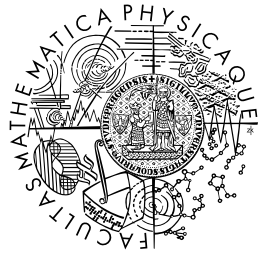

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Karel Maděra

Accelerating cross-correlation with GPUs

Name of the department

Supervisor of the master thesis: Supervisor's Name

Study programme: Computer Science

Study branch: ISS

Prague 2022

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

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

In ..... date ....

Author's signature

Dedication.

Title: Accelerating cross-correlation with GPUs

Author: Karel Maděra

Department: Name of the department

Supervisor: Supervisor's Name, department

Abstract: Abstract.

Keywords: key words

Contents

Introduction

The field of Signal processing is present everywhere in 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.

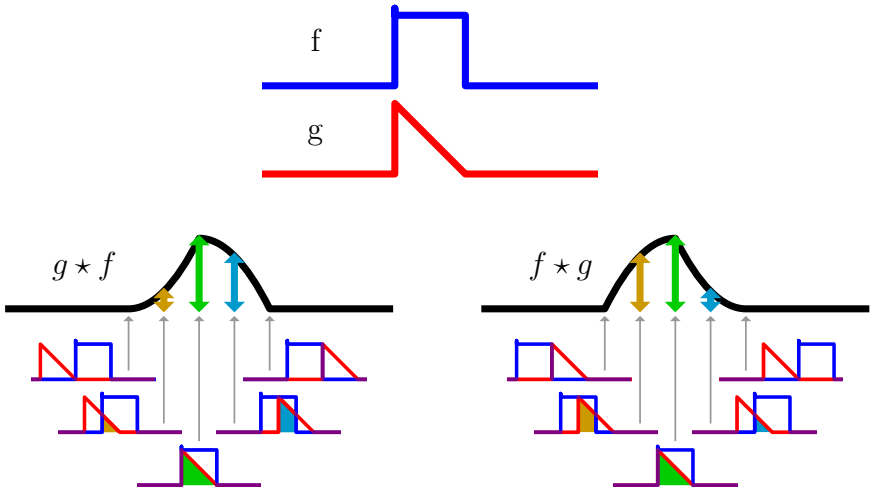

Figure 1.1: Cross-correlation of two functions. Wikimedia Commons contributors [2021]

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.

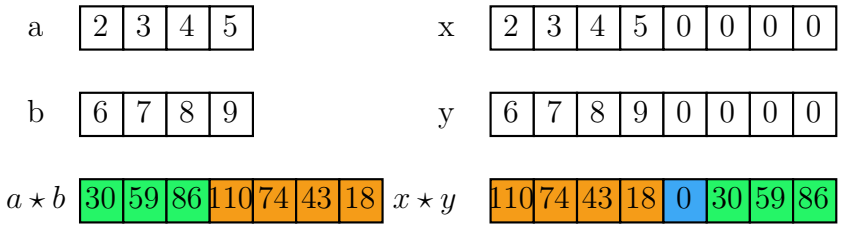

Figure 1.2: Comparison of linear and circular cross-correlation [Bali, 2020].

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.

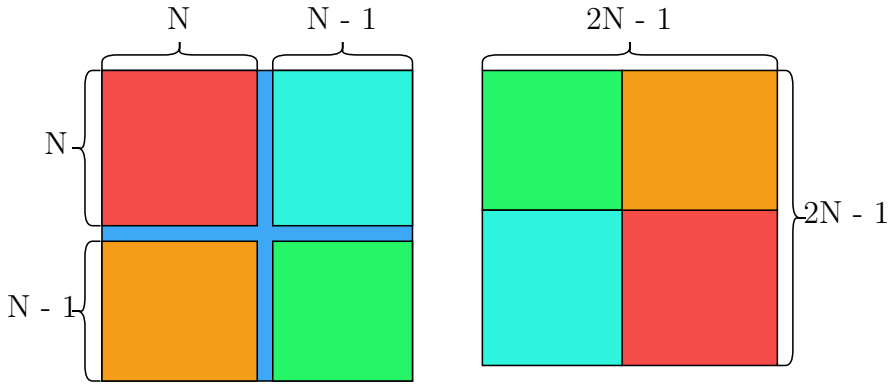

Figure 1.3: Result quadrant swap.

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].

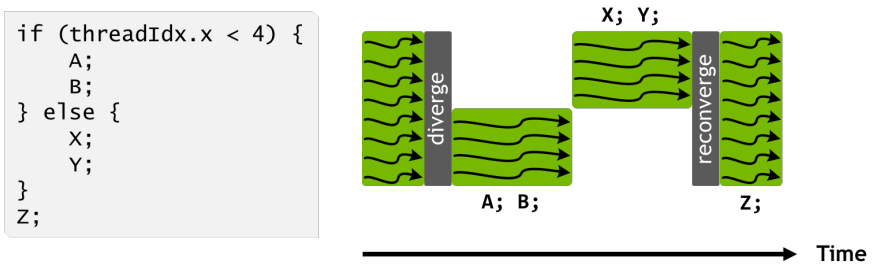

Figure 2.1: Branching in device [Nvidia, 2022].

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.

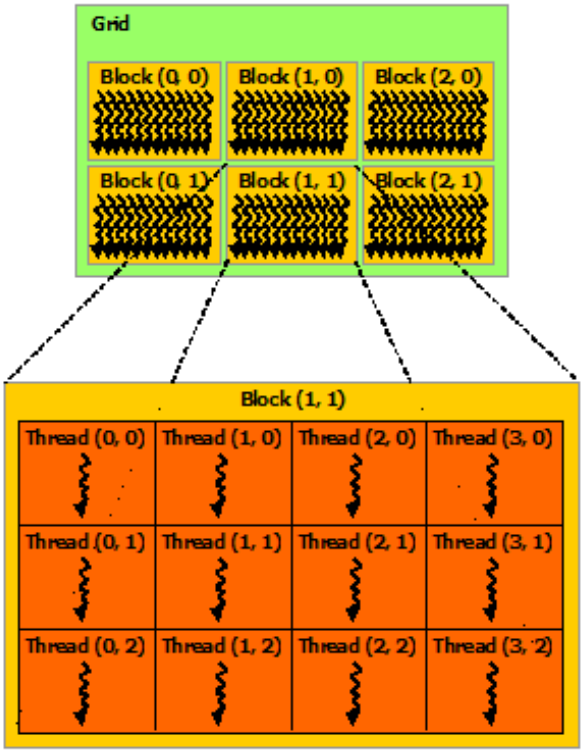

Figure 2.2: Thread grouping hierarchy [Nvidia, 2022].

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.

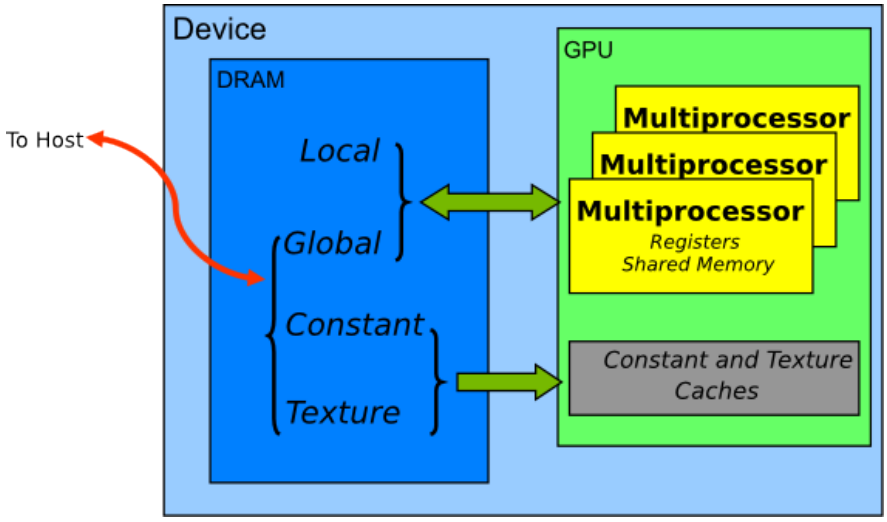

Figure 2.3: Memory types on a CUDA device [Nvidia, 2022].

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.

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.

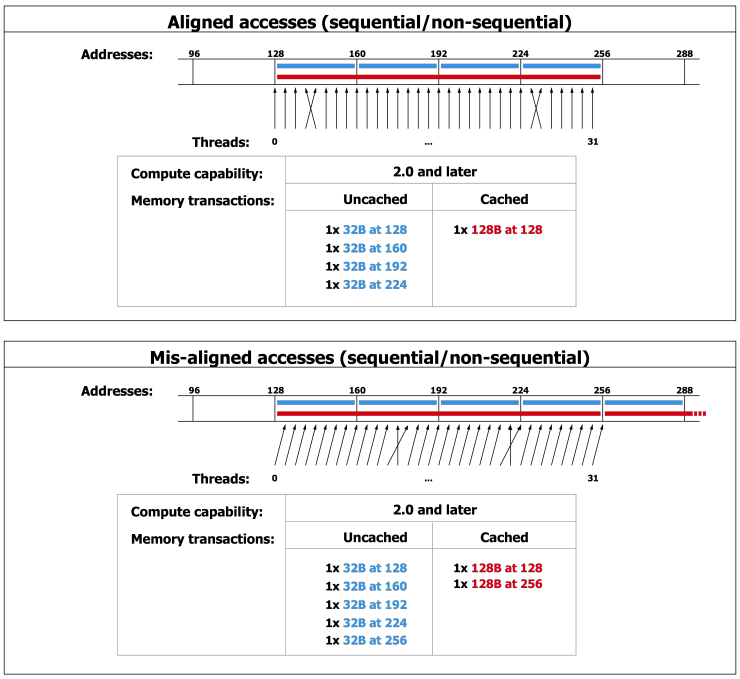

Figure 2.5: Global memory access [Nvidia, 2022].

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.

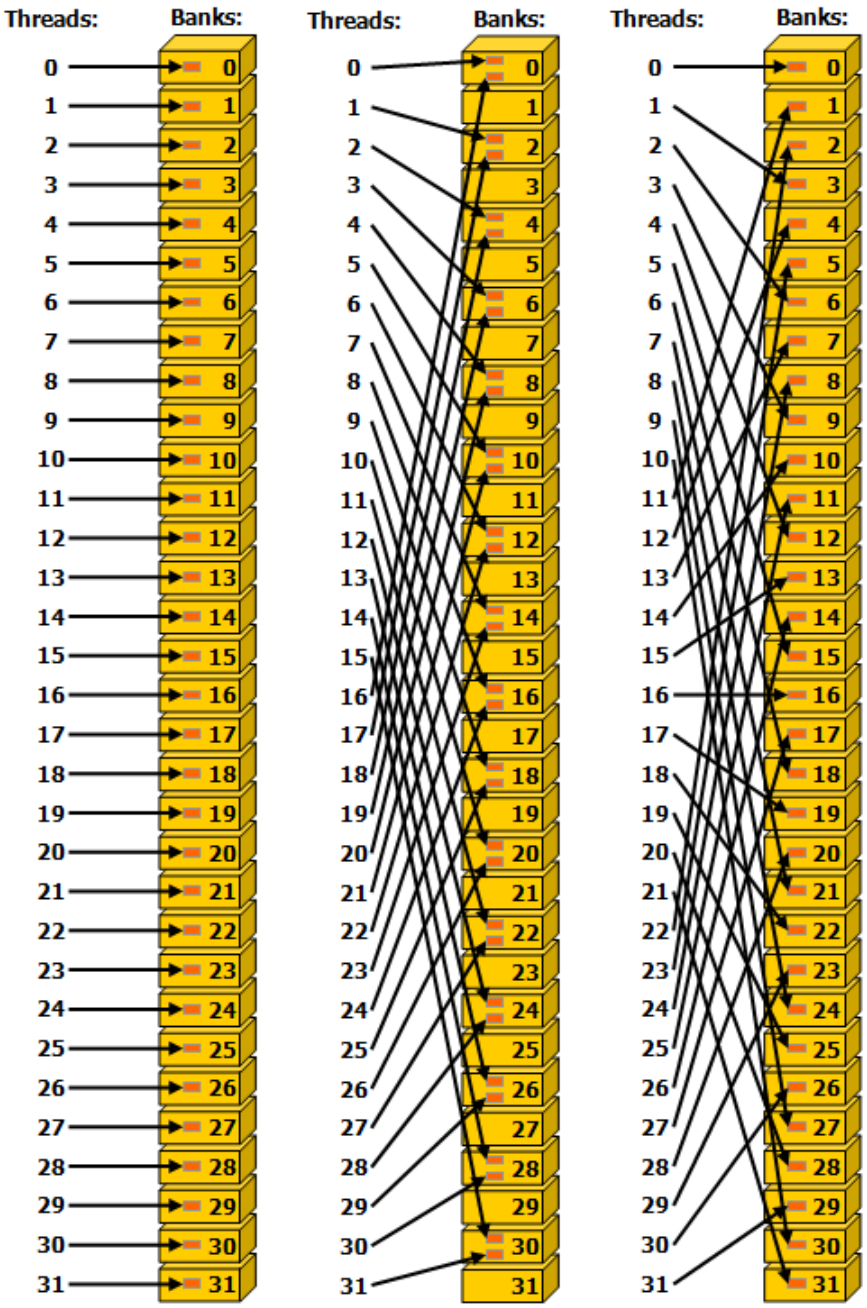

Figure 2.6: Shared memory access patterns [Nvidia, 2022].

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. Next we describe several such implementations.

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 for each element. As described in Section 1.3, each of these elements can be computed independently in parallel.

Each overlap defines a unique set of element pairs which are to be multiplied. Each pair of overlapping elements is uniquely assigned to a single shift of the two matrices.

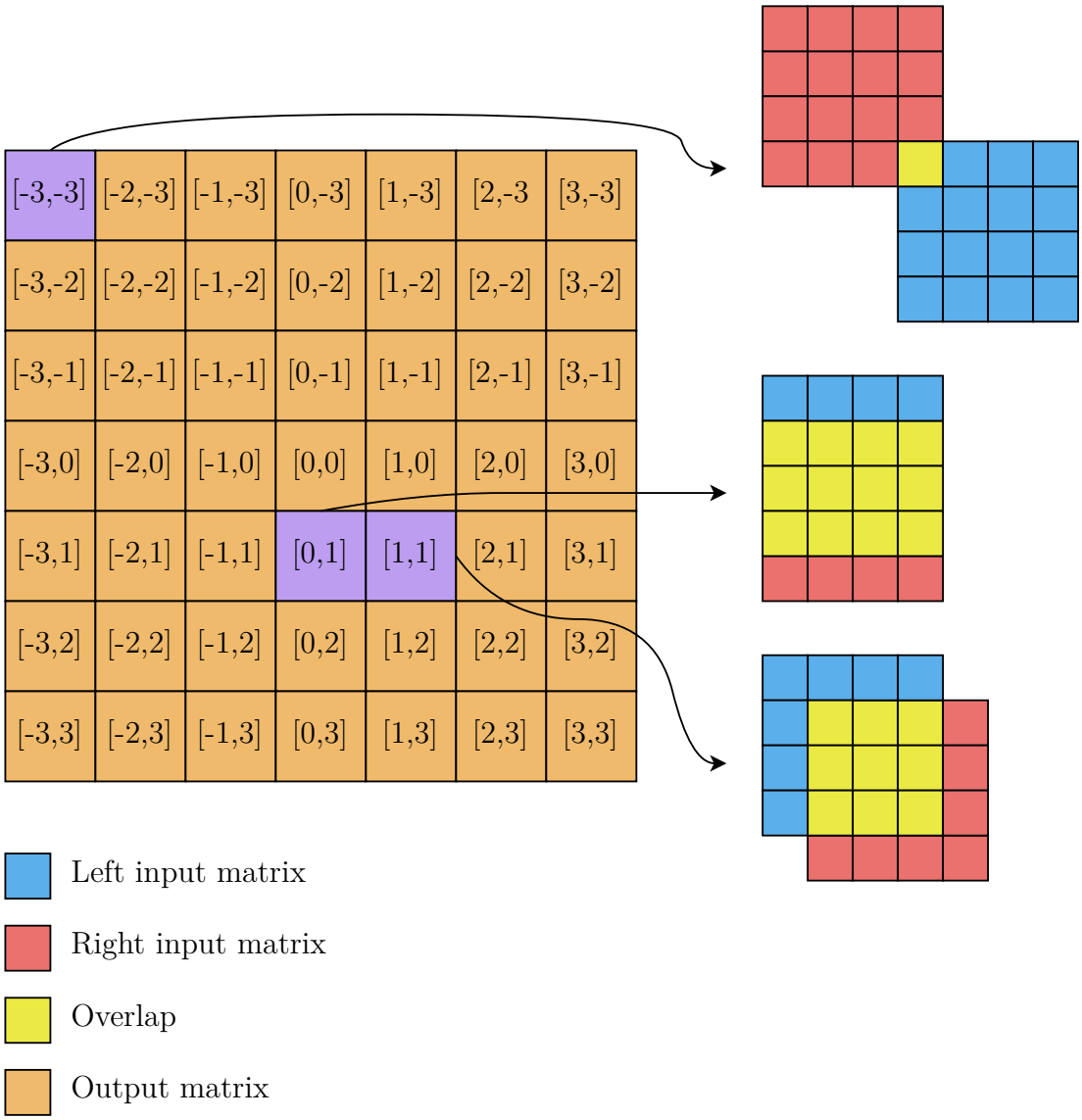

Figure 3.1: Result matrix with corresponding relative shifts.

3.1 Parallelization

In this section, we first reformulate the definition-based cross-correlation algorithm into the language of independent parallel tasks, then we 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 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 set. The results of all subtasks in a set have to be summed into the result of the parent task. The set of subtasks defines a continuous submatrix in both input matrices.

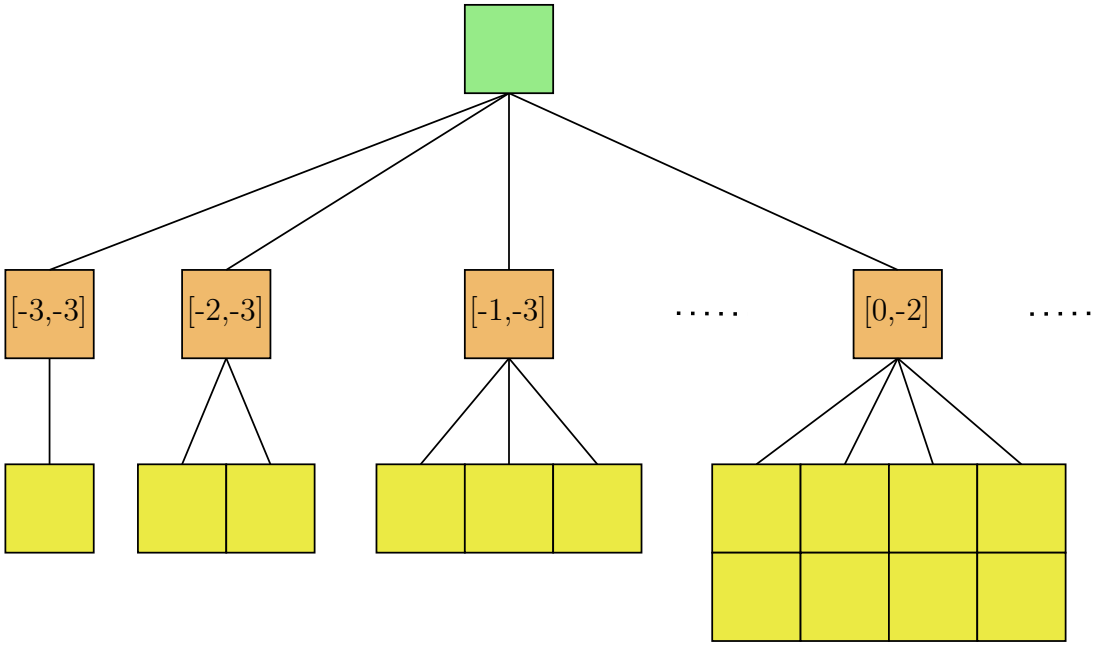

Figure 3.2: Tasks hierarchy in definition-based one-to-one cross-correlation.

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 can be computed. 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 matrix for use in all computations.

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 between these two 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.

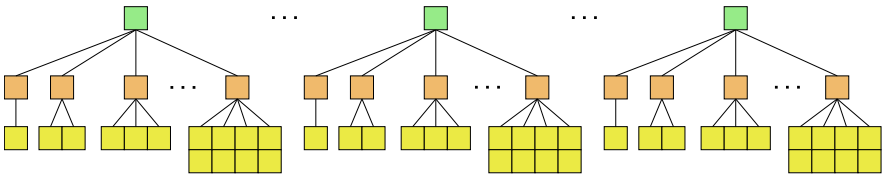

Figure 3.3: Task hierarchy of types with many matrices.

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 result 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 an orange box have to be summed together. Results of the children of a green box have to be written into a single matrix in memory, each into a different element without any collisions.

3.1.3 CUDA workers

Section 2.2.2 described 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 size of CUDA thread group.

Based on the choice of the CUDA thread group size, we can use 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 built on Warp Shuffle instructions. We first introduce a simple version of the implementation, later improving it step by step until we arrive at the algorithm actually implemented in the code accompanying this thesis.

Warp shuffle instruction 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 tasks representing a single relative shift of the two matrices (which is equivalent to a single element in the output matrix), as can be seen in Figure 3.4.

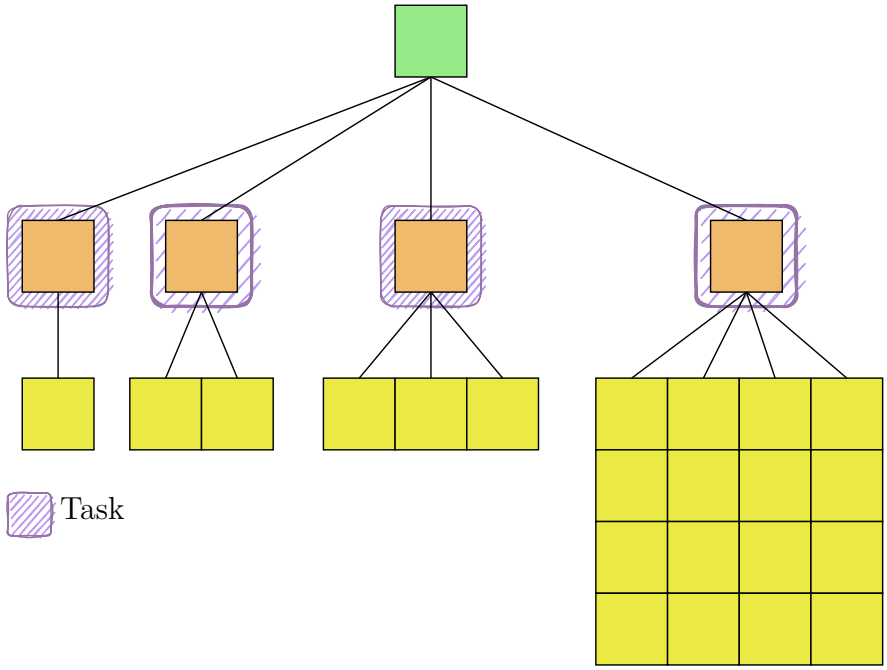

Figure 3.4: Tasks in simple warp shuffle algorithm.

The main idea behind this algorithm is illustrated in Figure 3.5. 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. Figure 3.5 shows how two neighboring threads process elements of the two input matrices. The numbers represent iterations of a *for loop* in the code.

As can be seen, the element from the left matrix processed by thread t in iteration k is required by thread t - *1* in iteration k + *1*. This holds for all threads

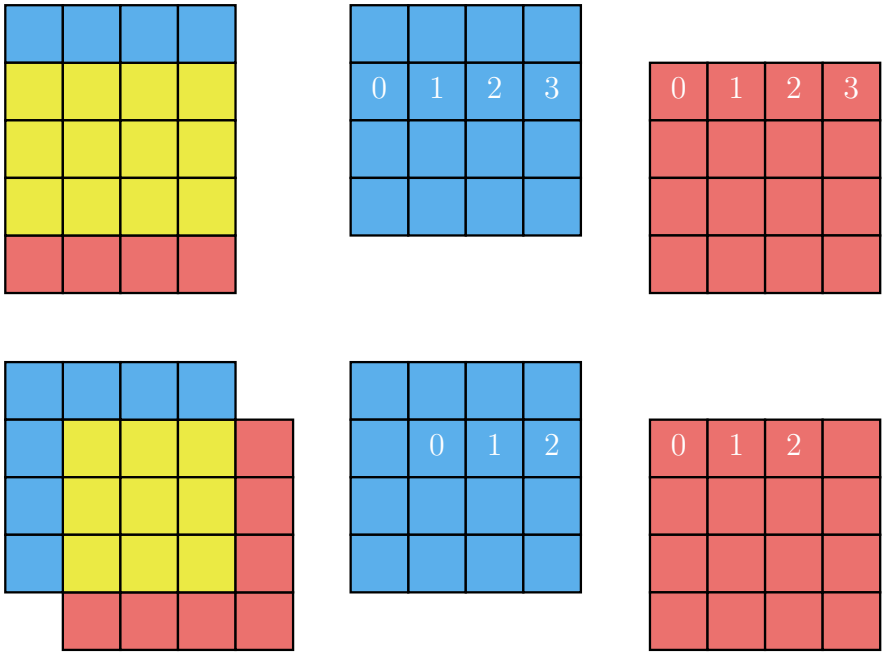

Figure 3.5: Work done by two neighboring threads.

of a warp and maps exactly onto the Warp Shuffle Down function, described in Section 2.2.3.

In any given iteration, all threads of a warp 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.

The problem of iteration *3*, 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. More GPU friendly implementation needs to prevent thread divergence of a warp by executing the range check for each thread 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 0 which is then added to the sum, making it a *noop* and preventing thread divergence.

The final algorithm can be described by the following pseudocode:

```
float sum = 0;
for (
  size_t warp_y_right = warp_right_start.y;
  warp_y_right < warp_right_end.y;
  ++warp_y_right
) {
  size_t warp_y_left = warp_y_right + warp_min_shift.y;


}
```

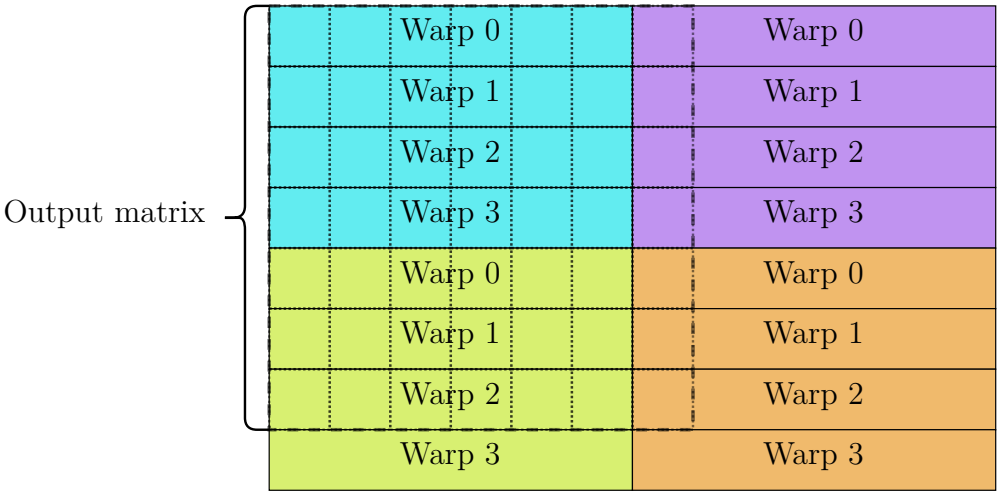

Figure 3.6: Distribution of shifts between CUDA threads, warps and blocks.

3.2.1 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.

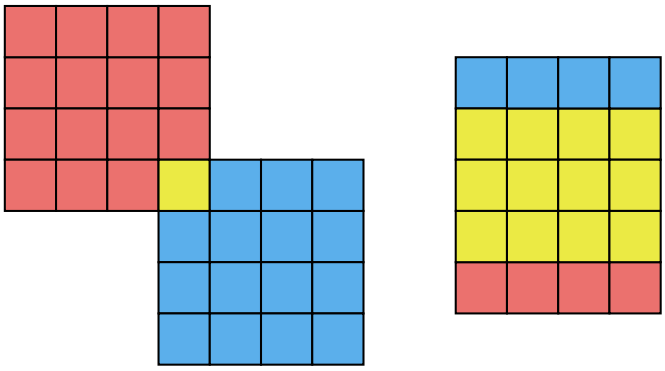

Figure 3.7: Task size difference in the simplified algorithm.

To distribute the work more evenly, we need to change what is considered a task processed by a worker. Compared to the simplified implementation, , 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 tasks 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. The maximum number of rows in a task is provided as an argument to the algorithm, and influences the number of workers created.

We provide several algorithms to derive the number of workers started and the distribution of tasks between workers from the provided argument and the size of the input. The provided algorithms are:

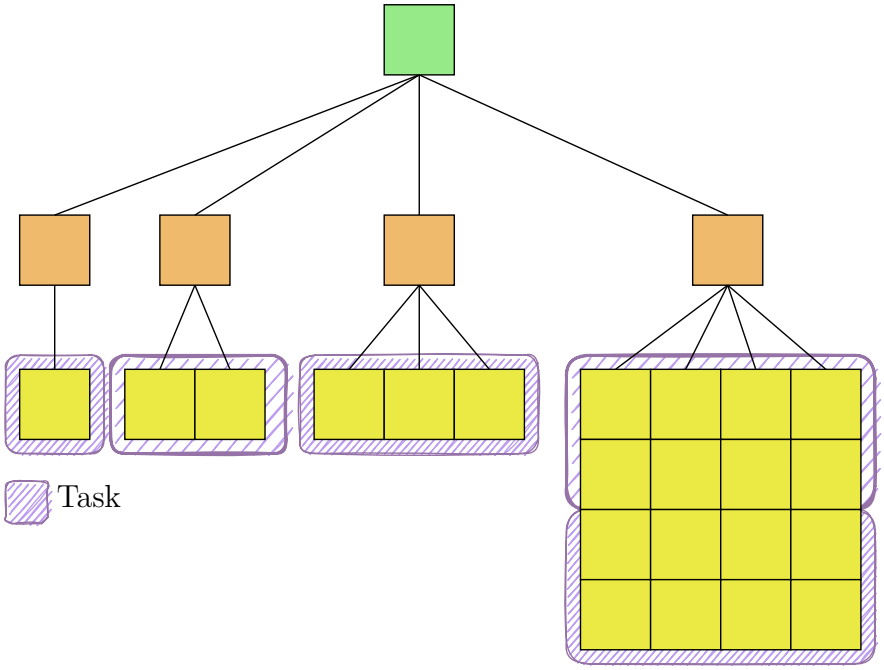

Figure 3.8: Tasks in warp shuffle algorithm with load balancing.

- 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. As we can see in Figure **??**, the worker ID is shared by all threads with the same rank in the x axis. This means that worker ID is assigned based on the y axis rank of each thread and is shared by all threads of a warp. The number

The *None* distribution is provided mainly to measure the overhead of the code changes required to implement work distribution. This distribution behaves identically as the simplified algorithm with no distribution.

The *Rectangle* distribution computes the maximum number of tasks required for any shift, and starts this maximum number of workers for all shifts, creating a rectangle of workers as can be seen in Figure 3.9. The worker API is designed to stop workers which are not required, stopping the redundant workers immediately after work assignment. The tasks are assigned using simple modulo operati

The *Triangle* distribution starts exactly one worker for each task. The disadvantage of this distribution is the complex computation required to assign worker to task. This computation includes many multiplications, divisions and most importantly a low throughput square root instruction. For small inputs where the size of tasks is small, the overhead of triangle distribution may be greater than any gains provided by work distribution.

The total number of workers started for given number of tasks can be seen in Figures 3.9 and 3.10. The first figure shows maximum work distribution, with each worker processing exactly one row of input. The second figure shows the same inputs, but with each worker processing at most 2 rows.

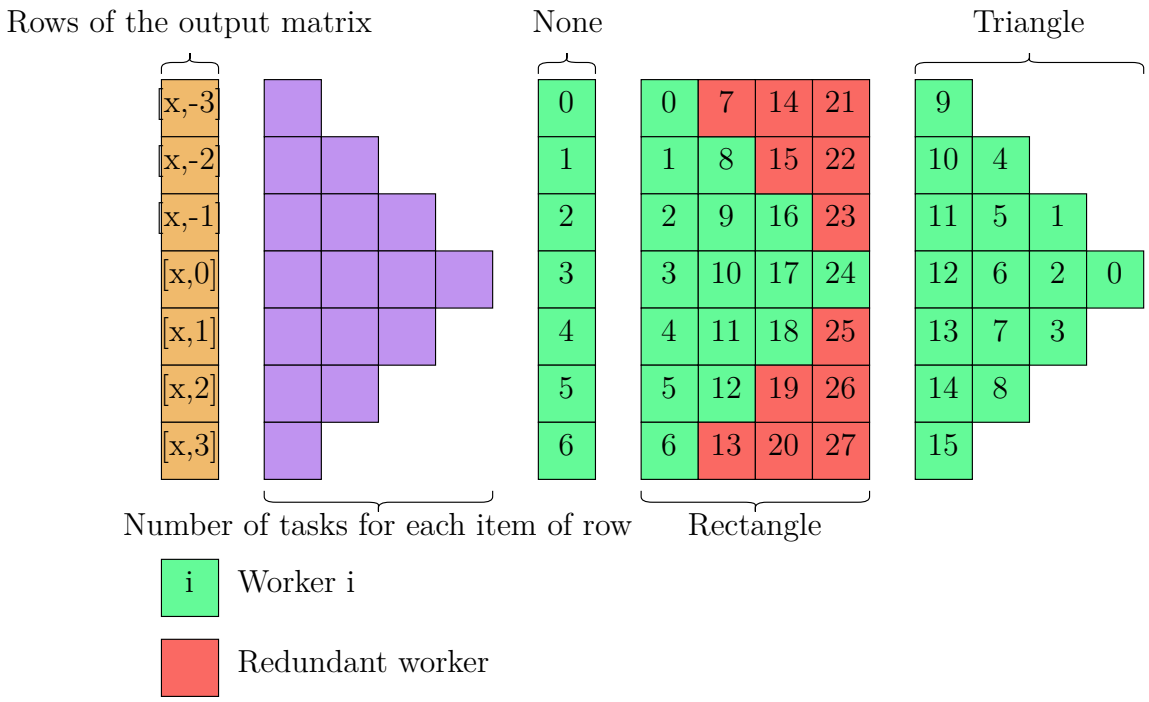

Figure 3.9: Work distribution of 4x4 input with 1 row per task.

This algorithm modifies the number of blocks started for each kernel. In the simplified algorithm, block size is configurable, but is pretty much static. The x axis size of a block is used to simplify grouping threads into warp, and as such has a hardcoded size of 32. The y axis size of a block is used to configure number of warps per block, and is provided as an argument to the algorithm. The number of blocks is then chosen to cover the output matrix in both x and y axes with one thread per output matrix element, with some possible overallocation due to the difference between size of the output matrix and the preset size of the block.

This algorithm retains the same size of a block and use of the x axis of grid size. As all threads with the same value of x axis always process the same rows, we only need to multiply the number of workers by using the y axis of the grid size to start the number of workers required by given work distribution algorithm.

Again, all threads with the same x axis thread rank (combination of the x axis of the thread block and the x axis of the thread itself) have the same `worker_id`, and us such all either run the same rows or end if the worker they represent is redundant. This can be seen in Figure **??**.

3.2.2 Improving ratio of arithmetic instructions

Another problem of the simplified implementation, shared with the work distributing implementation, is the ratio of warp shuffle instructions to arithmetic instructions. For each multiplication of the pair of input values, represented by a single yellow box, and addition of this result to the total sum, we must execute three warp shuffle instructions. This makes warp shuffle instructions the bottleneck in these implementations, as can be seen in Figure **??**. We can also see that the multiplication and addition are implemented as fused multiply-add (FMA) instruction.

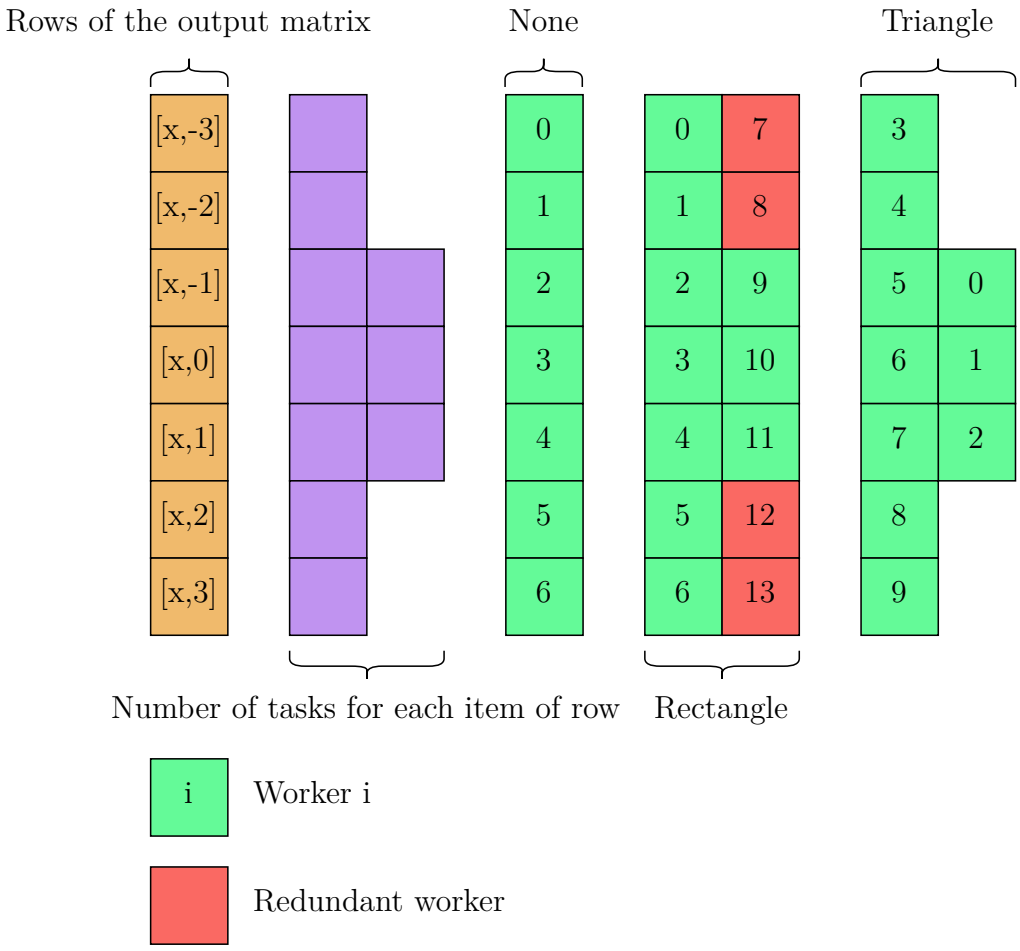

Figure 3.10: Work distribution of 4x4 input with maximum of 2 rows per task.

There are several ways to improve the ratio of warp shuffle instructions to arithmetic instructions. The simplest one is to utilize the *one-to-many* type of computation, and let single worker compute cross-correlation between one left matrix and many right matrices at once. This first allows data reuse, as the data from the left matrix is loaded only once to be used to compute multiple results. The second advantage is that we only need a single additional warp shuffle for each additional right matrix, which also adds a single multiplication and addition. The ratio of warp shuffles to fused multiply-add operations can then be expressed as $2 + r : r$, which is much improved from the $3 : 1$ ratio of the simplified warp shuffle algorithm.

The effects of this optimization can be seen when we compare Figure **??** with Figure **??**.

This optimization heavily relies on

More complex version of this optimization is to process multiple rows from the same right matrix, which can be used to improve the ration even for *one-to-one* type of computation. 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. These shifts differ in the y axis, and represent consecutive elements in a column of the output matrix. Each of these shifts represents different overlap of the two input matrices,

requiring different bounds in the y axis.

Due to these different bounds for the shifts computed by each thread, we have to add explicit initialization and finalization code which handles the case where only some of the rows of the right matrix overlap with given row from the left matrix.

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 to with maximum number of shifts per thread times the number of threads per block, which is used to accumulate the final result of the computation of all shifts for all threads of the thread block.

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 through registers and the reversal of the results for given row group

3.2.3 Problems with local memory

The optimizations described in the previous section 3.2.2 hint at further possibilities, such as using multiple left matrices and multiple rows from the left matrix in combination with the previous optimizations. The problem we encountered is that these additional changes lead to much more complex indexing which the *nvcc* compiler cannot expand into static compile time indexing, even though logically we can see it is.

This leads to compiler pulling the `thread_right`, `thread_left_top`, `thread_left_bottom` and `sum` to be allocated from local memory instead of from register file. The change from local memory results in a substantial overhead, which can be seen in Figure **??**.

3.3 Occupancy improvement

For small inputs, processing a single shift or even just several rows per thread may not start enough threads and lead to low occupancy.

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