceil \frac{\mathit{output\_matrix\_size.y}}{2 * r} \rceil \tag{3.4}$$

To map the worker ID i to row of the output matrix and job ID in the corresponding overlap, we first compute the triangle row x of the worker using equation 3.2. We then compute the output matrix row assigned to the first worker on row x as $first\_out\_row = r * (h - x)$. We also compute the worker ID of the first worker on row x using equation 3.1 and call it $first\_id$. The output matrix row assigned to worker i is then computed as $first\_out\_row + (i - first\_id)$. Job ID is then equal to $h - x$, i.e. the row counted from the bottom of the triangle.

As with the rectangle distribution, worker ID is derived only from the y component of thread rank, so all threads in a given row of the whole grid have the same worker ID, which also means that all threads of each warp have the same worker ID. This again enables us to stop any redundant worker right after job assignments, as only whole warps may be redundant.

3.4.3 Local array optimization

The following optimizations heavily rely on the ability of the CUDA compiler to place arrays such as the following into registers:

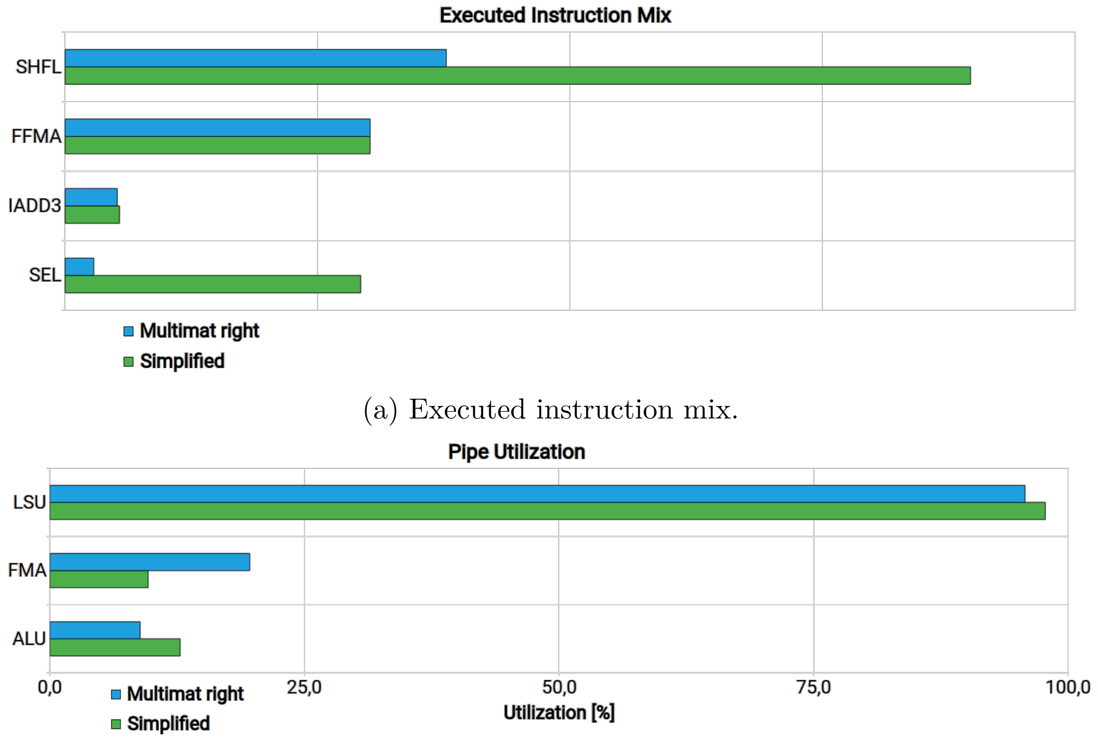
```
template<size_t LENGTH>
__device__ void foo(...) {
  ...
  float bar[LENGTH];
  for (size_t i = 0; i < LENGTH; ++i) {
    bar[i] = some_function(...);
  }
  ...
}
```

If an array is small and only accessed using static indexing, where all indices are known constants at compile time, the CUDA compiler places all elements of the array into registers. The array can also be accessed in small for loops with known compile time bounds, which are unrolled by the compiler and again result in static indexing. If for any access the index cannot be computed during compile time, the whole array is placed into Local memory, described in Section 2.2.5. Local memory is part of the device memory, and as such is the slowest memory accessible from device code. Local memory access also utilizes the same pipeline as warp shuffle instructions, competing for the throughput of this pipeline.

3.4.4 Utilizing multiple right matrices

Another problem of the simplified implementation not solved by work distribution described in Section 3.4.2, is the ratio of warp shuffle instructions to arithmetic instructions. In the simplified algorithm, for each multiplication and addition represented by a single yellow box, we must execute three warp shuffle instructions. This makes warp shuffle instructions the bottleneck in the simplified implementation, as shown in Figure 3.15a. The warp shuffle instructions (SHFL) dominate the executed instruction mix, which results in 97% utilization of the *LSU* pipeline implementing the shuffle instructions, as shown in Figure 3.15b. Compare this to the fused multiply-add instructions (FFMA) implementing the multiplication and addition, which are executed by the *FMA* pipeline with less than 10% utilization for the simplified algorithm.

There are several ways to improve the ratio of SHFL instructions to FFMA instructions. The one with simplest changes to the code of the simplified implementation is to utilize the *one-to-many* type of computation, and let each worker



(a) Executed instruction mix.

(b) Pipeline utilization.

Figure 3.15: Comparison of *one-to-many* simplified algorithm with the multiple right matrix optimization.

compute cross-correlation between one left matrix and many right matrices at once, as described in section 3.2.1 under the name *multimat*. We call this exact implementation usable for *one-to-many* and *n-to-mn* types the *multimat_right* optimization, as each job contains multiple overlaps of a single left matrix with multiple right matrices. The obvious advantage is data reuse, as the data from the left matrix is used to compute multiple results. The main advantage is that each additional right matrix only adds a single SHFL instruction, while also adding one FFMA instruction. The ratio of SHFL to FFMA instructions can then be expressed as $2 + r : r$, where r is the number of overlaps (from r right matrices) computed by each worker, which for any value greater than 1 is much improved from the $3 : 1$ ratio of the simplified warp shuffle algorithm.

The code changes required to implement this are very straight forward. As each job contains the same overlap between one left matrix and multiple right matrices, the three for loops and their bounds described in Section 3.4.1 are left unchanged. The main difference is that the *sum* and the *thread_right* variables in each thread are changed into arrays, utilizing the CUDA local array optimization described in Section 3.4.3, as shown in the following snippet:

```
template<size_t NUM_RIGHTS, typename T>
__device__ void warp_shuffle_multimat_right_impl(...) {
  ...
  T sum[NUM_RIGHTS];
  for (size_t r = 0; r < NUM_RIGHTS; ++r) {
    sum[r] = 0;
  }
  ...
  T thread_right[NUM_RIGHTS];
  for (size_t r = 0; r < NUM_RIGHTS; ++r) {
    thread_right[r] = load_with_bounds_check(...);
  }
  ...
  for (dsize_t r = 0; r < NUM_RIGHTS; ++r) {
    // Broadcast right buffer
    auto right_val = warp.shfl(thread_right[r], i);
    sum[r] += thread_left_bottom * right_val;
  }
}
```

We introduce the term *matrix group*, describing the right input matrices from which overlaps are grouped into a single job. As the size of the matrix group must be known at compile time to utilize the CUDA local array optimization, we need to replicate the *device* function implementing the algorithm for each supported matrix group size. This introduces a trade-off between compilation time, generated code size and the number of required registers on one hand and possible gains during run-time on the other. The actual matrix group size used during run-time is specified as run-time argument for the algorithm, which then chooses the correct implementation. If the number of matrices is not divisible by the matrix group size, the last few matrices will form a smaller matrix group which will choose the implementation based on its size. The maximum supported matrix group size is configurable during compile-time.

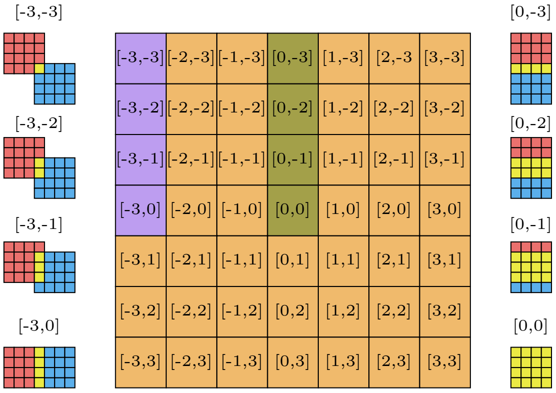
As with the simplified algorithm, thread block size is fixed to warp size in the x axis and configurable by a run-time argument in the y axis. The grid is then extended, this time in the x axis (to allow simple combination with work distribution) to cover one output matrix from each matrix group. For each of these output matrices, there might be redundant workers due to the fixed size of thread block. These workers are handled the same way as in the simplified algorithm, by not writing to any output matrix or being stopped immediately if whole warp is redundant. Due to the x component of the thread block size set to warp size, there will not be a redundant warp due to grid extension, but there may still be redundant warps due to the fixed thread block size in the y axis as in the simplified implementation.

Thanks to separately covering each output matrix, all threads of each thread block are always assigned overlaps from the same matrix group, which allows for reuse of much of the simplified algorithm code.

The effects of this optimization shown in Figure 3.15, which compares the simplified algorithm against the optimized algorithm using 8 right matrices (default maximum group size due to compile times) per worker on input of size 256x256 with 1 left matrix and 16 right matrices, which are enough to saturate the RTX 2060 used for profiling. As we can see, the `LSU` pipeline is still a bottleneck even for the optimized algorithm, but the utilization of the `FMA` pipeline, which does the useful part of the computation, has increased from 9% to 20%. The most visible change is in the mix of the executed instructions, where we see a very noticeable improvement in the ratio of shuffle instructions (SHFL) to the floating point fused multiply-add instructions (FFMA). As expected, the ratio improves from 3 : 1 to 10 : 8. The small improvement in the `LSU` pipeline utilization can be explained by the comparatively low throughput of the SHFL instructions compared to the FFMA instructions. This is exacerbated by our use of Compute Capability 7.5 card for profiling, which has half the warp shuffle throughput of all other Compute Capabilities.

3.4.5 Multiple rows from the right matrix

Another way to improve the instruction ratio is to process multiple overlaps from the same output matrix, which can be used even for the *one-to-one* type of computation. We call this the *multirow_right* optimization based on how rows of the input matrices are loaded and processed. With this optimization, multiple consecutive overlaps from multiple rows of a single column of the output matrix are grouped into a single job. This is illustrated in Figure 3.16, where we see two different jobs, each containing 4 overlaps, i.e. *job_size* is 4.



[-3,-3]	[-2,-3]	[-1,-3]	[0,-3]	[1,-3]	[2,-3	[3,-3]
[-3,-2]	[-2,-2]	[-1,-2]	[0,-2]	[1,-2]	[2,-2]	[3,-2]
[-3,-1]	[-2,-1]	[-1,-1]	[0,-1]	[1,-1]	[2,-1]	[3,-1]
[-3,0]	[-2,0]	[-1,0]	[0,0]	[1,0]	[2,0]	[3,0]
[-3,1]	[-2,1]	[-1,1]	[0,1]	[1,1]	[2,1]	[3,1]
[-3,2]	[-2,2]	[-1,2]	[0,2]	[1,2]	[2,2]	[3,2]
[-3,3]	[-2,3]	[-1,3]	[0,3]	[1,3]	[2,3]	[3,3]

Figure 3.16: Overlaps grouped into two different jobs by the *multirow_right* algorithm with 4 overlaps per job.

The core of the implementation is similar to the simplified algorithm. We compute the warp submatrix of the right input matrix containing all elements used by any of the overlaps in jobs assigned to the threads in the given warp. We then iterate over this submatrix, computing all *job_size* overlaps in a single pass. This reduces parallelism, as simplified algorithm would have split these overlaps into different jobs and computed them in parallel, but allows for data reuse, which is advantageous when the GPU is already saturated.

As each overlap in a given job has different shift in the y axis, each overlap will contain different number of rows as shown in Figure 3.16, but thanks to the column-wise grouping of overlaps, corresponding overlaps in each job assigned to threads of a single warp will contain the same group of rows. This again allows us to reuse much of the simplified implementation code, which expects a warp to process overlaps with the same number of rows.

Due to the different shifts of the overlaps in the y axis, first 0 to $job\_size - 1$ and last 0 to $job\_size - 1$ rows of the warp submatrix may not be used for all *job_size* overlaps. Because of this, we must separate first few and last few iterations of the outer loop over rows of the warp submatrix into what we call *Init* and *Finish* iterations, illustrated in Figure 3.17. Overlaps in each job will always result in exactly *job_size* $-$ 1 steps split between Init and Finish steps depending on the exact overlap grouping.

Jobs containing overlaps with positive y axis shift, as shown in Figure 3.17a, require Init steps. Jobs containing overlaps with negative y axis shift, as show in Figure 3.17b, require Finish steps. Jobs spanning shift 0 on the y axis require both Init and Finish steps.

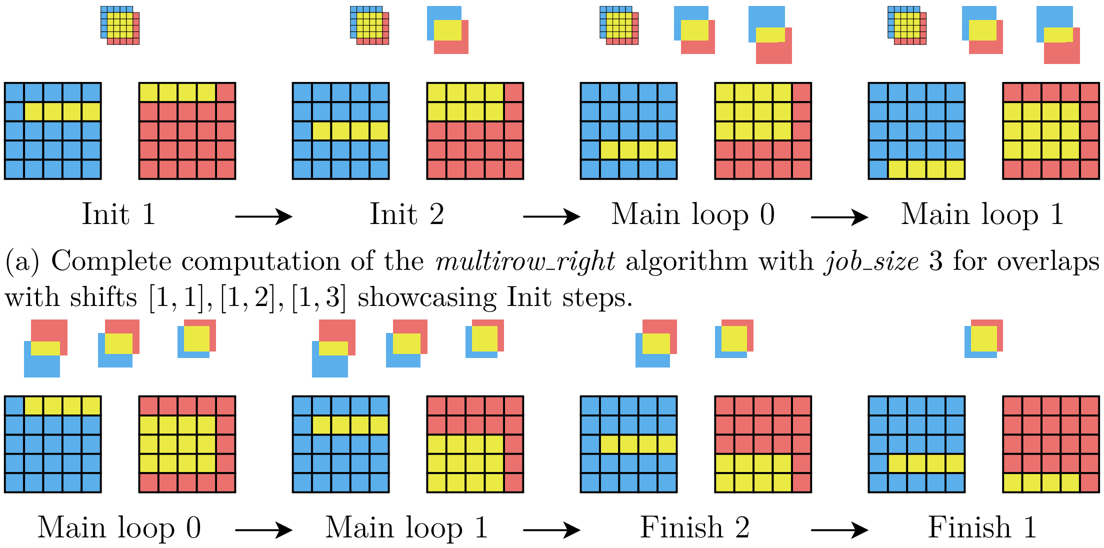
Code changes for this implementation start similarly to changes for *multimat_right* from Section 3.4.4. We again change the *sum* and *thread_right* variables into arrays, utilizing the Local array optimization described in Section 3.4.3. For this, we require that the number of overlaps grouped into a job, and consequently the number of rows loaded into the *thread_right* buffer, be known at compile time.

The three loops, the outer over rows of the warp submatrix, the middle over columns of the submatrix and the core loop over threads of the warp are separated. The middle and core loops are moved into a separate *device* function named *compute_row_group*, as they need to be reused for the Init, Finish and Main loop parts with different number of overlaps in each. The *compute_row_group* is one of the functions that needs to be generated for each allowed value of *job_size*, as that determines the maximum number of overlaps grouped into a job, which corresponds to the maximum number of rows processed in a single iteration, be it Init, Finish or Main loop iteration.

As described above, the outer loop is split into the three parts. There are always *job_size* $-$ 1 Init calls generated before the Main loop, where each Init call checks if the job assigned to the current thread contains overlaps with shifts requiring the given Init step. If required, Init step then calls *compute_row_group* with the required number of overlaps, as shown in Figure 3.17a with Init 1 calling *compute_row_group* for 1 overlap and Init 2 calling *compute_row_group* for 2 overlaps. Main loop then goes through the warp matrix row by row, calling *compute_row_group* each time on *job_size* overlaps. Finish steps then again always check if they are required before potentially calling *compute_row_group* with the required number of overlaps.

One of the disadvantages of this algorithm is the repeated reading of right input matrix rows by each worker. As warp shuffle is already utilized for data reuse in the given row, there is no simple mechanism to reuse data between rows as we traverse the warp submatrix from top to bottom. With 3 overlaps grouped into a job, each row of the right input matrix processed by the main loop is read 3 times by the worker, once for each of the overlaps assigned to the worker. Each time, it is used with different left row. This can be improved by also utilizing multiple rows from the left input matrix, computing multiple iterations of the current main loop at once. This implementation, named *multirow_both* as we read multiple rows in each iteration from both input matrices, is described in Section 3.5.3.

When profiling this optimization, as shown in Figure 3.18, we see similar improvements as with the previous *multimat_right* optimization. We have to keep in mind that the *multirow_right* algorithm improves the *one-to-one* type of computation, which cannot be improved by the previous optimization. These two optimizations can also be combined, which is described in Section 3.5.4. As before, the *LSU* pipeline remains a bottleneck, but the utilization of *FMA* pipeline is improved from 9% to 17%. With 4 overlaps per job, the ratio improves from



(a) Complete computation of the *multirow_right* algorithm with *job_size* 3 for overlaps with shifts $[1, 1], [1, 2], [1, 3]$ showcasing Init steps.

(b) Complete computation of the *multirow_right* algorithm with *job_size* 3 for overlaps with shifts $[1, -3], [1, -2], [1, -1]$ showcasing Finish steps

Figure 3.17: Illustration of Init and Finish steps of the *multirow_right* algorithm with overlaps processed in each step displayed above the step.

$3 : 1$ to $6 : 4$ as expected from the $2 + r : r$ theoretical ratio. As this optimization improves the *one-to-one* computation, it is more sensitive to occupancy reduction when workers process more than one overlap, mainly due to the smaller input size compared to the *one-to-many* computation.

3.5 Advanced warp shuffle optimizations

The optimizations of the warp shuffle algorithm described up to this point hint at further possibilities, such as:

- extending *multimat_right* optimization to also use multiple left matrices, creating *multimat_both* optimization;
- extending *multirow_right* to also use multiple rows from the left matrix, creating *multirow_both*,
- combining several optimizations.

This section first illustrates problems we encountered with the Local array optimization, introduced in Section 3.4.3, when implementing the *multimat_both* and *multirow_both* optimizations. We then describe our solution to this problem, demonstrating the effects of this solution on the performance of the two advanced optimizations. We then present the details of *multimat_both* and *multirow_both* implementation, which extend the basic optimizations described in Section 3.4. Lastly we combine all the basic and advanced optimizations together, creating the final implementation of the warp shuffle algorithm.

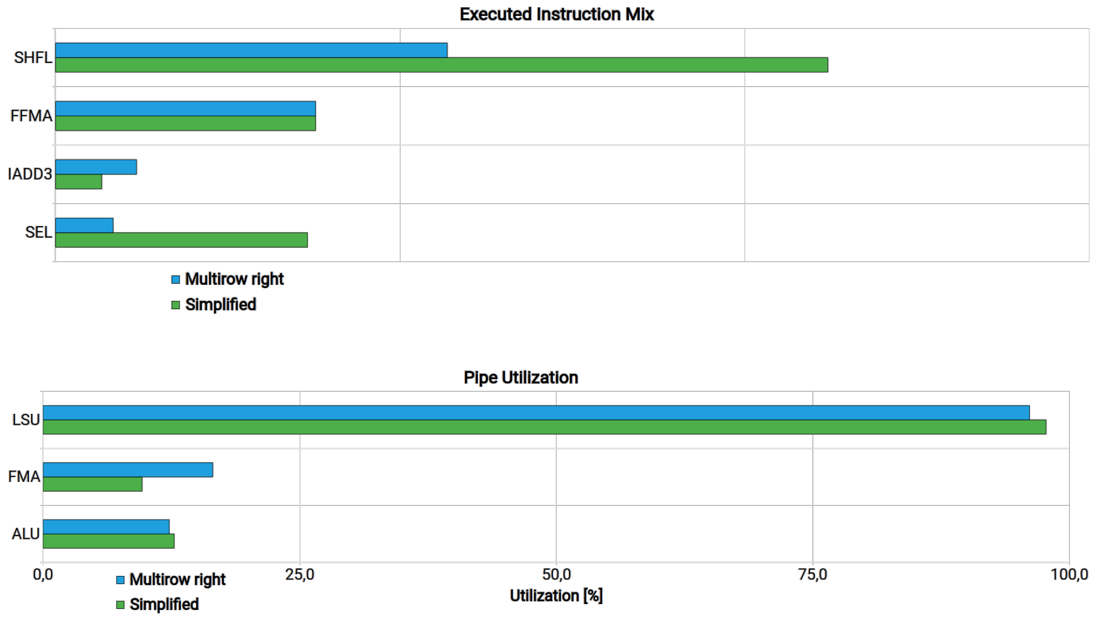


Figure 3.18: Comparison of *one-to-one* simplified algorithm with the *multirow_right* optimization.

3.5.1 Advanced optimizations and local arrays

When implementing the *multimat_both* and *multirow_both* optimizations, described in detail in the following sections 3.5.2 and 3.5.3 respectively, we encountered a problem with the *nvcc* compiler not optimizing the local arrays into registers.

Using profiling and examining the SASS, we isolated the problem to the *thread_left_bottom* and *thread_left_top* arrays. We further isolated it to the following part of the code, which in its original form shared by the simplified, *multimat_right* and *multirow_right* implementations looks like this:

```
thread_left_bottom = warp.shfl(
  warp.thread_rank() != 0 ? thread_left_bottom : thread_left_top,
  warp.thread_rank() + 1
);
thread_left_top = warp.shfl_down(thread_left_top, 1);
```

To process multiple values from left input matrices, which is the basis of both *multimat_both* and *multirow_both* optimizations, the code needs to be changed into the following:

```
#pragma unroll
for (size_t l = 0; l < NUM_LEFTS; ++l) {
  thread_left_bottom[l] = warp.shfl(
    warp.thread_rank() != 0 ? thread_left_bottom[l] :
       thread_left_top[l],
    warp.thread_rank() + 1
  );
  thread_left_top[l] = warp.shfl_down(thread_left_top[l], 1);
}
```

The *nvcc* compiler should be able to unroll this loop, and thanks to static indexing, the *thread_left_bottom* and *thread_left_top* local arrays should be optimized into registers. Unfortunately, as we can see in Listing 3.1, the compiler behaves as if dynamic indexing was used and pushes the arrays into local memory. Due to the closed source nature of the *nvcc* compiler, we can only speculate on the reasons why the loop unrolling does not result in static indexing. One possibility, based on the generated SASS instructions seen in Listing 3.1, is that the ternary operator is optimized into dynamic array indexing, which then prevents the local array optimization. As there is no visible branching in the unrolled loop, the base address of either the *thread_left_bottom* or the *thread_left_top* is loaded into register R63, which is then reused in all loads, resulting in dynamic indexing.

```
LDL R0, [R63+0x4]
SHFL.IDX PT, R8, R0, R57, 0x1f
SHFL.DOWN PT, R0, R32, 0x1, 0x1f
STL [R62], R8
STL [R61], R0
```

Listing 3.1: SASS instructions without local array optimization

We experimented with several solutions, with the following version compiling into static indexing:

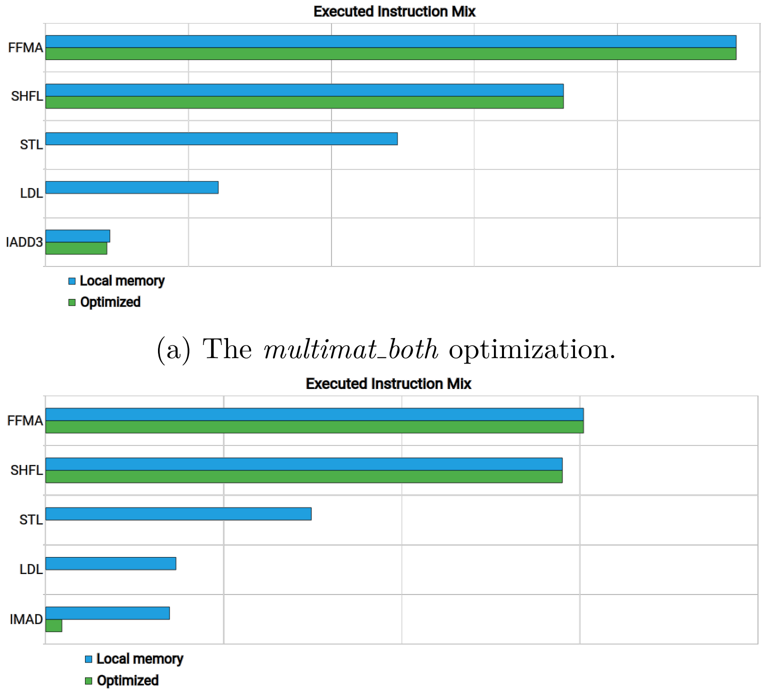
```
#pragma unroll
for (size_t l = 0; l < NUM_LEFTS; ++l) {
  T bottom_shift_val;
  if (warp.thread_rank() != 0) {
    bottom_shift_val = thread_left_bottom[l];
  } else {
    bottom_shift_val = thread_left_top[l];
  }

  thread_left_bottom[l] = warp.shfl(bottom_shift_val,
      warp.thread_rank() + 1);
  thread_left_top[l] = warp.shfl_down(thread_left_top[l], 1);
}
```

```
SEL R42, R52, R20, !P4
SHFL.IDX PT, R53, R48, R53, 0x1f
SHFL.DOWN PT, R52, R52, 0x1, 0x1f
```

Listing 3.2: SASS instructions with local array optimization

The only change is the expansion of the ternary operator into an equivalent if statement. As shown in Listing 3.2, the body of the updated loop results in a single *SEL* instruction which selects the top or the bottom part of the buffer. This version of the loop is used by the advanced warp shuffle optimizations described in the following sections.

The difference is also visible in Figure 3.19, which shows the additional local memory store (STL) and load (LDL) instructions in both the *multimat_both* and *multirow_both* without local array optimization. Apart from these additional instructions, the number of remaining instructions is generally the same, with some additional integer multiply-add instructions (IMAD) in Figure 3.19b due to local memory address computations.



(a) The *multimat_both* optimization.

(b) The *multirow_both* optimization.

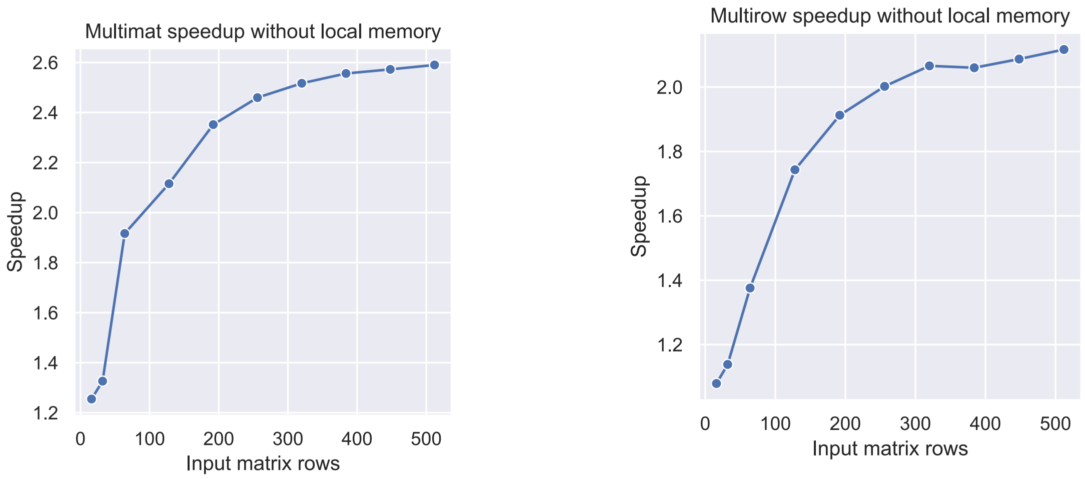
Figure 3.19: Comparison of the instruction mix between a version with arrays in local memory and a version with arrays in registers.

Based on this observation, the measured speedup in Figure 3.20 is caused solely by the application of the local array optimization. The speedup for smaller inputs is limited due to low occupancy, but is still present. As an example, Figure 3.20a shows that for input matrices of size 64 by 64, the solution without local memory access is already 2 times faster. Figure 3.20b shows that there is slightly smaller improvement in the *multirow_both* optimization compared to the *multimat_both*. This is most likely caused by higher general overhead due to the greater overall complexity of the optimization.

3.5.2 Multiple left matrices

A natural step following the use of the same overlap of multiple right matrices is to also group the same overlaps between multiple left matrices and multiple right matrices into a single job, as described in Section 3.2.1. This optimization is only usable for the *n-to-m* computation type, as the *n-to-mn* type correlates each left matrix with a different set of right matrices.

To implement this optimization, we start with the the code of the *multimat_right* optimization. We add additional template parameter *NUM_LEFTS* to the implementing function and change both *thread_left_bottom* and *thread_left_top* from simple variables into local arrays, as was done for the *thread_right* variable by the *multimat_right* optimization. The number of overlaps computed



(a) The *multimat_both* optimization.

(b) The *multirow_both* optimization.

Figure 3.20: Improvement of the two optimizations without local memory access.

by each thread is also changed to $NUM_LEFTS * NUM_RIGHTS$ from only NUM_RIGHTS. The changes are illustrated by the following snipped:

```
template<size_t NUM_RIGHTS, size_t NUM_LEFTS, typename T>
__device__ void warp_shuffle_multimat_both_impl(...) {
  ...
  T sum[NUM_RIGHTS * NUM_LEFTS];
  for (size_t s = 0; s < NUM_RIGHTS; ++s) {
    sum[s] = 0;
  }
  ...
  T thread_left_bottom[NUM_LEFTS];
  for (size_t l = 0; l < NUM_LEFTS; ++l) {
    thread_left_bottom[l] = load_with_bounds_check(...);
  }
  ...
  T thread_left_top[NUM_LEFTS];
  for (size_t l = 0; l < NUM_LEFTS; ++l) {
    thread_left_top[l] = load_with_bounds_check(...);
  }
  ...
  for (size_t r = 0; r < NUM_RIGHTS; ++r) {
    auto right_val = warp.shfl(thread_right[r], i);

    for (size_t l = 0; l < NUM_LEFTS; ++l) {
      sum[l * NUM_RIGHTS + r] += thread_left_bottom[l] * right_val;
    }
  }
  ...
}
```

Similarly to the *multimat_right* optimization, we group input matrices into matrix groups, this time separately grouping left matrices into *left_matrix_groups* and right matrices into *right_matrix_groups*, where *right_matrix_groups* are equivalent to the original matrix groups from the *multimat_right* optimization. The number of matrices in a left matrix group is configured independently of the number of matrices in a right matrix groups. As shown in the code snippet above, maximum size of left matrix group and right matrix group has to be known at compile time so that we can generate the *warp_shuffle_multimat_both_impl* for all combinations of these values from 0 to the configured maximum. At run-time, the implementation chooses the correct method. The last matrix groups from both left and right input matrices may be smaller as the number of left or right matrices may not be divisible by the value of the run-time argument. The implementation then chooses the function with the correct matrix group size.

As with the simplified algorithm, the block size is set to warp size in the x axis and configurable by run-time argument in the y axis. When extending the grid, we require expansion in four axes. Two to cover a single output matrix, one for left matrix groups and one for right matrix groups. These four axes need to be compressed into a three dimensional grid. To achieve this, we use the x axis to both cover columns of a single output matrix while also utilizing it to expand the grid across left matrix groups. The y axis is used to expand grid over the right matrix groups. The last axis, z, is used to cover the rows of the single output matrix . We choose to do this using the last grid axis to behave similar to the *multimat_right* and prepare the *multimat_both* optimization for combination with the work distribution optimization.

This optimization improves the ratio of warp shuffle to fused multiply-add instructions to $2 * l + r : l * r$, where l is the number of left matrices and r the number of right matrices utilized by each worker. This results in a noticeable improvement in the executed instruction mix shown in Figure 3.19a.

3.5.3 Multiple rows from both matrices

When improving the *one-to-one* computation using the *multirow_right* optimization described in Section 3.4.5, which processes multiple overlaps from consecutive rows of a single column of the output matrix, we hinted at a further improvement using multiple rows from the left matrix in the main loop. This not only improves the ratio of warp shuffle to fused multiply-add instructions, but also reduces the number of times every row from the right input matrix is read.

The main change is that the main loop now advances by multiple left input matrix rows instead of a single row. The exact number of left rows to advance by is configured using run-time argument. An additional stage between the multistep main loop and Finish is also needed to compute the remaining rows from the left input matrix when the total number of left input matrix rows is not divisible by the main loop step. This stage utilizes the original single step main loop. The Init and Finish parts are left unchanged.

The algorithm parameters are changed from just the number of right rows processed in each main loop iteration (which corresponds to the number of overlaps assigned to a single thread) to a pair of parameters, the number of left rows to process in each iteration of the main loop and the number of overlaps, also called shifts in the code, to be processed by each thread. The number of right rows to be loaded is now derived from the number of overlaps and the number of left rows. In each iteration of the main loop, the left row 0 is processed with right rows $[0, NUM\_SHIFTS - 1]$, left row 1 with right rows $[1, NUM\_SHIFTS]$ and left row l with right rows $[l, NUM\_SHIFTS - 1 + l]$. This gives us the number of right rows to load, which is $NUM\_SHIFTS + NUM_LEFT_ROWS - 1$ so that we can process all the left rows simultaneously.

The ratio of warp shuffle instructions (SHFL) to fused multiply-add instructions (FFMA) for this optimization is $l + (s - 1 + l) : l * s$ in the main loop, where l is the number of left rows processed by each iteration of the main loop and s is the number of shifts processed by each thread. The Init, single step main loop and Finish parts share the original ratio of the *multirow_right* implementation. This ratio is also apparent in Figure 3.19b.

3.5.4 Combining the optimizations

We have implemented the following optimizations of the Warp Shuffle algorithm:

- work distribution,
- *multimat_right*,
- *multirow_right*,
- *multimat_both*,
- *multirow_both.*

As described in Sections 3.5.2 and 3.5.3, the *multimat_both* and *multirow_both* optimizations are implemented as extensions to the *multimat_right* and *multirow_right* optimizations respectively.

All of the optimizations listed above can be combined with the following restrictions:

1. *multirow* optimizations cannot be combined with work distribution,
2. *multimat_both* can only be used to optimize the *n_to_m* computation.

With the *multirow* optimization, each job contains several different overlaps of the two input matrices, where each overlap has different number of rows. As our work distribution optimization is based on the number of rows of the overlap, the current implementation cannot be easily reused. This is not a problem with the *multimat* optimizations, in which each thread computes the same overlap in multiple output matrices.

The *n_to_m* computation type is the only type where multiple left matrices share the same right matrix, which makes it possible to reuse the data from the left matrices.

With these restrictions in mind, we have implemented the following versions of the warp shuffle algorithm:

- multimat_right,

- multimat_right_work_distribution,
- multimat_both_work_distribution,
- multirow_right,
- multirow_right_multimat_right,
- multirow_both,
- multirow_both_multimat_right,
- multirow_both_multimat_both.

Both *multimat* optimizations were already prepared for combination with work distribution, as hinted at in their respective sections. The core computation code does not change, the only difference is that we split the original warp submatrix the same way we did when implementing work distribution for simplified warp shuffle algorithm, described in Section 3.4.2. Each time we utilize the last used grid size axis to multiply the number of workers started. For *multimat_right*, it is the y axis, for *multimat_both*, it is the z axis.

Combination of *multimat* and *multirow* optimizations takes the implementation of *multimat* optimization, separates the middle and inner loop as is done to the simplified implementation in the implementation of *multirow* and splits the outer loop iterations into init, main loop, single step main loop, and finish parts.

These implementations are measured and compared in Section 4.2.2.

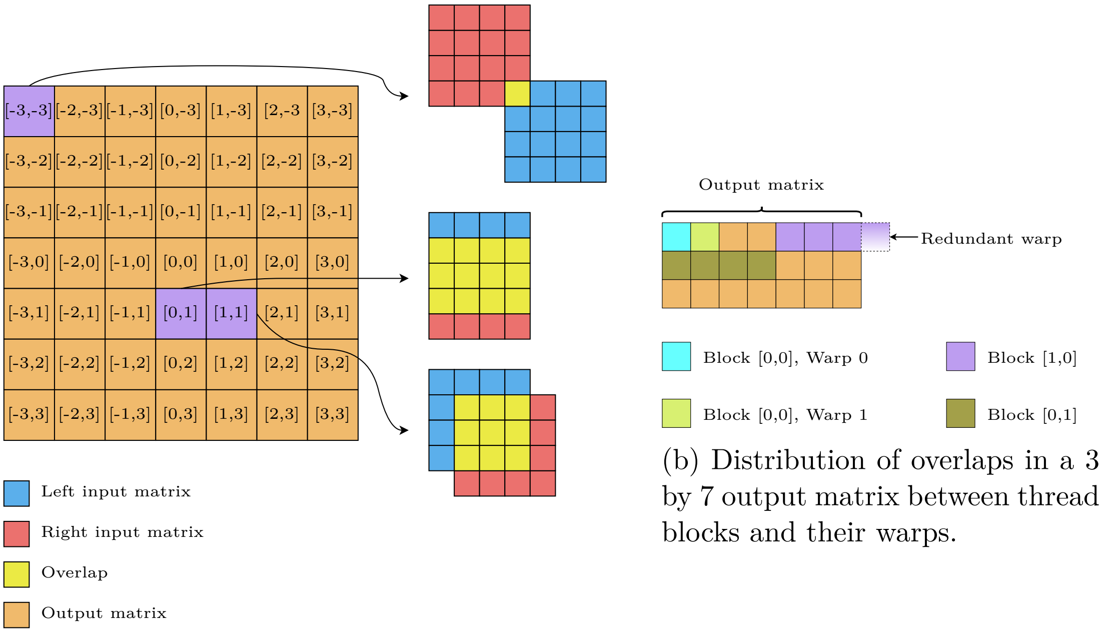
3.6 Occupancy improvement

For small inputs, processing a single overlap or a row group per thread may not split the computation into enough jobs to saturate the whole GPU, leading to low occupancy. As described in Section 2.3.1, low occupancy prevents the GPU from hiding the high latency of each instruction, resulting in poor performance. To increase the number of threads started for smaller inputs, we increase the size of each worker from a single thread to a whole warp or even a whole thread block. This increase in number of threads in combination with the small input data size leads to a need of balancing the overhead of each thread with the reduced workload. The overhead consists mainly of scheduling, repeated data reads, and computation of array indexes for each access.

In this section, we first introduce a simplified implementation of the *warp per shift* algorithm utilizing whole warps as workers with each job containing a single overlap, also called shift. We then introduce several improvements of this algorithm, first by using shared memory and then by further increasing the number of jobs using work distribution optimization, adapted from the implementation described in Section 3.4.2. Lastly we implement an algorithm utilizing whole thread blocks as workers, again with each job containing a single overlap.

3.6.1 Warp per shift

Implementation of the *Warp per shift* algorithm is very simple compared to the warp shuffle-based algorithm described in Section 3.4. The basic implementation utilizes whole warps as workers and assigns each worker a job consisting of a single overlap. Each overlap is uniquely identified by the shift of the two input matrices, and the shift is what is stored and propagated throughout the code. This is why we call this the *warp per shift* algorithm.



(a) Output matrix and the corresponding overlaps.

(b) Distribution of overlaps in a 3 by 7 output matrix between thread blocks and their warps.

Figure 3.21: Output matrix and its distribution between warps.

As in the Basic algorithm, introduced in Section 3.3, this implementation uses two dimensional grid of two dimensional thread blocks to cover output matrix. The difference is in how we use each dimension to map threads to overlaps, this time based on which warp and thread block they are part of. All threads of a warp are assigned the same overlap and they cooperate to compute all the tasks (yellow boxes) which belong to this overlap. The mapping from thread blocks and warps to overlaps of the output matrix is illustrated by Figure 3.21b. The x axis of thread block size is again set to warp size to simplify grouping of threads to warps, with the y axis again configurable using run-time argument. Warps of each thread block are assigned consecutive overlaps in a row of the output matrix using the y component of thread block size, which determines the number of warps in each thread block. This means that each thread block is assigned y overlaps in one of the output matrix rows. The x and y components of grid size are then used to cover the matrix using these thread blocks.

Each thread of a warp is assigned subset of the multiplication tasks (yellow boxes), as shown in Figure 3.22, which it computes and holds the sum in local variable. This distribution is done using the following code:

```
for (
  size_t i = warp.thread_rank();
  i < total_items;
  i += warp.size()
) {
  size_t overlap_row = i / overlap_size.x;
  size_t overlap_row_offset = i % overlap_size.x;

  size_t right_idx = ...;
  size_t left_idx = ...;

  sum += left[left_idx] * right[right_idx];
}
```

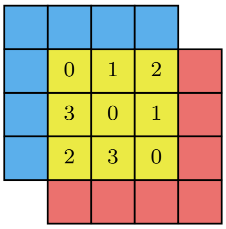


Figure 3.22: Thread computing given task with basic indexing (example warp size 4).

This distribution of tasks minimizes thread divergence but may lead to uncoalesced global memory access. Another possible disadvantage of this implementation is the division and the modulo instruction in each loop iteration, which is used to determine the position in the overlapping submatrix to be computed by the current thread.

The sums computed by each thread of the warp are then combined using the `cooperative_groups::reduce` function, which may even be hardware accelerated if running on the newest Ampere GPUs. The final result is then written by thread with warp rank 0 to the output matrix.

3.6.2 Simplified indexing

The Simplified indexing is an attempt to solve the problems highlighted in the previous section, namely the uncoalesced global memory access and the low throughput division instruction in the main loop. To fix these problems, we change the distribution of tasks between threads as shown in Figure 3.23. Compared to basic indexing described in the previous section, each row of the overlap is processed independently and fully before continuing to the next row. This assures coalesced access to the global memory, but leads to thread divergence if the row size of the overlap is not a multiple of warp size, as is the case in Figure 3.23.

Thread divergence is the major problem of this implementation. Based on our profiling, the simplified indexing leads to an average of only 15.31 of the 32 threads executing each instruction and not being predicated (masked by a predicate). With basic indexing on the same hardware, this average improves to 26.85, which is almost twice the work done per instruction. The main reason for this difference is shown in Figure 3.24. This figure shows the worst case scenario, where simplified indexing leads to the rows being processed sequentially, whereas

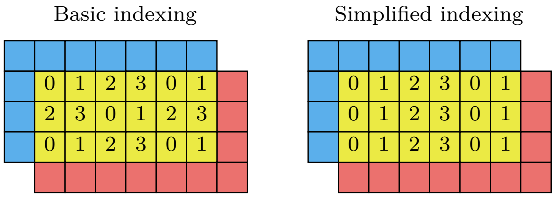


Figure 3.23: Comparison of task assignment with basic and simplified indexing (example warp size 4).

basic indexing executes this in a single iteration. Overlaps such as this make up a sizable part of every cross-correlation computation. Because of this, simplified indexing does not improve execution times.

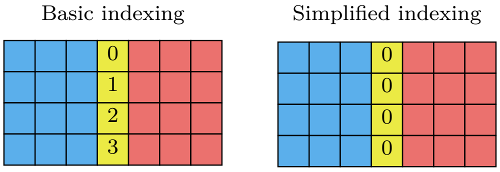
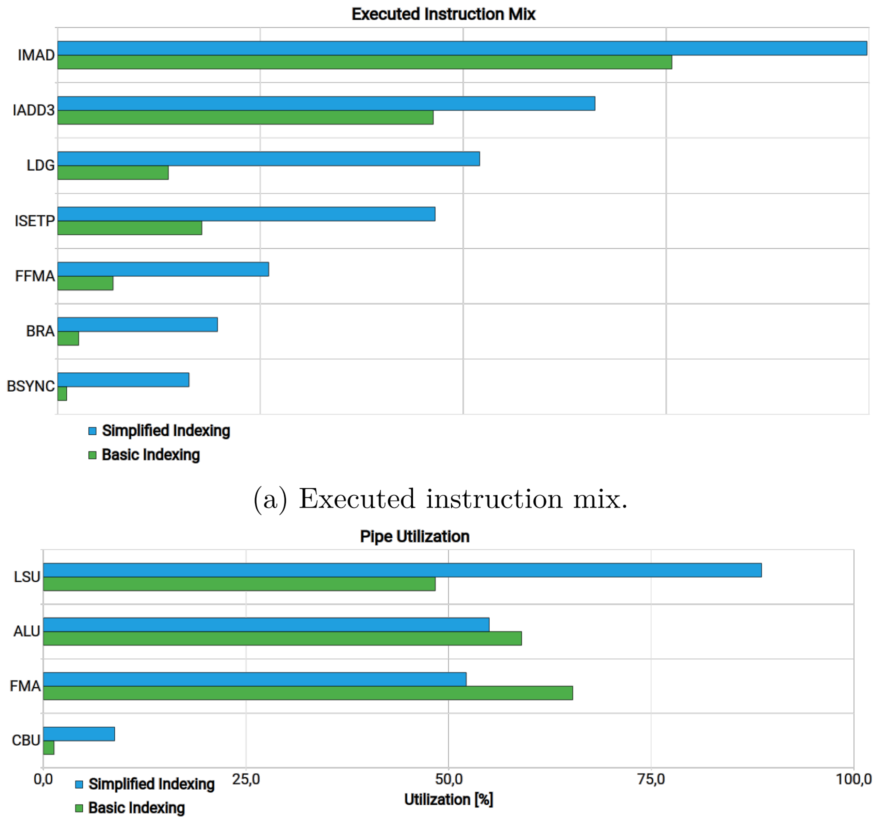


Figure 3.24: Task assignment with basic and simplified indexing to showcase thread divergence when using simplified indexing.

Even though simplified indexing should theoretically lead to better coalescing of global memory access, the opposite seems to be the case as illustrated by Figure 3.25a. This may be highly dependent on Compute Capability of the underlying hardware, but on CC 7.5 of RTX 2060, we observe a 47% increase in the number of global memory requests when using simplified indexing. This corresponds to high *LSU* pipeline usage of 88% visible in Figure 3.25b, which becomes a bottleneck. The profiling was done on input matrices of size 64 by 64 containing 32bit floating point numbers. In these matrices, each row is 256B in size. In many overlaps, the last item of each row will be less than 128B (global memory transaction size) from the first item in the next row. This leads to coalescing of the global memory read with basic indexing but results in 2 separate accesses with simplified indexing. This may explain the 47% difference in global memory requests.

Another visible difference in Figure 3.25a is the increase in number of instructions across the board, most visible with branching (BRA) and barrier synchronization (BSYNC) instructions. This is caused by the increase in number of iterations each warp executes to process the same data. Even with the increase in number of instructions, the *ALU* and *FMA* pipelines are less utilized than with basic indexing. This is caused by warps waiting for warp recombination on the barrier synchronization points and loads from global memory.

The effects of different pipeline utilization can also be seen in Figure 3.26. Warps of basic indexing algorithm are mostly stalled due to not being selected, i.e. there are multiple eligible warps and only one of them can be issued. This indicates that there may be too many warps for the size of the GPU. As these benchmarks were run on a RTX 2060 mobile, which is a rather small GPU, this was to be expected. The other main reason of warp stall is `Stall Math Pipe Throttle`, which is caused by the high utilization of *ALU* and *FMA* pipelines. These pipelines are responsible for computing the indices and the actual results of cross-correlation, which represents the useful work done by the GPU.



(a) Executed instruction mix.

(b) Pipeline utilization.

Figure 3.25: Comparison of basic and simplified indexing.

Simple indexing warps on the other hand are more often stalled on the `Stall Wait`, which represents warps waiting for a fixed latency execution dependency, i.e. a data dependency between two instructions or an instruction dependency on predicate computation. We can also see noticeable increase in stalls due to access to global memory (`Stall Long Scoreboard` and `Stall LG Throttle`) together with stalls due to branching. This is consistent with the properties of simple indexing described above.

3.6.3 Shared memory

This optimization takes inspiration from the warp shuffle algorithm and its *multirow* optimization. Instead of sharing input values by shuffling and broadcasting across threads of a warp, we load input values into shared memory and reuse them by all warps of the thread block.

Similarly to the warp shuffle algorithm, we utilize two alternating parts forming a ring buffer for data from the left input matrix and a single buffer for data from the right input matrix. Compared to the warp shuffle algorithm, these buffers are placed in the shared memory and are shared by all warps of each thread block. The windows of the input data loaded into buffers are not moved along the rows of the input matrices, but along a group of columns from top to bottom, processing the whole columns group before moving onto the next column group, as shown in Figure 3.27. With appropriately sized column groups, this enables us to maximize the throughput when loading data from global memory

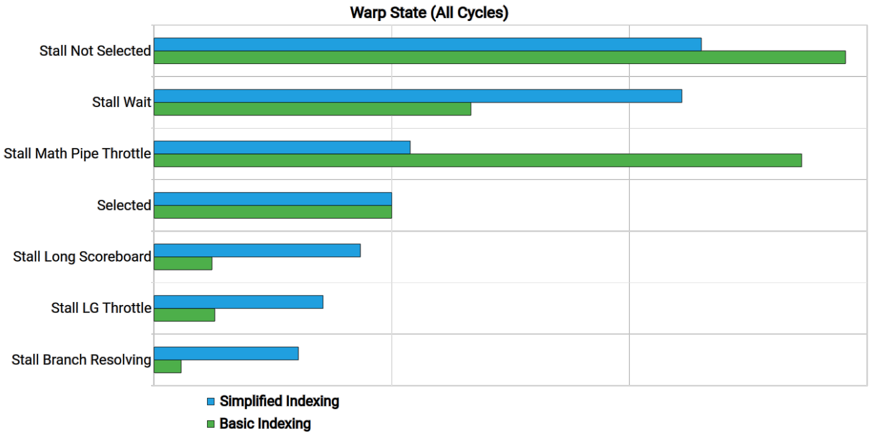


Figure 3.26: Comparison of warp stall reasons between the basic and simplified indexing.

using coalesced loads. The appropriate size here is multiple of 32, i.e. warp size. We call this the *shared_mem_row_size*, as it is also the size of each row of the shared memory buffers. The number of rows in each of the three shared memory buffers (two for data from the left matrix and one for data from the right matrix) is a run-time algorithm argument and stored in variable *shared_mem_num_rows*. Another reason we choose column groups instead of row groups is because of the way we assign overlaps to warps of a thread block. To prevent bank conflicts when accessing shared memory, we assign consecutive overlaps from a single column of the output matrix to the warps of a single thread block. This is explained in more detail further in this section.

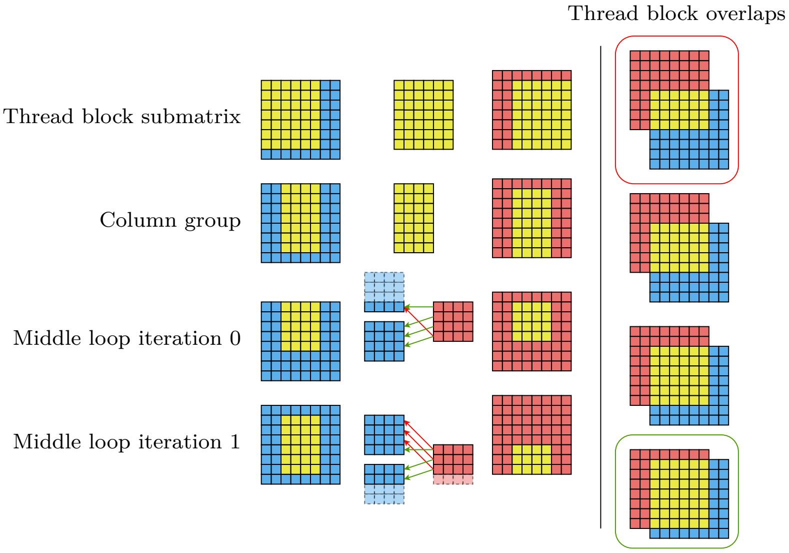


Figure 3.27: Computation of a single column group showcasing different warp offsets and left bottom buffer preload offset.

As warp shuffle algorithm had warp submatrix, i.e. submatrix of each of the input matrices containing elements required by any of the threads of the given warp, this algorithm computes thread block submatrix containing elements of the input matrices required by overlaps assigned to any of the warps of the given thread block . This thread block submatrix is what we partition into column groups and what we iterate over and load into the shared memory buffers.

The implementation is again made up of three nested for loops. The outer loop iterates over column groups, the middle loop iterates over rows of the column group in *shared_mem_num_rows* sized steps. Threads of the whole thread block go through these two loops synchronously to allow cooperation when loading data to the shared memory buffers.

```
T thread_sum = 0;
for (
  size_t column_group_start_x = block_matrix_start.x;
  column_group_start_x < block_matrix_end.x;
  column_group_start_x += shared_mem_row_size
) {
  // Bottom part of the left shared memory buffer
  left_bottom_s.load_submatrix(...);

  for (
    size_t right_buffer_start_row = block_matrix_start.y;
    right_buffer_start_row < block_matrix_end.y;
    right_buffer_start_row += shared_mem_num_rows
  ) {
    left_top_s.load_submatrix(...);
    right_s.load_submatrix(...);

    __syncthreads();

    compute_from_shared_mem_buffers(left_bottom_s, right_s, ...);

    compute_from_shared_mem_buffers(left_top_s, right_s, ...);

    swap(left_bottom_s, left_top_s);

    __syncthreads();
  }
}
```

In each iteration, we compute the parts of the right buffer which are part of the overlap assigned to the current warp and have the corresponding overlapping row from the left matrix in the given part of the left buffer, first the bottom part, then the top part.

```
template<typename T>
__device__ void compute_from_shared_mem_buffers(
  const shared_mem_buffer<T>& left_buffer,
  const shared_mem_buffer<T>& right_buffer,
  T &thread_sum,
  ...
) {
  // Offset in shared memory buffer
  int warp_right_start_offset = ...;
  int warp_right_end_offset = ...;

  // Offset between buffers
  int buffer_offset = ...;
  for (
    int right_idx = warp_right_start_offset + warp.thread_rank();
    right_idx < warp_right_end_offset;
    right_idx += warp.size()
  ) {
    l = left_buffer[right_idx + buffer_offset];
    r = right_buffer[right_idx];
    thread_sum += l * r;
  }
}
```

This innermost loop iterates over the rows from the two shared memory buffers which are overlapped in the overlap assigned to the current warp. Shared memory accesses in this loop are without bank conflicts thanks to way we assign overlaps between warps of a thread block. If we were to assign overlaps from a single row of the output matrix to warps of a thread block, as is done by the basic implementation of *warp per shift* algorithm, we would encounter a problem illustrated in Figure 3.28. For the purposes of this example, we work with warps of 4 threads, shared memory with 4 banks and assume each input matrix fits into one shared memory buffer. The figure shows the left and right input matrix and how their elements map into banks of shared memory when loaded by the thread block. For simplicity, we have chosen a thread block whose thread block submatrix contains whole input matrices as one of the overlaps computed by the thread block is the overlap with shift $[0, 0]$. When warps of a given thread block are assigned overlaps along a row of the output matrix, the overlaps differ in the number of columns. This leads to an access pattern with different stride in each warp. The strides for warps 1 and 2 result in 2-way bank conflicts, the stride for warp 0 results in 4-way bank conflict. With 32 threads per warp, this type of access would cause up to 32-way bank conflict, which would severely limit the shared memory throughput.

Due to this, we choose to assign overlaps from a single column to warps of a thread block. This assignment leads to access illustrated in Figure 3.29. This figure depicts two iterations of the innermost loop, with the same simplifications as the previous figure. As the overlaps differ in the number of rows, not columns, the access to shared memory has different starting and ending offset, but between these the access is perfectly linear and coalesced, each thread accessing different bank. Left buffer is accessed independently of the right buffer, so sharing banks between these buffers does not lead to bank conflicts.

When loading the bottom part of the left shared memory buffer directly in the outer loop, we need to limit which rows are loaded to the buffer and offset the loaded rows, as shown in Figure 3.27. If we were to just load the full buffer as shown in Figure 3.30, we would encounter a situation where rows loaded in the second iteration of the middle loop into the right buffer overlap rows which were loaded during the first load of bottom left buffer, which at that point is already

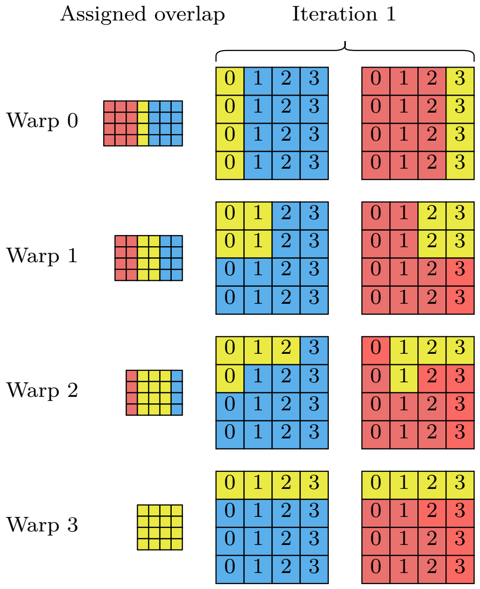


Figure 3.28: Input matrices with numbers designating their mapping to shared memory banks and how they are accessed by 4 different warps of a single thread block in iteration 1 when assigned along a row of the output matrix.

overwritten.

Even with the throughput of shared memory, the load from shared memory *LDS* instructions become a bottleneck, as shown in Figure 3.31. The memory input/output stall is caused by the memory input/output queue being full. This queue handles special math instructions, dynamic branches and most importantly for us the shared memory access instructions.

3.6.4 Loading data to shared memory

When loading data to shared memory, we have a choice between two different access patterns, shown in Figure 3.32:

- strided warps,
- continuous warps.

With strided warps, each warp loads chunks of *warp size* items with stride of *thread block size* items.

With continuous warps, the whole block of data is divided into equal parts, one for each warp. Each warp then loads the single continuous part of the data. The advantage compared to strided loads is that this access pattern should better utilize caches, as each warp does sequential access. The disadvantage is mainly in the fact that each warp will load some part of the data, even if the data is small and could be loaded by a subset of warps with strided loading.

3.6.5 Shared memory with multiple right matrices

Similarly to the warp shuffle algorithm, we can increase the ratio of arithmetic instructions to shared memory loads by computing the same shift between a single left matrix and multiple right matrices. This optimization is limited by

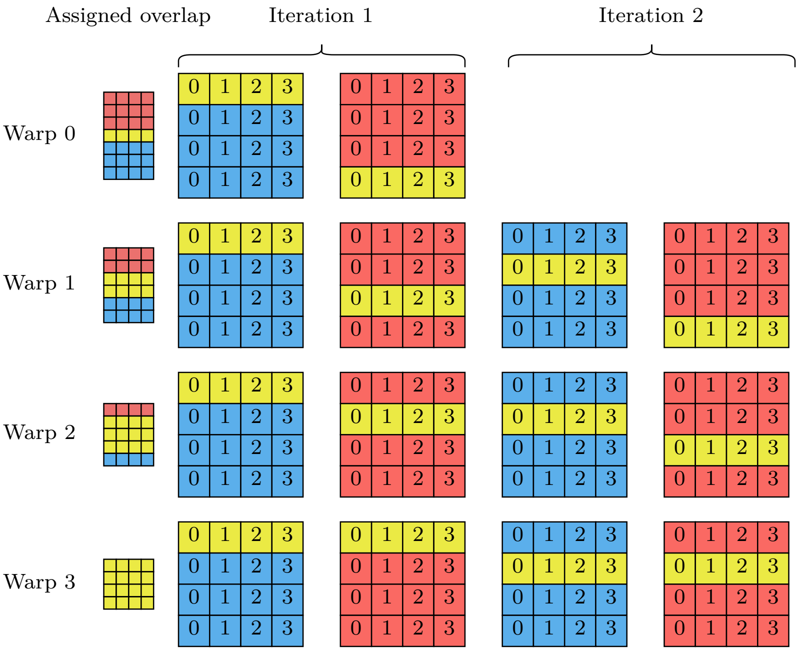


Figure 3.29: Input matrices with numbers designating their mapping to shared memory banks and how they are accessed by 4 different warps of a single thread block in the first two iterations when assigned along a column of the output matrix.

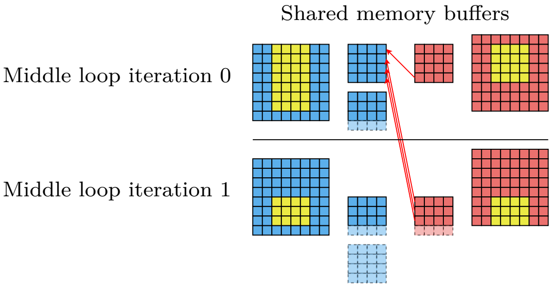


Figure 3.30: Cross-iteration dependency when the load of the bottom part of the left buffer is not limited and offset.

the size of shared memory, as we need to fit multiple right shared memory buffers described in the previous section into shared memory at once. The code changes are very similar to the warp shuffle changes. We change the *thread_sum* and *right_s* variables into arrays and wrap every access into an unrollable for loop. As we are again computing overlaps with the same shift from different output matrices, any loop bounds computed hold for all the overlaps.

The effects of this optimization are shown in Figure 3.33. The number of shared memory load instructions (LDS) and memory access index computations using the integer multiply-add (IMAD) is significantly reduced. Even with this reduction, the utilization of the *LSU* pipeline is still a bottleneck, most likely due to the reduced throughput of shared memory in the Compute Capability 7.5 we use for profiling. The higher utilization of *FMA* pipeline hints at better performance, which will be shown in Section **??**.

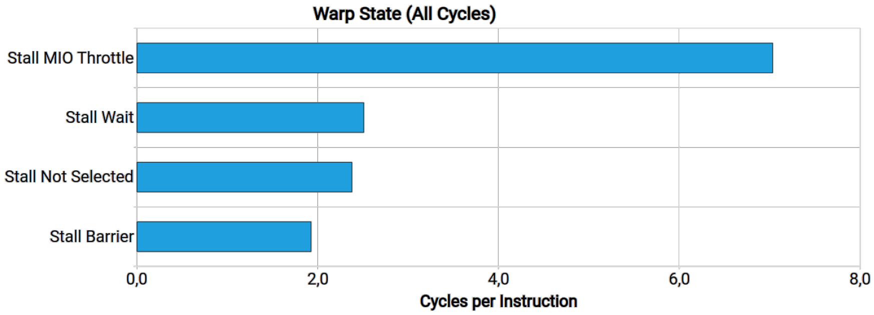


Figure 3.31: Memory input/output (MIO) stall caused by excessive shared memory access.

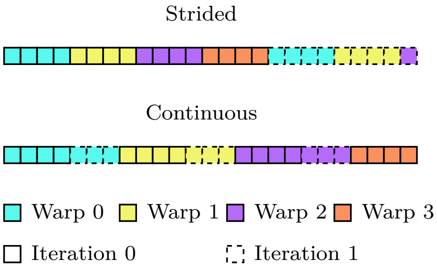


Figure 3.32: Difference between strided and continuous loading patterns.

3.6.6 Shared memory with single column group per block

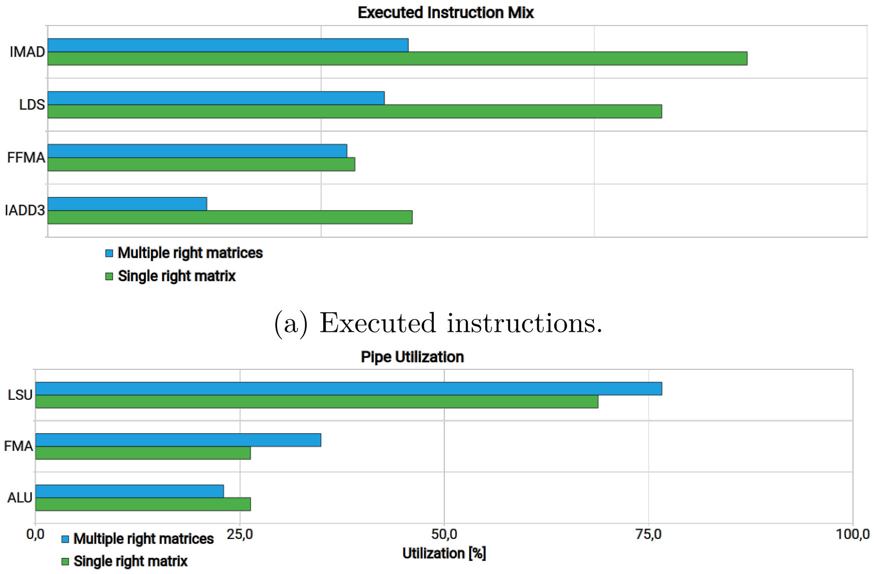
As described in Section 3.6.3, the thread block matrix is split into column groups. Each of these column groups can be processed independently, with the partial result added to the total using *atomicAdd* instruction as is done for work distribution in warp shuffle algorithm, described in Section 3.4.2. This optimization borrows from the rectangle work distribution, first computing the maximum number of column groups per shifts m and then starting m workers for each shift.

We utilize the z dimension of the grid size to multiply the number of workers by m. Based on thread block z index, each warp computes its assigned column group of the given thread block matrix. The loop bounds are computed to only include the assigned column group, after which the code from the original implementation with multiple column groups can be reused without any changes.

3.6.7 Work distribution

As described in Section 3.4.2 with the warp shuffle algorithm, there are massive differences between work done by different workers in the basic algorithm. The implementation shares much of the code with the warp shuffle algorithm, only difference being the size of workers. We can choose from the `triangle` or `rectangle` distributions and set the maximum number of rows processed by a worker.

In our benchmarks using RTX 2060 GPU, this optimization was only beneficial for inputs smaller than 32 by 32, as shown in Figure 3.34. For larger inputs, the GPU is already saturated and with each warp processing a single shift, the workload per thread is already small enough that the work distribution does not bring any noticeable improvement. Additionally, without the use of shared



(a) Executed instructions.

(b) Pipeline utilization.

Figure 3.33: Profiling of the effects of multiple right matrices on shared memory optimization.

memory described in Section 3.6.3, the added load onto global memory makes this implementation slower for larger inputs.

With larger GPUs, the boundary where the benefits of increased occupancy are diminished by added strain onto global memory is moved onto larger inputs, as is shown in Figure **??**.

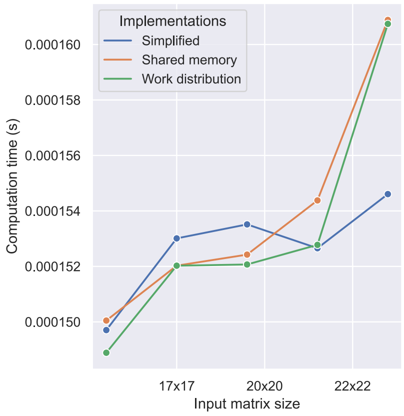


Figure 3.34: Comparison of computation time between simplified warp per shift algorithm, warp per shift with shared memory and warp per shift with work distribution.

3.6.8 Block per shift

Another possible way to further increase occupancy is to switch from warps as workers to whole thread blocks as workers. This can massively increase the number of threads started for given size of input, saturating the GPU even for smaller inputs. The implementation is rather simple. The thread block index directly maps to the position in the output matrix and consequently to the overlap computed by given thread block. We then compute the bounds of the overlapping submatrix and iterate over the overlapping elements using block stride loop. We then utilize `reduce` function provided by Cooperative Groups API to sum results in each warp, which are then stored into shared memory and reduced again by warp 0. The final result is then stored into the output matrix.

Based on our testing with RTX 2060, the block per shift algorithm does not significantly improve the run time over the simple warp per shift algorithm even for small input matrices, as shown in Figure 3.35. This is most likely caused by the simple warp per shift algorithm already saturating the GPU.

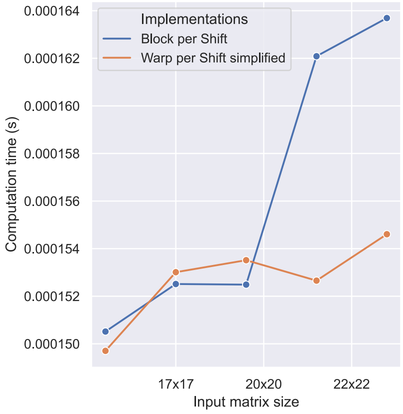


Figure 3.35: Comparison of the simple warp per shift algorithm and block per shift algorithm.

For larger GPUs, this implementation may be beneficial, but may also suffer from the low workload per thread, unbalanced workload between workers and no data reuse.

4. Results

In this chapter, we first describe our setup for measurement and validation of the implementations described in the previous chapter. We then compare the definition-based cross-correlation implementations against each other before comparing them with an FFT-based CUDA implementation and existing real world implementations in Python SciPy library and in Matlab.

4.1 Experiments

As the main aim of th thesis is to compare implementations of an algorithm, the code is heavily instrumented to enable measurements and comparisons of different parts of the implementation. This instrumentation is designed to limit its impact and to allow measurements which minimize background noise and imprecision of the time measurement tools provided by the CUDA C++ language.

As part of this thesis, we have also developed a benchmarking tool which allows the use of declarative description of the set of benchmarks to be executed. This tool is used both for measurements and for validation of the implementation.

To simplify the implementation, we have placed several restrictions on the input matrices and the computation:

- both input matrices are of the same size,
- whole output matrix is computed.

Both restrictions are used to simplify the implementation and reduce the number of variables when measuring.

4.1.1 Code instrumentation

All implementations, including definition-based, FFT-based and CPU-based, are split into the following steps:

- *Load* loads the input matrices into host memory,
- *Prepare* allocates device memory and precomputes things derived from the input data size,
- *Transfer* moves data from host to device memory,
- *Run* executes the computation,
- *Finalize* moves data from device to host memory,
- *Free* releases resources allocated in the *Prepare* step,
- *Store* stores results from host memory.

Each of these steps can be individually measured and compared. The main focus of this thesis is the *Run* step, but to properly compare the behavior of the FFT-based implementations with definition-based implementations, we will have to compare other steps as well.

We also provide simple CPU based single threaded definition-based implementation, for which most of these steps are empty. This implementation is provided for basic validation of results and the benchmarking infrastructure.

We can compare the algorithms on three separate levels:

- *Compute* measuring the duration of the Prepare, Transfer, Run, Finalize and Free together;
- *CommonSteps* measuring every implementation step separately;
- *Algorithm* measuring algorithm-specific parts such as individual kernels or library calls.

For parts which can be executed repeatedly such as computations and data transfers, we utilize measurement with an adaptive iteration count. The number of iterations is automatically increased until the measured duration is longer than a configured minimum, most often a second. This type of measurement should mitigate jitter, for example due to scheduling, and get around problems with minimum clock resolution for quick running steps. It is used for the *Compute* measurement, for measuring the *Run* step and for measuring kernel and function call duration in each algorithm.

4.1.2 Benchmarking tool

To compare the implementations, we have developed a benchmarking tool which executes bechmarks based on a declarative description which includes input sizes, sets of implementations to measure, sets of arguments for each implementation, and other parameters. The suite then generates the input data, optionally utilizing known good implementation of cross-correlation such as Python SciPy or Matlab to generate results for validation. It then runs the specified benchmarks, providing measurement and validation results.

4.1.3 Experiment setup

Experiments were run on two systems:

- Intel Xeon Silver 4110 with 256GB of RAM and NVIDIA Tesla V100 PCIe 16 GB,
- AMD Ryzen 5 4600H with 16GB of RAM and NVIDIA GeForce RTX 2060.

Most showcased results are collected using the adaptive iteration count. The resulting value shown is the total measured time divided by the number of iterations. To further remove noise, all measurements are repeated 20 times and mean of these measurements is taken as the final result, as there are no significant outliers which would skew the mean.

4.1.4 Result validation

When validating results of a computation, we compare them to a valid results computed using SciPy, Matlab or our simple CPU definition-based implementation. The output matrix is compared element by element with the valid result, computing a matrix of differences between these elements using formula 4.1.

$$\text{Relative\_difference}(a, b) = \frac{|a - b|}{\max(|a|, |b|)} \tag{4.1}$$

We then take the maximum element in the matrix of differences as the error of the result.

4.2 Measurements

In this section, we first compare the basic definition-based implementation with simple warp shuffle implementation. We then compare optimizations of the warp shuffle implementation against each other. Next we measure the occupancy improving optimizations introduced in Section 3.6. Lastly we compare the best optimizations of the definition-based implementation with the FFT-based implementation and the existing SciPy and Matlab implementations.

Each of the *one-to-one*, *one-to-many*, *n-to-mn*, and *n-to-m* input types is compared separately, as it allows for different optimizations and may benefit certain implementations better than others. We show only the measurement with the best combination of arguments for each implementation. We choose the arguments which are fastest when averaged across all input sizes.

4.2.1 Basic implementation

We first compare the basic algorithm with simple warp shuffle implementation...

4.2.2 Warp shuffle optimizations

Implementations utilizing warp shuffle instructions are usable across a range of input sizes. We measure their behavior on the following input matrix sizes: 16x16, 32x32, 64x64, 128x128, 256x256, 512x512. These sizes were chosen as larger input sizes are faster computed using the FFT-based implementation, and as such the speed of the optimizations of the definition-based implementation is irrelevant for these sizes. Based on the input type, we also measure with different number of left and right input matrices to gauge the behavior when changing these input properties.

one-to-one

For this input type, we have the following four implementations with these arguments:

Implementation	Argument	Value
Simple	Warps per thread block	8
Simple with work distribution	Warps per thread block	8
	Rows per thread	1
	Distribution type	triangle
Multirow right	Warps per thread block	8
	Right rows per thread	4
Multirow both	Warps per thread block	8
	Shifts per thread	4
	Left rows per iteration	4

The results displayed in Figure 4.1 show us the relative speed between the Simple warp shuffle implementation and each optimization. For smaller inputs, the *multirow* optimizations are slower, up to 20% slower for the *multirow_both* and up to 80% slower for the *multirow_right*. This is caused by the reduction in number of threads and correspondingly reduced occupancy of the GPU. This is combined with each thread reading each row of the right input matrix multiple times, adding global memory access latency which the GPU is unable to hide due to the low occupancy. One of the reasons we added *multirow_both* was to reduce the number of times we need to read each row in the right matrix, which is reflected here in better performance for all input sizes when compared with the *multirow_right* optimization. For larger input sizes the better ratio of warp shuffle to fused multiply-add instructions of these two optimizations allows them to overtake the Simple implementation. This is above 128x128 for *multirow_both* and above 256x256 for *multirow_right*.

The behavior of work distribution optimization tells us that the GPU is not fully saturated by the Simple implementation up until the 512x512 input matrix size. This is an expected problem with the *one-to-one* input type. For inputs smaller than 512x512, the large number of additional threads allows us to fully utilize the GPU, making the work distribution optimization up to 80% faster. As we increase the input size, the benefit of additional threads diminishes, completely disappearing at 512x512 input size. where the overhead introduced by the additional threads negates the increased GPU saturation.

one-to-many

This input type has six implementations with ran with the following arguments:

Implementation	Argument	Value
simple		
simple with work distribution		
multimat_right		
multimat_right with work distribution		
multirow_right multimat_right		
multirow_both multimat_right		

From the results in Figure **??**, we see that ....

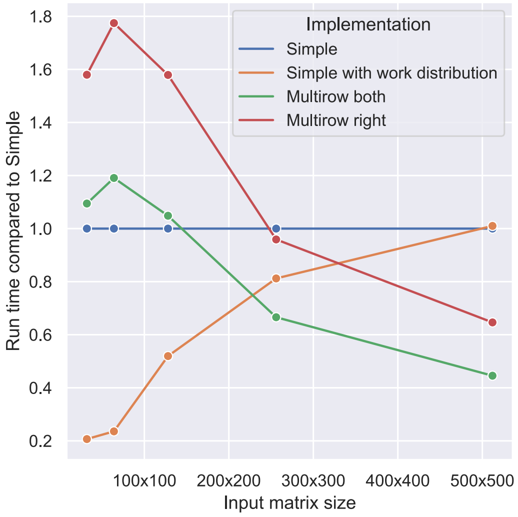


Figure 4.1: Relative speed of warp shuffle optimizations.

n-to-mn

This input type shares implementations with the *one-to-many* type, as it does not provide any additional possibilities for data reuse, because each left input matrix is cross-correlated with a different set of m right input matrices. This means that each implementation of *n-to-mn* type just executes the *one-to-many* implementation kernel n times in parallel, once for each left input matrix. Due to this, the main factor here is how many of the *one-to-many* kernels can we run in parallel, or more precisely how many thread blocks from these kernels can fit on a single SM.

The arguments differ slightly due to the higher GPU utilization:

Implementation	Argument	Value
simple		
simple with work distribution		
multimat_right		
multimat_right with work distribution		
multirow_right multimat_right		
multirow_both multimat_right		

n-to-m

The final input type has a group of implementations specifically optimized for this type, the *multimat_both* optimization and its combinations with other optimizations. We also reuse some *one-to-many* implementations, running them n times in parallel, once for each left input matrix, this time all with the same set of right input matrices.

Implementation	Argument	Value
simple		
simple with work distribution		
multimat_right		
multimat_right with work distribution		
multirow_right multimat_right		
multirow_both multimat_right		

4.2.3 Occupancy improving optimizations

Occupancy is mainly a concern when working with the *one-to-one* input type and small input matrices. Because of this, we implement these optimizations mostly for the *one-to-one* input type and will compare them only on this input type. There are some exceptions which also work with more input matrices which will be compared on other input types with other implementations in the following section 4.2.4.

Table **??** lists the occupancy improving optimizations with the arguments used for their measurement.

Implementation	Argument	Value
Warp per shift	Shifts per thread block	16
Warp per shift with work distribution	Shifts per thread block	8
	Rows per warp	10
	Distribution type	triangle
Warp per shift with shared memory	Shifts per thread block	16
	Shared memory row size	128
	Load with stride	True
	Column group per block	True
Block per shift	Block size	256

From the results in Figure 4.2, we see that the Warp per shift optimization is enough to saturate the GPU, so the Block per shift and Work distribution optimizations do not improve the run time. The increasing speed of work distribution is most likely caused by the increasing size of each job and correspondingly decreasing proportion of the overhead caused by distributing the jobs and collecting the results.

The shared memory optimization is not beneficial for inputs smaller than 64x64, as it adds synchronization overhead between the threads of a thread block when loading the data into shared memory. For larger input data sizes, the reduced number of global memory accesses gives this optimization an advantage over pure Warp per shift implementation.

4.2.4 Best definition-based implementations

Now that we have compared the optimizations of each implementation between each other, we now choose the best of these optimizations and compare the speedup we have gained against the basic definition-based implementation described in Section 3.3.

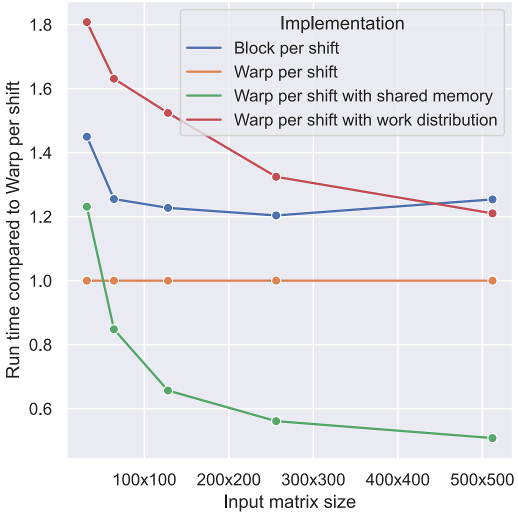


Figure 4.2: Relative speed of warp per shift optimizations.

4.2.5 FFT-based implementation

The FFT-based implementation used in this thesis is adapted from the one used by Bali [1]. It uses the cuFFT library for the Fast Fourier Transform and custom kernel for Hadamard multiplication.

Before comparing this FFT-based implementation with the definition-based implementations, we first show a few properties of this implementation caused by the use the Fast Fourier transform in general and of the cuFFT library in particular. First is the dependency between the precise size of the input and computation time illustrated in Figure **??**. As described in the cuFFT library documentation, cuFFT provides ”Algorithms highly optimized for input sizes that can be written in the form $2^a * 3^b * 5^c * 7^d$. In general the smaller the prime factor, the better the performance, i.e., powers of two are fastest.” [6]. From our measurements, we see up to 30% slowdown between power of two input size and input size one smaller.

Another feature is illustrated in Figure 4.3, where we show computation time for inputs smaller than 128x128. From these measurements we see that the cuFFT library computes inputs up to 90x90 faster in double precision than in single precision, even though the double precision version works with twice the amount of data and uses operations with lower throughput. The cause is shown in Figure 4.4, which shows the algorithm steps measured separately. The *Prepare* step, which allocates device memory and in this implementation runs the *cufftPlan** functions, is slower for single than for double. The Free step then deallocates these plans. Due to the closed source of the cuFFT library we can only speculate on the cause, but it is most likely due to some additional precomputation done for the single version. We also see the dependency on the precise size of the input. Measurement of each step separately adds some overhead, which is why the sum of individual steps is larger than the total computation time in Figure 4.3.

When compared to the definition-based implementations ...

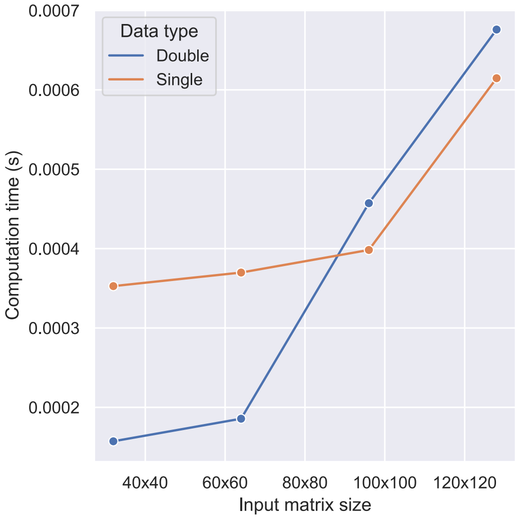


Figure 4.3: Comparison of FFT-based computation in single and double precision for small inputs.

4.2.6 Real world implementations

We now compare the best of our definition-based implementations with cross-correlation implementation from existing libraries and toolkits. We have chosen the Python SciPy library as a generally used CPU implementation of 2D cross-correlation and Matlab with Parallel Computing Toolbox for GPU accelerated 2D cross-correlation. Due to licensing limitations, Matlab will only be compared on the RTX 2060 system.

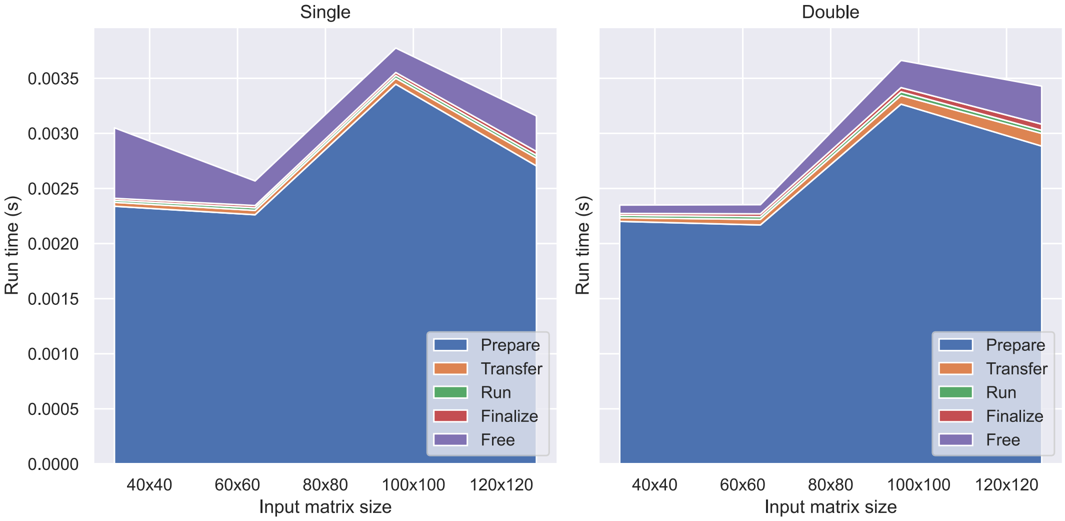


Figure 4.4: Individual steps of the FFT-based algorithm for small inputs.

Conclusion

4.3 Future work

Bibliography

[1] Michal Bali. Employing gpu to process datafrom electron microscope. Master's thesis, Charles University, 2020.

[2] M. A. Clark, P. C. La Plante, and L. J. Greenhill. Accelerating radio astronomy cross-correlation with graphics processing units. July 2011.

[3] Konstantin Kapinchev, Adrian Bradu, Frederick Barnes, and Adrian Podoleanu. Gpu implementation of cross-correlation for image generation in real time. pages 1–6, Cairns, QLD, Australia, 2015. IEEE. ISBN 978-1-4673-8117-8. doi: 10.1109/ICSPCS.2015.7391783.

[4] NVIDIA. Nvidia tesla v100 gpu architecture: The world's most advanced data center gpu. Technical report, NVIDIA, 2017.

[5] Nvidia. CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2022.

[6] NVIDIA. cuFFT. `https://docs.nvidia.com/cuda/cufft/index.html`, 2022. URL `http://web.archive.org/web/20220624091956/https://docs.nvidia.com/cuda/cufft/index.html`.

[7] NVIDIA. Kernel Profiling Guide. `https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`, 2022. URL `https://web.archive.org/web/20220514025303/https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`.

[8] Sergi Ventosa, Martin Schimmel, and Eleonore Stutzmann. Towards the processing of large data volumes with phase cross-correlation. *Seismological Research Letters*, May 2019. doi: 10.1785/0220190022.

[9] Chen Wang. *Kernel learning for visual perception.* PhD thesis, Technological University, Singapore, 2019.

[10] Wikimedia Commons contributors. Visual comparison of convolution, cross-correlationand autocorrelation. `https://commons.wikimedia.org/w/index.php?title=File:Comparison_convolution_correlation.svg&oldid=607616339`, November 2021. URL `https://commons.wikimedia.org/w/index.php?title=File:Comparison_convolution_correlation.svg&oldid=607616339`.

[11] Wikipedia contributors. Cross-correlation. `https://en.wikipedia.org/w/index.php?title=Cross-correlation&oldid=1065983922`, March 2022. URL `https://en.wikipedia.org/w/index.php?title=Cross-correlation&oldid=1065983922`.

[12] Lingqi Zhang, Tianyi Wang, Zhenyu Jiang, Qian Kemao, Yiping Liu, Zejia Liu, Liqun Tang, and Shoubin Dong. High accuracy digital image correlation powered by gpu-based parallel computing. *Optics and Lasers in Engineering*, 69:7–12, 2015. ISSN 0143-8166. doi: https://doi.org/10.1016/j.optlaseng.

2015.01.012. URL https://www.sciencedirect.com/science/article/pii/S0143816615000135.

A. Attachments

A.1 First Attachment