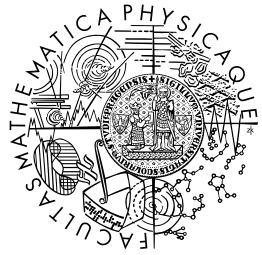

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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. For many of these applications, the computational complexity 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 to utilize the high throughput and massive amounts of computational power provided by massively parallel systems in the form of GPUs, or Graphical processing units.

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 of 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 its variants. We first define one-dimensional cross-correlation, extending it into multiple dimensions and introducing periodic and circular cross-correlation. We then describe how these variations are used to compute cross-correlation using Fast Fourier transform. Lastly we describe the possibilities for optimization and parallelization of cross-correlation, with real-world usage examples where these optimizations could 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.

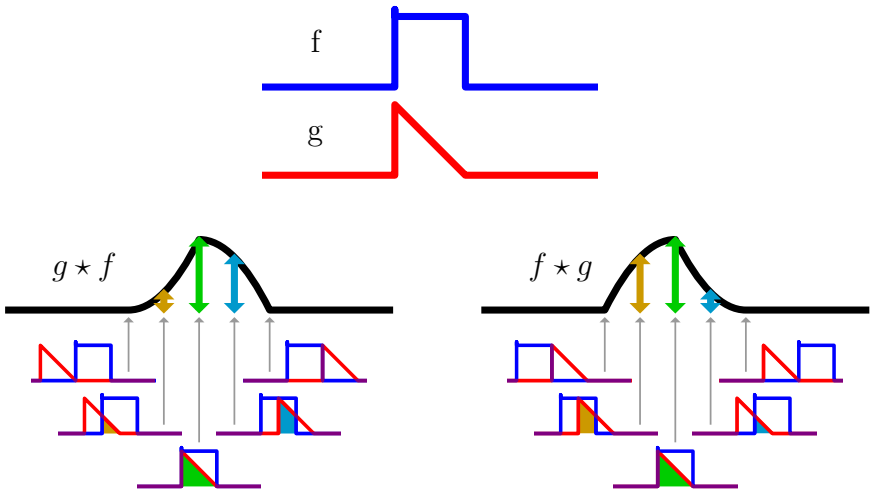

Figure 1.1: Cross-correlation of two functions. Wikimedia Commons contributors [2021]

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 when the two images overlap, which limits the computation to $(w_1 + w_2 - 1) * (h_1 + h_2 - 1)$, 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 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.

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.

For two series $\{x_n\}$ and $\{y_n\}$, we would thus first compute their discrete Fourier transforms $\{X_n\}$ and $\{Y_n\}$. Then we would compute the complex conjugate $\overline{X}$ of $\{X_n\}$, multiply the corresponding elements of $\overline{X}$ and $\{Y_n\}$ and finally

compute the inverse Fourier transform of the result. The time complexity of this algorithm is $\mathcal{O}(N \log(N))$ for the Fourier transform, $\mathcal{O}(N)$ for complex conjugate, $\mathcal{O}(N)$ for element-wise multiplication and $\mathcal{O}(N \log(N))$ for inverse Fourier transform. This gives us the resulting resulting asymptotic time complexity of $\mathcal{O}(N \log(N))$.

Circular cross-correlation can be imagined as a cross-correlation of two periodic discrete functions. When computing circular cross-correlation by definition, for each shift the last element shifted off the series end is taken from the back and moved to the front. This is the circular shift done by the modulo operator. If we assume infinite periodic functions, circular shift is equivalent to linear shift, as the first element is first in each period. If we then compute the sum from linear cross-correlation only over a single period, we can see that the result is equivalent to circular cross-correlation of the two functions.

We call this the periodic cross-correlation, for finite series $n \in \mathbb{N}\{x\}_n = x_0, x_1, ..., x_{n-1}, \{y_n\} = y_0, y_1, ..., y_{n-1} \in \mathbb{C}^N$ defined as:

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

To calculate linear (non-circular) cross-correlation of finite series using circular cross-correlation, we observe the following equality.

Compared to non-periodic linear cross-correlation, periodic linear cross-correlation is summed only over the length of the period, and is thus equivalent to a linear cross-correlation of one period zero extended to infinity.

If we observe the corresponding linear shift in periodic linear cross-correlation, we can observe on Figure **??** that the multiplied values for each shift are the same in both computations. Thus the results are the same for both when computing linear cross-correlation only over the length of the period.

Thus if we zero extend the series to twice its length and compute the circular cross-correlation of this series, the result is the same as linear cross-correlation of the original series zero extended to infinity.

The same equivalence holds for two-dimensional linear and circular cross-correlations as well.

Result is a matrix $z \in \mathbb{R}^2$ of size $2w * 2h$ with the result of cross-correlation in the bottom right quadrant.

1.3 Parallelization and 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 matrix in another image. This can be done to for example find a displacement of a certain point of interest between images of one item 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 is more general and can be used in works mentioned above. 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 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 different inputs (n to m) Clark et al. [2011].

There are several subtypes which can also be optimized for. For n left inputs with disjoint sets of m right inputs for each of the 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 the 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.

1.3.2 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 summed to compute the intensity of each pixel [Kapinchev et al., 2015].

Any post-processing is outside of 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 GPUs 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

CPUs are optimized to process a single stream of instructions working on a single stream of data as fast as possible. This requires CPU design to minimize instruction latency, which is achieved by using branch predictions, multiple levels of caching, and other such mechanisms. On the other hand, GPUs are optimized for throughput of a single stream of instructions working on multiple 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 assigning each thread separate registers in the register file.

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 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 (or a single GPU slice since Ampere).

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

both for host and device code. The available *function execution space specifiers* are:

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

Without any specifiers, function is compiled as part of the host code. **Kernel** is a function with the `__global__` specifier, which is callable from the host code but is executed on a device. Kernels serve as entry points which the host code can use 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, potentially in parallel. 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 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. [1] 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 the number of elements processed by a single instruction is directly exposed in

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

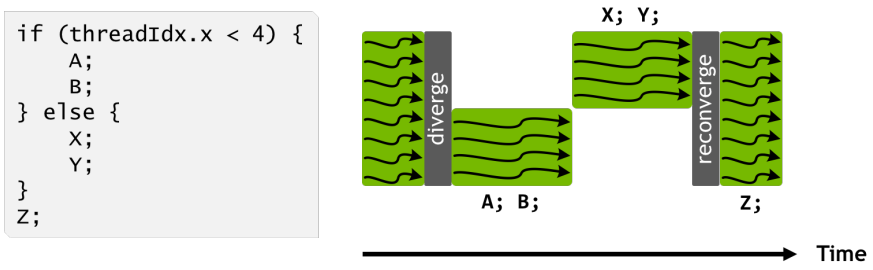

Figure 2.1: Branching in device code.

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 for work distribution 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.

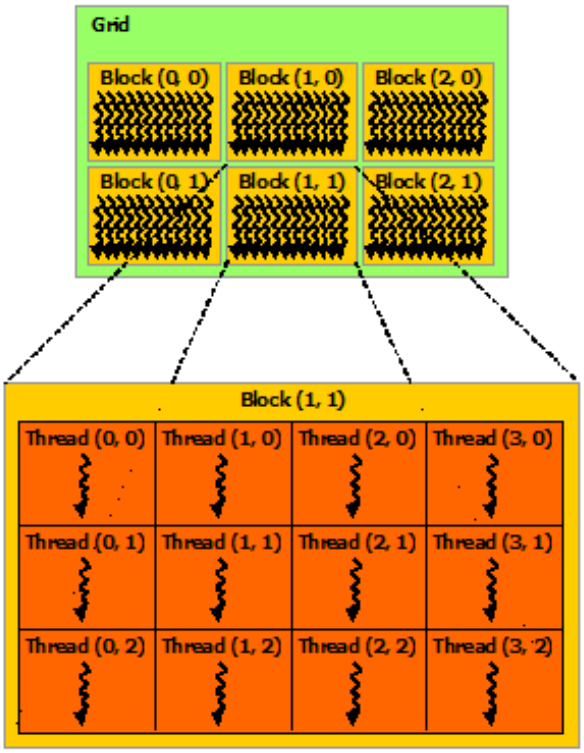

Figure 2.2: Thread grouping hierarchy.

Each thread is assigned an index, accessible through `threadIdx` built-in vari-

able. 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.

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.

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 the subsection 2.2.4.

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.

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 any power of 2.

Only direct lane indexing performs lane index wrap around. If the given lane index is out of range $[0 : width - 1]$, the actual lane index is computed as $srcLane \bmod width$. 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 a 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 (representing multiple grids each running on a separate device).

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 for the creation of handles for 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 other 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].

2.2.5 Memory hierarchy

Each CUDA device has its own DRAM memory, so called *device memory* or *VRAM* in GPU properties, 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 has to be transferred from or to the *device memory* over the PCI-e bus, either explicitly by calls to *cudaMemcpy* 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 on Figure 2.3. For this thesis, the main types are:

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

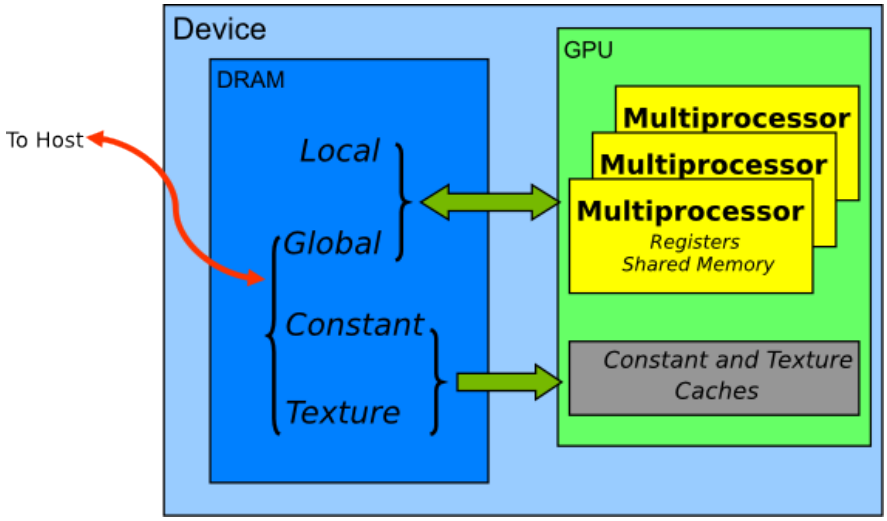

Figure 2.3: Memory types on a CUDA device.

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 the 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 reside in. 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. 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.

Figure 2.4: Streaming multiprocessor.

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 and other features. 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.

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 (program counters, registers, etc.) switching, the whole execution context for all warps is kept on-chip 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 only 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

As we can see in Figure 2.5, the access to global memory is grouped into 128 B naturally aligned chunks, where any chunk accessed by any of the threads of the warp has to be transferred from slow global memory. The maximum performance

is achieved when access to memory is aligned and coalesced, i.e. all threads of a warp access 32 consecutive 32bit elements of an array which are aligned to 128 B. Any other form of access introduces overhead in a form of unnecessary data being transferred from memory.

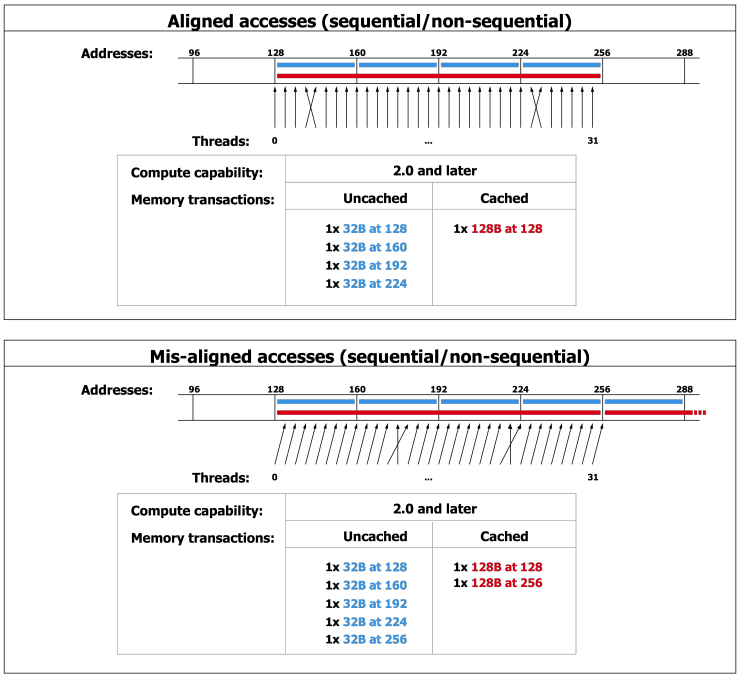

Figure 2.5: Global memory access [Nvidia, 2022].

2.3.4 Shared memory access

To achieve high bandwidth, shared memory is divided into 32 banks. The optimal access pattern shared memory is designed for is that each thread of a warp accesses 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 successive 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 being divided by the maximum number of different addresses accessed in any of the banks. This is called a *bank conflict*. Access to a the same address by multiple threads does not lead to a bank conflict, broadcasting the value between the threads instead.

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 as many as possible 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.

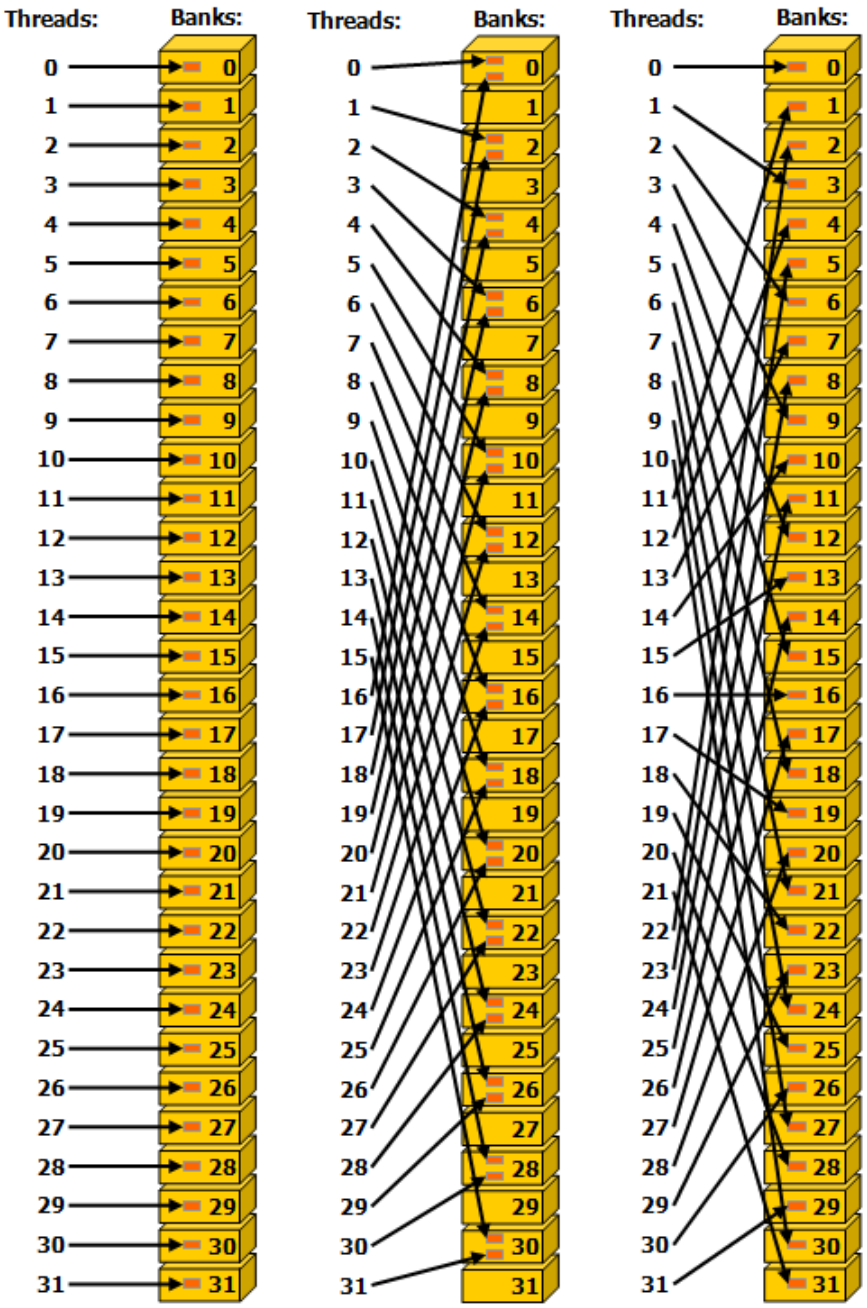

Figure 2.6: Shared memory access patterns [Nvidia, 2022].

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