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

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 the 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 [Kapinchev et al., 2015]. For many of these applications, the computation time of cross-corellation 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" [Bali, 2020], which uses both basic definition based cross-corellation 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, will be faster than the FFT based implementation.

Goals

The goal 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.

In steps, this thesis will:

- analyze optimizations of the definition based algorithm, focused on parallelization using CUDA platform,

- compare the optimized implementations with one based on Fast Fourier transform,
- measure different input sizes and types for both the optimized definition based and Fast Fourier transform based algorithms

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 [Wikipedia contributors, 2022]. 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.

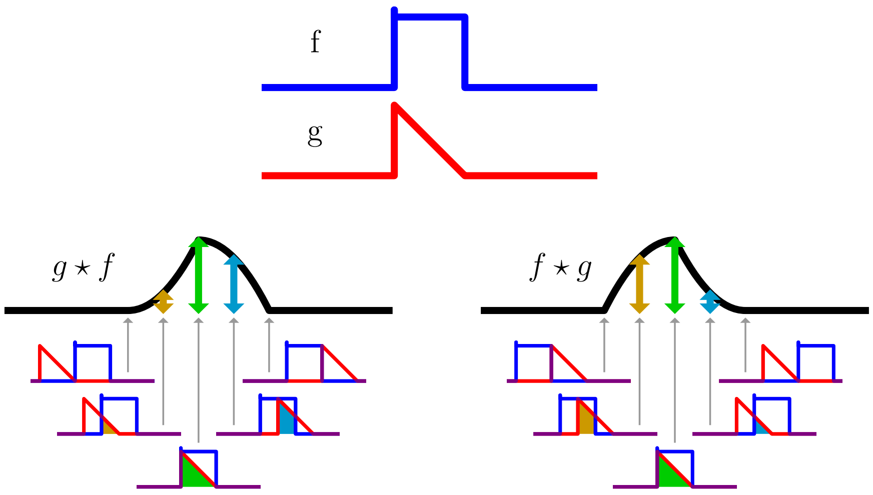


Figure 1.1: Cross-correlation of two functions. Wikimedia Commons contributors [2021]

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 [Wang, 2019], 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 [2020], 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.

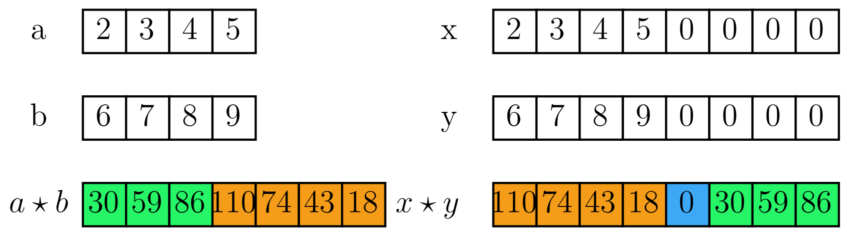


Figure 1.2: Comparison of linear and circular cross-correlation [Bali, 2020].

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 [Bali, 2020], as shown by Figure 1.3.

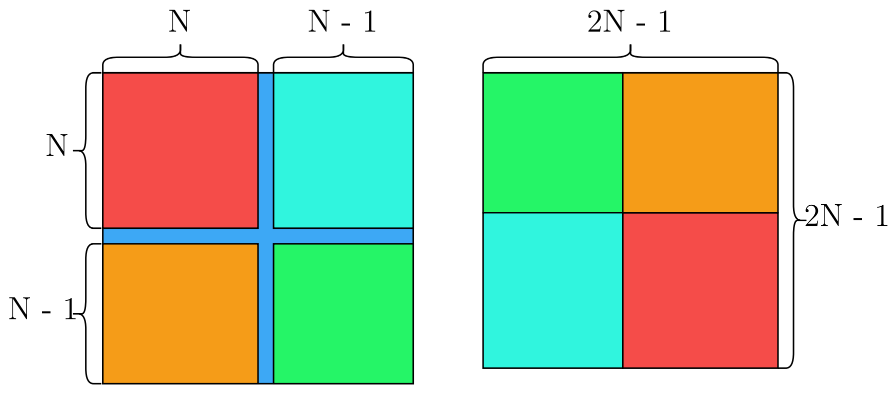


Figure 1.3: Result quadrant swap.

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, 2020], 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 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 [2020] and Zhang et al. [2015].

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 parallellism

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 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 different inputs. The two basic forms are:

1. n left inputs, each with m different right inputs (n to mn) Bali [2020] Zhang et al. [2015] Kapinchev et al. [2015],

2. x left inputs, each with all y right inputs (n to m) Clark et al. [2011].

There are several subtypes which can also be optimized for. For n disjoint sets of m right inputs, one set for each of the n left inputs, we have the following subtypes:

1. one to one,
2. one to many,
3. large number of pairs.

With these subtypes, we can more aggressively cache and reuse the left input. Any implementation capable of processing the general input form can also be used to implement all of these subtypes. Inversely, optimized implementation of any of the above subtypes can be used to implement the general n to mn input type and transitively any of the other subtypes.

Any implementation of the *one to many* subtype can also be used to implement the other major type, the *x to y* type, by running the *one to many x* times, possibly in parallel.

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 [Zhang et al., 2015] [Bali, 2020].

In the field of Seismology, cross-correlation is used for picking, ambient noise monitoring, waveform comparison and signal, event and pattern detection. [Ventosa et al., 2019]

In optical coherence tomography, the whole result of cross-correlation is summed to compute the intensity of each pixel [Kapinchev et al., 2015].

Any post-processing is outside the scope of this thesis. Result of cross-correlation will be taken as-is and validated against preexisting cross-correlation implementations.

2. CUDA

This chapter describes the Compute Unified Device Architecture, better known by its acronym CUDA, a ”general purpose parallel computing platform and programming model” [Nvidia, 2022], which allows simplified utilization of NVIDIA Graphics processing units (GPU) for solving complex computational problems.

First we describe the advantages and disadvantages of the GPU hardware. Next we describe the basic programming model, after which we provide more detail about features useful for parallelization of cross-correlation.

2.1 GPU

Central processing unit (CPU) is 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, GPU is optimized for throughput of a single stream of instructions working on many streams of data. The single stream of instructions is executed many times in parallel, which allows GPU 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.

This leads to the principle of **occupancy**, where GPU requires high number of threads to properly hide the high latency of each operation. Especially for small inputs, care needs to be taken so that the processing is split between enough threads to saturate the GPU.

The need to assign each thread separate registers in the register file as part of the execution context also highlights one of the limiting factors of occupancy, **register pressure**. Code requiring too many registers may limit the number of threads which can actually be executed in parallel, limiting occupancy, or leading to register spilling into slower types of memory, limiting performance.

2.2 Programming model

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

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

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. The abstraction defining the behavior of the device code, called the SIMT execution model, is described in the subsection 2.2.1.

2.2.1 Single Instruction, Multiple Threads

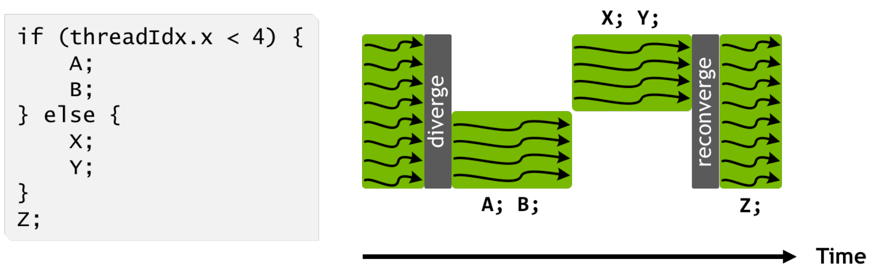
The device code, written in CUDA C++ as a part of *global* or *device* function, describes the behavior of a single thread running on the device. Compared to host code running on the CPU, the device code is always ran by many threads simultaneously.

In the Single instruction, multiple threads(SIMT) 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.[2] If branching occurs, as can be seen in Figure 2.1, 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 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.

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

On the other side of the spectrum, the SIMT execution model can be compared to the Single instruction, multiple data (SIMD) execution model, where

[2]Since Compute Capability 7.0 Volta, threads of a warp can be scheduled more independently and do not execute strictly in lockstep [NVIDIA, 2017].



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

Figure 2.1: Branching in device [Nvidia, 2022].

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.

2.2.2 Thread hierarchy

Apart from being grouped into warps, threads on the device are also 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 2.2. 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.

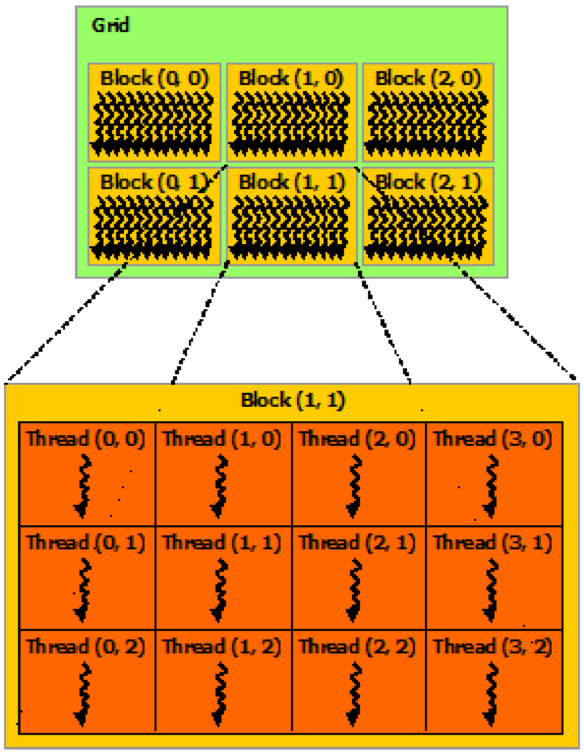


Figure 2.2: Thread grouping hierarchy [Nvidia, 2022].

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.

2.2.3 Thread cooperation

CUDA provides several mechanisms for thread cooperation. Threads can cooperate on the following levels of thread hierarchy, with increasing 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" [Nvidia, 2022].

Shared memory is a small on-chip memory, described in more detail in the subsection 2.2.5. Each thread block has private shared memory, accessible only from 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 access to shared memory between threads of the thread block, 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

On 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 operations, warps can perform:

- reduce-and-broadcast operations,
- broadcast-and-compare operations,
- reduce operations,

For data exchange, CUDA C++ provides several warp shuffle instructions. There are four source-lane addressing modes:

- direct lane index,
- copy from lane with ID lower by *delta*,
- copy from lane with ID higher by *delta*,
- copy from lane based on bitwise XOR of provided *laneMask* and own lane ID.

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.

Only direct lane indexing performs lane index wrap around. If the given lane index is out of the range $[0 : width - 1]$, the actual lane index is computed as: $srcLane \mod width$. In other addressing modes, the lanes with out of range source lane index are left unchanged, receiving the value they pass in. The wrap around mechanism allows us to rotate data between threads instead of just shifting. The direct lane indexing can also be used to broadcast a value from a single lane to all other lanes.

For warp level operations, the reduce-and-broadcast operations receive a single integer value from each lane which they compare to zero, making it effectively a boolean. The results of the comparison are then reduced in one of the following ways and the result is broadcast to all threads:

- result is non-zero if and only if all of the values are non-zero,
- result is non-zero if any of the values are non-zero,
- result contains single bit for each lane which is set if the value given by the lane was non-zero

The broadcast-and-compare operations broadcast the value given by each lane and compare it to the value given by the current lane, returning:

- mask of lanes that have the same value,
- mask if all threads in mask gave the same value, 0 otherwise.

Finally, there are the general reduce operations. *Add*, *min*, *max* operations are implemented for signed or unsigned integer values. *And*, *or*, *xor* operations are implemented for unsigned integers only.

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 [Nvidia, 2022]. 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[3].

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 handles to objects of data types representing the implicit groups.

The other type are *explicit groups*, which must be explicitly created from one of the implicit groups.

- 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.

Creating a handle for an implicit group is a collective operation, in which all threads of the group must participate. Creating the group handle in a conditional branch may lead to deadlocks or data corruption. It may also introduce unnecessary synchronization points, limiting concurrency. Similarly to implicit group handle 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 implicit group handles and do all partitioning at the start of the kernel and pass const references throughout the code Nvidia [2022].

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

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 be transferred 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 2.3. For this thesis, the main types are:

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

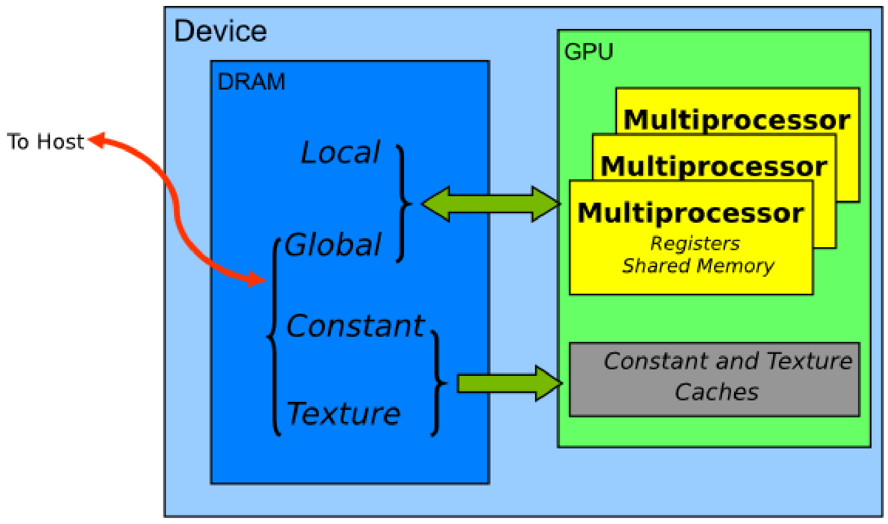


Figure 2.3: Memory types on a CUDA device [Nvidia, 2022].

Local and global memory are both allocated from device memory, and as such have very high access latency.

Local memory is 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.

Global memory is shared by all threads of a kernel, 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 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.

2.2.6 Streaming multiprocessor

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 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.

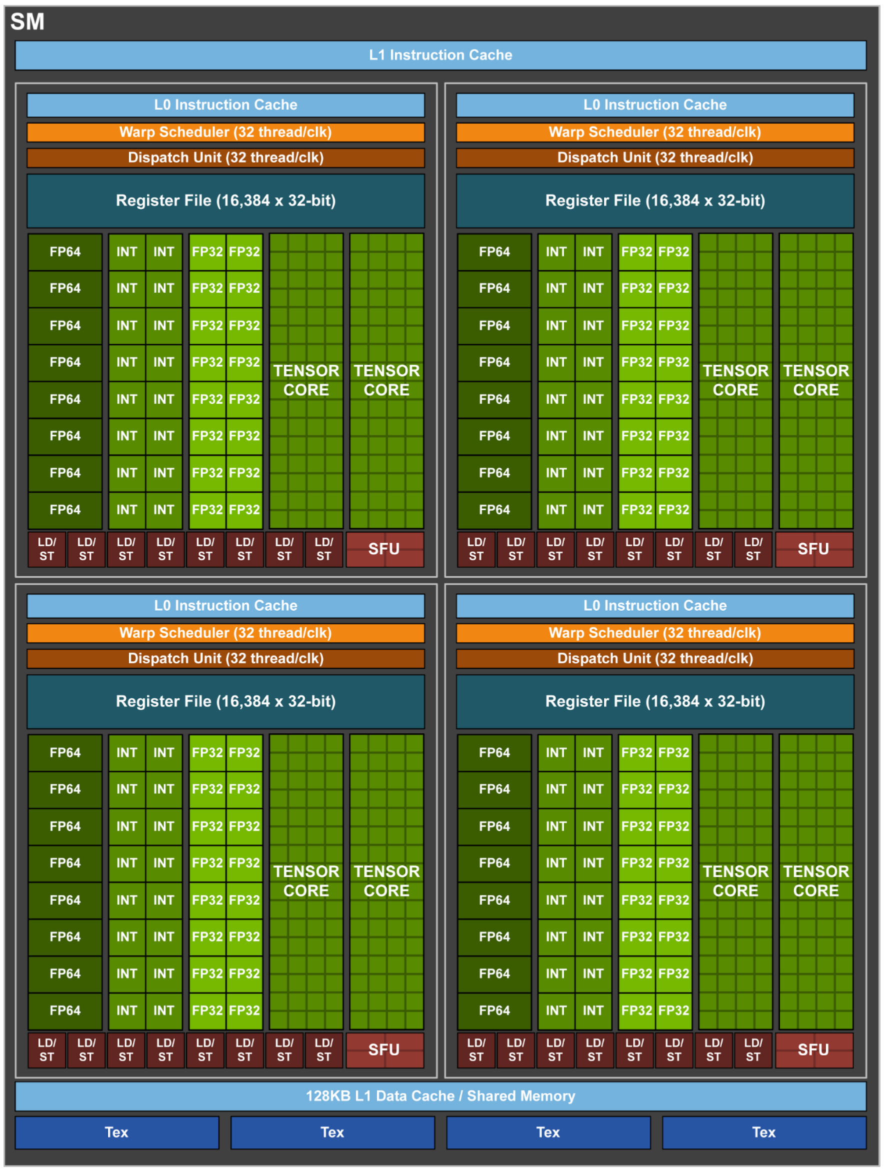

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 2.4: Streaming multiprocessor [NVIDIA, 2017].

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:

- 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, which if exceeded, makes the pipeline implementing the instruction saturated and unable to execute any other 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.

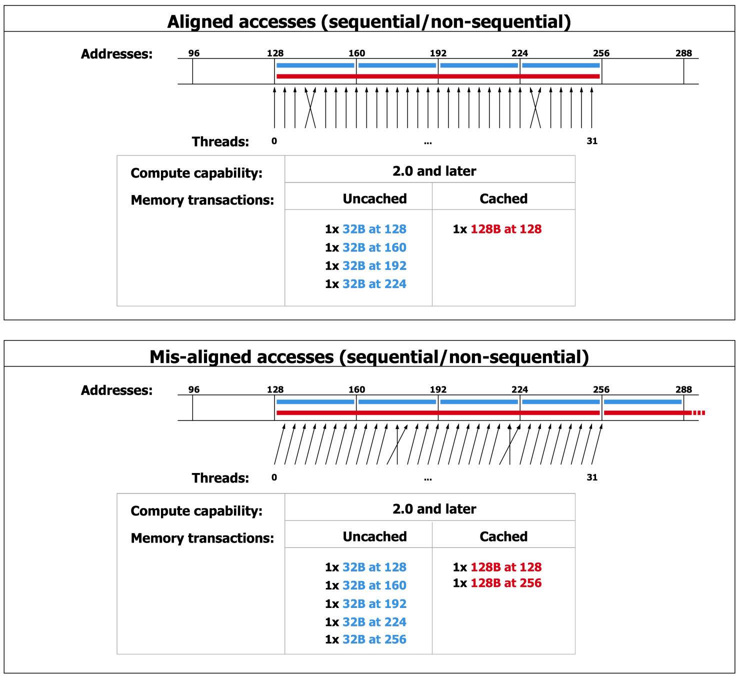
High LSU utilization reflects that the program may be memory bound, waiting for data from global or shared memory, or that the program executes many warp shuffle instructions, which are also implemented by the LSU pipeline. Due to this, the usage of shared memory together with warp shuffles is not advisable,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

2.3.3 Global memory access

Global memory access utilizes two levels of caching. 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. L1 cache is, by default, used for read-only data, such as the two input matrices in cross-correlation. L2 cache, with cache line size of 32 B, is still on-chip but is shared by all SMs. This cache is used by all accesses to global memory.

As we can see in Figure 2.5, the access to read-only data in 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.


Aligned accesses (sequential/non-sequential)
Addresses:
96
128
160
192
224
256
288
Threads:
0
...
31
Mis-aligned accesses (sequential/non-sequential)
Addresses:
96
128
160
192
224
256
288
Threads:
0
...
31

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

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 [Nvidia, 2022].

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, as can be see in the left column of Figure 2.6. If multiple threads access different addresses mapping to the same bank, as can be seen in the middle column of the figure, 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*. Access to the same address by multiple threads does not lead to a bank conflict, instead leading to a broadcast of the value between the threads.

2.3.5 General recommendations

We can summarize the information in previous subsections into few simple rules [Nvidia, 2022]:

1. Maximize parallel execution to achieve maximum utilization;
2. Optimize memory usage to achieve maximum memory throughput;
3. Optimize instruction usage to achieve maximum instruction throughput.

To maximize parallel execution, ensure 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.

To optimize memory usage, minimize transfers from lower bandwidth memory by reusing data in hardware cache or manually move data to shared memory. When accessing global memory, utilize coalesced accesses to minimize unnecessary data transferred. When accessing shared memory, minimize bank conflicts.

To optimize instruction usage, minimize 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. Minimize thread divergence to ensure all threads in a warp execute useful instructions.

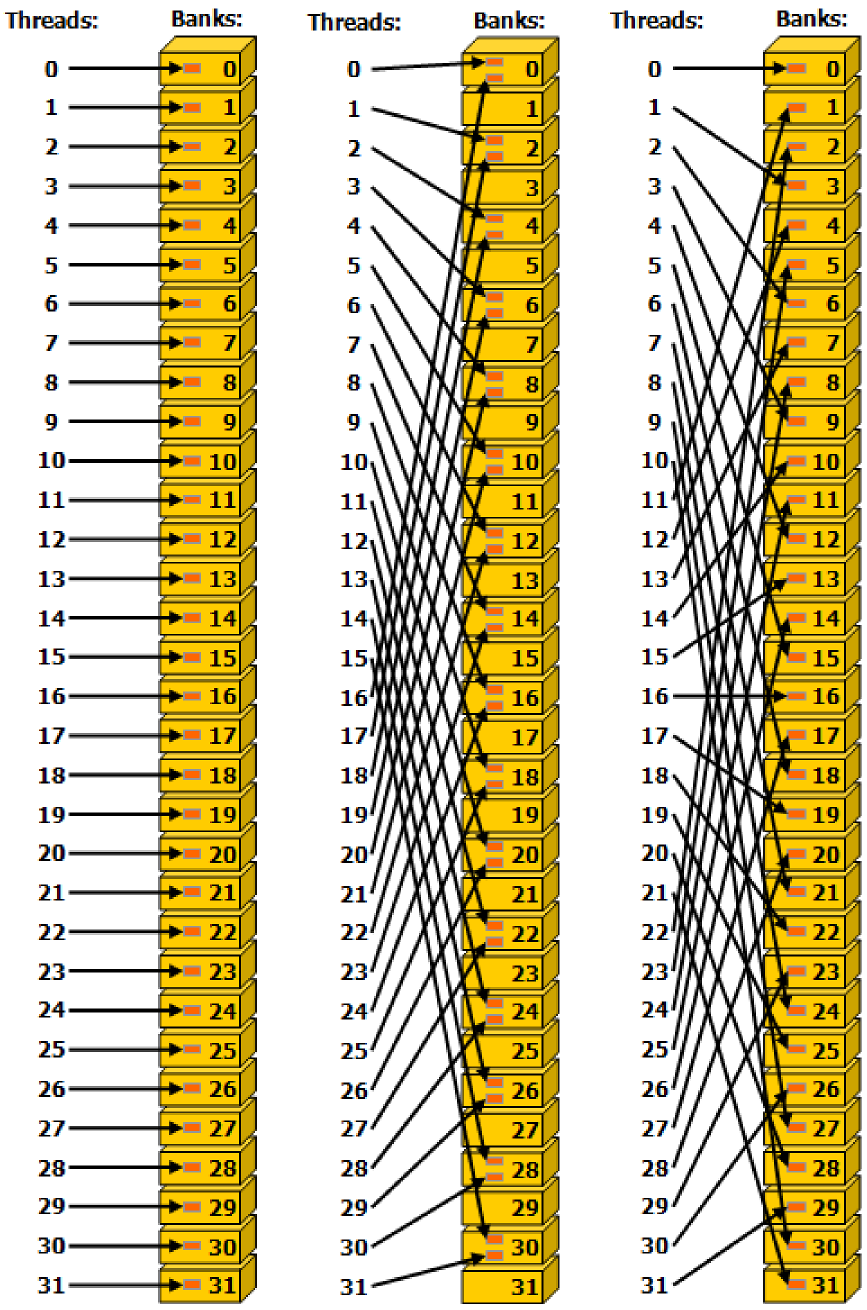


Figure 2.6: Shared memory access patterns [Nvidia, 2022].

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 an implementation based on Warp Shuffle instructions, with additional 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 implementation.

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. As described in Section 1.3, each element of the output matrix can be computed independently in parallel.

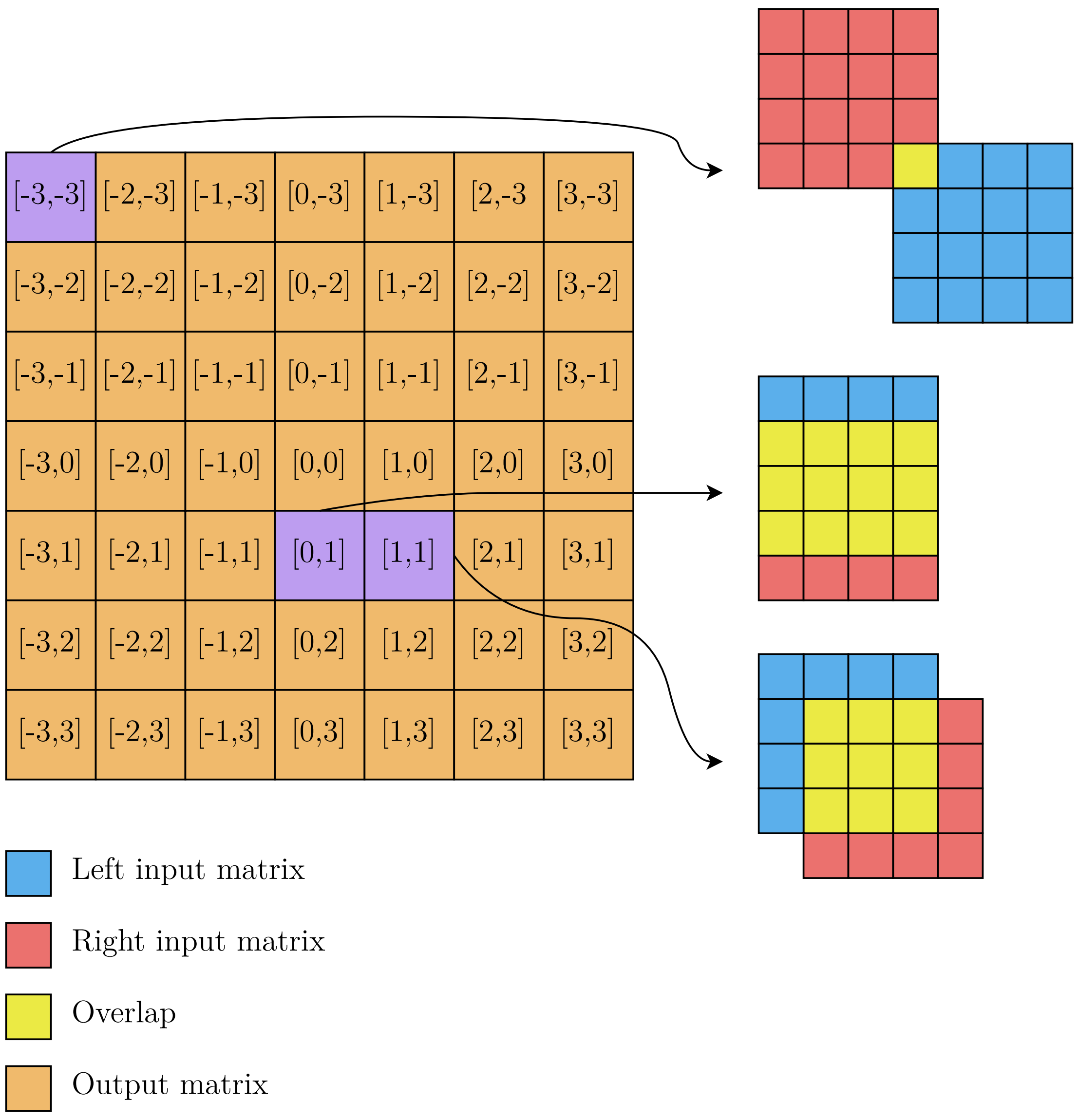


Figure 3.1: Result matrix with corresponding relative shifts.

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 shift 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. We then define the types of workers which can be derived from CUDA Thread hierarchy, described in Section 2.2.2. Next we introduce the possible distributions of tasks between different types of workers, with options for data reuse and load balancing.

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 can be seen in Figure 3.2. First level are the different relative shifts of the two input matrices, each represented by a single element in the output matrix. Each of these tasks has a set of independent subtasks corresponding to overlapping pairs of elements of the two input matrices. Each subtask is depicted as a yellow square in Figure 3.2. Each of these subtasks belongs to exactly one first level task, creating a tree structure. The results of all subtasks of a first level task have to be summed into the result of the parent task. The set of subtasks defines a submatrix in both input matrices, as can be seen in Figure 3.1.

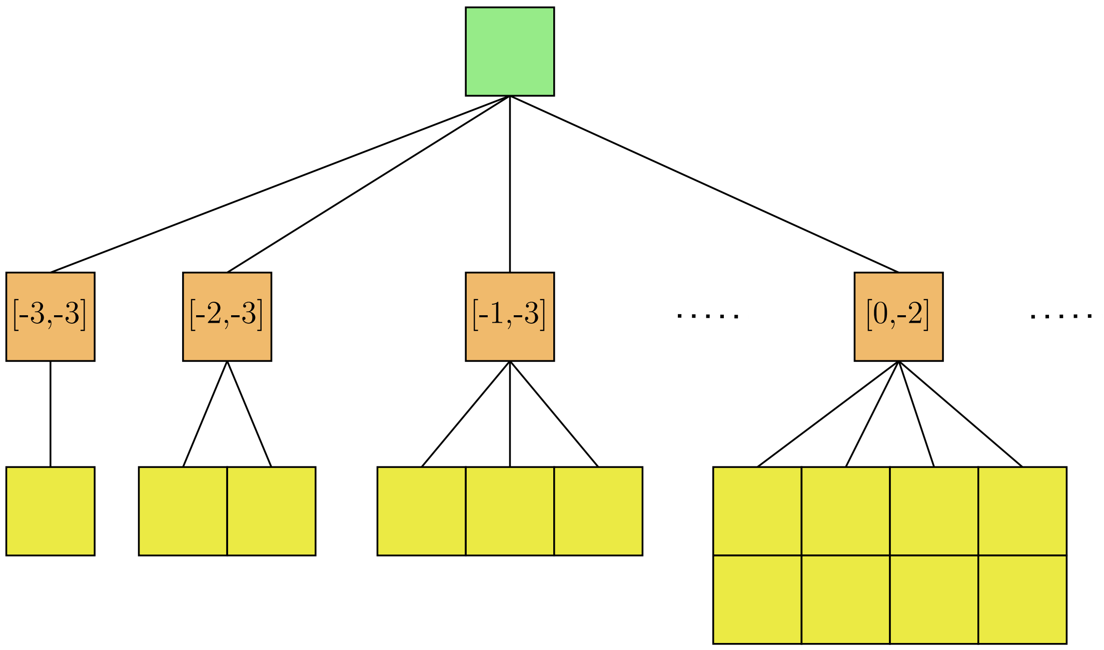


Figure 3.2: Tasks hierarchy in definition-based one-to-one cross-correlation.

The goal is to distribute the subtasks 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 can add additional levels to the task hierarchy shown in 3.2. 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*,
2. *n-to-mn*,
3. *n-to-m*.

As we can see, the *one-to-one* type, described in the previous section, together with the *one-to-many* type, are subtypes of the more general *n-to-mn* type. We separate the *one-to-one* and *one-to-many* types as they offer a great possibility for caching the single left matrix.

All of the described types can be partitioned into many *one-to-one* cross-correlations, as can be seen in Figure 3.3. 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 *n-to-mn* and *n-to-m* types is that in the *n-to-mn* type, each of the n left matrices is cross-correlated with a different set of m right matrices, whereas in the *n-to-m* type, all n left matrices are cross-correlated with the same m right matrices.

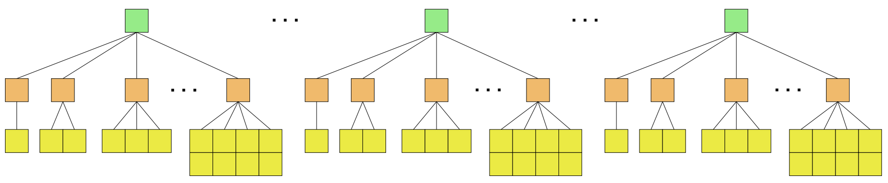


Figure 3.3: Task hierarchy of types with many matrices.

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 children of a green box have to be written into a single matrix, each into a different element without any collisions. Results of the 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 and sharing the same left matrix can be computed by reusing the data from the left matrix. Relations such as this can be used to group tasks for workers to reuse data or pass data to other neighboring workers to reduce the required memory throughput.

3.1.3 CUDA workers

Section 2.2.2 describes how CUDA threads are hierarchically grouped from smallest to largest as follows:

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

This thesis provides several implementations of the definition-based cross-correlation algorithm described in following sections, each mapping different level of task hierarchy shown in Figure 3.3 to different level of CUDA thread hierarchy.

Based on the choice of the CUDA thread group size, we can utilize smaller groups to compute the subtree of the assigned task, and primitives provided by larger groups to synchronize, communicate and combine results of the tasks between different workers.

3.2 Warp shuffle algorithm

This section describes the implementation of definition-based cross-correlation utilizing Warp Shuffle instructions. We first introduce a simple version of the implementation, later improving it step by step with optimizations evaluated in Section **??**.

In the simplified implementation, Warp Shuffle instructions are utilized to shift data loaded from the left matrix and broadcast data loaded from the right matrix between threads in a warp. CUDA threads are used as workers, with each task representing a single relative shift of the two matrices (which is equivalent to a single element of the output matrix), as can be seen in Figure 3.4.

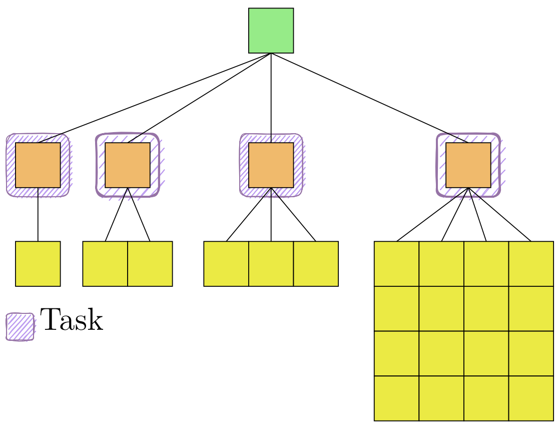


Figure 3.4: Tasks in simple warp shuffle algorithm.

The main idea behind this algorithm is illustrated in Figure 3.5. We see how two threads computing shifts $[0, 1]$ and $[1, 1]$ process elements of the two input matrices. The numbers represent iterations of a *for loop* in the code.

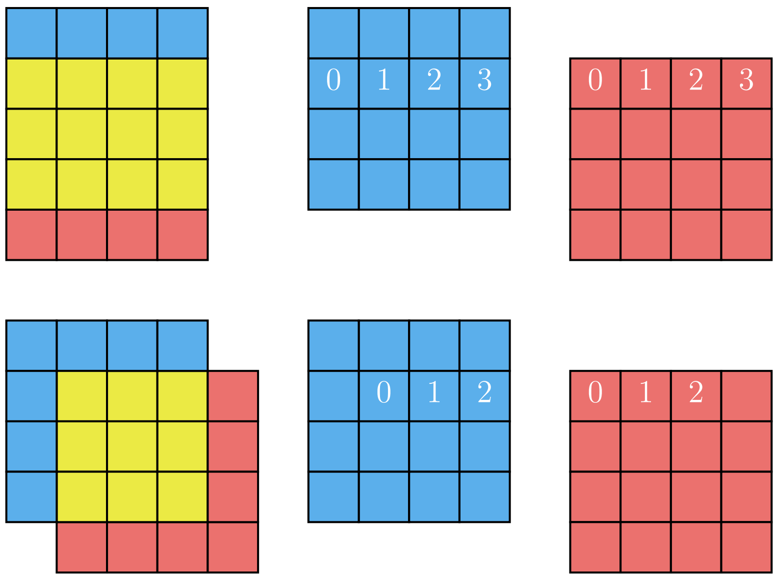


Figure 3.5: Work done by two neighboring threads.

As can be seen, 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.

To utilize these properties, threads of a single warp process 32 consecutive shifts in the x axis all with the same y axis value, as can be seen in Figure 3.6. 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 runtime algorithm argument. The grid size is set so that the output matrix is fully covered by threads. Any extra threads are handled by the bound checked reads described next.

The problem of iteration *3* in Figure 3.5, in which the lower thread 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, making it effectively a *noop* instruction 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.

3.2.1 Algorithm steps

The following description assumes *warp size* to be 32, as is the case for all existing Nvidia GPUs. To utilize coalesced loading from global memory, the left buffer

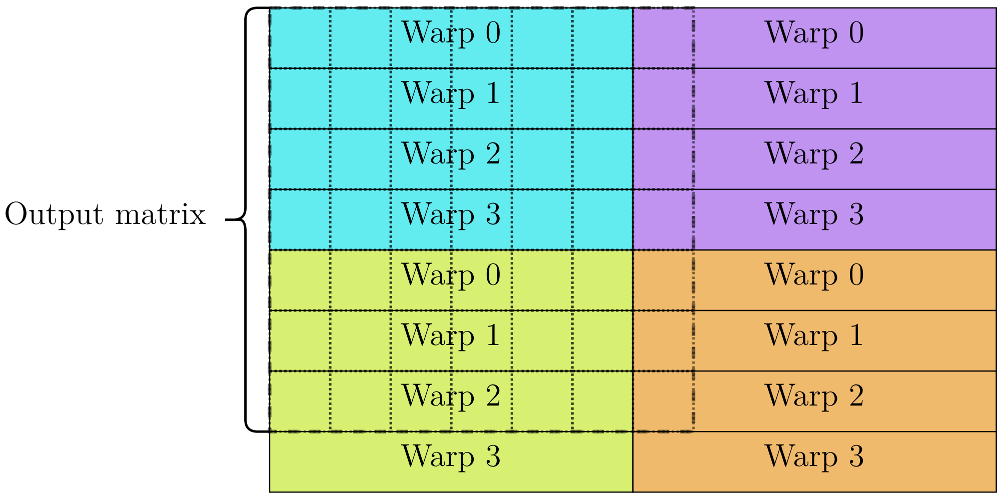


Figure 3.6: Distribution of shifts between CUDA threads, warps and blocks.

shuffled between threads is split into two parts of 32 items each, which together function as a single 64 item ring buffer. As described above, any out of bounds loads are range checked and load value 0 instead using code such as this::

```
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 the buffer, bound checked load shown above is used to load 32 consecutive values (or 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
);
```

As we can see, 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 shifted using the Warp Shuffle instructions.

The algorithm can be described by the following pseudocode:

```
COMPUTE warp overlap bounds
COMPUTE thread output position

SET sum = 0
FOR each overlapping row
  bound checked load of thread_left_bottom

  FOR every 32nd item in overlap row
```

```
    bound checked load of thread_left_top
    bound checked load of thread_right

    run 32 iterations of main loop
  ENDFOR
ENDFOR

IF thread output not out of bounds
  write sum to output
```

We compute bounds for the union of overlaps processed by the threads of the warp. The union of overlaps gives us a submatrix of the right matrix, which we then iterate over using the two nested for loops. When loading from the left matrix, each thread uses its assigned shift derived from the output position to read the correct element of the left matrix. As threads of a warp compute shifts which have the same y axis and are sequential in the x axis, the bound-checked loads of the left buffer result in sequential coalesced reads. As the loads of the right buffer are bound checked from the bottom by the overlap bounds, the bound-checked reads are utilized only in the last iteration when the size of the overlap is not divisible by warp size. We go through each row of the overlap in steps of 32 as that is the number of items processed by the main loop, based on the number of threads in a warp.

The main body of the algorithm is a for loop with 32 iterations, where in iteration i we broadcast right value from thread i and each thread multiplies this value with the value from the bottom left buffer stored by this thread. We then shift the left buffer using two warp shuffle instructions. After 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, allowing us to go ahead with the next iteration of the loop over row items.

The main loop is illustrated by the following code:

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

3.2.2 Work distribution

In the simplified algorithm described above, there are massive differences in work done by different threads. As we can see in Figure 3.7, the thread processing the left overlap has much less work than the thread processing the right overlap. This will lead to problems with occupancy once the threads with small amount of work are completed.

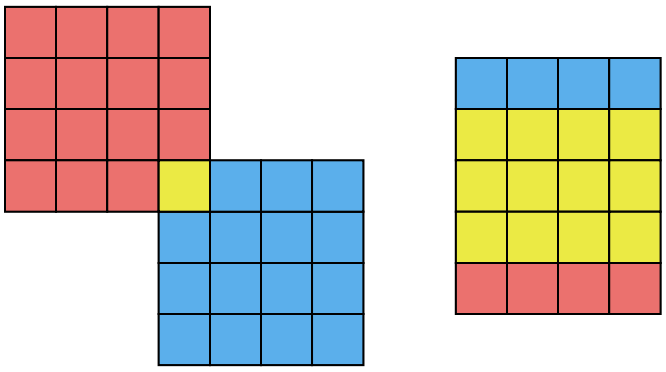


Figure 3.7: Task size difference in the simplified algorithm.

To distribute the work more evenly, we need to change what is considered a task processed by a worker. Compared to the simplified implementation, where task represents the whole overlap defined by the given shift, with work distribution a task represents several full continuous rows of overlapping pairs of elements (yellow boxes), as can be seen in figure 3.8. With this change, multiple workers may write to the same element in the output matrix. Each worker must add the final sum of the assigned task to sums of all other workers processing tasks of the same shift. As each worker needs to add the sum just once, 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 shift 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.

We provide several algorithms to derive the number of workers started and the mapping from thread ID to the row of the output matrix and the worker rank for given row. The provided algorithms are:

- None,
- Rectangle,
- Triangle.

The algorithms are illustrated in Figures 3.9 and 3.10. In these, the purple boxes represent number of tasks for each shift in the given row of the output matrix. Each task represents given number of full continuous rows of the overlap. For Figure 3.9, each purple box represents a single row of the overlap, or in other words, purple boxes represent number of overlapping rows for each shift in given output matrix row. As we define work distribution based on number

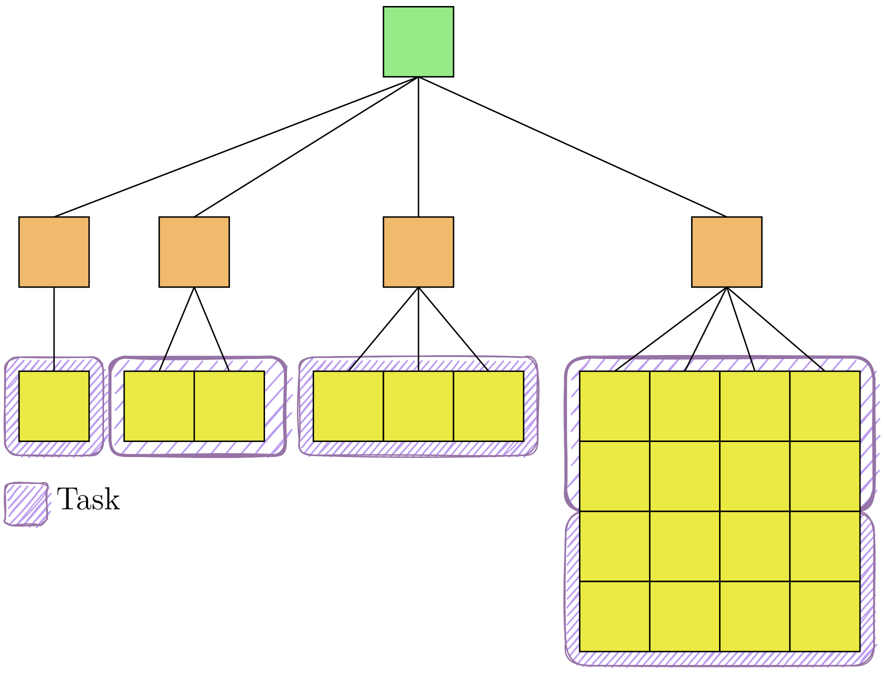


Figure 3.8: Tasks in warp shuffle algorithm with load balancing.

of overlapping rows, worker ID is the y axis of the thread ID, i.e. $blockIdx.y * gridSize.y + threadIdx.y$. This means that all threads with the given value of y axis share the same worker ID. Due to the way we utilize Warp Shuffle instruction to shift values along the x axis, i.e. along each row, the distribution of work into tasks takes into account only the number of rows, not the size of each row, as can be seen in Figure 3.8.

None distribution

The *None* distribution is provided mainly to measure the overhead of the code changes required to implement work distribution. This distribution behaves identically to the simplified algorithm with no distribution. As such, it starts a single worker for each shift in the output matrix. The workers are assigned by directly mapping their y axis ID to the row of the output matrix.

Rectangle distribution

The *Rectangle* distribution computes m, the maximum number of tasks required for any shift, and starts m workers for all shifts, creating a rectangle of workers which can be seen in Figure 3.9. The redundant workers are stopped immediately after work assignment. To start the required number of workers, the y axis of grid size is multiplied by m. The tasks are assigned using $worker\_ID \bmod output\_matrix\_size.y$ with worker rank computed as $worker\_ID/output\_matrix\_size.y$.

Triangle distribution

The *Triangle* distribution starts exactly one worker for each task. The disadvantage of this distribution is the complex computation required to assign workers to tasks. This computation includes many multiplications, divisions and most importantly a low throughput square root instruction. For inputs where the size of each task is small, the overhead of triangle distribution may be greater than any gains provided by load balancing and increased occupancy due to work distribution. The grid size is increased to start at least the required number of workers, with some overallocation due to the fixed size of thread blocks. To map worker ID to output matrix row and worker rank, we use the following formula, which computes the worker ID i of the worker at the start of the triangle row x:

$$r * x^2 - (3r - t) * x + (2r - t) = i$$

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. Solved for x, we get the following formula:

$$x = floor(\frac{(3r - t) + \sqrt{(3r - t)^2 - 4r(2r - t - i)}}{2r})$$

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

Lastly, we need to compute the variable t, the number of items on the top row of the triangle. The number of items on the top row can be computed as:

$$output\_matrix\_size.y \mod (2 * r)$$

.

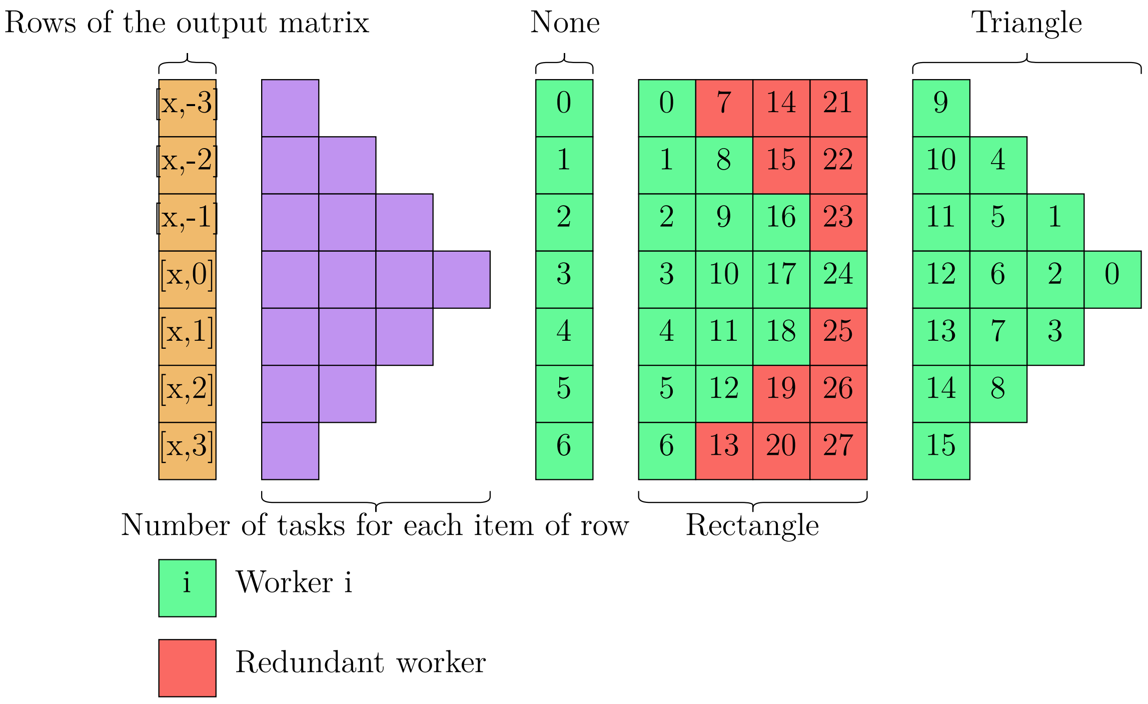


Figure 3.9: Work distribution of 4x4 input with 1 row per task.

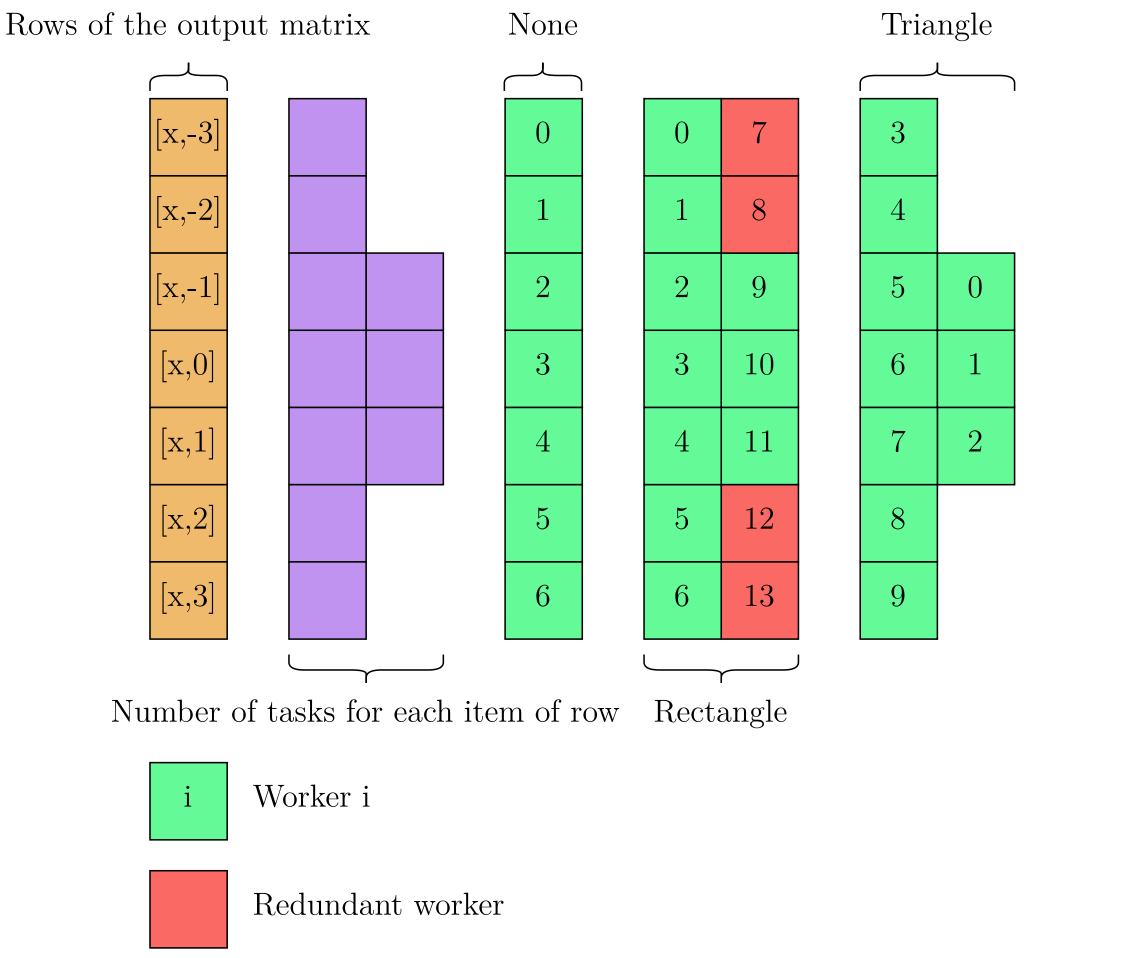


Figure 3.10: Work distribution of 4x4 input with maximum of 2 rows per task.

3.2.3 Utilizing multiple right matrices

Another problem of the simplified implementation not solved by work distribution, described in Section 3.2.2, is the ratio of warp shuffle instructions to arithmetic instructions. 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 can be seen in Figure 3.11. The warp shuffle instructions (SHFL) dominate the executed instruction mix, which results in 97% utilization of the *LSU* pipeline implementing the shuffle instructions, as can be seen in Figures 3.12. Compare this to the fused multiply-add instructions (FFMA) implementing the multiplication and addition, which is implemented by the *FMA* pipeline with less than 10% utilization for the simplified algorithm.

There are several ways to improve the ratio of warp shuffle instructions to the fused multiply-add instructions. The simplest way is to utilize the *one-to-many* type of computation, and let each worker compute cross-correlation between one left matrix and many right matrices at once. We call this the *multimat_right* optimization. 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 warp shuffle instruction, while also adding one fused multiply-add instruction. The ratio of warp shuffle to fused multiply-add instructions can then be expressed as $2 + r : r$, where r is the number of shifts (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 effects of this optimization can be seen in Figures 3.11 and 3.12. These figures compare the simplified algorithm against the optimized algorithm using 8 right matrices per worker on input of size 256x256 with 1 left matrix and 16 right matrices. 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 warp shuffle instructions compared to the fused multiply-add 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.

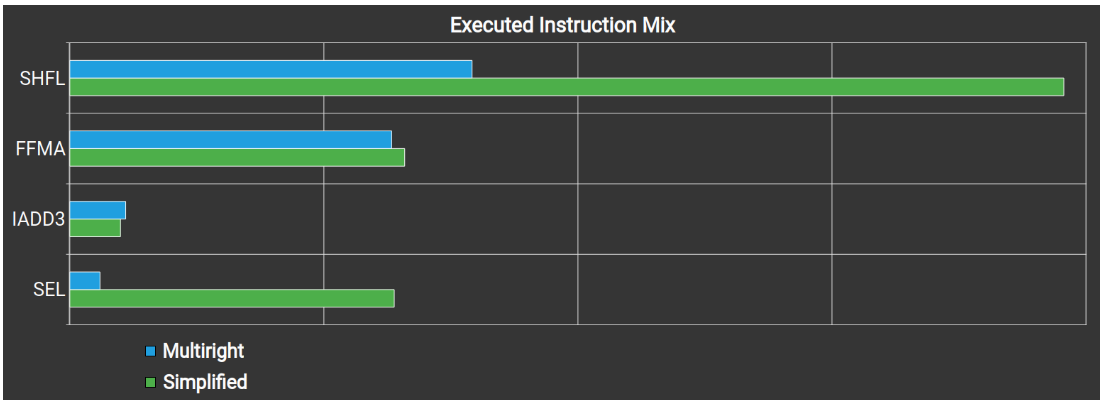


Figure 3.11: Comparison of instruction mix between *one-to-many* simplified algorithm and the multiple right matrix optimization.

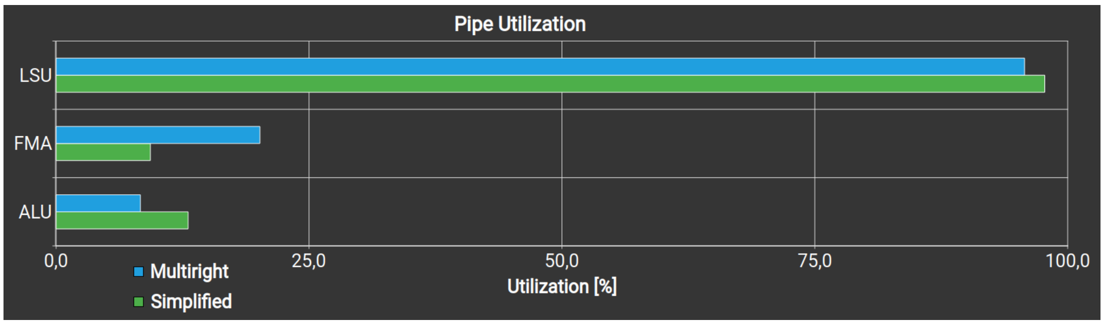


Figure 3.12: Comparison of pipeline utilization between *one-to-many* simplified algorithm and the multiple right matrix optimization.

3.2.4 Multiple rows from the right matrix

Another way to improve the instruction ratio is to process multiple rows from the same right matrix, which can be used even for the *one-to-one* type of computation. We call this the *multirow_right* optimization. There are several caveats when implementing this optimization. The main difference is that a single thread now computes multiple different shifts instead of the same shift in multiple matrices, as is done by the *multimat_right*. These shifts differ in the y axis, and represent consecutive elements in a column of the output matrix, as can be seen in Figure 3.13. Each of these shifts represents different overlap of the two input matrices, requiring different bounds in the y axis.

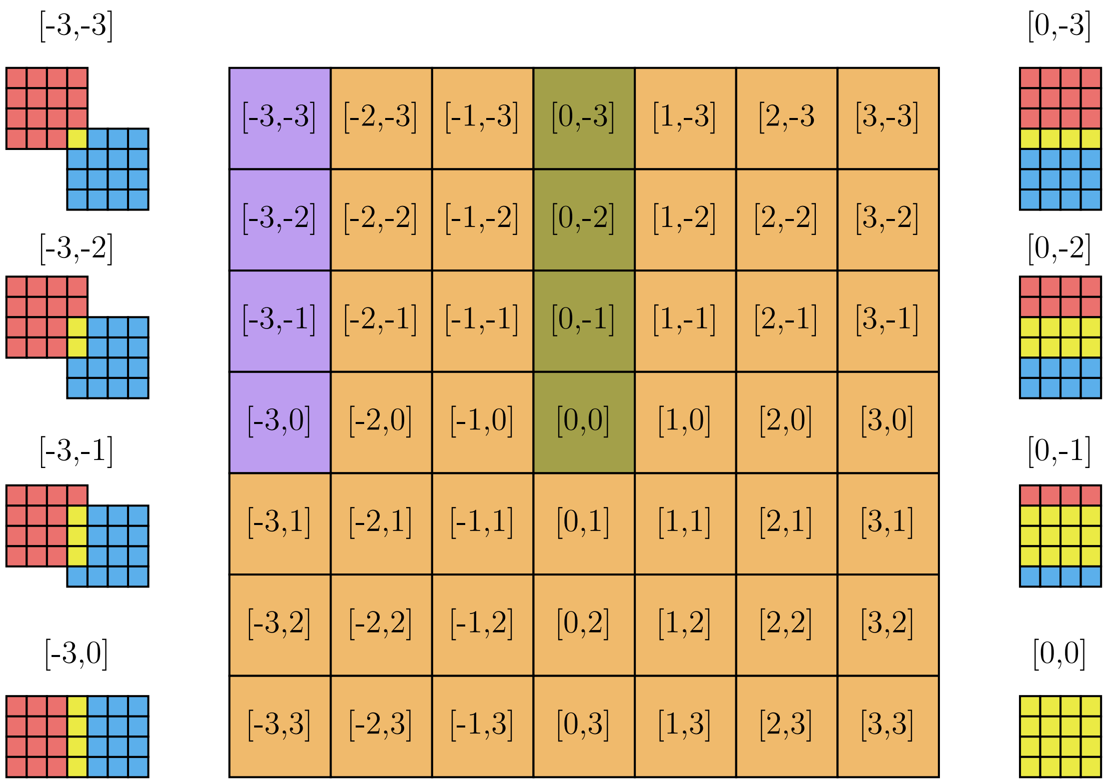


Figure 3.13: Shifts assigned to workers by the *multirow_right* algorithm with 4 shifts per worker.

Due to these different bounds for the shifts computed by the worker, we have to add explicit initialization and finalization code which handles the case where the current row being processed from the left matrix overlaps with the right matrix in a subset of shifts computed by the worker, as can be seen in Figure 3.14. This figure shows a run of the algorithm with three shifts per worker. For threads which compute shifts with top of the right matrix (in red) inside the left matrix (in blue), i.e. shift with a positive y axis, or where the top of the right matrix is less than $number\_of\_shifts\_per\_thread$ above left matrix, i.e. $y < 0$ and $y > -number\_of\_shifts\_per\_thread$, we must run separate initialization steps. These can be seen in Figure 3.14, where the first row processed from the left matrix overlaps in only a single shift computed by the thread, second row overlaps in two shifts and rest overlap in all three shifts. As such, we first run initialization step which processes a single row, after which we run an initialization step processing two rows. The maximum number of initialization steps is equal to $number\_of\_shifts\_per\_thread - 1$. If the top of the right matrix is outside the left matrix, i.e. the shifts computed by the thread have negative y axis, with $y0 = min(y)$ being the minimum vertical shift, the first $abs(y0)$ initialization steps are skipped.

Similar problem is present when thread computes shifts in which the bottom of the right matrix is inside the left matrix, i.e. $y < 0$, or less than $number\_of\_shifts\_per\_thread$ below the left matrix, i.e. $y > 0$ and $y < number\_of\_shifts\_per\_$
This situation is illustrated in Figure 3.15.

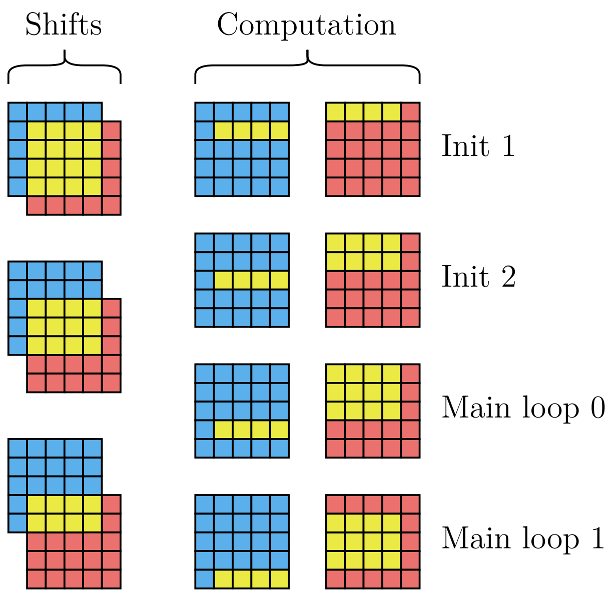


Figure 3.14: Initial steps of the *multirow_right* algorithm.

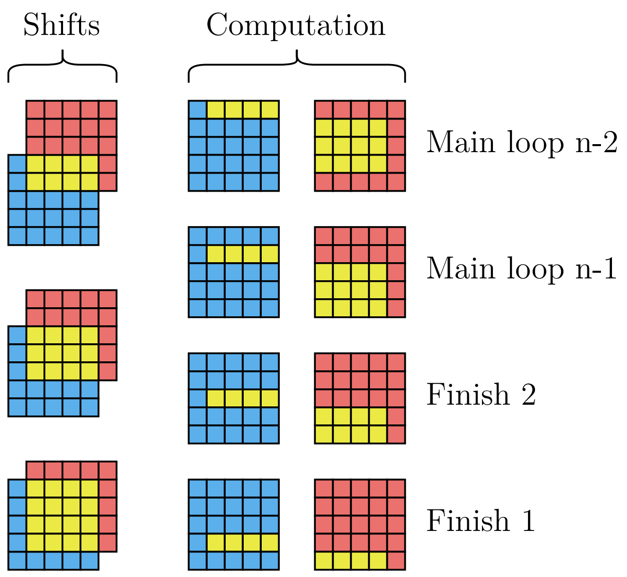


Figure 3.15: Final steps of the *multirow_right* algorithm.

After the initialization phase is finished, the main loop is conceptually very similar with the multiple right matrices case described previously. The only major differences are the accumulation of the final result through shared memory instead of registers and the reversal of the results for given row group.

The complexity of the code forces us to share the results of different function calls, such as the initialization, main body and finalization, through shared memory. We allocate a shared memory array with size $max\_shifts\_per\_thread * thread\_block\_size$, which is used to accumulate the final result of all shifts for all threads of the thread block. This array can be seen in Figure 3.16. The array begins with the first shift of each thread of the thread block, then second shift of each thread of the thread block, up to shift $max\_shifts\_per\_thread$, which is an algorithm argument. As all threads of a warp compute the same number of shifts, this ordering ensures access to shared memory causes no bank conflicts.

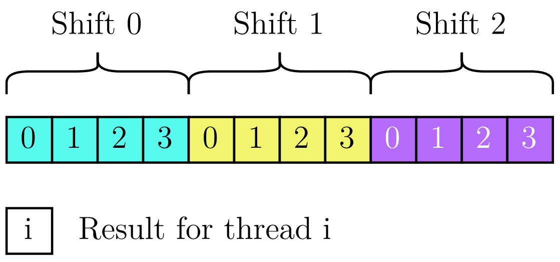


Figure 3.16: Shared memory allocation to threads.

When storing the result of main loop iteration into the shared memory array, we must reverse the results as can be seen in Figure 3.17. Multiple rows loaded from the right matrix are ordered as they are loaded from the matrix, indexing them 0 to n. As we can see, the shift in which given row from right matrix overlaps the row from the left matrix is given by $n - i$, which requires us to inverse the index when storing the result into shared memory. This needs to also be done during initialization and finalization steps, which is another reason why results need to be accumulated through shared memory, as we need to pass only the last s shifts to the step `Finish s`, for which the simplest implementation is pointer arithmetic on the shared memory array.

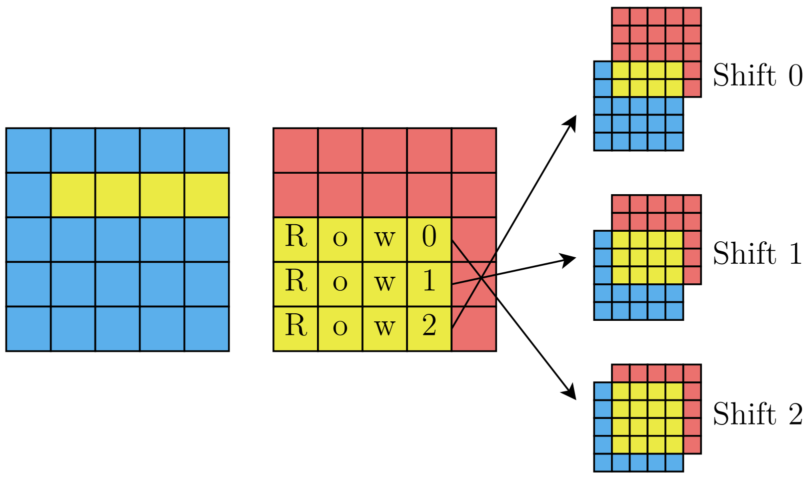


Figure 3.17: Reversal of results when accumulating to shared memory.

One of the disadvantages of this algorithm is the repeated reading of the same rows of the right matrix by each worker. As we already go through each row by shuffling from left to right, there is no simple way to reuse rows from top to bottom as we compute the overlap for given shifts. With 3 shifts per worker, each row of the right matrix processed by the main loop is read 3 times, once for each of the shifts processed by the worker. Each time, it is used with different left row. In the *multimat_right* optimization, these repeated reads are done by different workers, making them parallel and improving occupancy. This can be improved by additionally utilizing multiple left rows during the computation, reducing the number of times the rows from the right matrix need to be reread. This implementation is described in Section 3.3.4.

When profiling this optimization, as shown in Figures 3.18 and 3.19, we can see similar improvements as with the previous *multiright* 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. We can also combine these two optimizations, described in Section 3.3.6. As before, the *LSU* pipeline remains a bottleneck, but the utilization of *FMA* pipeline is improved from 9% to 17%. With 4 shifts per worker, the ratio improves from $3:1$ 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 shift, mainly due to the smaller input size compared to the *one-to-many* computation.

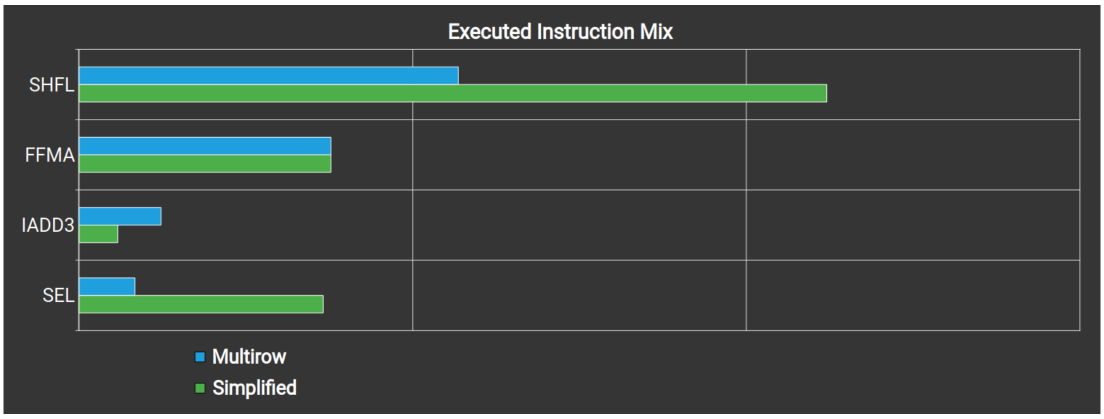


Figure 3.18: Comparison of instruction mix between *one-to-one* simplified algorithm and the *multirow_right* optimization.

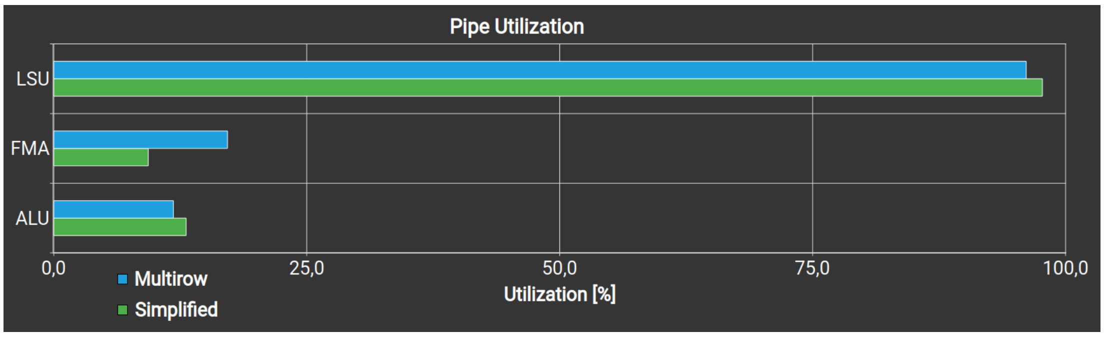


Figure 3.19: Comparison of pipeline utilization between *one-to-one* simplified algorithm and the *multirow_right* optimization.

3.3 Advanced warp shuffle optimizations

The optimizations of the warp shuffle algorithm described up to this point hint at further possibilities, such as using multiple left matrices and multiple rows from the left matrix in combination with the already described optimizations. This section first introduces the Local array optimization done by the *nvcc* compiler, on which our implementations heavily rely. Next we describe the problem with the *nvcc* compiler not performing the Local array optimization with our solution to this problem. We then utilize this solution to implement two further optimizations, which extend and are combined with the basic optimizations described in Section 3.2. Lastly we illustrate the effects of the Local array optimization on the implemented solutions.

3.3.1 Local array optimization

The *multimat_right* and *multirow_right* optimizations heavily rely on the CUDA compiler to place arrays such as the following into registers:

```
template<size_t NUM_RIGHTS, typename T, typename RES>
__device__ void warp_shuffle_impl(...) {
...
  RES 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(...);
  }
...
}
```

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. This allows for very fast access without placing any additional strain on the *LSU* pipeline. 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. Each access also utilizes the *LSU* pipeline, interfering with the warp shuffle instructions.

3.3.2 Advanced optimizations and local arrays

When implementing the *multimat_both* and *multirow_both* optimizations, described in Sections 3.3.3 and 3.3.4 respectively, we encountered a problem with the *nvcc* compiler not optimizing the local arrays into registers.

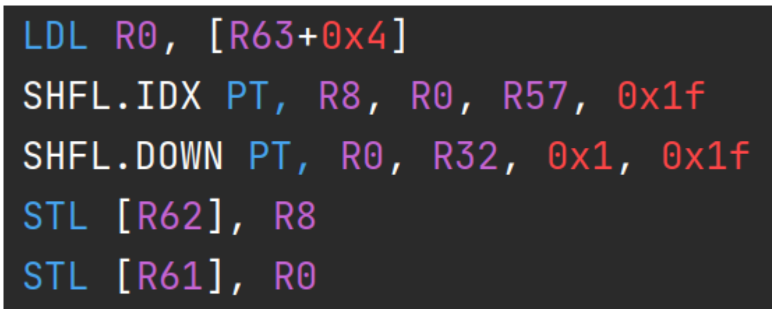
Using profiling and by examining the SASS, we isolated the problem to the *thread_left_bottom* and *thread_left_top* arrays. We further isolated it to the following part of the code, which in its original form shared by the simplified, *multimat_right* and *multirow_right* implementations looks like 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 a single or multiple left matrices, the code needs to be changed into the following:

```
#pragma unroll
for (dsize_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 Figure 3.20, the compiler behaves as if dynamic indexing was used and pulls the arrays into local memory. Due to the closed source nature of the *nvcc* compiler, we can only speculate on the reasons why this version worked while the other versions did not. One possibility, based on the generated SASS instructions seen in Figure 3.20, 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 R65, 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
```

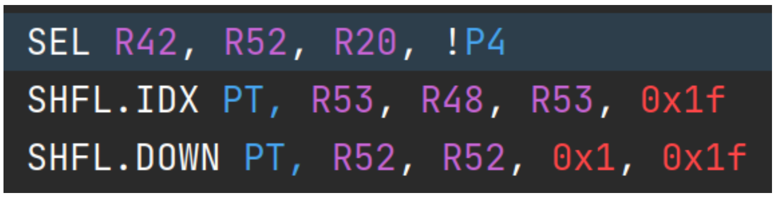
Figure 3.20: The loop body SASS instructions with local memory access.

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

```
#pragma unroll
```

```
for (dsize_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
```

Figure 3.21: The updated loop body SASS instructions with no local memory access.

As we can see in Figure 3.21, 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.

3.3.3 Multiple left matrices

A natural step following the use of multiple right matrices is to also utilize multiple left matrices. This can only be used in the *n-to-m* computation type, as the *n-to-mn* type and its subtypes correlate each left matrix with a different set of right matrices.

The changes to the code of the *multimat_right* optimization are straight forward. 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 shifts computed by each thread is also changed to $NUM_LEFTS * NUM_RIGHTS$ from only NUM_RIGHTS.

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 improvement can be seen in the executed instruction mix shown in Figure 3.22.

3.3.4 Multiple rows from both matrices

When improving *one-to-one* computation using the *multirow_right* optimization described in Section 3.2.4, which processes multiple rows from the same right matrix, we described a further improvement using multiple rows from the left matrix. This not only further improves the ratio of warp shuffle to fused multiply-add instructions, but also reduces the number of times every row from the right matrix is reread.

The initialization and finalization parts are left unchanged. The main difference is that the main loop now advances not by a single left row, but by the number of left rows given by the algorithm argument. We also need to add an additional stage between the main loop and finalization, where we utilize the original single step main loop to finish the leftover rows from the left matrix. This step is required when the number of left rows in the overlap is not divisible by the number of left rows processed in each iteration of the new multi-step main loop.

The algorithm parameters are changed from just the number of right rows to be processed by each thread to a pair of parameters, the number of left rows to process in each iteration of the main loop and the number of shifts to be processed by each thread. In the *multirow_right* algorithm, the number of shifts per thread was equal to the number of right rows to be processed by each thread. For clarity, we now invert this relationship, explicitly providing number of shifts and deriving the number of right rows to be loaded from the number of shifts 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 $[l, NUM\_SHIFTS - 1 + l]$. As such, we load $NUM\_SHIFTS + NUM_LEFT_ROWS - 1$ right rows, so that we can process all the left rows simultaneously.

The ratio of warp shuffles to fused multiply-adds 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 initialization, single step main loop and finalization share the original ratio of the *multirow_right* implementation. This ratio can be seen in Figure 3.23.

3.3.5 Effects of local array optimization

In this section, we compare the *multimat_both* and *multirow_both* algorithms with and without the local array optimization, described in Section 3.3.1. The only difference between the versions is described in Section 3.3.2, with a change to the loop shuffling the buffers containing data from the left matrices.

We can clearly see the additional local memory store (STL) and load (LDL) instructions in the instruction mix in Figures 3.22 and 3.23. Apart from these, we can see that the number of remaining instructions is the same.

When comparing the runtime of each implementation, the difference is even more noticeable. As we can see in both Figure 3.24 and Figure 3.25, the speedup for smaller sizes is limited due to low occupancy, but is still present. Figure 3.24 shows that for matrices of size 64 by 64, the solution without local memory access is already 2 times faster. Figure 3.25 shows that there is slightly smaller improvement in the *multirow_both* optimization compared to the *multimat_both*. This is most likely caused by the initialization and finalization parts of the im-

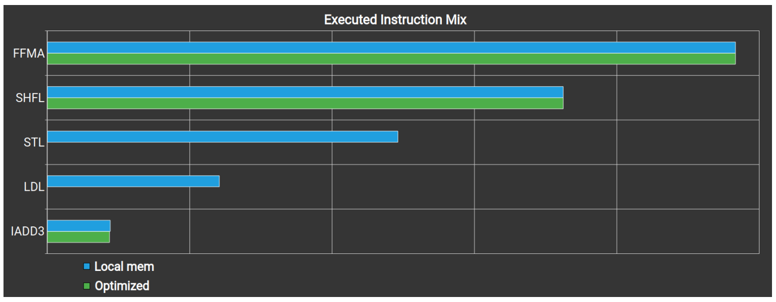


Figure 3.22: Instruction mix of the *multimat_both* optimization with local memory.

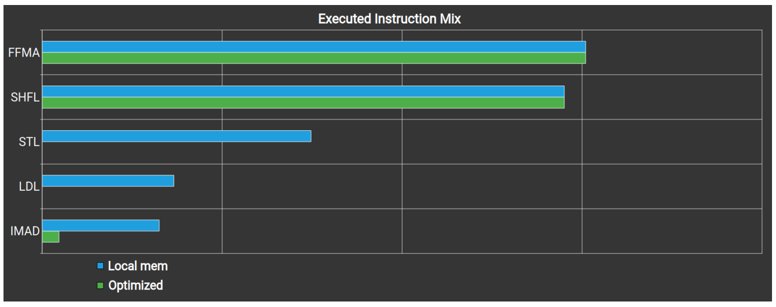


Figure 3.23: Instruction mix of the *multirow_both* optimization with local memory.

plementation, together with greater overall complexity of the optimization.

3.3.6 Combining the optimizations

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

- work distribution,
- multiple right matrices per worker (*multimat_right*),
- multiple rows from each right matrix per worker (*multirow_right*),
- multiple left matrices per worker (*multimat_both*),
- multiple rows from each left matrix per main loop iteration per worker (*multirow_both*).

As described in this section, the versions loading data from multiple left matrices or multiple rows of each left matrix 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_left* can only be used to optimize the *n_to_m* computation.

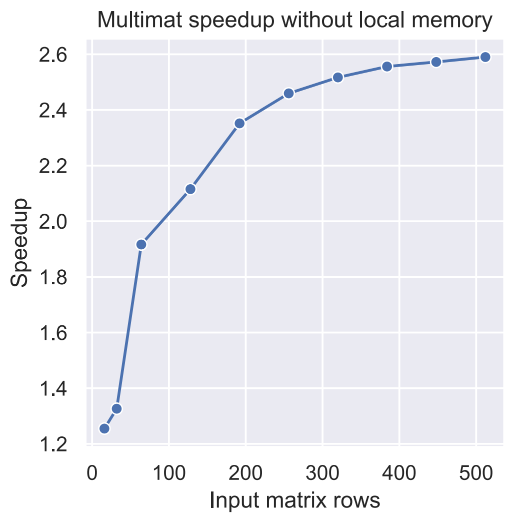


Figure 3.24: Improvement of the *multimat_both* optimization without Local memory access.

With the *multirow* optimization, each worker computes several different shifts of the input matrices, where each shift corresponds to a differently sized overlap. As our work distribution optimization is based on the size of the overlap, the current implementation cannot be reused. This is not a problem with the *multimat* optimizations, in which each thread computes the same shift from multiple matrices.

The *n_to_m* computation type is the only type where multiple left matrices share the same right matrix and as such can be reused in the computation.

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.

3.4 Occupancy improvement

For small inputs, processing a single shift or even just several rows per thread may not utilize enough threads to saturate the whole GPU, leading to low occupancy. As described in Section 2.3.1, low occupancy prevents the GPU from hiding the

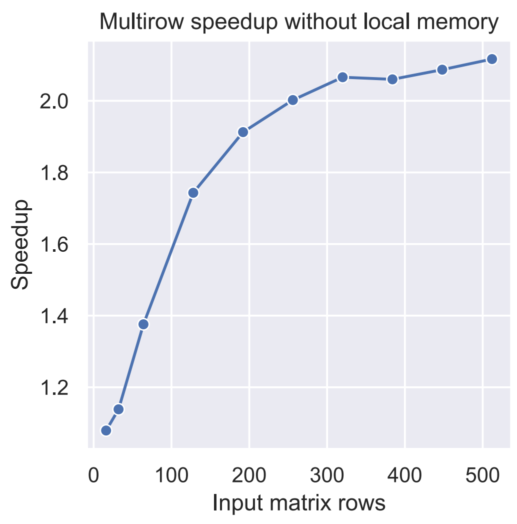


Figure 3.25: Improvement of the *multirow_both* optimization without Local memory access.

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 size of inputs leads to a need of balancing the overhead of each thread, such as bound computations, scheduling etc., with the reduced workload. For each implementation, there exists an input size for which the overhead results in pure CPU based implementation overtaking the GPU implementation. This bound is specific to each system, as it is based on the relative power of the CPU and GPU, together with the interconnection between GPU and CPU for data transfer. This topic is further explored in Section ??.

In this section, we first introduce a simplified implementation of an algorithm utilizing whole warps as workers. We then go through several improvements of this algorithm by using shared memory and further improving occupancy by increasing the number of tasks. Lastly we implement an algorithm utilizing whole thread blocks as workers.

3.4.1 Warp per shift

Basic implementation of the *Warp per shift* algorithm is very simple compared to the warp shuffle-based algorithm described in Section 3.2. Using the CUDA Cooperative Groups API, we determine the distribution of threads into warps and utilize whole warps as workers for this algorithm. For the basic version, we again make each element of the output matrix (which corresponds to a single shift of the two input matrices) a task.

Each thread individually computes the bounds of the submatrix assigned to its warp. The main loop goes through the pairs of overlapping elements in warp sized steps, as can be seen in Figure 3.26. As we can see, the main loop is iterated $warp_size$ less times than if the shift was processed by a single thread. We can also see that this type of indexing minimizes thread divergence but may lead to uncoalesced global memory access. Another possible disadvantage of this implementation is a division and 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 and the final result written by a single thread to the output matrix. We utilize the `cooperative_groups::reduce` function, which may even be hardware accelerated if running on the newest Ampere GPUs, to combine the results of threads in a warp.

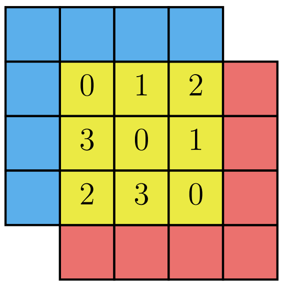


Figure 3.26: Item assignment with basic indexing (with warp size 4).

3.4.2 Simplified indexing

There were two problems highlighted in the previous section, the uncoalesced global memory access and the low throughput division instruction in the main loop. A way to fix these problems is to change the assignment between items and threads. In Figure 3.27, we can see a comparison of the basic indexing, described in the previous section, and the new *simplified* indexing. Each row is processed independently and fully before continuing with the processing of the next row. This assures coalesced access to the global memory, but leads to thread divergence if the row size of the overlapping submatrix is not a multiple of warp size, as is the case in Figure 3.27.

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 the 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 can be seen in Figure 3.28. This figure shows the worst case scenario, where simplified indexing leads to the rows being processed sequentially, whereas basic indexing executes this in a single iteration. Overlaps such as this make up a sizable part of every computation. Because of this, simplified indexing does not improve execution times.

Even though simplified indexing should theoretically lead to better global memory access coalescing, the opposite seems to be the case as can be seen in Figure 3.29. 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% as can be seen in Figure 3.30, which becomes a bottleneck. The profiling was done on input matrices of size

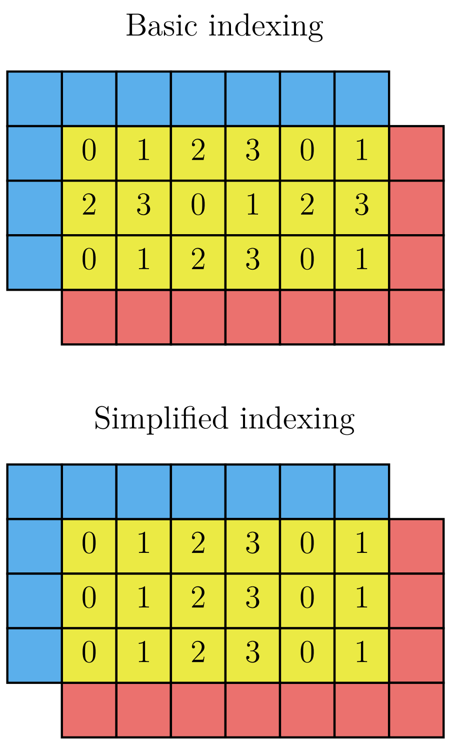


Figure 3.27: Comparison of basic and simplified indexing (with warp size 4).

64 by 64 containing 32bit floating point numbers. In these matrices, each row is 256B in size. In many shifts, the last item of each row will be less than 128B (L1 cache line 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 explains the 47% difference in global memory requests.

Another visible difference in Figure 3.29 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 loops each warp has to go through 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 3.31. 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 represent the useful work done by the GPU.

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

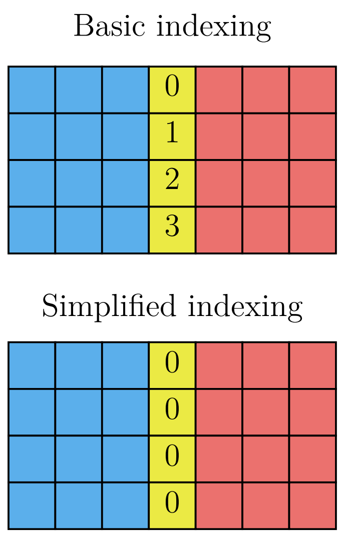


Figure 3.28: Thread divergence with simplified indexing.

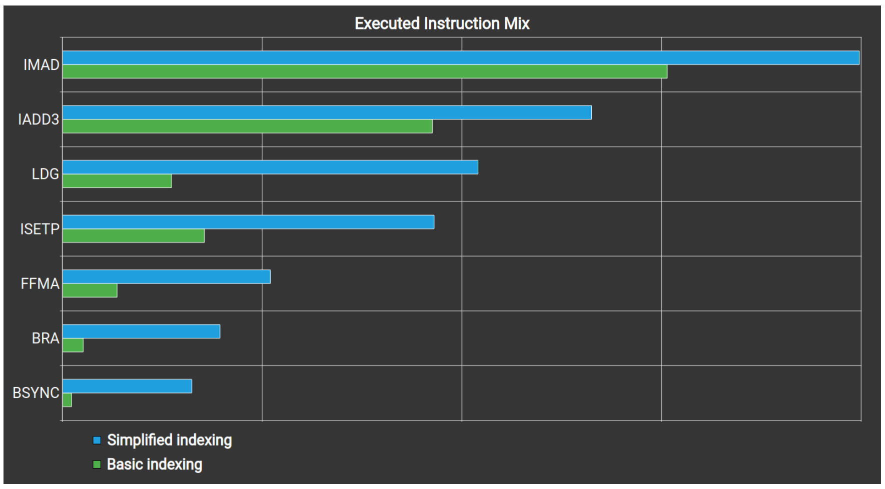


Figure 3.29: Comparison of the instruction mix executed by basic and simplified indexing.

with stalls due to branching. This is consistent with the properties of simple indexing described above.

3.4.3 Shared memory

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

Warps of a given thread block compute consecutive shifts in the y axis, as can be seen in Figure 3.32. This allows us to prevent bank conflicts, which are a major problem when using shared memory. Figure 3.34 illustrates shared memory access if we were to assign shifts along the x axis, with warp size 4 and 4 banks of shared memory. As we can see, this leads to strided access, where warps access blocks of consecutive values with a stride between the blocks. This is guaranteed to

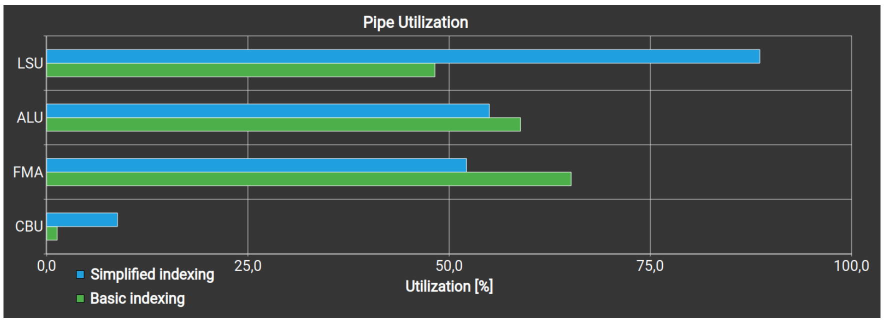


Figure 3.30: Comparison of pipeline utilization of the basic and simplified indexing.

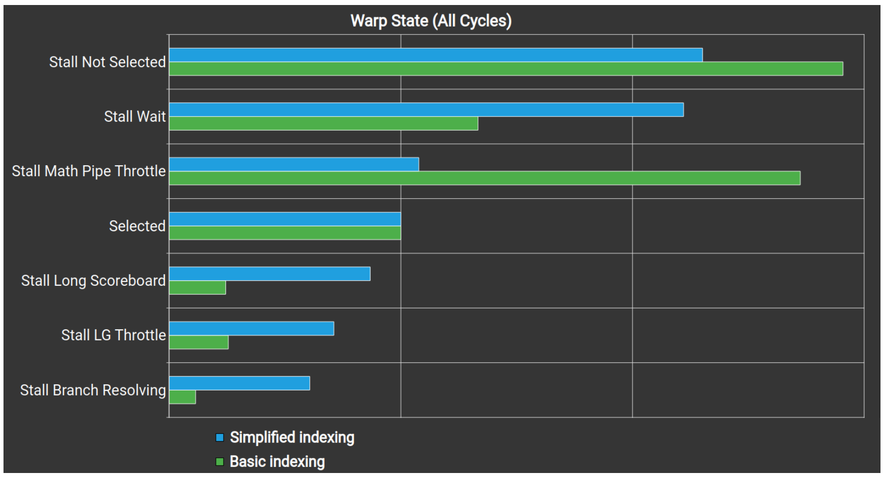


Figure 3.31: Comparison of warp stall reasons between the basic and simplified indexing.

cause up to 32 way bank conflicts, which would severely limit the shared memory throughput.

When assigning shifts along the y axis, all warps of a thread block compute the same shift in the x axis, which corresponds to the same row size of the overlapping submatrices, as can be seen in Figure 3.32. The only difference is the number of rows each overlapping submatrix contains. This leads to different starting and ending offsets for each of the warps, but allows us to utilize perfect linear access to shared memory, leading to optimal throughput. Left input matrix is accessed independently from the right input matrix, so sharing banks between the matrices does not lead to bank conflicts.

When loading data to shared memory, all warps of a block cooperate together to load the union of submatrices of all warps into shared memory. The union is divided into column groups of size `shared_mem_row_size` columns, where `shared_mem_row_size` is a runtime algorithm argument. These column groups are processed sequentially. The column groups can be seen in Figure 3.33. This is also a place for further parallelization similar to work distribution, where each column group can be processed independently and in parallel, which is described in Section 3.4.6.

As in the warp shuffle algorithm, the data from the left matrix is loaded into two buffers which serve as a ring buffer. We first preload the bottom part of

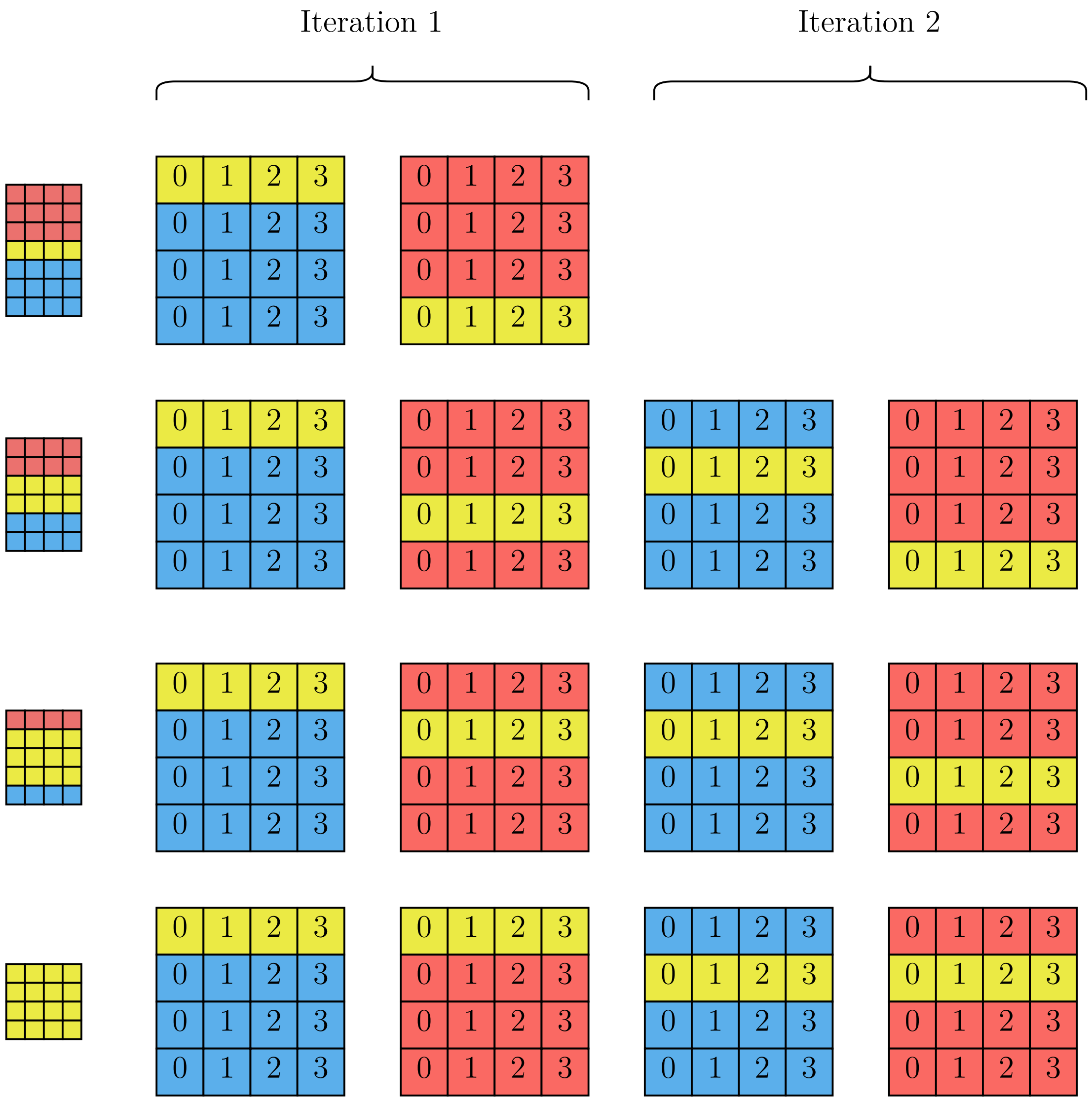


Figure 3.32: First two iterations with shifts assigned to warps along the y axis.

the buffer before the start of the outer loop. We have to limit the range of rows preloaded into the bottom left buffer to those which overlap with the rows loaded into the right buffer in the first iteration. If we were to preload the whole bottom left buffer, as can be seen in Figure 3.33, we would encounter a situation where data loaded into the right buffer in the second iteration requires data preloaded into the bottom left buffer which are already overwritten.

The size of each buffer is $shared\_mem\_row\_size * shared\_mem\_rows$, where $shared\_mem\_rows$ is another runtime algorithm parameter. This parameter has to be greater than or equal to the number of warps (and consequently computed shifts) per block.

The whole algorithm is described by the following pseudocode:

```
FOR each column group
  preload bottom part of left buffer

  FOR rows of overlap with step shared_mem_rows
```

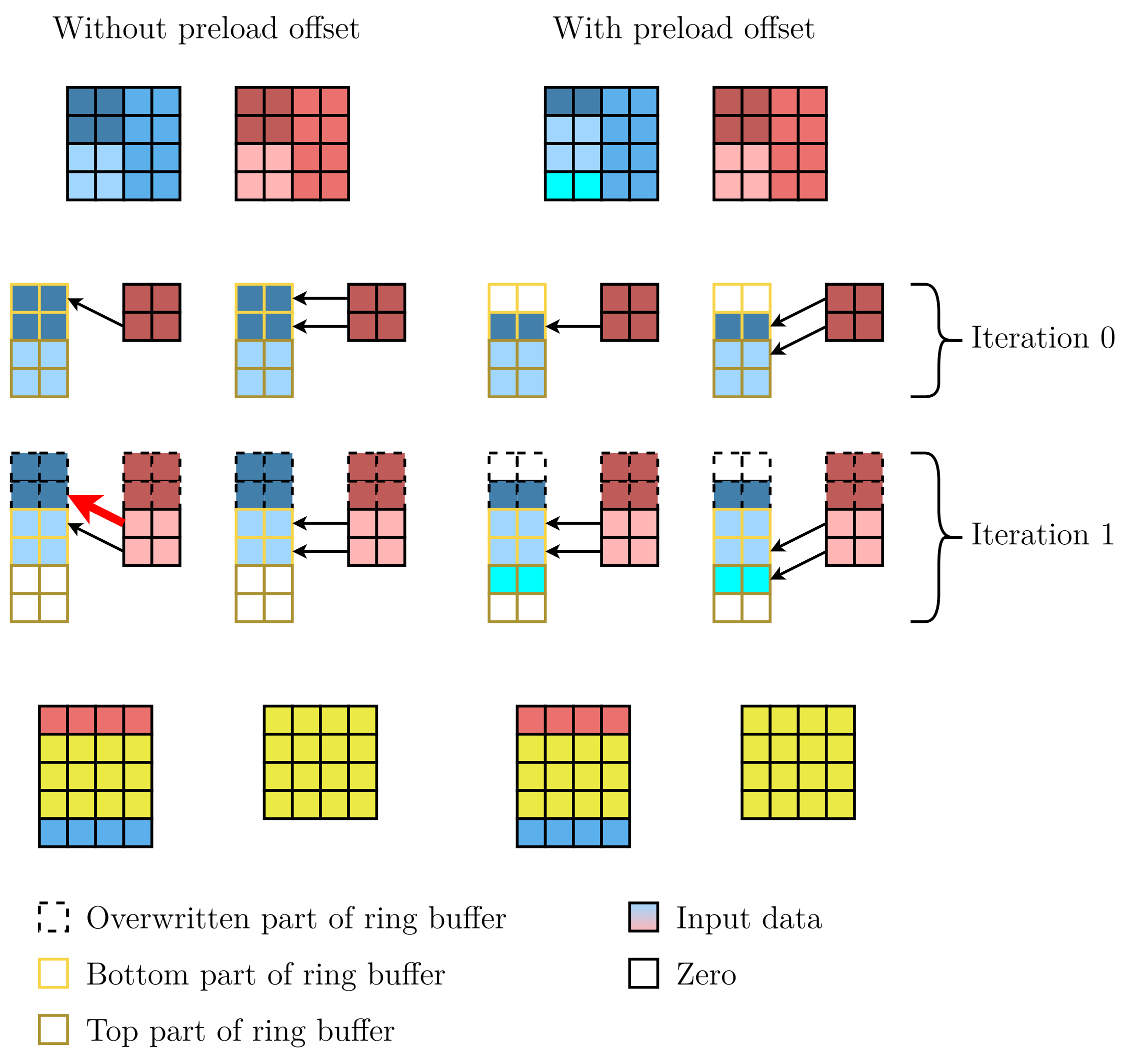


Figure 3.33: Why shared memory preload without offset does not work.

```
    load top part of left buffer
    load right buffer

    sync thread block

    compute bottom left buffer with right buffer
    compute top left buffer with right buffer

    swap top and bottom left buffer
    sync thread block
  END FOR
END FOR
```

Computation of left buffer half with right buffer computes the range of rows loaded into the two buffers which overlap in the shift assigned to the current warp. This gives two continuous ranges of the same size, one in left buffer half and one in the right buffer. Corresponding elements have to be multiplied and the result added to intermediate sum kept by each thread. After we process the whole overlapping submatrix, the intermediate sums in each thread are collected using the `reduce` function provided by Cooperative Groups API and the output is written to the output matrix by the first thread of the warp.

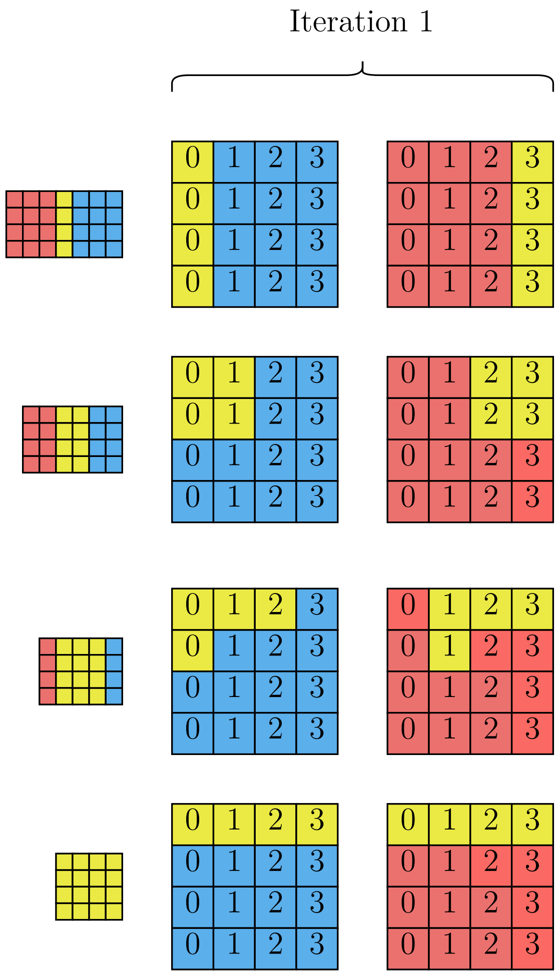


Figure 3.34: Bank conflicts with shifts assigned to warps along the x axis.

Even with the throughput of shared memory, the load from shared memory *LDS* instructions become a bottleneck, as can be seen in Figure 3.35. 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.

3.4.4 Loading data to shared memory

When loading data to shared memory, we have a choice between two different access patterns, shown in Figure 3.36:

- strided warps,
- continuous warps.

With strided warps, warp w in iteration i reads items based on the following formula:

$$[i * block\_size + w * warp\_size, i * block\_size + (w + 1) * warp\_size)$$

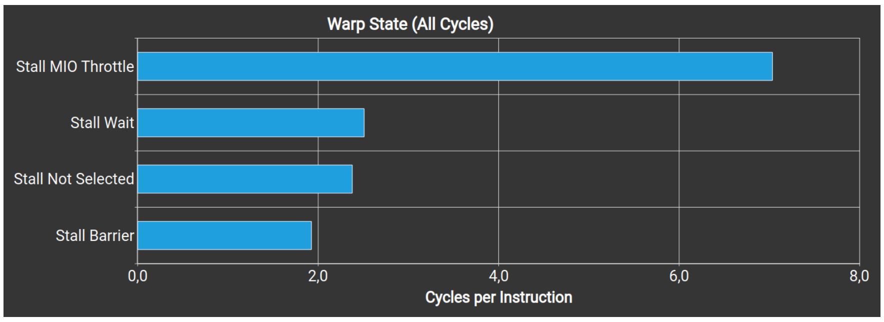


Figure 3.35: Memory input/output (MIO) stall caused by excessive shared memory access.

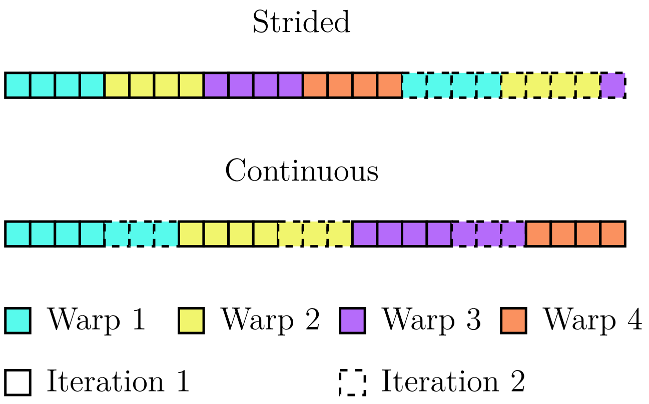


Figure 3.36: Difference between strided and continuous loading patterns.

With continuous warps, the whole block of data which is to be loaded 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 strided loading was used. For our input sizes and data access patterns, strided loading seems to be faster.

3.4.5 Shared memory with multiple right matrices

Similarly to 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. By reusing the data loaded from left matrix with multiple right matrices, we not only increase the ratio of arithmetic instructions, but also reduce the total number of loads from global memory and shared memory due to the reuse of data from the left matrix for computation of shifts of multiple right matrices. The effects of this optimization can be seen in Figures 3.37 and 3.38. 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 7.5 Compute Capability we used for profiling. The higher utilization of *FMA* pipeline hints at better performance, which will be shown in Section ??.

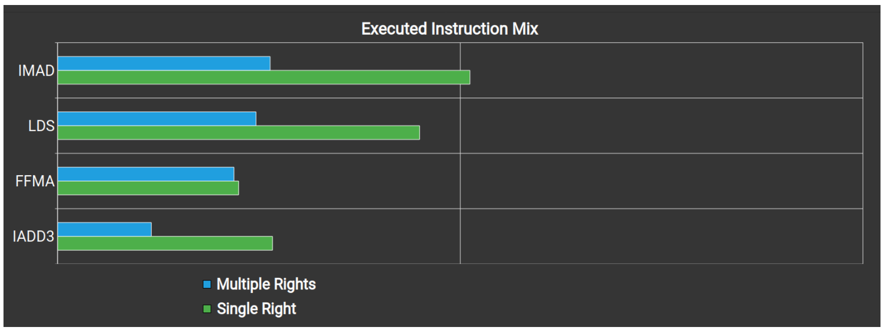


Figure 3.37: The effects of multiple right matrices optimization on executed instructions.

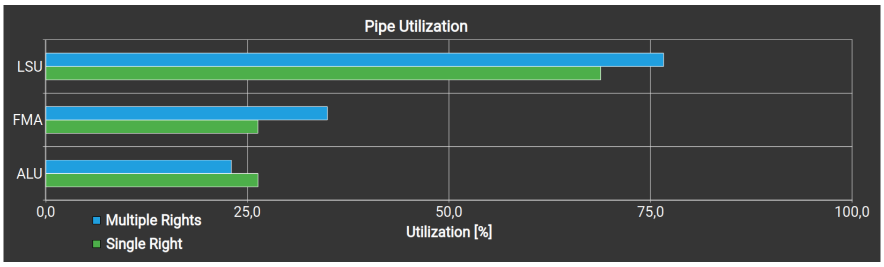


Figure 3.38: The effects of multiple right matrices optimization on pipeline utilization.

3.4.6 Shared memory with single column group per block

As described in Section 3.4.3, each column group processed by the shared memory optimization can be computed independently. This optimization borrows from the rectangle work distribution from Section 3.2.2, 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 shift. Redundant workers are stopped after this computation. The overlap 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.

3.4.7 Work distribution

As described in Section 3.2.2 with the warp shuffle algorithm, there are massive differences between work done by different workers in the basic algorithm. The implementation of this optimization is shared 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 can be seen in Figure 3.39. 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 much improvement. Additionally, without the use of shared memory described in Section 3.4.3, 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 can be seen in Figure **??**.

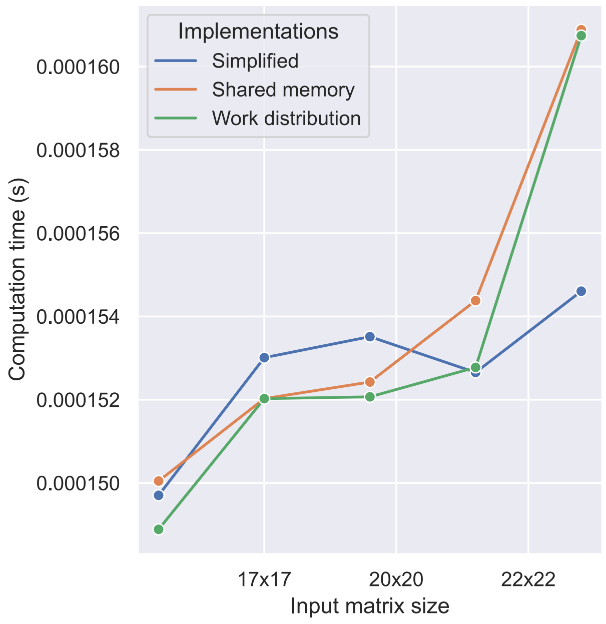


Figure 3.39: Difference between strided and continuous loading patterns.

3.4.8 Block per shift

Another step in increasing 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 block index directly maps to the position in the output matrix and consequently to the shift computed by given block. We then compute the bounds of the overlapping submatrix corresponding with the shift computed by the current block 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 in to the output matrix.

Based on our testing with RTX 2060, warps per shift with work distribution was enough to saturate the GPU even for smallest inputs. For inputs of size 32 by 32 or above, even work distribution was not required to saturate the GPU and the simplicity of the simplified algorithm and starts to take over, as can be seen in Figure 3.40.

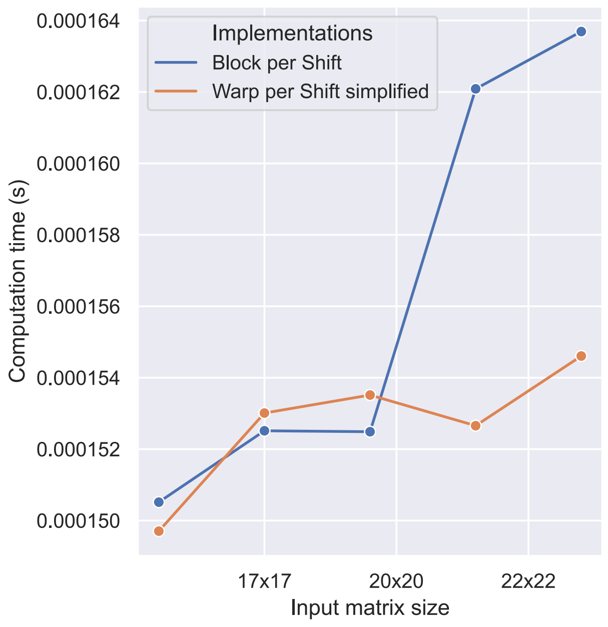


Figure 3.40: Difference between strided and continuous loading patterns.

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

4.1 FFT-based algorithm

Conclusion

Bibliography

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

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

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.

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

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

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.

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

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`.

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`.

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