Università Politecnica delle Marche

Embedded Systems

Implementazione dell'algoritmo di One-Sided Jacobi Rotation per la decomposizione SVD, tramite librerie NVIDIA CUDA C (codice C) su piattaforma embedded Jetson TK1 (GPU).

Matteo Orlandini
Jacopo Pagliuca

supervised by
Dott. Laura Falaschetti and Prof. Claudio Turchetti

April 15, 2020

Contents

1 Introduction

The following paper describes the work done for the design of the One-sided Jacobi algorithm for the Singular Value Decomposition (SVD) of a matrix, via NVIDIA CUDA C libraries. The code has been tested and designed for the embedded Jetson TK1 platform.

The Jetson TK1 is an embedded GPU made by NVIDIA that contains a Tegra K1 SoC in the T124 variant. It also has an Ubuntu Linux operating system and has installed the CUDA 6.5. The GPU's compute capability, that is the computing capacity that determines the general specifications and available functionality, is 3.2.

A C code for SVD optimized for CPU work had already been developed in University. Our job was to compare our results with those previously obtained and evaluate the possibility of an alternative implementation that would fully exploit the potential of the GPU parallel calculation.

As a basis for our work we studied the CUDA programming language and architecture in [3] and [6]. The papers [1], [2], [4] and [5] describing various approaches to the optimization of the GPU algorithm were then analysed and our version shows results that are very similar to those obtained in the papers.

We used the SSH protocol to work on the board from the University department, so we can connect to the board remotely. At the end of the project, however, it was not possible to access the department and then report the exact results of the computational time required by the algorithm. We have therefore used as a reference an NVIDIA GTX 610M, whose results are proportional to those obtained with the Jetson.

1.1 SVD

In linear algebra, the singular value decomposition (SVD) is a factorization of a matrix into three different matrices based on the use of eigenvalues and eigenvectors.

The decomposition of a matrix $\mathbf{A}$ is based on the fundamental theorem of linear algebra:

Given a matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ of rank ρ, the singular value decomposition of $\mathbf{A}$ is represented by the product of two unitary matrices $\mathbf{U} \in \mathbb{C}^{m \times m}$, $\mathbf{V} \in \mathbb{C}^{n \times n}$ and a diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$, as shown in (1).

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{1}$$

where $\mathbf{V}^*$ is the conjugate transpose of unitary matrix $\mathbf{V}$. The columns of $\mathbf{U}$ are called *left singular vectors* di $\mathbf{A}$ and those of $\mathbf{V}$ are the *right singular*

vectors of $\mathbf{A}$. In addition, $\mathbf{\Sigma}$ is a non-negative real matrix of type

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \sigma_\rho & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The diagonal elements of $\mathbf{\Sigma}$ are the *singular values* of $\mathbf{A}$ and are usually in descending order: $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_\rho > 0$, $\sigma_{\rho+1} = ... = \sigma_n = 0$.

In case $\mathbf{A} \in \mathbb{R}^{m \times n}$ the equation (1) becomes:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{2}$$

where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ are orthonormal matrices.

$$\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = I^{m \times m} \tag{3}$$

$$\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = I^{n \times n} \tag{4}$$

By joining these two properties to equation (2), the expression of $\mathbf{A}\mathbf{A}^T$ can be derived:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$$

Likewise, per $\mathbf{A}^T\mathbf{A}$:

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T$$

SVD has numerous applications in the linear algebra field. First of all, it provides important information about $\mathbf{A}$ matrix, such as its rank, its kernel and its image. It is used to define the pseudo-inverse of a rectangular matrix useful for solving the problem of least squares. It is also used in system resolution of homogeneous linear equations.

Another important application is the approximation of $\mathbf{A}$ matrix, with a lower rank (truncated SVD), used in image processing and signal processing.

SVD also has well-known applications in the field of core component analysis.

1.2 One Sided Jacobi SVD

To perform the SVD of a matrix, several algorithms have been developed with the aim of optimizing the number of operations carried out by the machine. One of the most widely used is the Jacobi's algorithm with its One sided Jacobi variant. The approach used is to apply successive rotations to the original matrix, in order to bring to zero the components that are outside the diagonal. Through different iterations, a diagonal matrix containing the required singular values is obtained as the final result.

1.2.1 Jacobi rotation

The Jacobi rotation is an operation that selectively bring to zero the specific elements of a matrix. Through a rotation in two dimensions, the elements (p, q) and (q, p) of the matrix are brought to zero.

The operation is based of the use of $\mathbf{J}(p, q, \theta)$ matrix of the type:

$$\mathbf{J}(p,q,\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & 0 & \\ & & c & \cdots & -s & & \\ & & \vdots & \ddots & \vdots & & \\ & & s & \cdots & c & & \\ & 0 & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5}$$

where $c = \cos(\theta)$ e $s = \sin(\theta)$, are applied only in the dimensions p and q.

Premultiply a vector for $\mathbf{J}(p, q, \theta)^T$ is equivalent to rotating it counter clockwise by a θ angle in the (p, q) plane. This rotation produces a vector resultant in which the q component has been brought to zero. In fact if $\mathbf{x} \in \mathbb{R}^n$ and

$$\mathbf{y} = \mathbf{J}(p, q, \theta)^T \cdot \mathbf{x} \tag{6}$$

The

$$y_p = cx_p - sx_q \tag{7}$$

$$y_q = sx_p + cx_q \tag{8}$$

$$y_i = x_i, i \neq p, q \tag{9}$$

From (8) you can see that y_q can be brought to zero by:

$$c = \frac{x_p}{\sqrt{x_p^2 + x_q^2}}, \quad s = \frac{-x_q}{\sqrt{x_p^2 + x_q^2}} \tag{10}$$

For a symmetric $\mathbf{A}$ matrix, you can bring to zero (p, q) and (q, p) applying (11).

$$\mathbf{B} = \mathbf{J}(p, q, \theta)^T \mathbf{A} \mathbf{J}(p, q, \theta) \tag{11}$$

1.2.2 Algoritmo One Sided Jacobi

The basic idea for this algorithm is to rotate the i and j columns of $\mathbf{A}$ to make them orthogonal. In this way, i and j columns of $\mathbf{A}^T\mathbf{A}$ will be implicitly orthogonal.

Let $\mathbf{J}(p, q, \theta)$ be the rotation matrix that applied to $\mathbf{A}$ produces:

$$\mathbf{B} = \mathbf{AJ} \tag{12}$$

where the i and j columns of $\mathbf{B}$ are given by

$$b_{:i} = ca_{:i} - sa_{:j} \tag{13}$$
$$b_{:j} = sa_{:i} + ca_{:j} \tag{14}$$

$(B^T B)_{ij}$ element will be

$$(B^T B)_{ij} = b_{:i}^T b_{:j} = (ca_{:i} - sa_{:j})^T (sa_{:i} + ca_{:j})$$

assuming $(B^T B)_{ij} = 0, i \neq j$ you get

$$cs(||a_{:i}||^2 - ||a_{:j}||^2) + (c^2 - s^2)(sa_{:i} + ca_{:j})$$

Dividing by c^2 and considering

$$\begin{aligned} t &= s/c \\ \alpha &= ||a_{:i}||^2 \\ \beta &= ||a_{:j}||^2 \\ \gamma &= a_{:i}^T a_{:j} \\ \tau &= (\beta - \alpha)/2\gamma \end{aligned}$$

the following equation is obtained:

$$t^2 + 2\tau t - 1 = 0 \tag{15}$$

Solving and choosing the square root with the minimum absolute value as solution

$$t = min| - \tau \pm \sqrt{1 + \tau^2}| \tag{16}$$

we obtain c and s values that represent cosine and sine of the rotation matrix:

$$c = \frac{1}{\sqrt{1 + t^2}}, \quad s = ct \tag{17}$$

With these c and s values, the $(B^T B)_{ij}$ and $(B^T B)_{ji}$ elements are equal to zero.[7]

2 CUDA

2.1 Parallel Computing

During the past several decades, there has been ever-increasing interest in parallel computation. The primary goal of parallel computing is to improve the speed of computation.

From a pure calculation perspective, parallel computing can be defi ned as a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently.

From the programmer's perspective, a natural question is how to map the concurrent calculations onto computers. Suppose you have multiple computing resources. Parallel computing can then be defined as the simultaneous use of multiple computing resources (cores or computers) to perform the concurrent calculations. A large problem is broken down into smaller ones, and each smaller one is then solved concurrently on different computing resources. The software and hardware aspects of parallel computing are closely intertwined together. In fact, parallel computing usually involves two distinct areas of computing technologies:

- Computer architecture (hardware aspect)
- Parallel programming (software aspect)

Computer architecture focuses on supporting parallelism at an architectural level, while parallel programming focuses on solving a problem concurrently by fully using the computational power of the computer architecture. In order to achieve parallel execution in software, the hardware must provide a platform that supports concurrent execution of multiple processes or multiple threads.

Most modern processors implement the *Harvard architecture*, as shown in Figure 1, which is comprised of three main components:

- Memory (instruction memory and data memory)
- Central processing unit (control unit and arithmetic logic unit)
- Input/Output interfaces

The key component in high-performance computing is the central processing unit (CPU), usually called the core. In the early days of the computer, there was only one core on a chip. This architecture is referred to as a uniprocessor. Nowadays, the trend in chip design is to integrate multiple

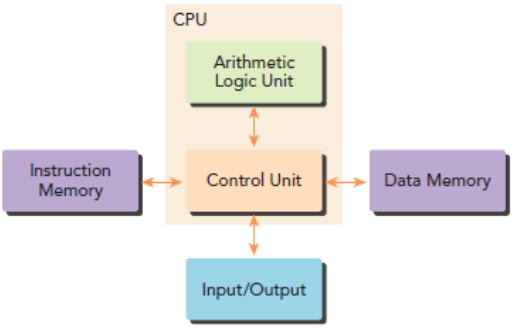


Figure 1: Harvard architecture [3]

cores onto a single processor, usually termed multicore, to support parallelism at the architecture level. Therefore, programming can be viewed as the process of mapping the computation of a problem to available cores such that parallel execution is obtained.

When implementing a sequential algorithm, you may not need to understand the details of the computer architecture to write a correct program. However, when implementing algorithms for multicore machines, it is much more important for programmers to be aware of the characteristics of the underlying computer architecture. Writing both correct and effi cient parallel programs requires a fundamental knowledge of multicore architectures. The following sections cover some basic concepts of parallel computing and how these concepts relate to CUDA programming.

2.2 Sequential and Parallel Programming

When solving a problem with a computer program, it is natural to divide the problem into a discrete series of calculations; each calculation performs a specified task, as shown in Figure 2. Such a program is called a *sequential program*. There are two ways to classify the relationship between two pieces

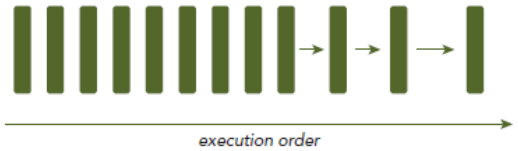


Figure 2: Sequential execution order [3]

of computation: Some are related by a precedence restraint and therefore must be calculated sequentially; others have no such restraints and can be calculated concurrently. Any program containing tasks that are performed

concurrently is a *parallel program*. As shown in Figure 3, a parallel program may, and most likely will, have some sequential parts.

From the eye of a programmer, a program consists of two basic ingredients: instruction and data. When a computational problem is broken down into many small pieces of computation, each piece is called a task. In a task, individual instructions consume inputs, apply a function, and produce outputs. A data dependency occurs when an instruction consumes data produced by a preceding instruction. Therefore, you can classify the relationship between any two tasks as either dependent, if one consumes the output of another, or independent.

Analyzing data dependencies is a fundamental skill in implementing parallel algorithms because dependencies are one of the primary inhibitors to parallelism, and understanding them is necessary to obtain application speedup in the modern programming world. In most cases, multiple independent chains of dependent tasks offer the best opportunity for parallelization.

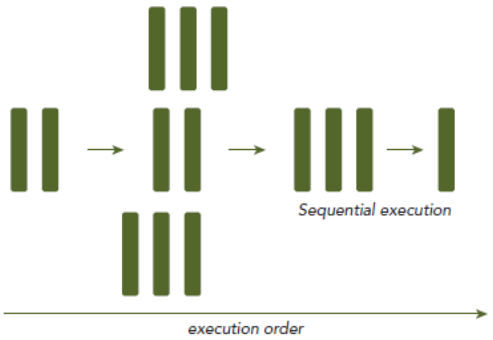


Figure 3: Parallel execution order [3]

2.3 CUDA Programming Structure

The CUDA programming model enables you to execute applications on heterogeneous computing systems by simply annotating code with a small set of extensions to the C programming language. A heterogeneous environment consists of CPUs complemented by GPUs, each with its own memory separated by a PCI-Express bus. Therefore, you should note the following distinction:

- **Host**: the CPU and its memory (host memory)
- **Device**: the GPU and its memory (device memory)

To help clearly designate the different memory spaces, example code in this book uses variable names that start with h_ for host memory, and d_ for device memory.

Starting with CUDA 6, NVIDIA introduced a programming model improvement called Unified Memory, which bridges the divide between host and device memory spaces. This improvement allows you to access both the CPU and GPU memory using a single pointer, while the system automatically migrates the data between the host and device. More details about the unified memory will be covered in Chapter 2.6. For now, it is important that you learn how to allocate both the host and device memory, and explicitly copy data that is shared between the CPU and GPU. This programmer-managed control of memory and data gives you the power to optimize your application and maximize hardware utilization.

A key component of the CUDA programming model is the kernel - the code that runs on the GPU device. As the developer, you can express a kernel as a sequential program. Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads. From the host, you define how your algorithm is mapped to the device based on application data and GPU device capability. The intent is to enable you to focus on the logic of your algorithm in a straightforward fashion (by writing sequential code) and not get bogged down with details of creating and managing thousands of GPU threads.

The host can operate independently of the device for most operations. When a kernel has been launched, control is returned immediately to the host, freeing the CPU to perform additional tasks complemented by data parallel code running on the device. The CUDA programming model is primarily asynchronous so that GPU computation performed on the GPU can be overlapped with host-device communication. A typical CUDA program consists of serial code complemented by parallel code. As shown in Figure 4, the serial code (as well as task parallel code) is executed on the host, while the parallel code is executed on the GPU device. The host code is written in ANSI C, and the device code is written using CUDA C. You can put all the code in a single source fi le, or you can use multiple source fi les to build your application or libraries. The NVIDIA C Compiler (nvcc) generates the executable code for both the host and device.

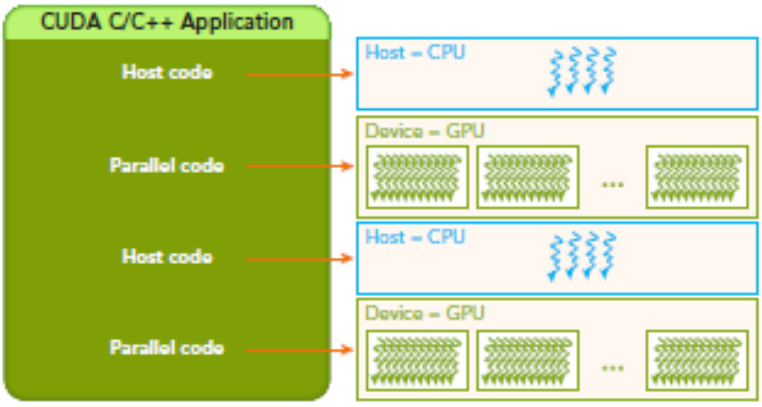


Figure 4: Cuda programming structure [3]

A typical processing fl ow of a CUDA program follows this pattern:

1. Copy data from CPU memory to GPU memory.
2. Invoke kernels to operate on the data stored in GPU memory.
3. Copy data back from GPU memory to CPU memory.

2.4 Managing Memory

The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory. Kernels operate out of device memory. To allow you to have full control and achieve the best performance, the CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory. Table 1 lists the standard C functions and their corresponding CUDA C functions for memory operations. The function used to perform GPU memory

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Table 1: Host and Device Memory Functions

allocation is cudaMalloc, and its function signature is:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

This function allocates a linear range of device memory with the specified *size* in bytes. The allocated memory is returned through *devPtr*. You may notice the striking similarity between cudaMalloc and the standard C runtime library malloc. This is intentional. By keeping the interface as close to the standard C runtime libraries as possible, CUDA eases application porting.

The function used to transfer data between the host and device is: cudaMemcpy, and its function signature is:

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t
    count, cudaMemcpyKind kind )
```

This function copies the specifi ed bytes from the source memory area, pointed to by *src*, to the destination memory area, pointed to by *dst*, with the direction specified by *kind*, where kind takes one of the following types:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

This function exhibits synchronous behavior because the host application blocks until cudaMemcpy returns and the transfer is complete. Every CUDA call, except kernel launches, returns an error code of an enumerated type cudaError_t. For example, if GPU memory is successfully allocated, it returns:

cudaSuccess

Otherwise, it returns:

cudaErrorMemoryAllocation You can convert an error code to a human-readable error message with the following CUDA runtime function:

```
char* cudaGetErrorString(cudaError_t error)
```

The cudaGetErrorString function is analogous to the Standard C strerror function. The CUDA programming model exposes an abstraction of memory hierarchy from the GPU architecture. Figure 5 illustrates a simplified GPU memory structure, containing two major ingredients: global memory and shared memory. You will learn more about the GPU memory hierarchy in Chapter 2.6. One of the more notable characteristics of the CUDA programming model is the exposed memory hierarchy. Each GPU device has a set of different memory types used for different purposes. You will learn much more detail about this hierarchy in Chapter 2.6. In the GPU memory hierarchy, the two most important types of memory are global memory and

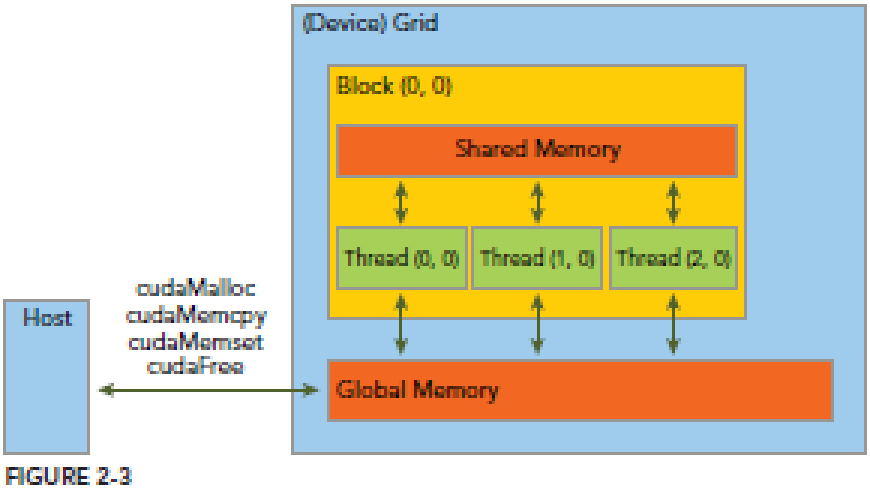


Figure 5: GPU Memory Structure [3]

shared memory. Global memory is analogous to CPU system memory, while shared memory is similar to the CPU cache. However, GPU shared memory can be directly controlled from a CUDA C kernel.

2.5 Organizing Threads

When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specifi ed by the kernel function. Knowing how to organize threads is a critical part of CUDA programming. CUDA exposes a thread hierarchy abstraction to enable you to organize your threads. This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks, as shown in Figure 6. All threads spawned by a single kernel launch are collectively called a grid. All threads in a grid share the same global memory space. A grid is made up of many thread blocks. A thread block is a group of threads that can cooperate with each other using:

- Block-local synchronization
- Block-local shared memory

Threads from different blocks cannot cooperate. Threads rely on the following two unique coordinates to distinguish themselves from each other:

- blockIdx (block index within a grid)
- threadIdx (thread index within a block)

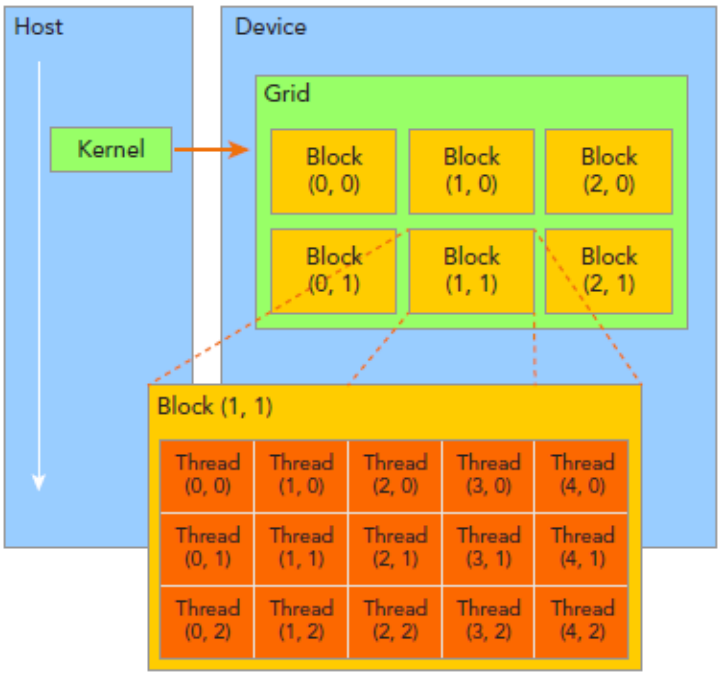


Figure 6: Thread hierarchy [3]

These variables appear as built-in, pre-initialized variables that can be accessed within kernel functions.

When a kernel function is executed, the coordinate variables blockIdx and threadIdx are assigned to each thread by the CUDA runtime. Based on the coordinates, you can assign portions of data to different threads.

The coordinate variable is of type uint3, a CUDA built-in vector type, derived from the basic integer type. It is a structure containing three unsigned integers, and the 1st, 2nd, and 3rd components are accessible through the fi elds x, y, and z respectively.

blockIdx.x
blockIdx.y
blockIdx.z
threadIdx.x
threadIdx.y
threadIdx.z

CUDA organizes grids and blocks in three dimensions. Figure 6 shows an example of a thread hierarchy structure with a 2D grid containing 2D blocks. The dimensions of a grid and a block are specified by the following two built-in variables:

- blockDim (block dimension, measured in threads)

- gridDim (grid dimension, measured in blocks)

These variables are of type dim3, an integer vector type based on uint3 that is used to specify dimensions. When defi ning a variable of type dim3, any component left unspecifi ed is initialized to 1. Each component in a variable of type dim3 is accessible through its x, y, and z fi elds, respectively, as shown in the following example:

blockDim.x

blockDim.y

blockDim.z Usually, a grid is organized as a 2D array of blocks, and a block is organized as a 3D array of threads. Both grids and blocks use the dim3 type with three unsigned integer fields. The unused fields will be initialized to 1 and ignored.

There are two distinct sets of grid and block variables in a CUDA program: manually-defi ned dim3 data type and pre-defi ned uint3 data type. On the host side, you defi ne the dimensions of a grid and block using a dim3 data type as part of a kernel invocation. When the kernel is executing, the CUDA runtime generates the corresponding built-in, pre-initialized grid, block, and thread variables, which are accessible within the kernel function and have type uint3. The manually-defined grid and block variables for the dim3 data type are only visible on the host side, and the built-in, pre-initialized grid and block variables of the uint3 data type are only visible on the device side.

2.6 CUDA Memory Model

To programmers, there are generally two classifi cations of memory:

- Programmable: You explicitly control what data is placed in programmable memory.
- Non-programmable: You have no control over data placement, and rely on automatic techniques to achieve good performance.

In the CPU memory hierarchy, L1 cache and L2 cache are examples of non-programmable memory. On the other hand, the CUDA memory model exposes many types of programmable memory to you:

- Registers
- Shared memory
- Local memory

- Constant memory
- Texture memory
- Global memory

Figure 7 illustrates the hierarchy of these memory spaces. Each has a different scope, lifetime, and caching behavior. A thread in a kernel has its own

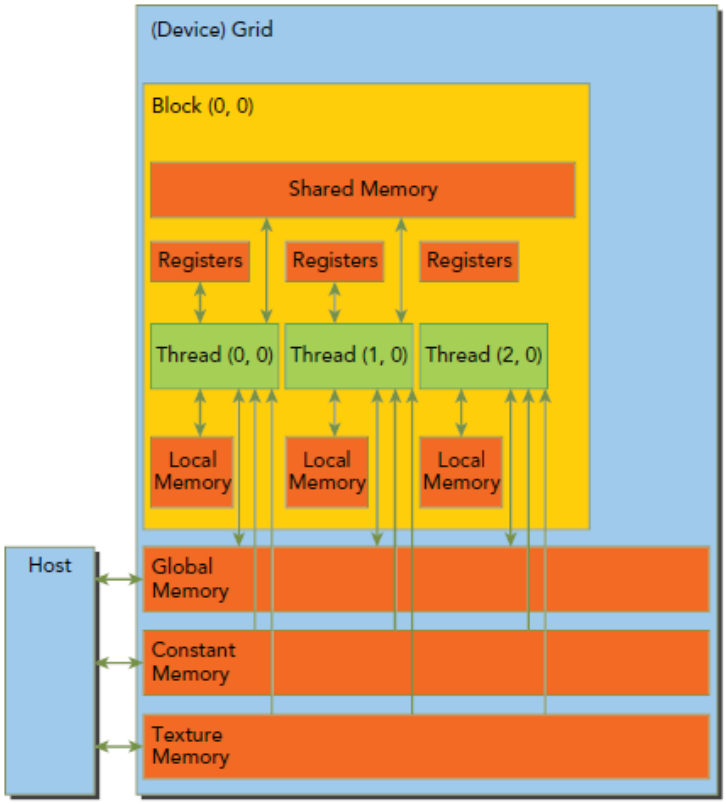


Figure 7: GPU Memory Hierarchy [3]

private local memory. A thread block has its own shared memory, visible to all threads in the same thread block, and whose contents persist for the lifetime of the thread block. All threads can access global memory. There are also two read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different uses. Texture memory offers different address modes and fi ltering for various data layouts. The contents of global, constant, and texture memory have the same lifetime as an application.

2.6.1 Registers

Registers are the fastest memory space on a GPU. An automatic variable declared in a kernel without any other type qualifi ers is generally stored in a register. Arrays declared in a kernel may also be stored in registers, but only if the indices used to reference the array are constant and can be determined at compile time.

Register variables are private to each thread. A kernel typically uses registers to hold frequently accessed thread-private variables. Register variables share their lifetime with the kernel. Once a kernel completes execution, a register variable cannot be accessed again.

Registers are scarce resources that are partitioned among active warps in an SM.

2.6.2 Local Memory

Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory. Variables that the compiler is likely to place in local memory are:

- Local arrays referenced with indices whose values cannot be determined at compile-time.
- Large local structures or arrays that would consume too much register space.
- Any variable that does not fit within the kernel register limit.

The name "local memory" is misleading: Values spilled to local memory reside in the same physical location as global memory, so local memory accesses are characterized by high latency and low bandwidth and are subject to the requirements for efficient memory access. For GPUs with compute capability 2.0 and higher, local memory data is also cached in a per-SM L1 and per-device L2 cache.

2.6.3 Shared Memory

Variables decorated with the following attribute in a kernel are stored in shared memory:

__shared__

Because shared memory is on-chip, it has a much higher bandwidth and much lower latency than local or global memory. It is used similarly to CPU L1 cache, but is also programmable.

Each SM has a limited amount of shared memory that is partitioned among thread blocks. Therefore, you must be careful to not over-utilize shared memory or you will inadvertently limit the number of active warps.

Shared memory is declared in the scope of a kernel function but shares its lifetime with a thread block. When a thread block is fi nished executing, its allocation of shared memory will be released and assigned to other thread blocks.

Shared memory serves as a basic means for inter-thread communication. Threads within a block can cooperate by sharing data stored in shared memory. Access to shared memory must be synchronized using the following CUDA runtime call:

```
void __syncthreads();
```

This function creates a barrier which all threads in the same thread block must reach before any other thread is allowed to proceed. By creating a barrier for all threads within a thread block, you can prevent a potential data hazard. They occur when there is an undefi ned ordering of multiple accesses to the same memory location from different threads, where at least one of those accesses is a write. _syncthreads may also affect performance by forcing the SM to idle frequently.

The L1 cache and shared memory for an SM use the same 64 KB of on-chip memory.

2.6.4 Constant Memory

Constant memory resides in device memory and is cached in a dedicated, per-SM constant cache. A constant variable is decorated with the following attribute:

__constant__

Constant variables must be declared with global scope, outside of any kernels. A limited amount of constant memory can be declared — 64 KB for all compute capabilities. Constant memory is statically declared and visible to all kernels in the same compilation unit.

Kernels can only read from constant memory. Constant memory must therefore be initialized by the host using:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const
    void* src, size_t count);
```

This function copies count bytes from the memory pointed to by src to the memory pointed to by symbol, which is a variable that resides on the device

in global or constant memory. This function is synchronous in most cases.

Constant memory performs best when all threads in a warp read from the same memory address. For example, a coeffi cient for a mathematical formula is a good use case for constant memory because all threads in a warp will use the same coefficient to conduct the same calculation on differentdata. If each thread in a warp reads from a different address, and only reads once, then constant memory is not the best choice because a single read from constant memory broadcasts to all threads in a warp.

2.6.5 Texture Memory

Texture memory resides in device memory and is cached in a per-SM, read-only cache. Texture memory is a type of global memory that is accessed through a dedicated read-only cache. The readonly cache includes support for hardware filtering, which can perform fl oating-point interpolation as part of the read process. Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance. For some applications, this is ideal and provides a performance advantage due to the cache and the fi ltering hardware. However, for other applications using texture memory can be slower than global memory.

2.6.6 Global Memory

Global memory is the largest, highest-latency, and most commonly used memory on a GPU. The name *global* refers to its scope and lifetime. Its state can be accessed on the device from any SM throughout the lifetime of the application.

A variable in global memory can either be declared statically or dynamically. You can declare a global variable statically in device code using the following qualifier:

__device__

In the Chapter 2.4, you learned how to dynamically allocate global memory. Global memory is allocated by the host using cudaMalloc and freed by the host using cudaFree. Pointers to global memory are then passed to kernel functions as parameters. Global memory allocations exist for the lifetime of an application and are accessible to all threads of all kernels. You must take care when accessing global memory from multiple threads. Because thread execution cannot be synchronized across thread blocks, there is a potential hazard of multiple threads in different thread blocks concurrently modifying

the same location in global memory, which will lead to an undefi ned program behavior.

Global memory resides in device memory and is accessible via 32-byte, 64-byte, or 128-byte memory transactions. These memory transactions must be naturally aligned; that is, the first address must be a multiple of 32 bytes, 64 bytes, or 128 bytes. Optimizing memory transactions are vital to obtaining optimal performance. When a warp performs a memory load/store, the number of transactions required to satisfy that request typically depends on the following two factors:

- Distribution of memory addresses across the threads of that warp.
- Alignment of memory addresses per transaction.

In general, the more transactions necessary to satisfy a memory request, the higher the potential for unused bytes to be transferred, causing a reduction in throughput efficiency.

For a given warp memory request, the number of transactions and the throughput efficiency are determined by the compute capability of the device. For devices of compute capability 1.0 and 1.1, the requirements on global memory access are very strict. For devices with compute capabilities beyond 1.1, the requirements are more relaxed because memory transactions are cached. Cached memory transactions exploit data locality to improve throughput efficiency.

3 Sequential implementation

Jacobi rotation is a rotation of a 2-dimensional subspace in an n-dimensional space, denoted by $\mathbf{J}$. A pair of elements of a symmetric matrix $\mathbf{B} \in \mathbb{R}^{n \times n}$ are set to zero after the application of this transformation, denoted as $\mathbf{B} \longmapsto \mathbf{J^T B J} = \mathbf{B'}$ where $c = \cos(\theta)$, $s = \sin(\theta)$ and θ is the rotation angle in the (i, j)-plane. Only i-th and j-th rows of $\mathbf{B}$ are affected. Similarly, only the j-th and i-th columns are affected. The elements b'_{ij}, b'_{ji}, b'_{ii} and b'_{jj} in $\mathbf{B}$ are used to calculate the angles in the rotation matrices. Rotation matrices eliminate the elements b'_{ij} and b'_{ji} as shown in Eq. (18).

$$\mathbf{J}(i,j,\theta) = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & i & & j & \\ \end{array} \\ \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \end{array} \tag{18}$$

Jacobi algorithm performs a sequence of orthogonal updates, where each new matrix $\mathbf{B}$ is more diagonal than its predecessor. Eventually the off-diagonal elements are small enough, so they can be considered as zeros. In particular, each Jacobi rotation involves a pre-multiplication and a post-multiplication of $\mathbf{B}$ by orthogonal matrices. In general, $(n^2 - n)/2$ rotations are performed (in the case of a symmetric matrix) trying to make zero all off-diagonal elements. This $(n^2 - n)/2$ transformations constitute a sweep. Commonly, the Jacobi rotations are applied by using one of the following approaches: performing cyclical rotations per row or performing cyclical rotations per column. In these approaches, the pair (i, j) is selected in a row by row or column by column fashion, respectively. For example, if $n = 4$, the rotation sequence is: $(i, j) = (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)$. [1]

Now the CUDA implementation of the sequential algorithm is shown. Il kernel che esegue l'algoritmo One-Sided Jacobi sequential è compreso in un ciclo **while** mostrato di seguito.

```
while (!host_exit_flag) {
  ++iter;
  host_exit_flag = true;
  cudaMemcpy( exit_flag, &host_exit_flag, sizeof(bool),
      cudaMemcpyHostToDevice);
  for (int j = columns - 1; j >= 1; --j)
```

```
    for (int i = j - 1; i >= 0; --i) {
      rotate<<<1, rows>>> (B, i, j, rows, exit_flag);
  }
}
```

Listing 1: Sequential loop

iter variable stores the iterations needed to complete One-Sided Jacobi rotation algorithm successfully.

host_exit_flag contains the flag that allow to exit from the **while** loop when the matrix columns are sufficiently orthogonals.

The i and j pair represent the i^{th} and j^{th} pair of columns that is rotating, B contains the matrix from which extract the singular values.

The *rotate* kernel executes the One-Sided Jacobi rotation described in chapter 1.2. To call this function, only one block and a number of threads equal to the rows are used, as can be seen from the angular brackets, so that through the thread index you can access the k^{th} element of the i^{th} and j^{th} pair of column vectors. The kernel is defined as follows

```
1  __global__ void rotate (float * B, int i, int j, int rows,
       bool * exit_flag){
2    int k = threadIdx.x;
3    __shared__ float alpha, beta, gamm, limit, tao, t, c, s;
4    float *pi, *pj;
5    if (k < rows) {
6      alpha = beta = gamm = 0;
7      __syncthreads();
8      pi = B + rows * i + k;
9      pj = B + rows * j + k;
10     atomicAdd(&alpha, *pi * *pi);
11     atomicAdd(&beta, *pj * *pj);
12     atomicAdd(&gamm, *pi * *pj);
13     __syncthreads();
14     if (* exit_flag) {
15       limit = fabsf(gamm) / sqrtf(alpha * beta);
16       if (limit > eps)
17         * exit_flag = false;
18     }
19     tao = (beta - alpha) / (2 * gamm);
20     t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
21     c = expf(-0.5f * log1pf(t * t));
22     s = c * t;
23     const float tmp = *pi;
24     *pi = c * tmp - s * *pj;
```

```
25      *pj = s * tmp + c * *pj;
26    }
27  }
```

In line 3, *alpha*, *beta*, *gamm*, *limit*, *tao*, *t*, *c* and *s* with attribute *__shared* reside in shared memory as they are common to all threads in the block.

In line 6 the *alpha*, *beta* and *gamm* variables are initialized.

In line 7, *__syncthreads*() waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to *__syncthreads*() are visible to all threads in the block.

The matrix *B* B is formed by column vectors and pointers *pi* e *pj* point to the memory location where the k^{th} element is contained, that is the k^{th} row, of the pair of i^{th} and j^{th} vectors column.

The *atomicAdd* function, in lines 10, 11 e 12, is defined as

```
float atomicAdd(float* address, float val);
```

reads the 32-bit floating point *old* located at the address *address* in global or shared memory, computes (*old* + *val*), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns *old*. It is used to calculate the norm of the i^{th} and j^{th} column vectors and their dot product and write it in *alpha*, *beta* and *gamm* respectively.

If the variable *exit_flag* is true, *limit* is calculated and if it's greater than the global variable *eps*

```
static const float eps = 1e-4;
```

Listing 2: eps variable declaration

then *exit_flag* becomes false.

The variables *c* and *s* represent $\sin(\theta)$ and $\cos(\theta)$ functions, respectively, described in equation (18).

In lines 24 and 25the matrix is rotated multiplying appropriately the content of the pointers *pi* and *pj* for *c* and *s*.

Next, in the program the kernel *computeSingVals* is called and it's performs the calculation of the singular values.

```
computeSingVals<<<columns, rows>>>(B, AUX1, rows, columns);
```

A number of block equals to *columns* and a number of theads equals of *rows* are used. In this way using the block and thread indexes you can acced to che j^{th} column vector and to his k^{th} element. The variables *AUX1* is an array that contains the singular values.

The *computeSingVals* kernel is defined as

```
__global__ void computeSingVals (float * B, float * AUX1,
    int rows, int columns){
  int k = threadIdx.x;
  int j = blockIdx.x;
  __shared__ float t;
  if ((j < columns) && (k < rows)){
    float *pj = B + rows * j + k;
    t = 0;
    atomicAdd(&t, *pj * *pj);
    AUX1[j] = sqrtf(t);
  }
}
```

Listing 3: Kernel computing singular values

The 32-bit floating point t variable, initilized to zero, contains the square of the j^{th} columns vector norm.

To obtain the singular values you need to do the square root of t with the *sqrtf* function. *AUX1* array stores the calculated values.

4 Parallel implementation

Parallelizing the One-Sided Jacobi entails partitioning the $n(n-1)/2$ pairs of columns that must be orthogonalized each sweep into sets of independent column pairs. Each sweep is then processed one set at a time, orthogonalizing the column pairs within the current set in parallel.

Column pairs for each set are generated using a round-robin tournament scheduling algorithm. Conceptually, each round in the tournament represents a set, and the pairings of players within a round correspond to the pairings of columns within that set. As an example, the following is all possible sets containing their respective column pairs for $n = 6$:

$$\begin{aligned}
\text{Set } 1 &= \{(1,2),(3,4),(5,6)\} \\
\text{Set } 2 &= \{(1,4),(2,6),(3,5)\} \\
\text{Set } 3 &= \{(1,6),(2,3),(4,5)\} \\
\text{Set } 4 &= \{(1,5),(2,4),(3,6)\} \\
\text{Set } 5 &= \{(1,3),(2,5),(4,6)\}
\end{aligned}$$

In general, each set contains $\hat{n}/2$ pairs of columns that are orthogonalized in parallel, where $\hat{n}$ is the next even integer greater than or equal to n. If n is odd, then each set will have one column paired with a "ghost" column; pairs containing the ghost column are not orthogonalized. Under this scheme, it takes $\hat{n}/2 - 1$ sets to complete a full sweep.

The column pair (p', q') orthogonalized by the i^{th} rotation in a set is computed directly from the corresponding column pair (p, q) orthogonalized by the i^{th} rotation in the previous set. In practice, this generation scheme is better suited for execution on a GPU where compute bandwidth greatly exceeds memory bandwidth. [5]

In the traditional sequential implementations of the Jacobi algorithm, the parallelism is not exploited, mainly due to the data dependence of a rotation with its predecessor. The parallel Jacobi algorithm exploits the maximum parallelism for the decomposition of a symmetric matrix. This algorithm requires $n-1$ number of steps, each step having $n/2$ rotations. All rotations within a step can be executed in parallel. [1]

It is now possible to explore using CUDA to implement a parallel SVD based on the One-Sided Jacobi method described in 1.2. The $m \times n$ input matrix A is taken to be real-valued with $m \geq n$. If $m < n$, then the SVD of A can be computed from the SVD of $A^T = V\Sigma U^T$. Entries are stored in column-major order as single-precision floating point numbers. [5]

The CUDA implementation of the parallel algorithm is now shown. The code that invokes the kernel that executes the One-sided Jacobi parallel algorithm is as follows.

```
while(!host_exit_flag) {
  ++iter;
  host_exit_flag = true;
  cudaMemcpy(dev_exit_flag, &host_exit_flag, sizeof(bool),
      cudaMemcpyHostToDevice);
  for(int set = 0; set < cols; set++) {
    scheduling<<<1, 1>>> (dev_v1, dev_v2, cols);
    round <<<cols/2, rows>>> (dev_B, dev_v1, dev_v2, cols,
        rows, dev_exit_flag);
  }
  cudaMemcpy( &host_exit_flag, dev_exit_flag, sizeof(bool),
      cudaMemcpyDeviceToHost);
}
```

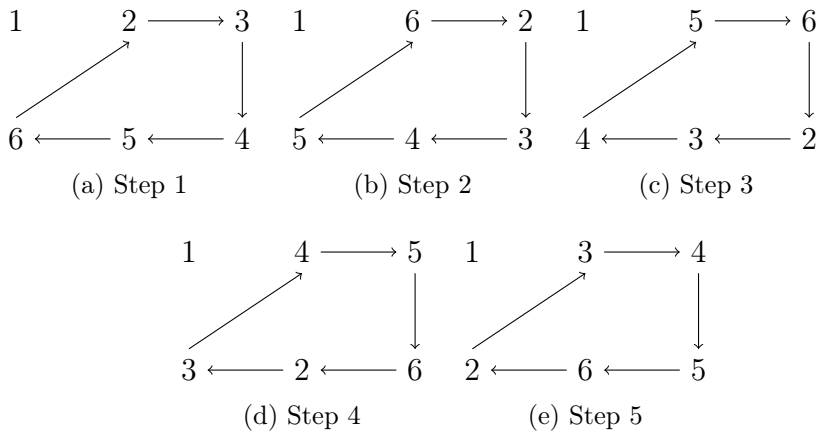
Listing 4: Parallel loop

The variables *host_exit_flag*, *iter* and B perform the same functions as described in code 4.

The variable *set* represents the current round of the round robin scheduling. As explained above, the number of sets needed to form all possible combinations are $n - 1$, where n is the number of columns of the original matrix.

The vectors *v1* and *v2* represent the index of the pairs of columns of the matrix to rotate. For each set the vector elements with the same index represent a pair of columns to be mutually orthogonal. The update of the vectors *v1* and *v2* is performed by the kernel *scheduling*.

The kernel *scheduling* rotates the *v1* and *v2* vectors at each iteration of the loop where the value of *set* is increased. In fact, to perform the round robin algorithm it is sufficient to keep fixed the first element of the first vector and to rotate the others as if they were a single vector. Thus, the element 0 representing the first column will always be at the first position of the first vector. The other elements of the same vector will shift to the right, while the second vector will shift to the left. The outgoing element of the second vector is moved to the vacancy left by the *v1* element 1 and the last element of the first vector becomes the last element of the second. Assuming you have a matrix with 6 columns, the arrays *v1* and *v2* contain integers 1 to 3 and 4 to 6 respectively. In this case, the round robin algorithm is presented in figure 8.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5

Figure 8: Round Robin scheduling

Each iteration of this algorithm is carried out on the device. In this way, you don't need to copy the vectors from host to device and vice versa. The computational cost for launching a kernel compensates by far what would be needed for the *cudaMemcpy* function.

```
__global__ void scheduling (int *v1, int *v2, int cols){
  int tmp = v2[0];
  for (int i = 0; i < (cols/2) - 1; i++)
    v2[i] = v2[i+1];
  v2[cols/2 - 1] = v1[cols/2 - 1];
  for (int i = (cols/2) -1; i > 1; i--)
    v1[i] = v1[i-1];
  v1[1] = tmp;
}
```

The *round* kernel concurrently orthogonalizes each pair of columns in a set, giving it a linear compute grid consisting of $n/2$ thread blocks. For each set, the b^{th} thread block is responsible for orthogonalizing the columns a_{pb} and a_{qb} , where (p_b, q_b) is the bth column pair of the current set.

Each thread block consists of m threads and needs at most $16m$ bytes of shared memory to store the column vectors a_{pb}, a_{qb} , v_{pb} , and v_{qb} . Threads with consecutive thread identifiers read from global memory into shared memory consecutive elements of each of the previously listed column vectors. This has the benefit of coalescing global memory accesses into fewer transactions, thereby improving the memory bandwidth of the kernel.

With one thread allocated per element for one of the column vectors read into shared memory, it is possible to update each vector component in parallel, resulting in increased thread-level parallelism. Since each thread

undergoes the same computation, divergent warps only occur when fewer than m threads need to be used to update the components of v_{pb} and v_{qb}.[5]

The chapters 4.1, 4.2 and 4.3 present three variants of the kernel *round* depending on the memory in which the matrix B is stored.

The calculation of singular values is performed by the kernel *computeSingVals*, code 3, described above.

4.1 Global Memory

This chapter shows the kernel *round* that uses the global memory.

In lines 2 and 3 of code 5, once the index values of the current block and thread are saved in the variables *blockId* and *threadId* respectively, from *v1* and *v2* is extracted the pair of indices i and j corresponding to i^{th} and j^{th} column vectors that is going to become orthogonal. The indices are saved in variables i and j as you can see in lines 6 and 7.

The pointers *pi* and *pj* will point to the items each thread should work on. Therefore, i and j represent the columns obtained from *v1* and *v2*, while the thread id indicates the row to work on (that is the item in the corresponding column). Taking into account the fact that matrix B is in memory in column-major order, you can access the specific items as in lines 8 and 9 of code 5.

The variables *alpha*, *beta* and *gamm* represent homonyms described in the theory of Jacobi algorithm in chapter 1.2. These variables are saved in the shared memory; in this way their value is shared among the threads of the same block. The function *atomicAdd* allows each thread to increase the value of the variables sequentially, avoiding collisions due to parallel access. Each thread then calculates its *alpha*, *beta* and *gamm* value for the row they represent and adds that to the total value that takes into account the single pair on which the block is working on.

The value of *limit* is calculated for the purpose of being compared with the eligible value meeting convergence. When the threshold is exceeded, the variable *exit_flag* is modified, which allows to exit the iterations once the variable is copied from the device to the host.

Following the instructions explained above, we obtain the values of the variables s and c, which represent sine and cosine of the rotation matrix. These variables depend on *alpha*, *beta* and *gamm*. They are therefore common for each pair of columns.

Finally, the rotation matrix is applied by rotating the columns with the appropriate values of c and s.

```
1 __global__ void round (float *B, int *v1, int *v2, int cols
    , int rows, bool * exit_flag) {
```

```
2   int blockId = blockIdx.x;
3   int threadId = threadIdx.x;
4   __shared__ float alpha, beta, gamm;
5   if ((blockId < cols/2) && (threadId < rows)){
6     int i = *(v1 + blockId);
7     int j = *(v2 + blockId);
8     float * pi = B + rows * i + threadId;
9     float * pj = B + rows * j + threadId;
10    alpha = beta = gamm = 0;
11    __syncthreads();
12    atomicAdd(&alpha, *pi * *pi);
13    atomicAdd(&beta, *pj * *pj);
14    atomicAdd(&gamm, *pi * *pj);
15    __syncthreads();
16    if (*exit_flag) {
17      const float limit = fabsf(gamm) / sqrtf(alpha * beta)
          ;
18      if (limit > eps){
19        *exit_flag = false;
20      }
21    }
22    const float tao = (beta - alpha) / (2 * gamm);
23    const float t = sign (tao) / (fabsf(tao) + sqrtf(1 +
        tao * tao));
24    const float c = expf(-0.5f * log1pf(t * t));
25    const float s = c * t;
26    const float tmp = *pi;
27    *pi = c * tmp - s * *pj;
28    *pj = s * tmp + c * *pj;
29  }
30 }
```

Listing 5: Codice parallel global

4.2 Semi Shared Memory

In this implementation the variables *alpha*, *beta*, *gamm*, *limit*, *tao*, *t*, *c*, *s*, *i* and *j* are stored in the shared memory and are identified by the attribute *__shared*. The pointers *pi* and *pj* that contain the memory address of the rows of the i^{th} and j^{th} column vectors are not stored in the shared memory.

```
__global__ void round (float *B, int *v1, int *v2, int cols
    , int rows, bool * exit_flag) {
```

```
  int blockId = blockIdx.x;
  int threadId = threadIdx.x;
  float * pi, *pj;
  __shared__ float alpha, beta, gamm, limit, tao, t, c, s;
  __shared__ int i, j;
  if ((blockId < cols/2) && (threadId < rows)){
    i = *(v1 + blockId);
    j = *(v2 + blockId);
    pi = B + rows * i + threadId;
    pj = B + rows * j + threadId;
    alpha = beta = gamm = 0;
    __syncthreads();
    atomicAdd(&alpha, *pi * *pi);
    atomicAdd(&beta, *pj * *pj);
    atomicAdd(&gamm, *pi * *pj);
    __syncthreads();
    if ( *exit_flag) {
      limit = fabsf(gamm) / sqrtf(alpha * beta);
      if (limit > eps){
        *exit_flag = false;
      }
    }
    tao = (beta - alpha) / (2 * gamm);
    t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
    c = expf(-0.5f * log1pf(t * t));
    s = c * t;
    const float tmp = *pi;
    *pi = c * tmp - s * *pj;
    *pj = s * tmp + c * *pj;
  }
}
```

4.3 Shared Memory

In the implementation where shared memory is used to work with matrix B you can see that between the angle brackets indicating the blocks and threads allocated to the kernel there is a third variable as can be seen in the following code.

```
round <<<cols/2, rows, 2*rows*sizeof(float)>>> (dev_B,
   dev_v1, dev_v2, cols, rows, dev_exit_flag);
```

This allows to dynamically allocate shared memory, which can be used when the amount of shared memory is not known at compile time. In this case the shared memory allocation size per thread block must be specified (in bytes) using an optional third execution configuration parameter.

```
__global__ void round (float *B, int *v1, int *v2, int cols
    , int rows, bool * exit_flag) {
  int blockId = blockIdx.x;
  int threadId = threadIdx.x;
  __shared__ float alpha, beta, gamm, limit, tao, t, c, s,
     tmp;
  extern __shared__ float arr[];
  __shared__ int i, j;
  if ((blockId < cols/2) && (threadId < rows)){
    i = *(v1 + blockId);
    j = *(v2 + blockId);
    arr[threadId] = *(B + rows * i + threadId);
    arr[threadId+rows] = *(B + rows * j + threadId);
    alpha = beta = gamm = 0;
    __syncthreads();
    atomicAdd(&alpha, arr[threadId] * arr[threadId]);
    atomicAdd(&beta, arr[threadId+rows] * arr[threadId+rows
       ]);
    atomicAdd(&gamm, arr[threadId] * arr[threadId+rows]);
    __syncthreads();
    if (*exit_flag) {
      limit = fabsf(gamm) / sqrtf(alpha * beta);
      if (limit > eps){
        *exit_flag = false;
      }
    }
    tao = (beta - alpha) / (2 * gamm);
    t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
    c = expf(-0.5f * log1pf(t * t));
    s = c * t;
    tmp = arr[threadId];
    arr[threadId] = c * tmp - s * arr[threadId+rows];
    arr[threadId+rows] = s * tmp + c * arr[threadId+rows];
    *(B + rows * i + threadId) = arr[threadId];
    *(B + rows * j + threadId) = arr[threadId+rows];
  }
}
```

The dynamic shared memory kernel, *round*, declares the shared memory array

using an unsized extern array syntax (note the empty brackets and use of the extern specifier). The size is implicitly determined from the third execution configuration parameter when the kernel is launched.

```
extern __shared__ float arr[];
```

5 Performance

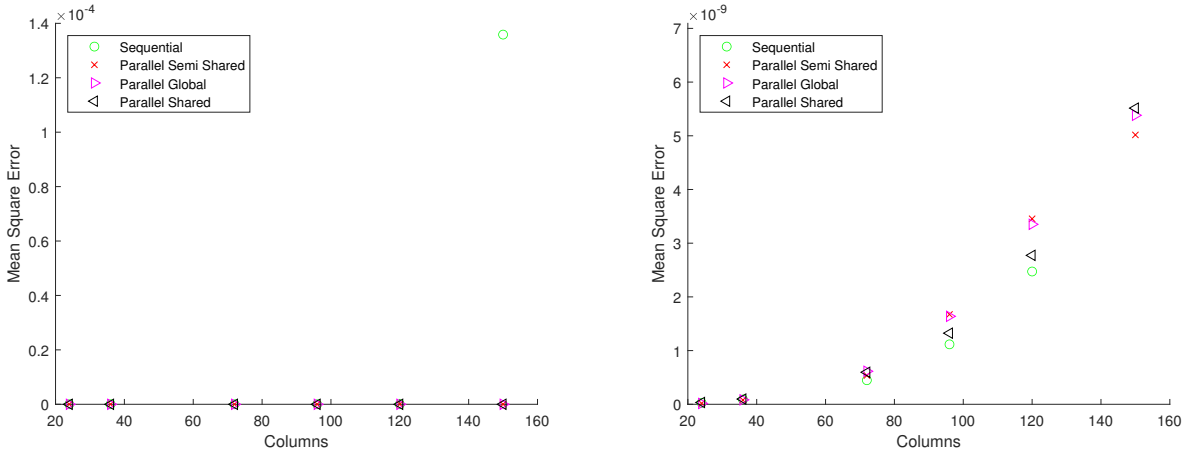
To test the algorithms presented in chapters 3, 4.1, 4.2 and 4.3, six Matlab generated matrices were used via the function *rand* that returns a single uniformly distributed random number in the interval $(0, 1)$. These matrices have an aspect ratio, that is the ratio between the width (columns) and the height (rows) of a matrix, equal to $4 : 3$. The matrices generated have 32, 48, 96, 128, 160 and 200 rows.

The charts presented in this chapter were generated using the Nvidia GTX 610M GPU because it was not possible to use the JETSON board to produce the results.

The figure 9a and 9b show how the mean square error (MSE) of the singular values change varying both the One-Sided Jacobi algorithm and the number of columns of the matrix used. The MSE is calculated as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

where n is the number of matrix columns, Y is the vector that contains the singular values calculated by the Matlab function *svd* and $\hat{Y}$ is the *AUX1* array, shown in code 3, which contains the singular values calculated by the GPU. In figure 9a you can see that the maximum sqaure error is 10^{-4}, consistent with the variable *eps* equal to this value in code 2. Figure 9b shows that for all matrices with less than 150 columns the mean square error is in the around 10^{-9}.



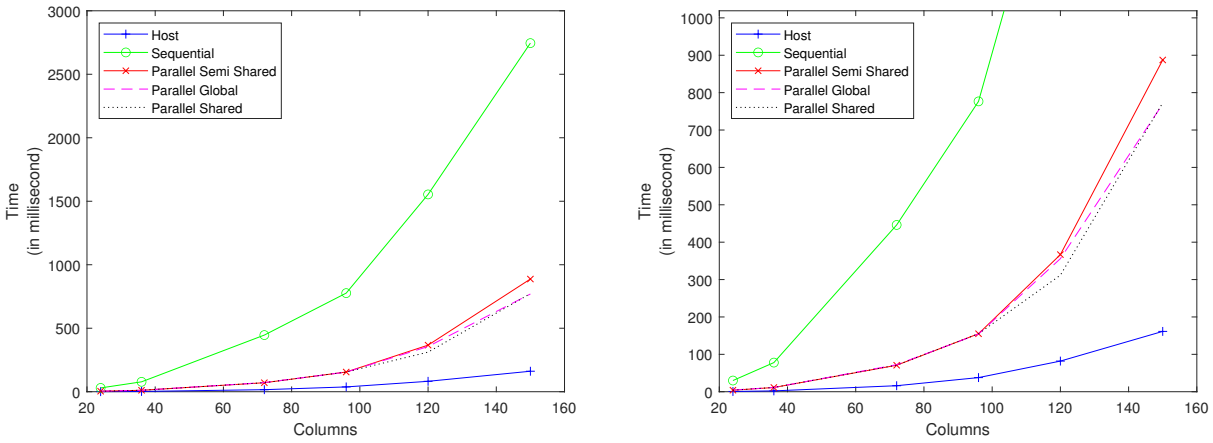
(a) Mean square error

(b) Mean square error zoom

Figure 9: Mean square error comparison

The following graphs were obtained using the function *cudaEventRecord* that records an event. This function allows to manage two *event*, called *start* and *stop*, and to measure the time elapsed between the two. *Start* is recorded before the *while* loop presented in code 1, while *stop* is recorded when you exit this loop. As one sided Jacobi rotation is performed in the *while* loop, the time elapsed between the entry and exit of the cycle is measured.

The figures 10a and 10b show how the execution time of the One-Sided Jacobi algorithm changes with the matrix size, in particular with the number of columns because all test matrices have the same aspect ratio. The fastest algorithm remains the one that runs on the host, while among the algorithms tested on the device the best is the parallel one that uses shared memory. This is also confirmed by the fact that this is the fastest memory and can be up to 100 times faster than global memory. [3] [6]



(a) Columns vs Time

(b) Columns vs Time zoom

Figure 10: Columns vs Time comparison

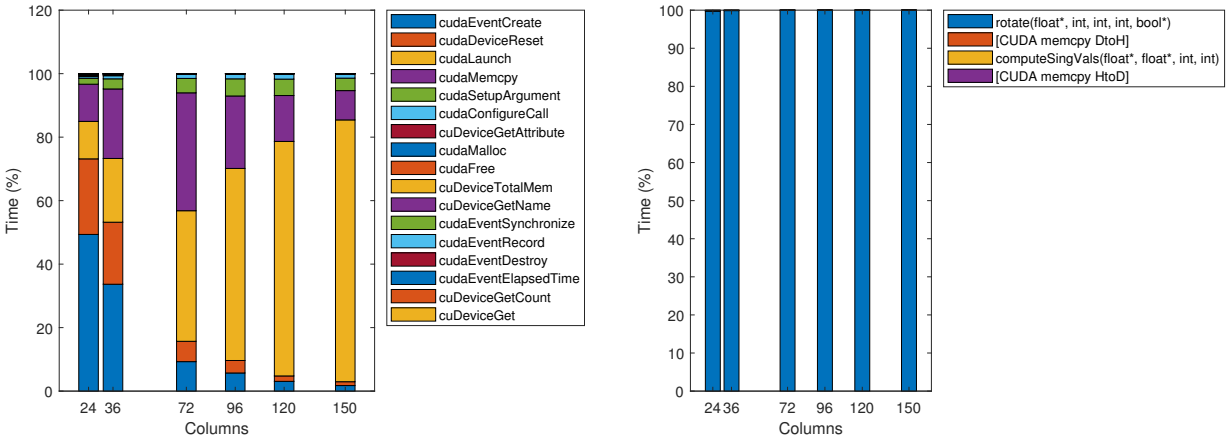
The following graphs were obtained using the *nvprof* tool. The nvprof profiling tool enables you to collect and view profiling data from the command-line. nvprof enables the collection of a timeline of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, memory set and CUDA API calls and events or metrics for CUDA kernels. Profiling options are provided to nvprof through command-line options. In the left graphs you can see the calls to the CUDA API, while in the right one the kernel performance.

On the ordinate axis, the measurement taken into account is the percentage of the total time of the program executed on the device. The size

of the matrix, in relation to the number of columns, is shown on the axis of the ascisse. The graphs presented are divided by the type of algorithm considered.

The figures 11a and 11b show the sequential algorithm performance. In figure 11a we note that for the matrix with 24 columns the functions that occupy the most time in percentage are *cudaEventCreate* (49%), *cudaDeviceReset* (23%), *cudaLaunch* (11%) and *cudaMemcpy* (11%), while for the matrix with 150 columns are *cudaLaunch* (82%) and *cudaMemcpy* (9%). The functions *cudaDeviceReset* and *cudaLaunch* destroy all allocations and reset all state on the current device in the current process e launches a device function, respectively. The increase in the percentage of time relative to *cudaLaunch* as the matrix size increases is due to the kernel *rotate* that is called $n(n-1)/2$ times, where n is the number of columns, as explained in chapter 3 and as you can see in code 1. The number of times the function on the device is called is quadratic with the increase of columns.

The kernel *rotate* is the most time consuming kernel, as shown in figure 11b with a time percentage greater than 99%.



(a) API Profiling Sequential

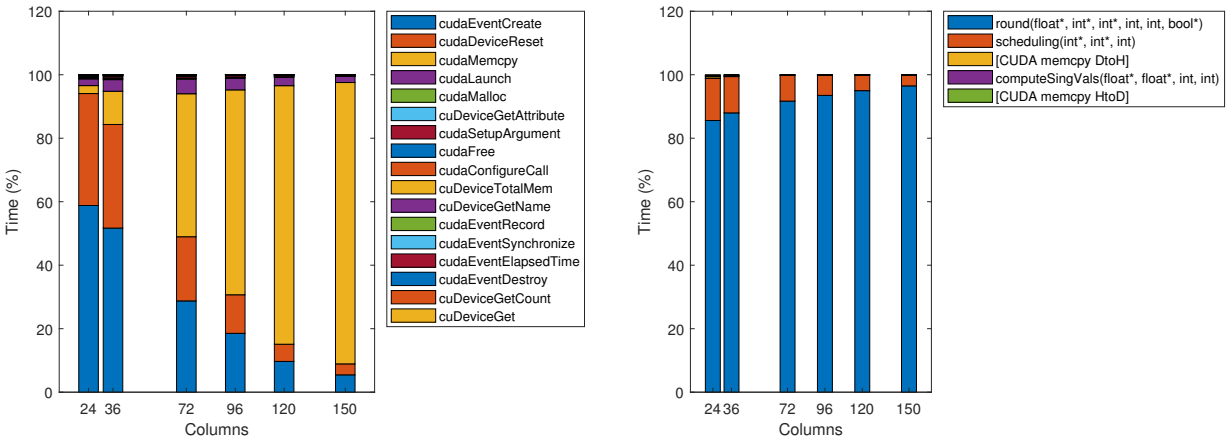
(b) Kernel Profiling Sequential

Figure 11: Sequential performance

The figures 12a and 12b show the performance of the parallel algorithm with the matrix B saved in the global memory. As shown in figure 12a, the functions that require more time for the smaller matrix are *cudaEventCreate* (58%), *cudaDeviceReset* (35%), *cudaMemcpy* (2%) and *cudaLaunch* (2%). For the larger matrix are instead *cudaMemcpy* (88%), *cudaEventCreate* (5%), *cudaDeviceReset* (3%) and *cudaLaunch* (2%). As the matrix size increases,

the function *cudaMemcpy* (88%) predominates from the temporal point of view because it is necessary to make more copies of the variable *host_ exit_ flag* from the host memory to the device, and vice versa, as shown in code 4.

The kernel that takes longer in percentage is *round* with l'85% for the smaller matrix and 96% for the larger one, while *scheduling* employs 13% e il 3%, respectively.



(a) API Profiling Parallel Global

(b) Kernel Profiling Parallel Global

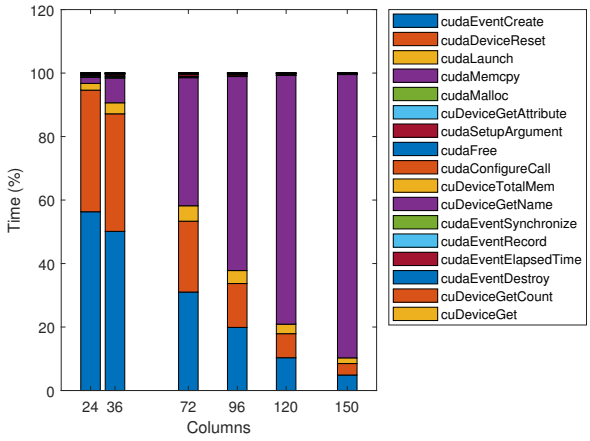
Figure 12: Parallel global performance

The figures 13a and 13b show the performance of the parallel algorithm with the matrix B saved in the global memory and with the variables inside the kernel saved in the shared memory.

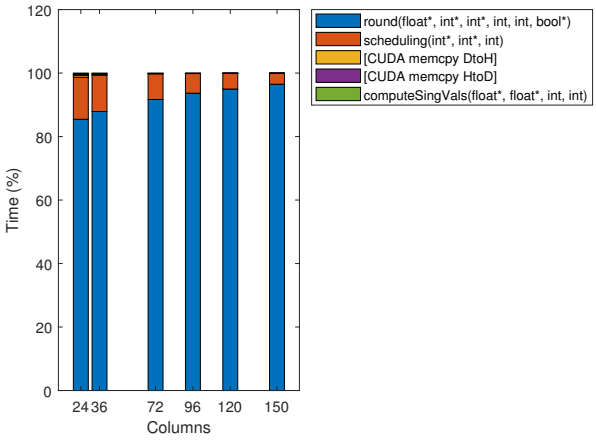
The most time consuming functions for the smallest matrix are *cudaEventCreate* (56%), *cudaDeviceReset* (38%), *cudaMemcpy* (2%) and *cudaLaunch* (2%). For the larger matrix are instead *cudaMemcpy* (89%), *cudaEventCreate* (4%), *cudaDeviceReset* (3%) and *cudaLaunch* (2%).

The kernel *round* takes l'85% of the global time for the smaller matrix and 96% for the larger one, while *scheduling* employs 13% and 3%, respectively.

The results are similar to those described above: the most important variation is in the execution time, shown in figure 10b.



(a) API Profiling Parallel Semi-Shared

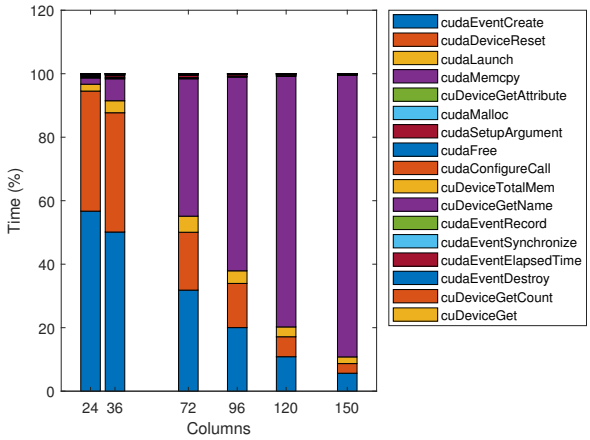


(b) Kernel Profiling Parallel Semi-Shared

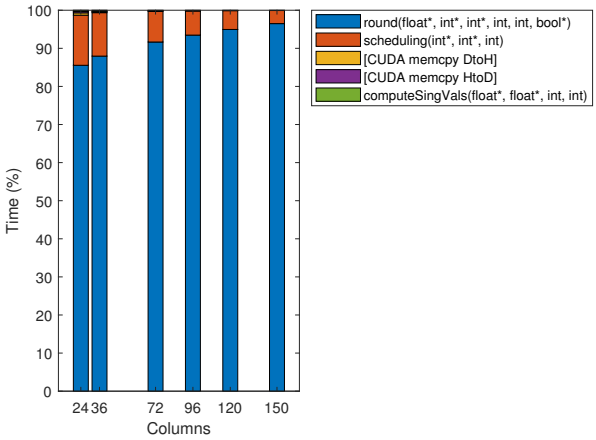
Figure 13: Parallel semi shared performance

The figures 14a and 14b show the parallel algorithm performance with matrix B and the variables used in the kernel saved in the shared memory.

Again, the results are similar to those described above. The use of shared memory to allocate the vector decreases the time to complete the kernel *round*, but not the percentage of time required by individual functions.



(a) API Profiling Parallel Shared



(b) Kernel Profiling Parallel Shared

Figure 14: Parallel shared performance

6 Conclusions

The CUDA programming model has an inherent design trait that complicates the implementation of *round*. In short, it is impossible to synchronize the execution and global memory transactions of threads in different thread blocks within a running kernel. Without loss of generality, for *round* this means that at the end of a set when thread block b_1 writes its orthogonalized columns back to global memory, there is no way to guarantee that thread block b_2 will see and read in either of the updated columns at the start of the next set.

In practice, the only way for threads in different thread blocks to synchronize and share data is to relaunch the kernel after each thread block has written its initial results to global memory. This restricts the scope of *round* and forces the host to launch the kernel once per set. This results in the following undesirable consequences:

- For every run of *round*, each thread block must read the matrix columns from global memory into shared memory or only from global memory in base of the implemented algorithm.

  In contrast, if synchronization across thread blocks were possible, it is possible a more intelligent column pair partitioning scheme could be devised. Ideally, such a scheme would minimize the number of global memory reads a thread block requires at the start of a new set.

- Convergence testing must be done on the host. This requires a counter to be stored in global memory that is incremented for each thread block whose column pair is already sufficiently orthogonal. The host then needs to read this value at the end of each set to determine if it needs to continue calling *round*.

  Since the host must read memory from the device after every set, this has the effect of implicitly synchronizing the host with the device after every invocation of *round*. This effectively reduces, if not eliminates, possibilities for concurrent kernel execution, which in turn limits total throughput.

In principle, an SVD based on the one-sided Jacobi method can be adapted to exploit readily available devices with massively parallel capabilities. With it, it is possible to achieve a high degree of thread-level parallelism that is on the order of the number of elements in the matrix.

However, core characteristics of both the programming model and hardware architecture of NVIDIA GPUs make the above approach a bit of a

mismatch for them in practice. Most notably a lack of grid-level thread synchronization forces the orthogonalization process to be spread out over multiple kernel invocations. Consequently, the host is forced to synchronize with the device after each set, and the device is unable reduce global memory transactions between sets through caching of data in shared memory. Both of these result in additional overhead that ultimately limits the ability to fully exploit the compute capabilities of the GPU.[5]

Another approach for the SVD is presented in [2]. It's very important when we can no longer store the entire matrix in shared memory, we have to operate on the matrix in the slower global memory. Instead of repeatedly reading and updating the columns one at a time, block algorithms can operate with blocks of columns.

There are two global memory block Jacobi algorithms that differ only in the way block columns are orthogonalized and there is a comparison of their performance with parallel streamed calls to the cuSOLVER 8 library routines. Our Jetson GPU cannot use this library because we are using CUDA 6.5 and it has a 3.2 compute capability.

The runtime performance shown in chapter 5 for the implementation of the approach given above was measured on a pc con CPU Intel i7-3630 QM @2.40 GHz, 8GB di RAM e una GPU NVIDIA Geforce GTX 610M. For further work it's recommended to perform the algorithms described with a high-performance GPU that can use cuSOLVER libraries and with a ≥ 3.5 compute capability: this allows the programmer to increase the parallelism using kernels that are invoking other kernels.

References

[1] R. I. Acosta-Quiñonez, D. Torres-Roman, R. Rodrìguez-Avila, and D. Robles-Valdez. *A parallel implementation of One-Sided Jacobi SVD for non-symmetric squared matrices on a high-performance GPU*. Jalisco: Instituto Politécnico Nacional, 2016.

[2] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief, and David E. Keyes. *Batched QR and SVD algorithms on GPUs with application in hierarchical matrix compression*. Thuwal: King Abdullah University of Science and Technology (KAUST), 2017.

[3] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. Ed. by Inc. John Wiley & Sons. Indianapolis, 2014.

[4] Sheetal Lahabar and P.J. Narayanan. *Singular Value Decomposition on GPU using CUDA*. Hyderabad: Center for Visual Information Technology International Institute of Information Technology, 2009.

[5] Michael V. Romer. *Computing the singular value decomposition in parallel on graphics processing units using a one-sided Jacobi method*. Austin: University of Texas, 2013.

[6] Jason Sanders and Kandrot Edward. *CUDA by example : an introduction to general-purpose GPU programming*. Ed. by Addison-Wesley. Boston, 2010.

[7] Claudio Turchetti, Laura Falaschetti, and Lorenzo Manoni. *Singular Value Decomposition Algorithms for Embedded Systems: A Comprehensive Treatment*. Università Politecnica delle Marche,