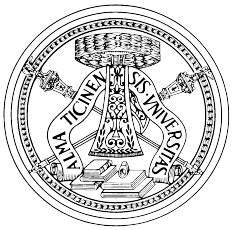
UNIVERSITÀ DEGLI STUDI DI PAVIA

Facoltà di Ingegneria
Dipartimento di Ingegneria Industriale e dell'Informazione



DOCTORAL THESIS IN COMPUTER SCIENCE
XXXI CICLO

High Performance Architectures for Hyperspectral Image Processing and Machine Learning

Supervisor:
Chiar.mo Prof. Francesco LEPORATI
Coordinator:
Chiar.mo Prof. Paolo DI BARBA

Author:
Alessandro FONTANELLA

Contents

Introduction **1**

1 Remote Sensing and Hyperspectral Imaging **3**
1.1 Remote Sensing . . . 3
1.2 Electromagnetic Radiation Sources . . . 4
1.3 Energy Interaction with Earth Surface . . . 5
1.4 Hyperspectral Images . . . 6

2 Compute Unified Device Architecture **9**
2.1 Graphics Processing Unit . . . 9
2.2 CUDA Programming Model . . . 12
2.2.1 Compiling CUDA Program . . . 17
2.2.2 GPU Architecture . . . 18
2.2.3 CUDA Memories . . . 19

3 Band Selection for Hyperspectral Images **23**
3.1 Hyperspectral Unmixing Chain . . . 24
3.2 Image Dataset . . . 26
3.3 Band Prioritization Algorithms . . . 28
3.3.1 SNR-based Criterion . . . 28
3.3.2 ID-based Criterion . . . 39

4 Virtual Dimensionality Estimation for Hyperspectral Images **51**
4.1 Harsanyi–Farrand–Chang (HFC) Algorithm . . . 53

5 Unknown Class Problem in Image Classification **67**
5.1 Neural Networks . . . 68
5.2 Convolutional Neural Networks . . . 70
5.3 Network Architecture and Development . . . 72
5.4 Unknown Class Problem . . . 75

6 Embedded Real-Time Fall Detection **79**
6.1 Wearable Devices . . . 80
6.2 Deep Learning for Fall Detection . . . 82
6.3 Network Architecture . . . 86

6.4 Memory Occupancy, Computation Power and Power Consumption Metrics . . 87
6.5 Inference Module on Embedded Device 92

Conclusion **96**

A OpenMP API **98**

B cuBLAS Library **100**

Bibliography **110**

Introduction

In the last years, the complexity of scientific problems has continuously increased, requiring more and more computational power [1]. Most of these problems can be subdivided in independent sub-problems each one heavy from the computational point of view.
The processing time is one of the major community interest and, typically, it should be kept as low as possible especially in those applications subject to real-time constraints.
In order to deal with this requirement, several technologies providing high computational power have been developed. Among them, the multi-core and many-core devices are the most exploited ones.
This PhD works takes origin from the need of exploring in real application cases what kind of technologies and solutions must be devised to implement high performance computation systems for hyperspectral image processing and machine learning.
The *Custom Computing and Programmable System Laboratory* of University of Pavia focuses its research activities in the algorithms development using the previously mentioned technologies, in particular for hyperspectral image processing [2, 3].
The first part of the presented work, performed in collaboration with the *Hyperspectral Computing Laboratory* of Estremadura University, is focused on the algorithms development for the analysis of hyperspectral images with the aim of building a real-time elaboration system. This particular type of images allows, for instance, to identify the material present in a particular area.
The time required to acquire a scene by modern sensors is around 5 s. Due to the size of the hyperspectral image, which is around 140 MB, it is characterized by a high computationally complexity. For this reason, it is necessary to exploit multi-core and many-core technologies in order to be able to satisfy the real-time constraint. Several algorithms versions, based on programming languages like C and Python exploiting OpenMP API and CUDA framework, is developed and the performance improvements, which can be obtained through parallel technologies, are shown.
It has to be underlined that the description slant is more focused to the strategy which can be followed in order to parallelize an algorithm for the analysis of an hyperspectral image rather than the algorithm itself. A recent trend, particularly in machine learning field, concerns the development from a high performance system to an embedded device which elaborates data directly on the field after acquisition. Another research topic of the *Custom Computing and Programmable System Laboratory* concerns the algorithms development on embedded

devices [4, 5] with the aim of achieving high performance processing also in this case.
In particular, the second part of the presented work, related to the machine learning field, is divided in two projects.
The first one, in collaboration with the *Embedded Systems Laboratory* of Swiss Federal Institute of Technology (EPFL), concerns the development of a Convolutional Neural Network on an embedded device for image recognition. The TensorFlow code is cross-compiled for ARM architecture and the network execution time is compared with other implementations based on pure C programming language and ARM compute library. Moreover, the problem of classifying images, not seen during the network training, is tackled.
The second project, in collaboration with *Computer Vision Laboratory* of University of Pavia and in partnership with *STMicroelectronics*, regards the development of a fall detection system on a low power embedded device. A Recurrent Neural Network, trained on a high performance workstation and implemented on a wearable device, is able to detect in real-time the fall of elderly people on the basis of the values acquired through a three-axis accelerometer. Finally, a set of general metrics for evaluating the network requirements in terms of memory occupancy, computing power and power consumption are derived.

1 Remote Sensing and Hyperspectral Imaging

1.1 Remote Sensing

The purpose of remote sensing is to extract quantitative and qualitative information from an object or area through the analysis of data acquired from a device which is not in direct contact with the subject to measure.
Due to its development, the remote sensing process has become an essential instrument in several application such as the monitoring of the Earth surface.
The general scheme of the process is presented in Figure 1.1.

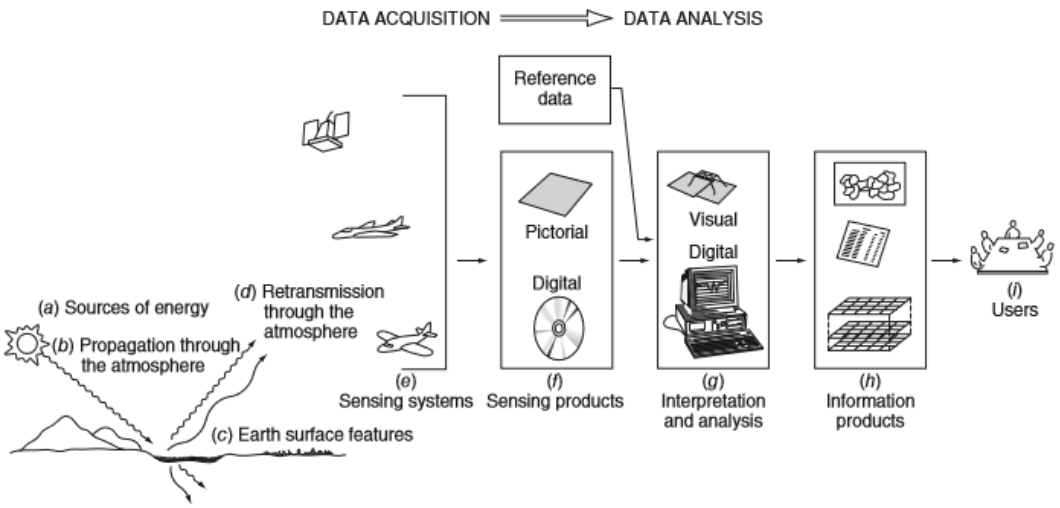


Figure 1.1: General Scheme of Remote Sensing Process [6]

Typical applications of this process are, for instance, the urban expansion estimation and protection of natural resources. However, it can be also applied to other purposes such as estimating a world poverty map [7].
As shown in Figure 1.1, the remote sensing process can be divided in two parts: data acquisition and data analysis.

The former is made up of data collection performed through sensors which acquire the electromagnetic energy emitted and reflected by the Earth surface.
The latter involves examining the acquired data through a computer which extracts all the required information and stores them in form of maps or table.
Finally, the results are presented to the users in order to facilitate and speed-up their decision-making process.

1.2 Electromagnetic Radiation Sources

As mentioned before, the data acquisition is based on the measurement of electromagnetic energy. The visible light is an example of this type of energy but even the radio waves, ultra-violet rays, radiant heat and X-rays belong to the same group. All these energies propagate through harmonic and sinusoidal motion at the speed of light as shown in Figure 1.2

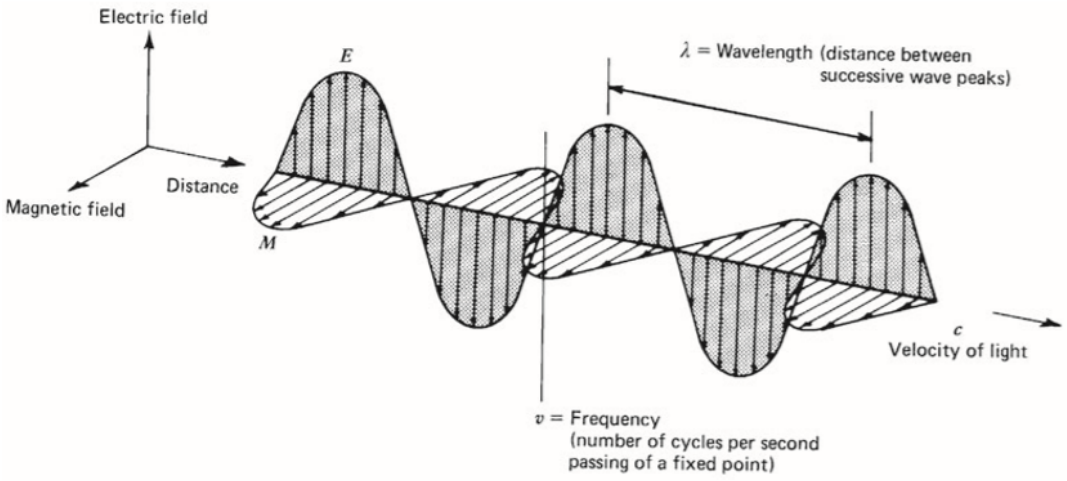


Figure 1.2: Electromagnetic wave [6]

The electromagnetic wave is made up of sinusoidal electric and magnetic waves both perpendicular to the propagation direction. The distance between two consecutive peaks of the wave and the number of peaks per unit time are called *wavelength* (λ) and *frequency* (v), respectively.
The wave obeys to the general equation:

$$c = \lambda v \tag{1.1}$$

Since c is a constant equal to $3 \cdot 10^8$ *m/s*, λ and v are inversely related. For this reason, each of them can be used to characterize a wave.
In remote sensing, it is more common to classify the waves according to their wavelength within the electromagnetic spectrum, as shown in Figure 1.3. Although names are assigned to each portion of the electromagnetic spectrum, it should be underlined that there is no a clear

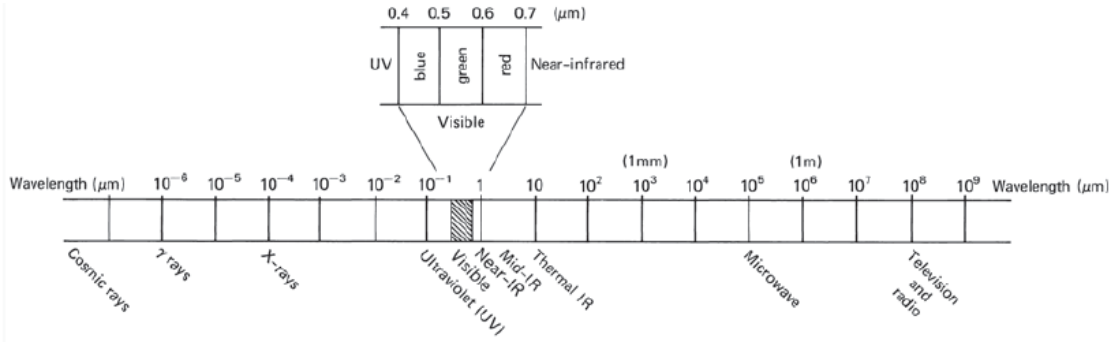


Figure 1.3: Electromagnetic spectrum [6]

separation between the different spectral regions.
The common remote sensing systems can detect waves that belong to the visible, infrared and microwaves spectrum. The visible portion of the electromagnetic spectrum is extremely small since the spectral human eye sensitivity is only between 0.4 μm and 0.7 μm. The blue color ranges from 0.4 μm to 0.5 μm, the green one from 0.5 μm to 0.6 μm and the red one from 0.6 μm to 0.7 μm. The infrared radiation (IR) is divided in: near-infrared from 0.7 μm to 1.3 μm, mid-infrared from 1.3 μm to 3 μm and thermal-infrared from 3 μm to 14 μm. Finally, the microwaves ranges from 1 mm to 1 m.

1.3 Energy Interaction with Earth Surface

Once the electromagnetic energy hits the Earth surface, three types of interaction can occur: reflection, absorption and transmission. By applying the principle of energy conservation, the following equation holds:

$$E_I(\lambda) = E_R(\lambda) + E_A(\lambda) + E_T(\lambda) \tag{1.2}$$

where $E_I(\lambda)$ is the incident energy, $E_R(\lambda)$ is the reflected energy, $E_A(\lambda)$ is the absorbed energy and $E_T(\lambda)$ is the transmitted energy. All these components are function of the wavelength λ.
An example of electromagnetic energy hitting a volume of water is reported in Figure 1.4.
The amount of energy reflected, absorbed and transmitted depends from the different ground characteristics, which themselves are strictly correlated to the type of material and its conditions, and from the wavelength of the incident wave.
Since most of the remote sensing systems work in the electromagnetic spectrum region where the predominant effect is the reflection, the reflectance properties of the Earth surface assume a fundamental role. Hence, Eq. 1.2 is usually rewritten as:

$$E_R(\lambda) = E_I(\lambda) - [E_A(\lambda) + E_T(\lambda)] \tag{1.3}$$

The reflectance characteristic of the Earth surface can be estimated by measuring the amount of incident energy which has been reflected. This entity, which is a function of λ, is called

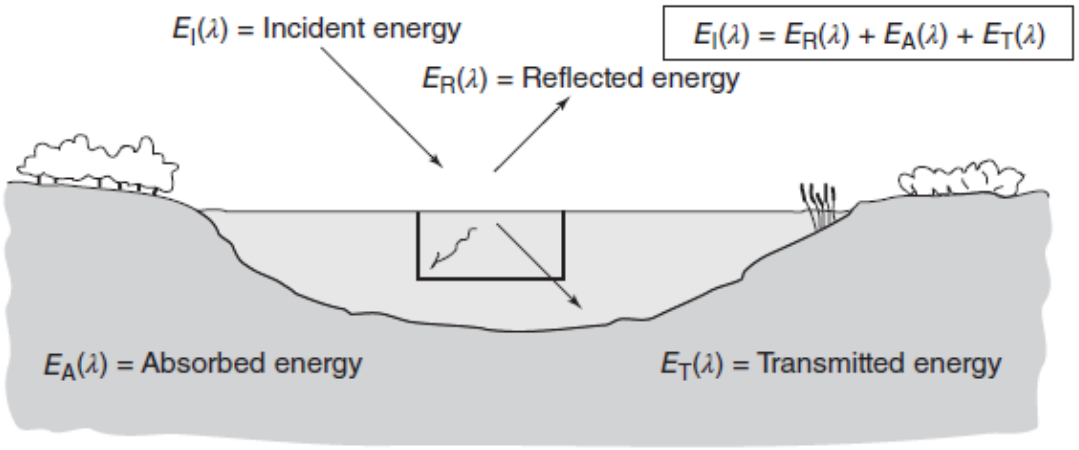


Figure 1.4: Interaction between electromagnetic energy and Earth surface [6]

spectral reflectance (ρ_λ) and is defined as:

$$\rho_\lambda = \frac{E_R(\lambda)}{E_I(\lambda)} \tag{1.4}$$

where ρ_λ is usually expressed as a percentage.
The so-called *spectral reflectance curve*, obtained by plotting the spectral reflectance as a function of the wavelength, provides information about the characteristics of a material and influences the choice of the wavelengths used to acquire spectral data for a particular application.

1.4 Hyperspectral Images

The first step of the remote sensing process is the acquisition and storage of the electromagnetic energy through electronic sensors which generate a signal proportional to the measured reflectance at different wavelengths. However, it is impossible to build sensors that cover all the electromagnetic spectrum. For this reason, each device is characterized by a spectral resolution.
Another sensor property is the spatial resolution which represents the minimum dimension for an object in order to be detected.
A particular type of sensor, called hyperspectral, acquires scenes at several different wavelengths of the electromagnetic spectrum, and produces, as output, the so-called *image cube*. Contiguous wavelengths are grouped into bands, and each pixel of the cube can be represented by three components (x, y, z), where x and y denote a spatial position on the scene, while z identifies a particular band. Thus, the hyperspectral cube represents the same image acquired at multiple different wavelengths. An example is reported in Figure 1.5.
The huge amount of information contained in hyperspectral images has opened their appli-

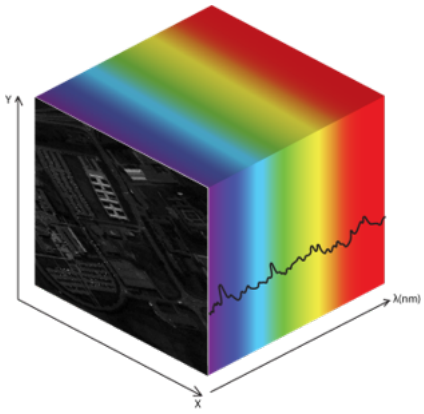


Figure 1.5: Hyperspectral image

cation in several fields such as the determination of mineral composition, water quality and erosion monitoring, agriculture controlling [6] and cancer detection [8].
The high spectral resolution of the hyperspectral sensor allows to plot, for each image pixel, the spectral reflectance curve, also called *spectral signature*. This type of plot identifies in unambiguous way the material which generated it. Thus, each pixel of the image can be seen as a linear combination of the spectral signatures of all the materials present in it, as shown in Figure 1.6.
This hyperspectral image has been acquired by NASA's Airborne Visible-Infrared Imaging Spectrometer (AVIRIS) system which is able to measure reflected energy in the wavelength region from 0.4 μm to 2.5 μm using 224 spectral bands [10]. Every 5 seconds, AVIRIS generates image cubes with shape equals to $512x614x224$ and bit resolution of 16. This means that each algorithm for the analysis of hyperspectral images must analyze 140 MB of data in a limited amount of time (5 seconds) in order to satisfy the real-time constraint. The analysis of hyperspectral images often involves computations which are intrinsically parallel. For this reason, it is possible to evaluate parallel technologies to speed-up the computation. GPUs are popular devices perfect for this kind of computations. Therefore, they are used to achieve real-time performances in hyperspectral images analysis satisfying constraints that using only CPU would be impossible to overcome.

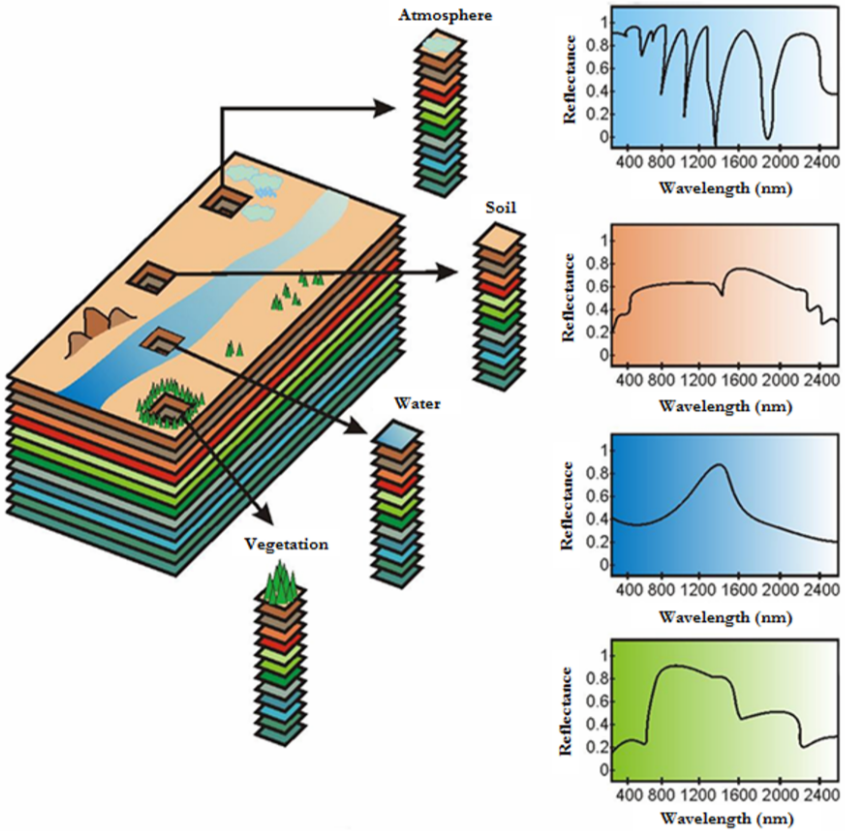


Figure 1.6: Spectral signature concept [9]

2 Compute Unified Device Architecture

2.1 Graphics Processing Unit

Starting from 80's, microprocessor based on single Central Processing Unit (CPU) produced a rapid performance increase and cost reduction in computer applications. However, since 2003 this trend has begun to slow-down due to energy consumption and heat dissipation issues that have limited the increase in clock frequency and the level of elaboration that can be performed during each clock period within a single CPU [11].
In order to tackle this problem, processor producers have developed new chip models in which each CPU is made up of more than one core, generating a real impact on the software developer community [12]. However, if the application is written as a sequential program there is no performance increase since it will not exploit all the available CPU cores. For this reason, most of recent software applications exploit parallel programming in which multiple threads cooperate in order to complete the work faster. The spread of parallel program development has been referred to as *concurrency revolution* [12].
Starting from 2003, the semiconductor industry has defined two main methods for microprocessor design: the *multi-core* and *many-core* approaches. The former aims at maintaining the computational speed of sequential programs while moving into multiple cores. An example is the Intel Core i7™microprocessor with four processor cores each of which is designed to maximize the execution speed of sequential program, supporting the out-of-order execution, multiple instruction issue, the full X86 instruction set and hyper-threading with two hardware threads.
The latter, which is the Graphics Processing Unit (GPU) design method, focuses more on increasing the throughput of parallel application, trying to maximize the number of cores at each generation. For instance, the GPU NVIDIA GTX 1080Ti has 3584 cores each of which is single instruction issue, in-order and multithreaded processor that shares its cache with several other cores. In Figure 2.1, the peak performance evolution, expressed in Giga Floating Point Operations per Second (GFLOPS), for a generic CPU and NVIDIA GPU is shown. This graph is not a real performance comparison since it does not necessarily represent the application speed, but can be used to understand the evolution of the gap between parallel and

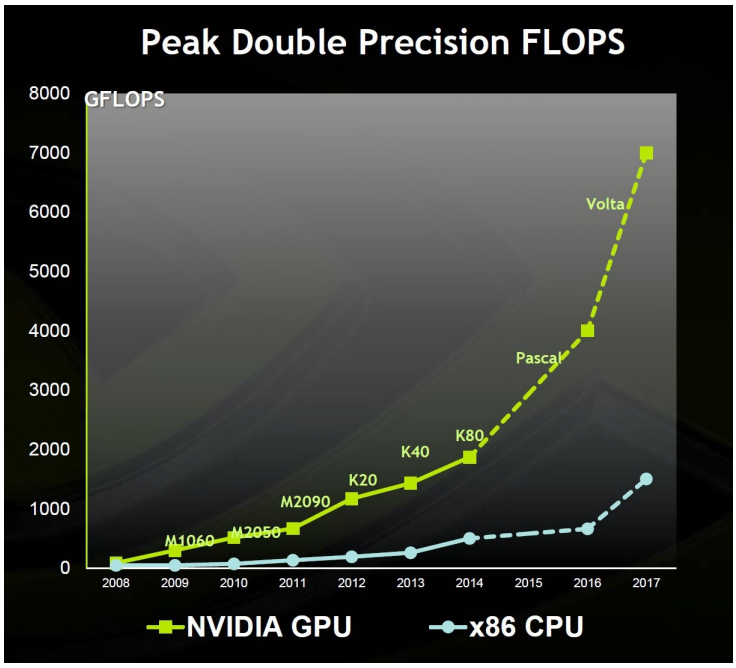


Figure 2.1: Floating-Point Operations per Second for CPU and GPU

sequential execution.
Due to the clear performance difference between CPU and GPU, many application developers has decided to execute the most computationally intensive part of their program by exploiting GPU technology.
The reason for this great performance difference is the designed strategy adopted for the two devices as illustrated in Figure 2.2.

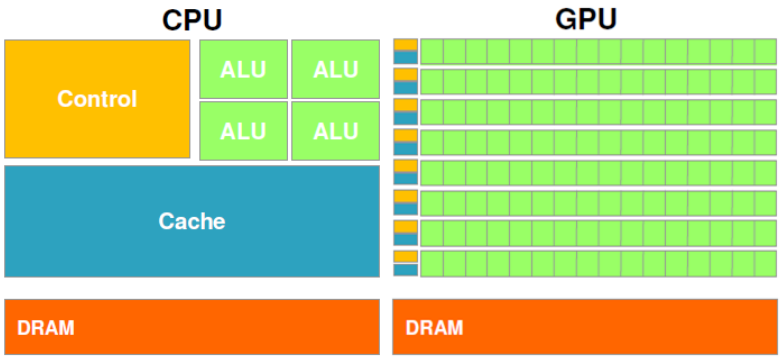


Figure 2.2: CPU and GPU design philosophies [11]

The CPU designed is latency-oriented and optimized for sequential execution. A large cache size memory avoids long-latency memory accesses as well as the arithmetic logic and sophisticated control unit are designed to reduce the operation latency. All these CPU parts shorten the latency of each single thread, but consumes chip area and power that could be used to

increase the number of arithmetic execution units.
On the other hand, the GPU design is throughput-oriented and optimized to maximize the total execution throughput of a large number of threads. Caches are provided to avoid that multiple threads which require the same data have to access the high-latency DRAM memory. The small dimension of caches together with the simple control logic allow to increase the total number of arithmetic execution units. Moreover, the GPU takes advantage of the high number of threads to hide the time required by threads which must access the long-latency memory.
Another important issue related to the different performance between CPU and GPU is the memory bandwidth. Indeed, the execution time of many applications is strictly related to the data transfer speed between the processor and memory system and vice versa. The memory bandwidth comparison is shown in Figure 2.3.

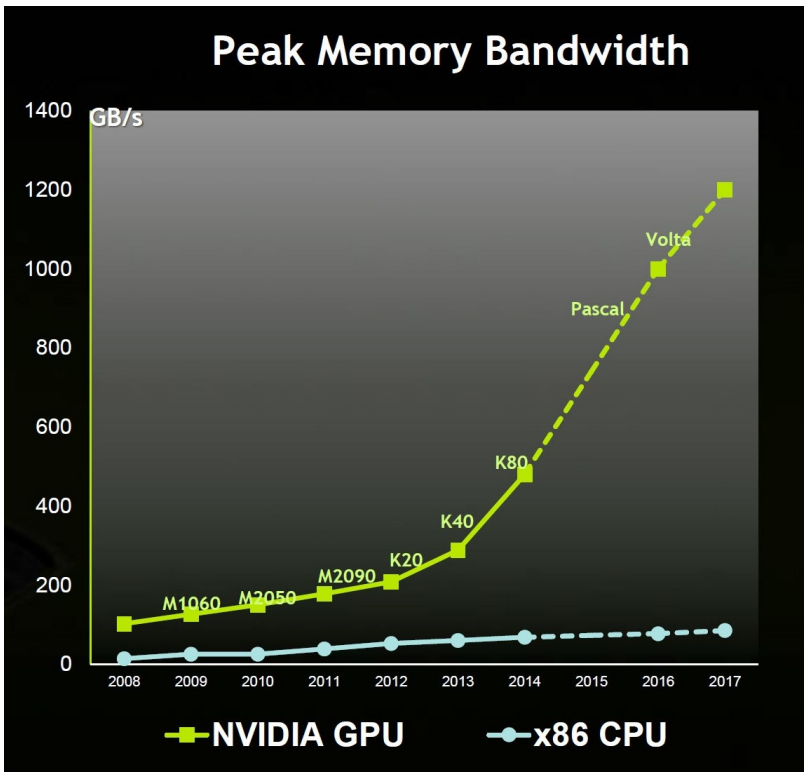


Figure 2.3: CPU and GPU memory bandwidth

This performance gap is due to CPU issues. Indeed, its design has to be compliant with I/O devices, applications and operating system making difficult to increase the memory bandwidth. It should be clear that the GPU is a parallel computing engine that will not perform well if the application has only few threads. In this case, the CPU will have much higher performance than GPU. Therefore, the current adopted strategy is to execute the sequential part on the CPU while numerical intensive part on the GPU. This is the reason why NVIDIA company decided to introduce in 2007 the Compute Unified Device Architecture (CUDA) programming model which supports a CPU-GPU execution of an application [13].

2.2 CUDA Programming Model

Before 2007, the only way to program a GPU device was to use graphics Application Programming Interface (API) such as OpenGL or Direct3D and this technique was called General Purpose GPU (GPGPU). However, due to the limited amount of general operations that could be performed with the graphics API, this programming methodology did not become widespread. In 2007, NVIDIA decided to release the new programming model CUDA which facilitated the use of GPUs for general purpose applications. Thus, several fields, such as physics, image processing and machine learning, started exploiting GPU technology.
The CUDA programming model subdivides the computing system in two parts: host and devices. The former is a traditional CPU while the latter are GPUs used to speed up the execution of computationally intensive parts of the code. Thus, the sequential parts of the program will be executed by the host while the parallel ones will be run on the devices.
The source code of a CUDA program contains both the code for the host and the device. Concerning the host code, it is possible to use programming languages like C/C++ or Fortran while the device code must be compliant with the host one and has to be extended in order to support CUDA features.
Each function executed by the device is called *kernel* and is made up of a set of parallel threads. The typical flow of a CUDA program is reported in Figure 2.4.

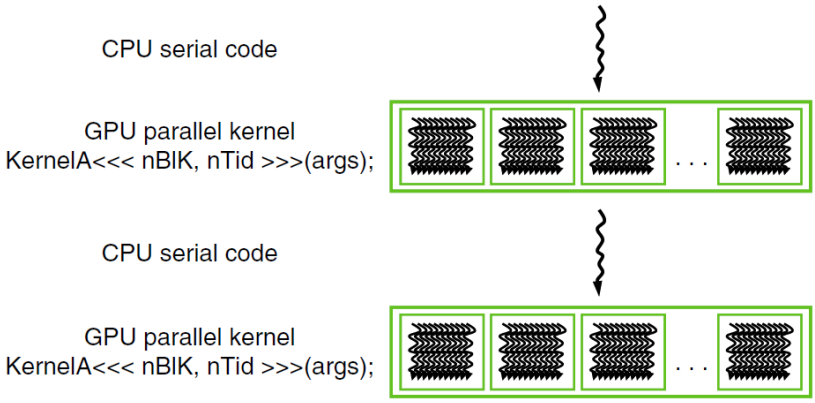


Figure 2.4: Typical CUDA program [11]

The program execution starts always from the host and continues until a kernel is invoked. At this point, the execution is performed by the GPU which generates a great number of threads. When it ends, the execution control returns to the CPU until a new kernel is invoked or the execution will terminate.
Host and device present physically separated memories. Thus, before invoking a kernel, it is necessary to allocate memory on GPU device and transfer all the data necessary for the elaboration. Once the execution is terminated, the results have to be copied back to the CPU and the GPU memory has to be released. The block diagram of these operations is shown in Figure 2.5.
The CUDA programming model provides a series of APIs which perform all the operations

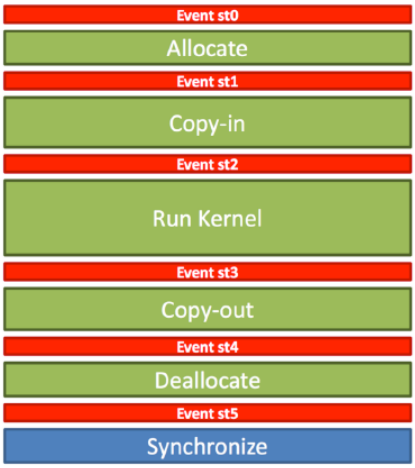


Figure 2.5: Kernel call example

previously described.
In order to allocate memory on the device, the function *cudaMalloc* can be invoked by the host. It has the following prototype:

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

where *devPtr* and *size* are the pointer address and the size, expressed in byte, of the object to be allocated, respectively. This function returns a data of type *cudaError_t* which reports if there were errors or not. Concerning the *cudaMalloc* function, two possible values are allowed: *CudaSuccess* or *CudaErrorMemoryAllocation*. These data can be used to check if there were enough memory on the device to be allocated.
Considering the *Copy-in* and *Copy-out* steps, CUDA provides the *cudaMemcpy* routine in order to manage the memory transfers. It has the following prototype:

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

where *dst* is the destination memory address, *src* is the source memory address, *count* is the size in byte to copy. Finally, *kind* identifies the type of transfer and it can assume four different distinct values: *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* and *cudaMemcpyDeviceToDevice*.
All these values are used to identify the position (host or device) of source and destination memory addresses. In particular, the *Copy-in* and *Copy-out* steps in Figure 2.5 are performed by using *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost*, respectively.
This function returns a *cudaError_t* data which can assume four possible values:

- *cudaSuccess*: the data transfer has been completed successfully

- *cudaErrorInvalidValue*: one or more parameters function are incorrect
- *cudaErrorInvalidDevicePointer*: the provided device memory pointer is not valid
- *cudaErrorInvalidMemcpyDirection*: invalid value assigned to *kind* variable

Before introducing the *Run Kernel* step, it is necessary to explain the threads organization. Inside the GPU, threads are organized in blocks and multiple blocks make up a grid. It is important to underline that each block must have the same number of threads.
In order to identify in a unique way each thread, CUDA model provides data structure containing 3D coordinates. Each single block can be identified inside the grid by using the variable *blockIdx.x*, *blockIdx.y* and *blockIdx.z*. Concerning the threads inside the block, each of them is accessible using the variable *threadIdx.x*, *threadIdx.y* and *threadIdx.z*. An example is shown in Figure 2.6.

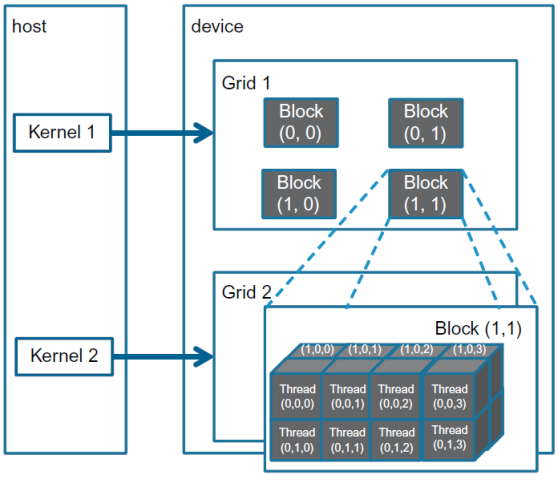


Figure 2.6: An example of CUDA grid organization [11]

Notice that it is not mandatory to use all the three indexes. Moreover, other two three-dimensional variables are provided: *blockDim* and *gridDim*. They are used to identify the dimension of each block and the grid, respectively.
The most common case, the bidimensional one, is reported in Figure 2.7.
In this case, each single thread can be seen as a matrix element and so it can be identified with two indexes as follows:

$$i = threadIdx.y + blockIdx.y * blockDim.y \tag{2.1}$$

$$j = threadIdx.x + blockIdx.x * blockDim.x \tag{2.2}$$

Moreover, there exist the possibility to synchronize all the threads within the same block by using the function __synchthreads. All the threads which will reach this barrier will wait for all the other threads inside the block before continuing the program execution. With the introduction of CUDA 9.0, a new way of synchronization has been introduced and it is called

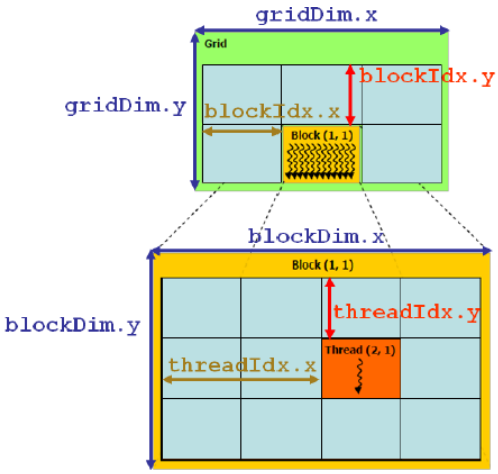


Figure 2.7: Thread Organization [13]

cooperative groups. Now the programmer can define synchronization at sub-block level, but even on the entire grid.
Once the GPU threads organization is known, it is possible to understand the sintax for calling a CUDA *kernel*:

```
kernel <<<dim3 grid, dim3 block>>>(arg1, arg2, ...)
```

where *grid* and *block*, as said before, are structure of type *dim3* used to specify the grid and blocks dimension, respectively, while *arg1* and *arg2* are kernel parameters which are optional. The last step of a CUDA program is to free the previously allocated device memory (*Deallocate* in Figure 2.5). Even in this case, CUDA provides a function, *cudaFree*, which has the following prototype:

```
cudaError_t cudaFree(void* devPtr)
```

where *devPtr* is the device memory pointer to deallocate. For this function, the *cudaError_t* return data can assume the following values:

- *cudaSuccess*: the memory has been successfully released
- *cudaErrorInvalidDevicePointer*: the given device pointer is not valid
- *cudaErrorInitializationError*: the CUDA driver and runtime cannot be initialized

Furthermore, the CUDA programming model allows to manage the synchronization between host and device.
All the CUDA functions can be divided in two categories: *synchronous* and *asynchronous*.

In the first case, the host will wait the results coming from the device and only then it will continue the program execution.
In the second one, the host continues the program execution immediately after the GPU function call.
For instance, kernel launches are asynchronous operations and they can be overlapped with host execution as shown in Figure 2.8.

Figure 2.8: Asynchronous kernel launches

On the other hand, the *cudaMemcpy()* functions belongs to the synchronous category.
However, it is possible to maximize parallel execution between host and device by using asynchronous memory functions (*cudaMemcpyAsync()*) which force to allocate the host memory through the *cudaMallocHost()* function. This function allocates bytes of host memory that is page-locked and accessible to the device. In this way, it is possible to exploit the *concurrency*; the ability to perform the following operations simultaneously:

- *CUDA kernel*
- *cudaMemcpyAsync() HostToDevice*
- *cudaMemcpyAsync() DeviceToHost*
- *Operations on the host*

In order to effect concurrency, it is possible to use the *stream*. The *stream* is a sequence of operations that execute in issue-order on the GPU. CUDA operations in different streams can run concurrently and be overlapped. An example is reported in Figure 2.9.
By overlapping memory transfers, device and host executions, it is possible to obtain performance improvements. However, in order to exploit CUDA *streams*, a suitable GPU engine must be provided.
Dealing with *streams*, it is necessary to manually set a synchronization between host and device whenever the CPU needs the GPU computation results. To this aim, the CUDA programming model provides the *cudaDeviceSynchronize()* function which blocks the host execution until all the issued CUDA calls are completed.

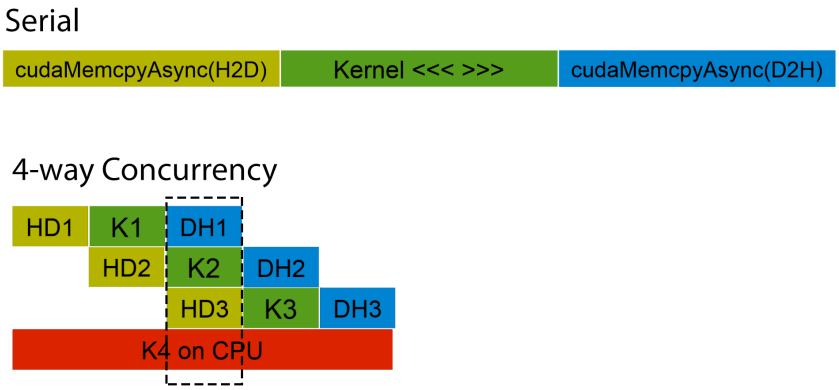


Figure 2.9: Stream usage example

2.2.1 Compiling CUDA Program

Considering the C programming language, CUDA programming model provides a proper compiler which is called *nvcc*. It is able to differentiate between the host and device code. The first one is assigned to a normal compiler such as gcc or cl, while the second one is compiled by *nvcc* into assembly form, called *Parallel Thread eXecution* (PTX) and/or binary form (*cubin* object). An overview of the compilation process for a CUDA program is reported in Figure 2.10.

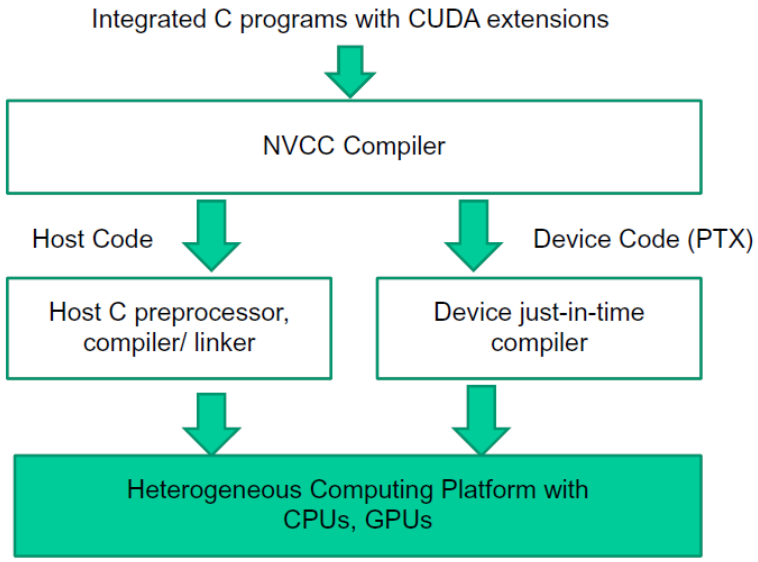


Figure 2.10: CUDA program compilation process [13]

Basically, the PTX code generation is due to compatibility issue. Indeed, any PTX code can be compiled by the GPU device driver during run-time. This strategy, called *just-in-time*

compilation, allows to obtain better performance thanks to compiler improvements present in new device drivers. Moreover, this is the only way to run applications on a device which did not exist when they were compiled. However, this strategy increases the application load time, but, once the PTX has been compiled, a copy of the binary code is stored inside the so-called *compute-cache*. In this way, it is possible to avoid subsequent compilations when calling the same application.
On the other hand, the binary compilation is *architecture-specific*. This means that the compatibility of applications across different device depends on the so-called *compute capability* which identifies the features supported by the GPU device. The *compute capability* of a device is represented by a number expressed as *X.Y* where *X* represents the major revision number while *Y* the minor one. Devices with the same major number have the same core architecture. The main compute capability versions are reported in Table 2.1.

Table 2.1: Compute capability and GPU core architecture

Compute Capability Version	GPU Architecture
1.0	Tesla
2.0	Fermi
3.0	Kepler
5.0	Maxwell
6.0	Pascal
7.0	Volta

A cubin object compiled for a device with compute capability X.y can only be executed by another one with compute capability X.z where z≥y.

2.2.2 GPU Architecture

As said before, the GPU is a many-core device. Its fundamental units are the so-called *CUDA cores* which are responsible for threads execution. On the chip area, several cores are grouped together in *streaming multiprocessors (SMs)* whose structure is reported in Figure 2.11.
Since most of the experiments have been run on a Kepler device, its structure has been analyzed. The SM includes: 192 CUDA cores for floating point operations (Core), 64 Double Precision units (DP unit), 32 Special Function Units (SFU) and 32 Load/Store units (LD/ST). Concerning the memories, each SM has 64 KB of on-chip memory that can be used as shared memory among threads of the same block or as L1 cache. In addition, a 48 KB read only data cache is provided. Finally, a total number of 65536 32 bits-registers is available to each block. Once a kernel is invoked, the generated threads are grouped into blocks. Each SM can manage up to 16 blocks only if enough resources, such as the number of registers or the shared memory, are enough for each threads. Otherwise, the number of assigned blocks will be reduced until the resource requirements are satisfied.
In order to be executed, each block assigned to a SM is split into groups of 32 consecutive threads, called *warps*. Each SM has four warp schedulers and eight instruction dispatch units.

Figure 2.11: General schema of Kepler SM [14]

In this way, it is possible to issue and elaborate four warps concurrently.
The main advantage of this architecture is the so-called *automatic scalability* whose example is illustrated in Figure 2.12.
Since the multithreated program is divided into blocks, a GPU with more SMs will be faster than one with less number of processing units.
The entire GPU is built by adding several SMs on the same chip area as illustrated in Figure 2.13.
The presented Kepler GPU is equipped with 15 SMs and is connected to the CPU through PCI Expresses Gen3 x16 interface. A *GigaThread Engine* manages the thread blocks distribution to each SM. A L2 cache of 1536 KB avoids continuously access to the 12 GB DRAM GDDR5 memory which is controlled by six *Memory Controller* providing a memory interface of 384 bits.

2.2.3 CUDA Memories

In order to obtain the best performance, it is important to efficiently manage the memory accesses. A block diagram of CUDA device memory model is reported in Figure 2.14.

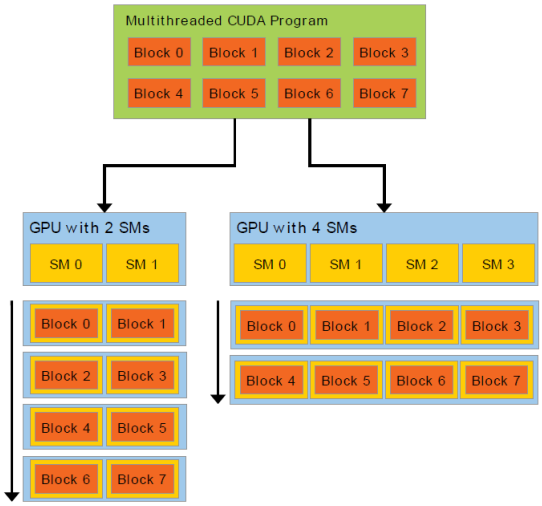


Figure 2.12: Automatic scalability example [13]

Figure 2.13: Kepler full chip block diagram [14]

The memories are subdivided in two types: *global* and *on-chip*.
The *global* memory coincides with the DRAM and it is used to exchange data between CPU and GPU by using the provided API functions *cudaMalloc()* and *cudaMemcpy()*. It can be read and written by each thread independently from its block. The main disadvantage of this memory is the long access latency (few hundreds clock cycles) and low access bandwidth.

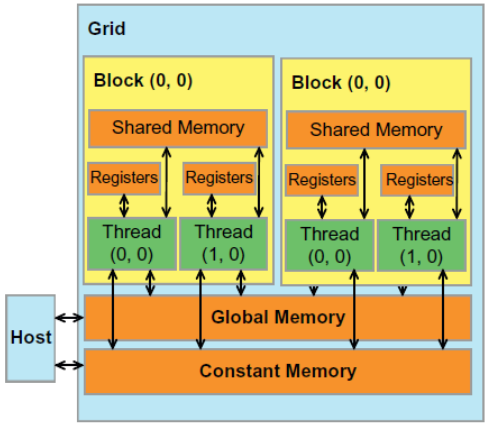


Figure 2.14: CUDA device memory model [11]

The *on-chip* memories, shared memory and registers, have a low access latency and a high access bandwidth. In particular, registers are assigned to individual threads and are perfect to store frequently used variables which are private to each thread.
On the contrary, shared memory can be accessed by all the threads belonging to the same block, supporting efficient data sharing between threads. In this way, the high cost of accessing the global memory is reduced.
The design methodology of the shared memory is the *interleaved memory* organized with 32 banks. It is able to manage n accesses to n distinct memory addresses even avoiding conflict if different threads will request the same memory location. Considering the read operation, if multiple threads request the same data, it will be broadcast while the writing operation will be performed by only one thread.
The access conflict happens if different threads try to request memory addresses mapped on the same bank. In order to avoid issues, the memory access will be serialized.
Examples of these events are shown in Figure 2.15.

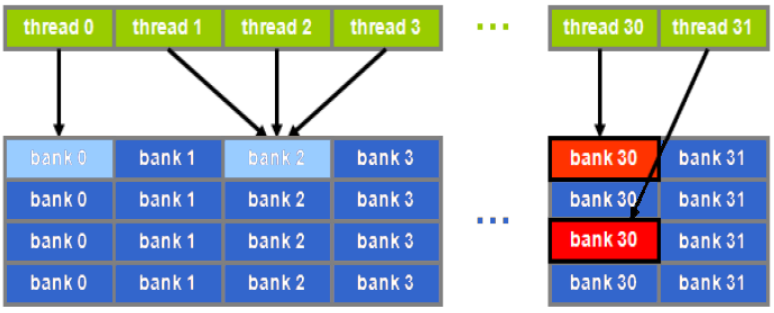


Figure 2.15: Example of a bank multiple access (left) and conflict (right)

It should be underlined that the performance bottleneck could be the high cost for accessing the global memory. However, this access cannot be avoided since this is the only memory

which can be used to share data among all the threads and between the host and device with enough size to store all the application data.
In order to limit this issue, two caches, L1 and L2, have been introduced between the global and on-chip memories. The L1 memory is private to each SM while the L2 one is shared among all the devices.
Concerning the global memory accesses, each of them must pass through the L2 cache. Concerning the store instruction, it will be automatically performed on L2 while for the load instruction two different modalities can be chose: *caching load* and *non-caching load*. In the former mode, the data search starts in L1 cache, then in L2 and finally into the global memory while in the latter mode the search starts directly inside L2.

3 Band Selection for Hyperspectral Images

The hyperpsectral image, also called *image cube*, represents the same scene acquired at different wavelengths. Typically, its memory size can range from hundreds of megabytes up to many gigabytes.
Thus, algorithms applied on this type of data have to deal with a great quantity of information. For this reason, the analysis of the image cube is characterized by high computational requirements.
In order to tackle this problem, the information redundancy, which is due to the high correlation between adjacent bands of the hyperspectral image, has to be reduced with different approaches specifically designed.
A common adopted strategy is *dimensionality reduction* (DR), that produces a new compact dataset by applying a transformation based on a suitable criterion. It can be divided in two categories: *feature selection* and *feature extraction*. The former tries to find a subset of the original data, while the latter transforms data from high-dimensional space to a lower one. For instance, the principal component analysis (PCA), which belongs to the second category, performs DR in order to maximize the variance of the transformed data in the new space.
DR algorithms have to deal with two issues: the choice of the dimension number needed to avoid significant loss of information and the transformation of the input data which can compromise or corrupt the original information [15].
Another approach to reduce information redundancy is the so-called *band selection* (BS), introduced in the early 1990s [16–18] and improved in recent years [19–21].
The BS technique can be divided into two categories: supervised and unsupervised. In supervised approaches, the knowledge about classes or objects is inside the hyperspectral image, while in unsupervised approaches no previous information is required.
BS techniques extract a bands subset of the input hyperspectral image that can represent the original data as well as possible. Since no transformation is applied on the input dataset, the original information are preserved. This property represents the main advantage compared to the DR *feature extraction* technique. However, even in this case it is necessary to deal with two issues: the choice of the number of bands to preserve and the BS criterion.
The first issue can be solved through algorithms for the estimation of the so-called *virtual*

dimensionality (VD) concept which represents a reasonable number of bands to preserve [22]. These algorithms are able to find how many different pure surface materials, called *endmembers*, are present inside the hyperspectral image.
Concerning the second issue, a specific BS methodology, called *band prioritization* (BP), has been adopted since it is the most recent and accurate one. This method assigns a score to each band using a suitable approach. Then, the BS is performed by extracting a certain number of bands on the basis of each score.
The key point in the BP approach is the choice of the criterion used to assign a score to each band. Different approaches have been analyzed. Some of them, like signal-to-noise ratio (SNR), kurtosis, entropy, information divergence (ID) and variance [23], are based on statistical techniques. Another criterion relies on the concept of linearly constrained minimum variance (LCMV) [15]. All of them can be adopted in BP algorithms, which therefore could feature a high computational complexity. The operations involved in those algorithms are however intrinsically parallel. For this reason, it is possible to develop efficient solutions based on specific many-core devices such as graphics processing units (GPUs) which provide high computational capability at low cost.
In the presented work, two criteria will be analyzed: SNR and ID.
The development phase started from a MATLAB implementation based on the proposed algorithms. A serial C code version has been derived, and then, the OpenMP API (See Appendix A) has been adopted for a multi-core CPU implementation.
Finally, a parallelization exploiting NVIDIA GPU has been carried out.
In particular, an accurate comparison among the C serial, OpenMP and CUDA solutions has been conducted in order to identify the most suitable implementation for each algorithm.
The OpenMP solutions have been developed only for allowing a comparison between multi-core and many-core approaches. Therefore, the OpenMP versions exploit simply directives and are not so optimized as the CUDA ones.

3.1 Hyperspectral Unmixing Chain

As previously said, the hyperspectral sensors have an high spectral resolution but a low spatial one. This implies that several endmembers can be present in a single pixel which, in this case, is called *mixed pixel*. If a pixel contains only one pure material, it is identified as *pure pixel*.
The main purpose of the hyperspectral image analysis is the endmembers identification inside the scene and the estimation of their *abundance fractions*, the percentage with which each pure material is present inside each pixel.
Typically, it is assumed that there is a linear combination between the incident energy and the ground. Thus, the hyperspectral image can be modelled as follows:

$$r(x,y) = M\alpha(x,y) + n \tag{3.1}$$

where *r(x,y)* is the reflectance value inside the pixel at position (*x,y*), M is the *endmember matrix* which contains the spectral signature of the pure material inside the scene, α is the

mixing matrix which contains the *abundance fraction* for the considered pixel and n is the noise.
The term *hyperspectral unmixing* refers to the estimation of the matrices M and α which are present in the model described by Eq. 3.1. This task can be accomplished through the so-called *hyperspectral unmixing chain* shown in Figure 3.1.

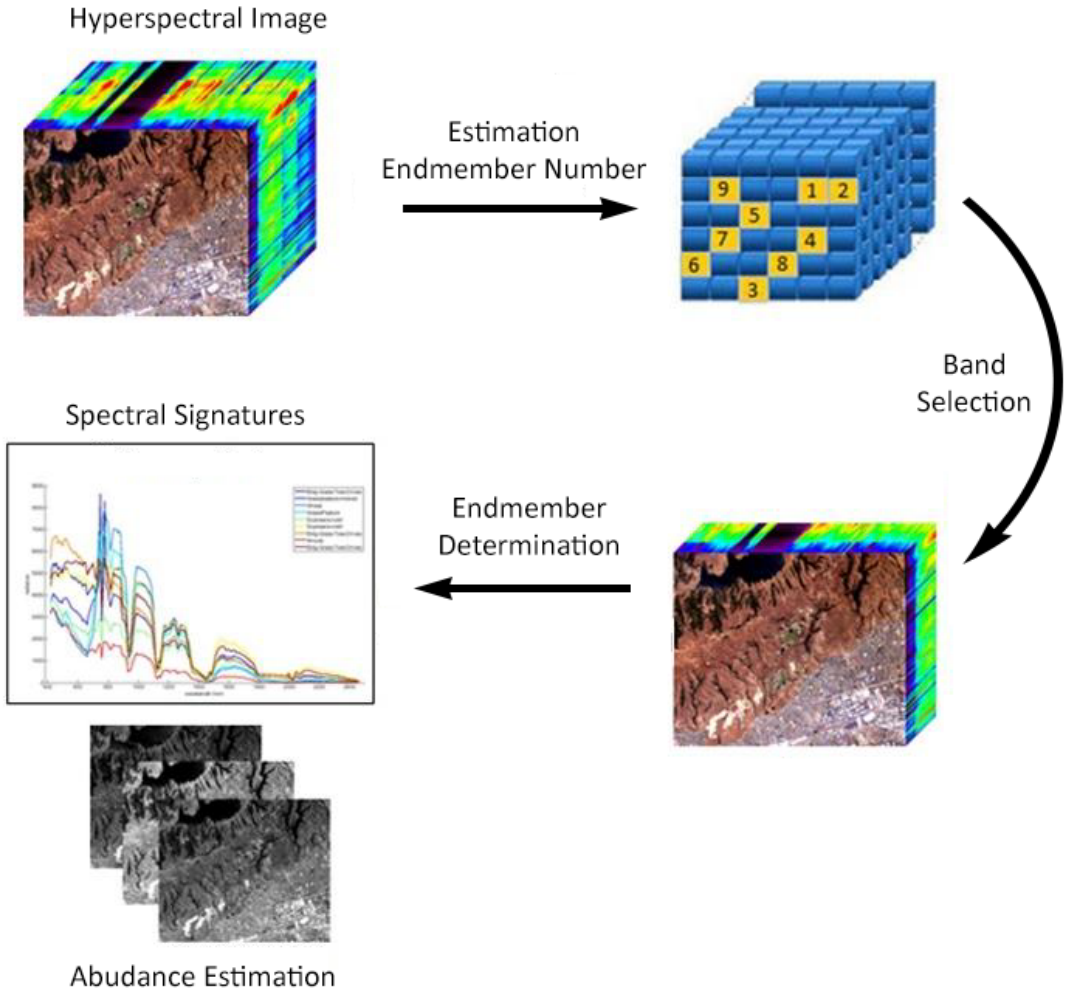


Figure 3.1: Hyperspectral unmixing chain block diagram

The first step is the extraction of the endmembers number from the hyperspectral image given in input. Several algorithms, such as Harsanyi–Farrand–Chang (HFC) [24] and Hyperspectral Subspace identification with minimum error (HySime) [25], can be used for this process part. After that, the *band selection* step extracts a new compact image by using the output of previous phase. By reducing the information redundancy, the computational complexity of the successive steps can be reduced.
The *Endember Determination* step identifies the pure materials present inside the scene. The

Vertex Component Analysis (VCA) algorithm can be exploited for this task [26]. This step finds, for each pure material inside the scene, the reflectance value for the considered bands and produces the *endmember matrix*.
This matrix is used in the final step (*Abudance Estimation*) in order to calculate the percentage with which each material is present inside each pixel (*mixing matrix*). This task can be performed such as through the Fully Constrained Least-Squares (FCLS) linear unmixing approach [27].
As pointed out before, the analysis of hyperspectral images should ends before five seconds in order to satisfy the real-time constraints. Thus, each step of the hyperspectral unmixing chain should be computed, as fast as possible.
This point of view should be kept in mind during the comparison of the different BS algorithm implementations in order to find the most suitable one.

3.2 Image Dataset

Before starting with a detailed description of each BS algorithm, a brief overview of the image dataset, used for testing and evaluating performance, is presented.
All the tests have been performed on a machine with the following characteristics:

- Processor: Intel i7 3770 (4 physical cores) running at 3.40 GHz
- RAM: 8 GB DDR3
- GPU: NVIDIA Tesla K40 (Kepler architecture), 2880 CUDA cores working at 875 MHz, 12 GB DDR5 DRAM

Each version of the algorithm has been developed through Visual Studio 2015 (64 bit Microsoft Windows 10) and CUDA version 8.0. Concerning the images, two different types were used: *synthetic* and *real*.
The synthetic images have been generated using the Matlab script developed by Nascimento et al. [25]. They have been used to profile and verify the correctness of the results i.e. the outputs of the different algorithm versions (Matlab, serial C, openMP and CUDA) had to be the same.
The synthetic dataset characteristics are shown in Table 3.1.
For each image name, the following characteristics are reported:

- *Bands*: total number of wavelengths used to acquire the scene
- *Rows*: number of pixel along x dimension
- *Columns*: number of pixel along y dimension
- *Pixels*: total number of pixel for each band
- *Endmembers*: number of material inside the scene
- *Size*: image size (expressed in MB) considering single floating point representation

Table 3.1: Synthetic image dataset

Image	Bands	Rows	Columns	Pixels	Endmembers	Size [MB]
p15N2k	224	50	40	2000	15	1.70
p15N10k	224	100	100	10000	15	8.54
p15N20k	224	100	200	20000	15	17.09
p15N50k	224	500	100	50000	15	42.72
p20N75k	224	750	100	75000	20	64.09
p20N100k	224	200	500	100000	20	85.45
p20N200k	224	500	400	200000	20	170.90
p25N500k	224	500	1000	500000	25	427.25

The real images used in our testing phase are a Hydice image collected over a forest (referred to as Hydice in the following), five AVIRIS scenes, two ROSIS images and one EO-1 scene. The five AVIRIS images are: the well-known Cuprite mining district, the World Trade Center, the Kennedy Space Center, the Indian Pines over the test site of Indiana and a Salinas Valley scenes (referred to in the following as Cuprite, WTC, KSC, Indian Pines and Salinas, respectively). The ROSIS images are acquired over the center of the Pavia city in Italy and over the Pavia University (referenced as PaviaC and PaviaU in the following). The EO-1 image was acquired over Okavango Delta, Botswana (referenced as Botswana in the following). The last two images are obtained by joining multiple Indian Pines images (indicated in the following as Indian Pine_EW and Indian Pine_NS).
The real dataset characteristics are shown in Table 3.2.

Table 3.2: Real image dataset

Image	Bands	Rows	Columns	Pixels	Endmembers	Size [MB]
Hydice	169	64	64	4096	18	2.64
Cuprite	188	250	191	47750	18	34.24
WTC	224	614	512	314368	23	268.63
PaviaC	102	2000	512	1024000	58	398.44
KSC	176	512	614	314368	25	211.06
Salinas	224	512	217	111104	22	94.94
Botswana	145	1476	256	377856	21	209.00
PaviaU	103	610	340	207400	35	81.49
Indian Pine	220	145	145	21025	19	17.64
Indian Pine_EW	220	1848	614	1134672	53	952.25
Indian Pine_NS	220	2678	614	1644292	59	1379.94

Each 3D image cube has been transformed into a 2D matrix, as shown in Figure 3.2, and stored in a text file with an *header* line containing four values which represents the bands, pixels, rows and columns of the considered hyperspectral image.

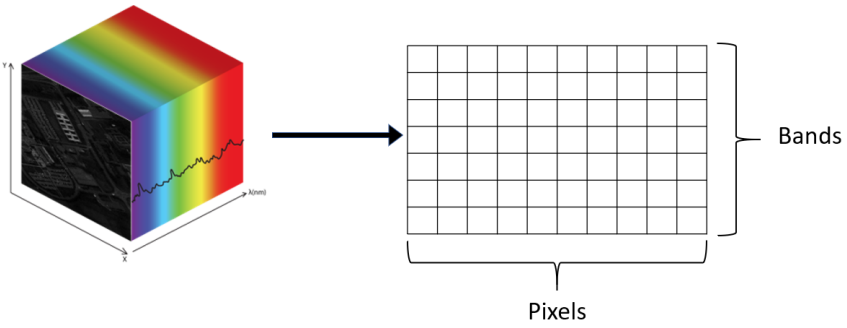


Figure 3.2: Stored 3D image cube as 2D matrix

3.3 Band Prioritization Algorithms

In this section each BP algorithm will be described in detail considering as input an hyperspectral image cube with L bands and N pixels.

3.3.1 SNR-based Criterion

The variance, evaluated for each hyperspectral band B_l, is one of the best known method to assign band priority score ρ as follows:

$$\rho(B_l) = \sigma_l^2 \qquad \text{for} \quad l = 1, \ldots, L \tag{3.2}$$

where σ_l^2 is the l-band variance.
In the past, band selection has been performed through an eigenanalysis without taking advantages of the relationship between eigenvalues and eigenvectors [16].
In [28], BP score started to be evaluated by using loading factors associated to each band. These loading factors were extracted from eigenvalues and eigenvector of spectral or eigendecomposition of a proper data matrix.
Thus, it is possible to derive an alternative formulation for Eq. 3.2, called *PCA-based BP*, using a set of loading factors ξ.
Let

$$\Sigma = \frac{1}{N} \sum_{k=1}^{N} (x_k - \mu)(x_k - \mu)^T \tag{3.3}$$

be the data sample covariance matrix where x_k is the k-th l-dimensional pixel vector and μ is the sample-mean vector. Since Σ is a symmetric nonnegative definite matrix, all its eigenvalues $\{\lambda_i\}_{i=1}^{L}$ are real and nonnegative and its associated l-dimensional eigenvectors $\{(v_{i1}, \ldots, v_{iL})^T\}_{i=1}^{L}$ can be chosen orthonormal.
The loading factor can be evaluated as follows:

$$\xi_{kl} = \sqrt{\lambda_k} v_{kl} \qquad \text{for} \quad k, l = 1, \ldots, L \tag{3.4}$$

The band priority score can be assigned to each band in this way:

$$\rho(B_l) = \sum_{k=1}^{L} \xi_{kl}^2 \qquad \text{for} \quad l = 1, \ldots, L \tag{3.5}$$

It can be proved that the score calculated with Eq. 3.2 and Eq. 3.5 is the same.
As pointed out in [29], variance is not an appropriate criterion for measuring bands quality. Because of that, the Maximum Noise Fraction (MNF) transformation, based on SNR criterion, was developed. This method has been later reinterpreted and called Noise-Adjusted Principal Component (NAPC) [30].
The idea of NAPC is to design a whitening matrix for the covariance one so that the noise covariance matrix becomes an identity one. In this way, the variance of each band can be interpreted as SNR.
In the NAPC approach, the observation model is the following:

$$z = s + n \tag{3.6}$$

where z is the observation vector with covariance matrix Σ_z, s a signal vector and n the noise vector independent from s with the covariance matrix Σ_n.
The estimated noise will be used to find the whitening matrix F in order to orthonormalize Σ_n so that:

$$F^T \Sigma_n F = I \qquad \text{and} \qquad F^T F = \Delta_n^{-1} \tag{3.7}$$

where Δ_n is the diagonal eigenvalues matrix of Σ_n.
Finally, the noise-adjusted sample covariance matrix is given by:

$$\Sigma_{adj} = F^T \Sigma_z F \tag{3.8}$$

As pointed out before, instead of using the variance to assign the band score, it is possible to resort to loading factors.
First of all, it is necessary to consider the set of eigenvalues of the noise-adjusted sample covariance matrix $\{\lambda_{adj,i}\}_{i=1}^{L}$ and their associated eigenvectors $\{v_{adj,i}\}_{i=1}^{L}$.
The loading factors for SNR-based method can be estimated as in Eq. 3.4:

$$\xi_{adj,kl} = \sqrt{\lambda_{adj,k}} v_{adj,kl} \qquad \text{for} \quad k, l = 1, \ldots, L \tag{3.9}$$

Using Eq. 3.9, the following band priority score can be evaluated:

$$\rho(B_l) = \sum_{k=1}^{L} \xi_{adj,kl}^2 \qquad \text{for} \quad l = 1, \ldots, L \tag{3.10}$$

It should be pointed out that PCA-based and SNR-based criterion are BP algorithms even if they are based on PCA which is a common DR technique [23].
The analysis of the SNR-based algorithm starts with the inspection of Matlab code provided by HyperComp laboratory, University of Extremadura (Spain).

The first code part concerns the reading of hyperspectral image.

```
1  % Read Hyperspectral Image
2  filename = 'image_name.txt';
3  delimiterIn = ' ';
4  headerlinesIn = 1;
5  data = importdata(filename, delimiterIn, headerlinesIn);
6
7  r = data.data;
8
9  in = zeros(1,4);
10 [token,remain] = strtok(data.textdata,' ');
11 for i=1:4
12   in(i) = str2double(token);
13   [token,remain] = strtok(remain,' ');
14 end
15
16 bnd = in(1,1);
17 pxl_no = in(1,2);
18 xx = in(1,3);
19 yy = in(1,4);
```

The variable *r* (line 7) is a 2D matrix which contains the hyperspectral image while *bnd* and *pxl_no* identify the number of bands and pixels, respectively.
The second step is the evaluation of the sample covariance matrix (Eq. 3.3).

```
20 % Sample Covariance Matrix
21 R = (r*r')/pxl_no;
22 u = mean(r,2);
23 K = R-u*u';
```

The variable *K* represents the sample covariance matrix. It should be underlined that this matrix has been evaluated with a different formula with respect to Eq. 3.3.
It can be proved that the result obtained is the same.

Proof. The following statement has to be proved:

$$\Sigma = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y}) = \frac{1}{N}(\sum_{i=1}^{N} x_i y_i - N \cdot \overline{xy})$$

Firstly, it is necessary to multiply out the brackets.

$$\sum_{i=1}^{N}(x_i-\overline{x})(y_i-\overline{y})=\sum_{i=1}^{N}(x_i y_i)-\sum_{i=1}^{N}(x_i\overline{y})-\sum_{i=1}^{N}(y_i\overline{x})+\overline{xy}\sum_{i=1}^{N}1$$

Since $\overline{x}$ and $\overline{y}$ are constants, they can be put in front of summations. Moreover, $\sum_{i=1}^{N}x_i=N\cdot\overline{x}$ (the same holds for y_i) and $\sum_{i=1}^{N}1=N$.

$$=\sum_{i=1}^{N}(x_i y_i)-N\cdot\overline{yx}\underbrace{-N\cdot\overline{xy}+\overline{xy}\cdot N}_{0}$$

In this way, it is demonstrated that:

$$\sum_{i=1}^{N}(x_i-\overline{x})(y_i-\overline{y})=\sum_{i=1}^{N}x_i y_i-N\cdot\overline{xy}$$

□

After that, the Δ_n matrix has to be estimated.

```
%=== Noise estimation ====%
K_Inverse=inv(K);
tuta=diag(K_Inverse);
K_noise=1./tuta;
K_noise=diag(K_noise);
```

The variable *K_noise* contains the diagonal eigenvalues matrix of Σ_n.

```
%==== Noise adjusted sample covariance matrix ===%
F=inv(sqrtm(K_noise));
K_adj = F'*K*F;
```

After having computed the whitening matrix *F*, the noise adjusted data covariance matrix (*K_adj*) can be evaluated as in Eq. 3.8.
In order to assign a score to each band, it is necessary to extract the eigenvalues and eigenvectors of *K_adj* matrix.

```
%=== Find eigenvector matrix of K_adj ===%
[V_adj,D_adj] = eig(K_adj);
for i = 1: size(V_adj, 2)
  norm = sqrt(V_adj(:, i)' * V_adj(:, i));
  V_adj(:, i) = V_adj(:, i)/norm;
```

```
37 end
```

The array *D_adj* contains the eigenvalues of Σ_{adj} while the matrix *V_adj* their associated normalized eigenvectors.
The final step is the evaluation of the loading factors to find the band priorities.

```
38 %=== Calculate priority ===%
39 band_priority = zeros(1, bnd);
40 for i = 1: bnd
41   sum = 0;
42   for j = 1: bnd
43     sum = sum + D_adj(j,j) * (V_adj(i,j) ^2);
44   end
45   band_priority(i) = sum;
46 end
```

The algorithm returns the array *band_priority* which contains the band priority scores. After having sorted the array, the bands to keep are chosen starting from the with the highest score. It is possible to see an example of the SNR-based criterion applied on the WTC hyperspectral image, shown in Figure 3.3.

Figure 3.3: WTC hyperspectral image

This image depicts the lower part of Manhattan and, through ENVI software, a different color intensity has been associated to each reflectance value acquired.
After having applied the SNR-based algorithm, the band priorities are obtained. In Figure 3.4, the comparison between two images in different bands is reported.
By inspecting the figure on the right, it is possible to observe that it is completely dark. Indeed, its band obtained a low priority score and so it can be discarded since it does not provide any

Figure 3.4: WTC images extracted from two different bands

additional information.
On the other hand, the band of the left picture obtained an high priority score and will be kept.
Before starting with the development of the algorithm exploiting parallel architectures, a profiling has to be performed in order to find the most time consuming part. For this reason, the first version developed is based on serial C programming language.
The algorithm starts with the reading of the hyperspectral image. Since the image has a bidimensional shape, it can ben stored inside a matrix structure, as it was done in Matlab.
In C language, the concept of matrix can be developed by using pointers to pointers. However, this method generates memory fragment problem and the memory accesses are not well optimized. For this reason, the hyperspectral image can be stored in a single array and each element can be accessed as follows:

$$matrix[i \cdot num\_columns + j]$$

where i and j indicate the element position while *num_columns* the total number of matrix columns.
In this way, the matrix is stored in continuous memory location avoiding memory fragmentation and exploiting optimized memory access.
In particular, the hyperspectral image has been stored in a C structure as follows:

```
1 struct hyperspectral_image{
2   int rows;
3   int columns;
4   int num_pixels;
5   int bands;
6   double *data;
7 };
```

The image is stored inside the array *data* while all image properties are represented by four integer variables.

The following step, evaluation of the covariance matrix (K), can be developed as a standard matrix-matrix multiplication.
A particular study has to be applied for the computation of the noise-adjusted data covariance matrix (K_adj). Indeed, it is necessary to develop a function which performs an inverse of a squared matrix. This task can be accomplished through *LU-Decomposition*.

LU-Decomposition

Computing the inverse of matrix $A \in \Re^{nxn}$ means finding a matrix $X \in \Re^{nxn}$ such that $AX = I$.
Let $X_{i,j}$ denote the j-th column of X, i.e. $X = (X_{:,1}, \ldots, X_{:,n})$.
The j-th column of the matrix-matrix product SX is given by the matrix-vector product $SX_{:,j}$, i.e. $SX = (SX_{:,1}, \ldots, SX_{:,n})$.
Let $e_j = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ be the j-th column of the identify matrix I.
Thus, it is possible to assert that $SX = (SX_{:,1}, \ldots, SX_{:,n}) = (e_1, \ldots, e_n) = I$. This implies that the columns $(X_{:,1}, \ldots, X_{:,n})$ of the inverse of A can be obtained by solving n systems of linear equations:

$$\begin{aligned} SX_{:,1} &= e_1 \\ &\vdots \\ SX_{:,n} &= e_n \end{aligned}$$

Notice that if the linear system $Ax = f$ has an unique solution x for every f, then there exists a unique X with $AX = I$.
The *LU-decomposition* means finding the matrices L and U such that:

$$A = LU$$

where L is a lower triangular matrix and U is an upper triangular matrix.
The *LU-decomposition* can be used to solve the linear system:

$$Ax = f$$

by replacing the A matrix with LU ones:

$$LUx = f$$

Applying the variable change y=Ux, two linear systems are obtained as follows:

$$Ly = f \qquad \text{and} \qquad Ux = y$$

Since L and U are lower and upper triangular matrices, the two systems can be easily solved.

The last non-trivial step is to compute the eigenvalues and eigenvectors of Σ_{adj} matrix. In

order to accomplish this task, the method described in [31] has been used.
The computation is performed by two subroutines: *tred2* an *tqli*. The first one performs the Householder algorithm reducing the *nxn* symmetric input matrix to tridiagonal form. This matrix is given as input to the *tqli* function, which performs a QL algorithm with implicit shifts for determining eigenvalues and eigenvectors.
The combination of *tred2* and *tqli* function is a very efficient technique for finding the eigenvalues and eigenvectors of a real symmetric matrix.
Once the algorithm has been completely developed through C programming language, the image dataset has been used to prove the implementation correctness. In particular, it has been necessary to adopt the double floating point representation in order to have same results between Matlab and C versions.
After having proved its correctness, it is possible to profile each step using the synthetic image dataset in order to identify the most time consuming part. The profiling results are shown in Table 3.3.

Table 3.3: Profiling results of SNR-based algorithm

Step	Time occupancy (%)
R matrix computation	99%
Other	1%

Thanks to the profiling, it is possible to understand that the computation of *R* matrix involves the 99% of the total execution time.
This result is not surprising. Indeed, that step involves a matrix product of the whole hyperspectral image. Since the matrix dimension is [*Band, Num_Pixel*], the multiplication implies an high computational complexity.
On the other hand, all the successive steps are based on the *R* matrix, whose dimension is [*Band, Band*], and have low computational complexity. Indeed, as shown in Table 3.1 and Table 3.2, the number of bands is greatly lower than the number of pixels. This property justifies the low computational complexity of all the other steps of the algorithm.
The first parallel version, which has been developed, is based on the OpenMP API. The portion of the code, which has been parallelized, is shown below:

```
1 /* Compute R Matrix */
2 #pragma omp parallel private(i, j, k, acc) shared(img,R_matrix)
3 {
4   for(i = 0; i<(img->bands); i++){
5     #pragma omp for schedule(static)
6     for(j = 0; j<(img->bands); j++){
7       acc = 0;
8       for(k = 0; k<(img->num_pixels); k++){
```

```
 9            acc += img->data[i*(img->num_pixels)+k] * img->data[j*(img->
       num_pixels)+k];
10          }
11          R_matrix[i*(img->bands)+j]= (double)(acc/(img->num_pixels));
12        }
13     }
14  }
```

R_matrix is an array that will contain the results of the matrix product.
The first *#pragma* directive is used to open the parallel region. After this statement, there will be four threads each one with its private copy of the loops variables (*i, j and k*) and *acc*, which is used as accumulator during the multiplication.
The *shared* variables are the input hyperspectral image (*img*) and the output matrix (*R_matrix*) since they must be read and written by all the threads, respectively. Concerning the write operation, it could happen that different threads access the same memory location generating a data race condition. However, for the considered case each output element will be computed by just one thread so that the conflicts are avoided.
This problem does not happen during read operation since the data will be broadcast to all the threads.
The second *#pragma* directive is used to split the work among the threads in an equitably way since the clause *schedule(static)* has been inserted.
As it was done for the serial C version, the correctness of the OpenMP algorithm version has been verified on the entire data set.
The last parallel implementation exploits the GPU technology. The GPU algorithm flow is reported in Figure 3.5.
Since the most time consuming part is the computation of *R_matrix*, this task has been assigned to the GPU that, after having received the hyperspectral image from the host, performs the matrix multiplication and transfers back the result to the CPU.
In particular, the *cuBLAS* library, which contains different highly optimized linear algebra routines, has been used to perform the matrix multiplication through the *cublasDgemm* function that evaluates the following expression:

$$C = \alpha op(A) op(B) + \beta op(C) \tag{3.11}$$

where A, B are input matrices, C is an input/output matrix and α and β are scalar values.
In the *cuBLAS* library, the matrices are read and stored by default in column-major format while the CUDA C language, chosen for the SNR-BS implementation, adopts the row-major format which is the standard array storing method in C language.
In order to tackle this problem, matrices can be read as if they were transposed by properly setting the *op()* parameter in Eq. 3.11. However, the result of an operation is stored in column-major format and this cannot be modified.
In this case, since the correlation matrix is symmetric, the output C matrix can be transferred

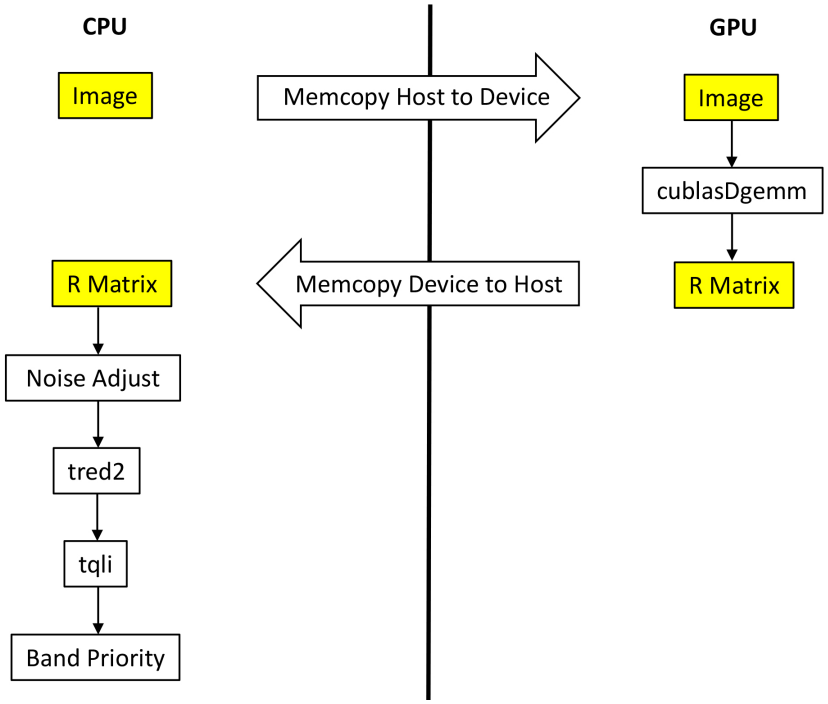


Figure 3.5: SNR-based algorithm flow. Data and operations are in yellow and white boxes, respectively [32].

back to the host and read as if it was stored in row-major format.
If the output matrix is not symmetric, it is possible to exploit the property of matrix product:

$$C = AB$$
$$C^T = (AB)^T = B^T A^T$$

By using *cuBLAS* library to computer C^T, the result will be stored in column-major order. The output, read in row-major order, will be equal to C matrix.
Concerning the choice of the two scalar values, α was set equal to the inverse of the number of pixels of the hyperspectral image while β was set equal to 0 since the C matrix is only an output matrix.
The R matrix is transferred back to the host memory since its dimensions are significantly lower than the hyperspectral image ones. Thus, the subsequent operations can be carried out in a sequential way.
In order to develop the GPU part of the presented flow, the following code has been used.

```
1 /* Compute R Matrix */
2
3 cudaMalloc((void**)&d_R,memsize);
4 cudaMalloc((void**)&d_img_data, memsizeC);
5
```

```
6 cudaMemcpy(d_img_data, img->data, memsizeC, cudaMemcpyHostToDevice);
7
8 cublasDgemm(*handle, CUBLAS_OP_T, CUBLAS_OP_N, img->bands, img->
      bands, img->num_pixels, &alpha, d_img_data, img->num_pixels,
      d_img_data, img->num_pixels, &beta, d_R, img->bands);
9
10 cudaMemcpy(R, d_R, memsize, cudaMemcpyDeviceToHost);
11
12 cudaFree(d_R);
13 cudaFree(d_img_data);
```

The device memory is allocated through the *cudaMalloc()* function. This memory is used to store the hyperspectral input image, copied from the host through *cudaMemcpy()* function, and the output matrix (*R*).
Then the matrix operation is performed through *cublasDgemm* function and the result is copied back to the host.
Finally, the memory is released through the *cudaFree()* function.
Even in this case, the GPU version of the SNR-based algorithm has been tested using the entire data set.
The processing times of the three algorithm versions using the synthetic and real image dataset is reported in Table 3.4 and Table 3.5, respectively.

Table 3.4: SNR-based algorithm processing time of serial, OpenMP and CUDA version using synthetic images dataset

Images	Size [MB]	Serial [s]	OpenMP [s]	CUDA [s]
p15N2k	1.70	0.114	0.068	0.047
p15N10k	8.54	0.421	0.253	0.052
p15N20k	17.08	0.79	0.469	0.057
p15N50k	42.72	1.914	1.086	0.078
p20N75k	64.08	2.839	1.586	0.098
p20N100k	85.44	3.765	2.131	0.124
p20N200k	170.89	7.504	4.207	0.213
p25N500k	427.24	19.271	11.492	0.473

In order to obtain a better comprehension of the different versions performance, it is possible to plot the processing times using the logarithmic scale as shown in Figure 3.6.
By analyzing the result of the SNR-based algorithm, it is possible to notice that CUDA version performs better than the other ones for each hyperspectral image.
Considering the WTC image, which is acquired in 5 sec, only the CUDA version is able to satisfy the real-time constraint while this does not happen in the serial and OpenMP versions.

Table 3.5: SNR-based algorithm processing times of serial, OpenMP and CUDA version using real images dataset

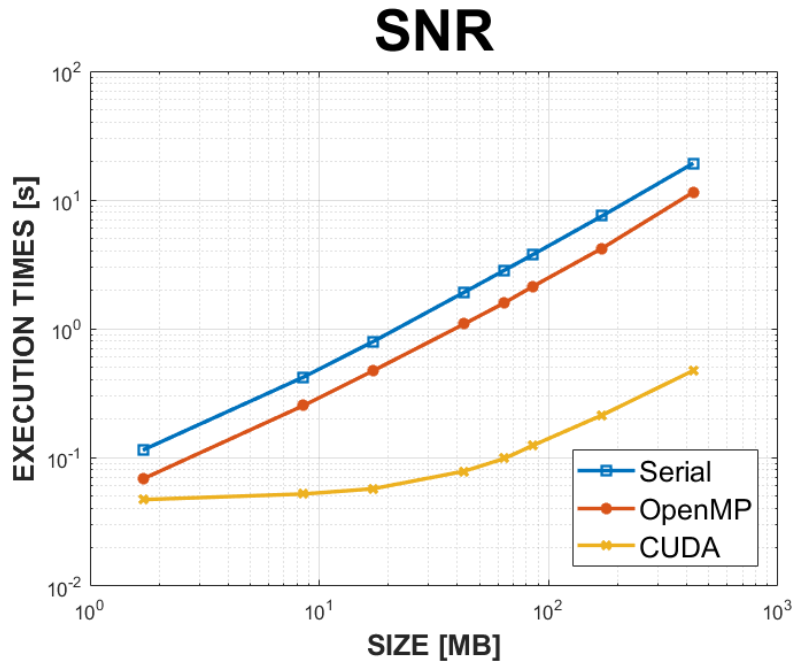
Images	Size [MB]	Serial [s]	OpenMP [s]	CUDA [s]
Hydice	2.64	0.101	0.054	0.017
Indian_Pines	17.64	0.835	0.501	0.047
Cuprite	34.24	1.36	0.753	0.054
PaviaU	81.49	1.784	0.991	0.079
Salinas	94.94	4.504	2.457	0.138
Botswana	209.00	7.164	3.774	0.213
KSC	211.06	8.718	4.435	0.224
WTC	268.63	14.077	7.205	0.310
PaviaC	398.44	11.006	5.214	0.366
Indian_Pines_EW	952.25	49.657	26.398	1.859
Indian_Pines_NS	1379.94	73.644	38.693	4.550

Another important characteristic is the lack of linear relation between the real hyperspectral image sizes and execution times (Figure 3.6b). Indeed, the time spent on the evaluation of R matrix is strictly dependent on the input hyperspectral image geometry (number of bands and pixels) which changes for each real image.
On the other hand, the synthetic hyperspectral images has the same number of bands and an increasing number of pixels. For this reason, a linear relation between the image size and execution time can be observed in Figure 3.6a. Concerning the CUDA version, it is possible to underline that for small images the GPU computational power is not completely exploited. Thus, since the execution time for small images is nearly the same, the first part of CUDA plot is approximately flat.
Furthermore, the number of bands influences the complexity of the remaining operations of the algorithm. Hyperspectral images with a lower number of bands have a lower computational weight on the operations that follow the correlation matrix computation.

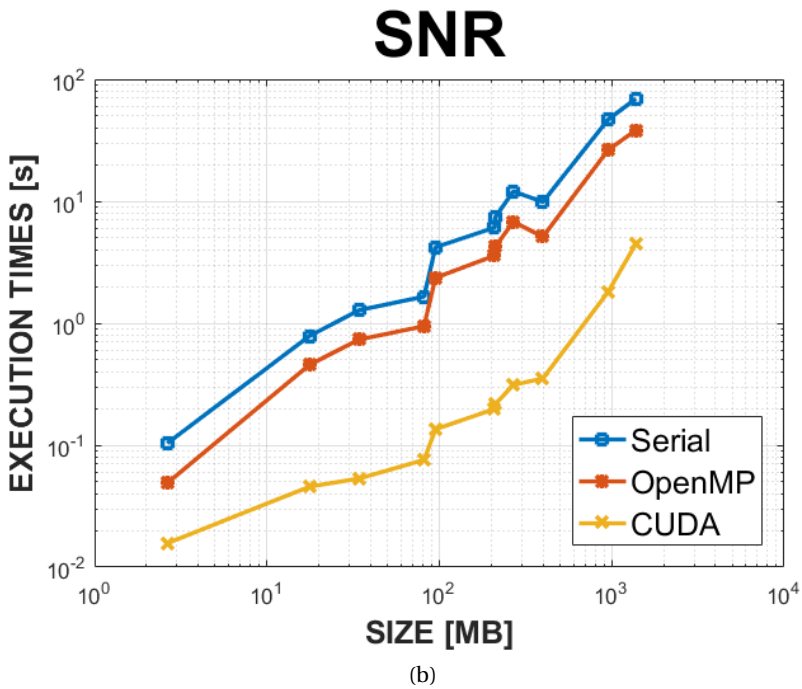
3.3.2 ID-based Criterion

The ID-based criterion is based on the concept of *relative-entropy* also called *Kullback-Leibler distance*.
The *relative-entropy* is a measure of the distance between two distributions [33]. To be more precise, the *relative-entropy D(p||q)* measures the inefficiency of considering that the data distribution is q when the true one is p.
The *relative-entropy* is defined as:

$$D(p||q) = \sum_{x \in X} p(x) log\left(\frac{p(x)}{q(x)}\right) \tag{3.12}$$



(a)



(b)

Figure 3.6: Processing time plot on logarithmic scale: (a) synthetic image dataset (b) real image dataset [32].

where *p(x)* and *q(x)* are two probabilities mass functions.
In Eq. 3.12, the following conventions have been adopted:

- $0\,log\left(\frac{0}{0}\right) = 0$
- $0\,log\left(\frac{0}{q}\right) = 0$
- $p\,log\left(\frac{p}{0}\right) = \infty$

An important property of the *relative-entropy* is reported below:

$$D(p||q) \geq 0 \tag{3.13}$$

with equality if and only if:

$$p(x) = q(x) \qquad \forall x \in X \tag{3.14}$$

Proof. In order to verify this property, it is necessary to introduce the *Jensen's inequality*:

$$E[f(X)] \geq f(E[X]) \tag{3.15}$$

where f is a convex function and X is a random variable.
Considering $A = \{x : p(x) > 0\}$ as the support set of $p(x)$, it is possible to obtain:

$$-D(p||q) = -\sum_{x\in A} p(x)\,log\left(\frac{p(x)}{q(x)}\right) \tag{3.16}$$

$$= \sum_{x\in A} p(x)\,log\left(\frac{q(x)}{p(x)}\right) \tag{3.17}$$

$$\leq log\left(\sum_{x\in A} p(x)\,\frac{q(x)}{p(x)}\right) \tag{3.18}$$

$$= log\left(\sum_{x\in A} q(x)\right) \tag{3.19}$$

$$\leq log\left(\sum_{x\in X} q(x)\right) \tag{3.20}$$

$$= log(1) \tag{3.21}$$

$$= 0 \tag{3.22}$$

Since $log(x)$ is a strictly concave function of x, the equality condition of Eq. 3.18, derived from Jensen's inequality, is satisfied if and only if $\frac{q(x)}{p(x)}$ is constant for all x. This means that:

$$q(x) = c\,p(x)$$

$$\sum_{x\in A} q(x) = c \sum_{x\in A} p(x) = c$$

The equality in Eq. 3.20 is satisfied if and only if:

$$\sum_{x\in A} q(x) = \sum_{x\in X} q(x) = 1$$

This implies that:

$$\sum_{x \in A} q(x) = c = 1$$

In this way, it has also been proved that:

$$D(p||q) = 0 \qquad \text{if and only if } \; p(x) = q(x) \quad \forall x$$

□

It should be pointed out that the *relative entropy* is not a true distance between distributions since it does not satisfy the triangle inequality and is not symmetric.
In [34], the *relative entropy* has been adopted to define a symmetric hyperspectral measure, called *Spectral Information Divergence* (SID), which is used to evaluate the spectral similarity between two pixels.
Considering an hyperspectral pixel $x = (x_1, \ldots, x_L)^T$, each component x_l is the reflectance value in the band B_l.
A probability vector $p = (p_1, \ldots, p_L)^T$ can be assigned to the pixel x and each component can be evaluated as follows:

$$p_j = \frac{x_j}{\sum_{l=1}^{L} x_l} \tag{3.23}$$

Considering another hyperspectral pixel y with its probability vector q, the relative entropy of y with respect to x can evaluated as:

$$D(x||y) = \sum_{l=1}^{L} p_l \, log\left(\frac{p_l}{q_l}\right) \tag{3.24}$$

The SID between the two pixels can be defined as:

$$SID(x, y) = D(x||y) + D(y||x) \tag{3.25}$$

The ID-based criterion is based on the same basics, but it is focused on finding the similarity between bands instead of pixels.
Let p_l be the image histogram of band B_l, transformed in a probability distribution, and g_l be the associated Gaussian distribution with mean and variance equal to the sample mean and sample variance of B_l. The band priority can be computed as:

$$\begin{aligned} \rho(B_l) &= D(p||g) + D(g||p) \\ &= \sum_{i=1}^{NBINS} p_{li} \, log\left(\frac{p_{li}}{q_{li}}\right) + \sum_{i=1}^{NBINS} q_{li} \, log\left(\frac{q_{li}}{p_{li}}\right) \end{aligned} \tag{3.26}$$

The ID criterion measures the non-Gaussianity of each band. Indeed, the higher the value of $\rho(B_l)$, the greater the distance between p_l from the Gaussian distribution g_l.

The analysis of ID-based criterion has been started from the Matlab version provided by HyperComp laboratory, University of Extremadura (Spain).
The reading of the input hyperspectral image is done in the same way as it was done in SNR-based criterion.

```
%==== Input variables ===%
data = img;
pxl_no = size(data,2);
bnd = size(data,1);
info_priority = zeros(1, bnd);
```

The matrix *data* contains the input hyperspectral image, *pxl_no* and *bnd* are the total number of pixels and bands, respectively, and *info_priority* is an array which contains the band priorities.
Then, for each band it is possible to evaluate the band priority as follows:

```
for i = 1: bnd
  test = uint16(data(i,:) * 255) + 1;
  symbol = [min(test(:)) : max(test(:))];

  P = hist(single(test), single(symbol));
  P = P/(pxl_no);

  % Generate associated gaussian distribution
  sample_mean = mean(single(test(:))-1);
  sample_std = std(single(test(:))-1);

  P_gaussian = zeros(1, length(symbol));
  for j = 1: length(symbol)
    P_gaussian(j) = exp(-(single(symbol(j)-1) - sample_mean)^2/(2 *
        sample_std * sample_std))/(sample_std * sqrt(2*pi));
  end

  % Calculate band priority
  D1 = 0;
  for k = 1: length(symbol)
    if ((P_gaussian(k) ~= 0) && (P(k) ~= 0))
      temp1 = log2(P(k)/P_gaussian(k));
      temp1 = P(k) * temp1;
      D1 = D1 + temp1;
    end
```

```
30    end
31
32    D2 = 0;
33    for k = 1: length(symbol)
34      if ((P_gaussian(k) ~= 0) && (P(k) ~= 0))
35        temp2 = log2(P_gaussian(k)/P(k));
36        temp2 = P_gaussian(k) * temp2;
37        D2 = D2 + temp2;
38      end
39    end
40    info_priority(i) = D1 + D2;
41  end
```

Let's consider for instance the band B_j. The first step is the extraction of the j-th *data* matrix row. Since the reflectance values usually range from 0 to 1, it is necessary to scale the data range (line 7) in order to fill the bins (256 in the example) of the band histogram.
After that, the band histogram is evaluated (line 10) and normalized in order to extract a probability distribution (line 11).
The next step is the computation of the Gaussian distribution (line 19) by applying the following formula:

$$g(x) = \frac{1}{\bar{\sigma}\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(x-\bar{\mu})^2}{\bar{\sigma}^2}\right)} \tag{3.27}$$

where $\bar{\mu}$ and $\bar{\sigma}^2$ are sample mean and variance, respectively.
The last step is the band priority computation using the Eq. 3.26. The result is stored inside the *info_priority* array which will be used to select the bands to keep.
Starting from the Matlab code, a new algorithm version has been developed exploiting the serial C programming language and its correctness has been verified.
Then, the complete application profiling, performed using the synthetic image dataset, showed that there is not a part of the algorithm which is more computationally intensive than other ones. This algorithm property has to be taken into account during the development of the parallel versions.
Concerning the OpenMP API, the following code has been adopted:

```
1 #pragma omp parallel for private(i, max, min, std, mean, var, P,
      P_Gaussian, D1, D2) shared(img, band_priority) schedule(static)
2 for (i = 0; i < img->bands; i++){
3
4   /* COMPUTE BAND PRIORITY */
5
6 }
```

Since the band priority computation is based on small simple steps, each single thread will be in charge of computing the priority for the assigned bands.
The hyperspectral image has been stored in a C structure variable (*img*) and declared as *shared* since it has to be read by all the threads.
All the other variables, necessary for the evaluation of the band priority, have been declared as private.
The *for* loop over the total number of bands have been equally subdivided among the threads using the clause *schedule(static)*.
Finally, the GPU technology has been exploited. Since in the ID-based criterion no portion of the code is more computationally expensive than others, the entire algorithm has been developed on the GPU. Moreover, by analyzing Eq. 3.26, it is possible to notice that the priority of each band is evaluated considering every time a different row of the hyperspectral image. This means that the memory transfer and device computation can be overlapped by using *streams*.
The ID-based algorithm flow for each stream is reported in Figure 3.7.

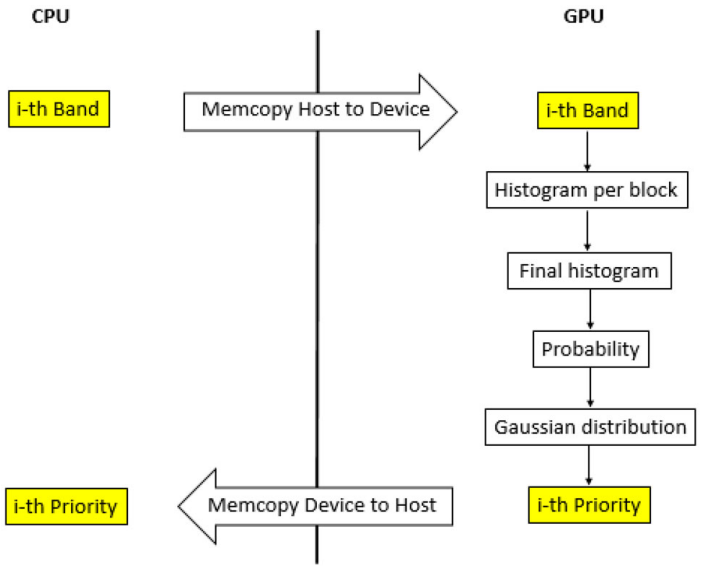


Figure 3.7: ID-based algorithm flow for a single stream. Data and operations are in yellow and white boxes, respectively [32].

Considering the *i*-th band, the first step is the evaluation of the image histogram.
The *i*-th row of the input hyperspectral image, which contains the pixel values of the considered band, must be transferred to the GPU global memory.
Then, each thread has to read a single pixel element and, depending on its value, it has to update the count of the corresponding histogram bin.
However, it could happen that several threads have to update concurrently the same bin value. This memory concurrency situation could generate wrong results that may differ from one execution to another one.
To tackle this problem, it must be guaranteed that only one thread per time can access a

precise memory location while the other ones have to wait until the operation is completed.
The CUDA programming model provides a series of functions, called *atomic*, which threat this kind of operations both in global and shared memory avoiding memory concurrency. It should be mentioned that *atomic* operations reduce the overall performance. Indeed, since the access of different threads to the same memory location cannot be done in parallel, it will be serialized. However, this is the only way to obtain correct results.
In particular, the *atomicAdd()* function has been adopted for the evaluation of the image histogram.
In a first attempt, the image histogram could be stored in an array inside the global memory. All the kernel threads can access the array and update the bins through *atomic* operations. In this case, the continue access to the global memory causes an high latency.
Thus, the idea is to exploit the low latency shared memory by splitting the construction of the histogram in two steps.
In the first one, each thread-block reads a sub-part of the input image from the global memory and creates, through atomic operations, a temporary histogram in the shared memory. Even in this case, different threads cannot update a memory location at the same time, but it is no longer necessary to deal with the high latency of the global memory.
Then, all the temporary histograms are stored inside the global memory in order to be used for the next phase.
In the second step, another kernel is launched with the aim of summing all the values present in the temporary arrays producing the final histogram.
An example of the described two-phase procedure is illustrated in Figure 3.8.

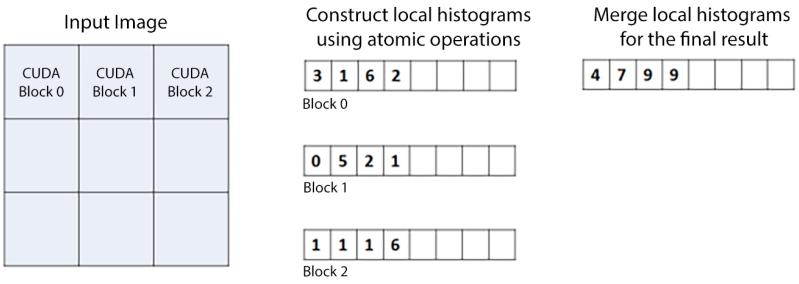


Figure 3.8: Two phase procedure for parallel histogram evaluation

Another tricky step is the development of the summation required by Eq. 3.26 for the band priority computation.
It is necessary to find a technique which exploits the GPU parallelism to find the summation of values present in array stored in the device global memory. This technique is called *reduction* and it is based on two successive kernel calls.
In the first kernel call, all the threads of each block store a sub-part of the original array inside the shared memory by loading a single element from the global memory.
Then, cyclically, half of the total available threads will be active and each of them will perform the sum of an element-pair until a single value will be obtained.
Finally, only one thread per block will write the summation of the sub-array, loaded by the

block, inside a new array allocated in the global memory.
This new array will be given in input to the second kernel that, using the same procedure as before, will evaluate the sum of the elements contained in the array producing the final result. An example of *reduction* technique is reported in Figure 3.9.

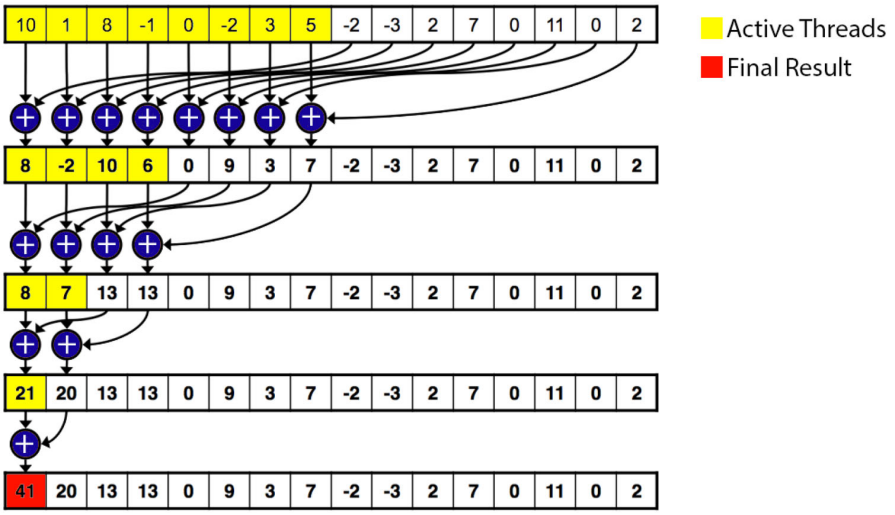


Figure 3.9: Example of *reduction* technique [32].

As said before, since the priority of each band is assigned in an independent way, it is possible to exploit the GPU *streams*. Operations in different streams can run concurrently which means that several kernel execution can be overlapped with data transfer from or to the device memory. In this way, while one stream is computing the band priority, other ones are loading new data inside the device memory and transferring back to the host the obtained results.
The overlapping memory transfers and kernel execution can be visualized, as shown in Figure 3.10, using the NVIDIA Visual Profiler tool.

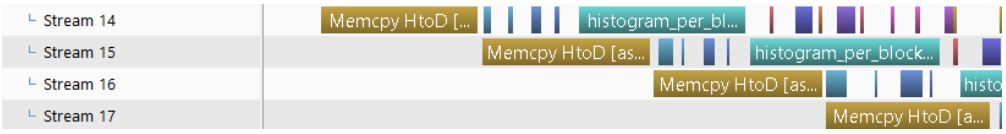


Figure 3.10: NVIDIA Visual Profiler output. Brown boxes represent memory transfers from host to device, while the other boxes concern different kernel executions.

As shown in Figure 3.10, the most time consuming part of the GPU version is the first part of the histogram computation (*Histogram per block* in Figure 3.7) which takes around 90 % of the total execution time. As expected, this low performance is due to the *atomic* operations which serialized the threads memory accesses. Moreover, the execution time of this function is not constant but it is strictly dependent on pixel values of the input hyperspectral image.
The processing times of the three algorithm versions using the synthetic and real image dataset are reported in Table 3.6 and Table 3.7, respectively.
The results shows that the ID-based criterion is not heavy from the computational point of

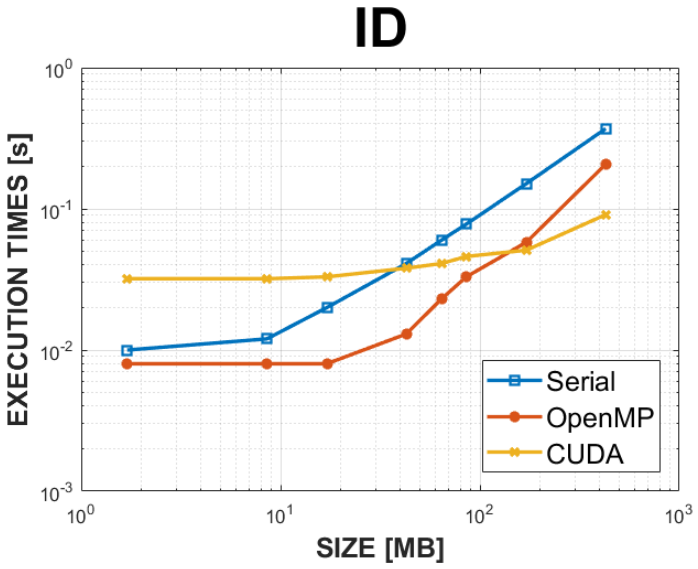
Table 3.6: ID-based algorithm processing time of serial, OpenMP and CUDA version using synthetic images dataset

Images	Size [MB]	Serial [s]	OpenMP [s]	CUDA [s]
p15N2k	1.70	0.01	0.008	0.032
p15N10k	8.54	0.012	0.008	0.032
p15N20k	17.08	0.02	0.008	0.033
p15N50k	42.72	0.041	0.013	0.038
p20N75k	64.08	0.06	0.023	0.041
p20N100k	85.44	0.078	0.033	0.046
p20N200k	170.89	0.151	0.058	0.051
p25N500k	427.24	0.37	0.207	0.091

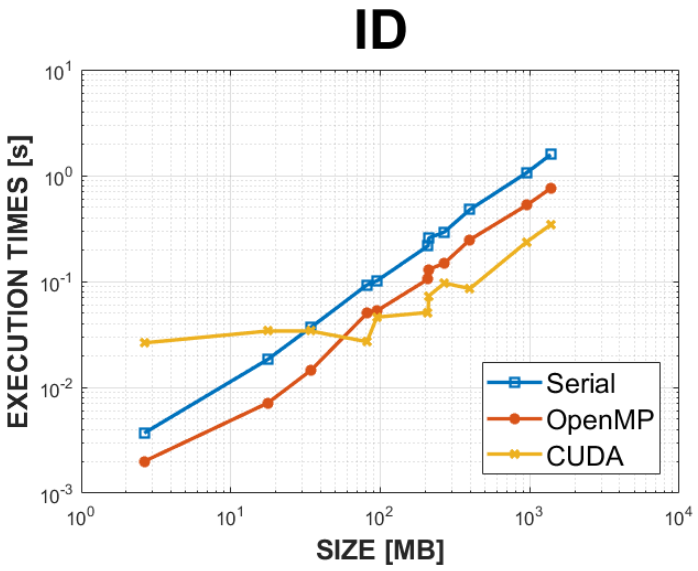
Table 3.7: ID-based algorithm processing time of serial, OpenMP and CUDA version using real images dataset

Images	Size [MB]	Serial [s]	OpenMP [s]	CUDA [s]
Hydice	2.64	0.003	0.002	0.026
Indian_Pines	17.64	0.018	0.007	0.034
Cuprite	34.24	0.036	0.014	0.034
PaviaU	81.49	0.092	0.05	0.027
Salinas	94.93	0.101	0.052	0.046
Botswana	209.00	0.217	0.105	0.051
KSC	211.06	0.257	0.13	0.073
WTC	268.62	0.293	0.148	0.096
PaviaC	398.43	0.478	0.248	0.085
Indian_Pines_EW	952.25	1.062	0.524	0.236
Indian_Pines_NS	1379.94	1.58	0.757	0.343

view. Indeed, even the C serial version is able to extract the band priority for the biggest hyperspectral image in a time below 5 sec.
However, it is important to underline that the BS is a only step of the whole hyperspectral unmixing chain. Thus, it would be better to spend the less possible time in order to perform the entire hyperspectral image analysis satisfying the real-time constraint.
As it was done for the SNR-based criterion, it is possible to plot the processing times using the logarithmic scale as shown in Figure 3.11.
In this case, the CUDA version is not always able to provide the best performance and this is due to different reasons.
The ID-based criterion does not present computationally intensive parts and so even solution based on C serial or OpenMP can achieve good performance.
On the other hand, the evaluation of the image histogram has an high impact on the GPU



(a)



(b)

Figure 3.11: Processing time plot on logarithmic scale: (a) synthetic image dataset (b) real image dataset [32].

execution performance. Indeed, this algorithm part is the one which takes more time since it is not possible to exploit the parallel execution but the memory accesses have to be serialized. Moreover, the GPU computation time of the histogram is related to the pixels values of the input hyperspectral image. For this reason, the relationship between execution time and image size is not linear as it happens for the serial C and OpenMP versions.
To the best of author's knowledge, those are the first parallel implementations of the algorithms,

therefore, it is not possible to perform direct comparisons with other works in literature.

4 Virtual Dimensionality Estimation for Hyperspectral Images

The aim of BS is to represent the input hyperspectral image in a lower dimensional space. This is a very important aspect due to the benefits that can be obtained in data storage, processing time and computational complexity.
Since it is necessary to avoid any loss of data information, the choice of the bands number to preserve is crucial.
This issue is tackled using the number of pure materials present inside the scene for the reduction of the hyperspectral image dimension. Indeed, typically the number of endmembers is significantly lower than the number of bands [35]. For this reason, the BS step of the hyperspectral unmixing chain receives as input the estimation of endmembers number.
As said before, this estimation can be performed exploiting the concept of *VD*. This term was introduced in [36] and is defined as the number of spectrally distinct signatures present in a given scene.
Several techniques have been developed to estimate VD such as *HFC* method together with its *noise-whitened HFC* (NWHFC) version [37], *Signal Subspace Estimate* (SSE) [38], *Hyperspectral Signal Identification by Minimum Error* (HySime), *Maximal Orthogonal Complement Algorithm* (MOCA) [39], *Linear Spectral Mixture Analysis* (LSMA)-based methods [40], *Maximum Orthogonal Subspace Projection* (MOSP) [41], etc.
These techniques develop the concept of VD in context of Neyman-Pearson (NP) detection theory in which the spectrally distinct signatures are considered as signal sources under binary hypotheses. These hypotheses can be defined using three aspects: *eigen-analysis* with eigenvalues or eigen/singular vectors, *Linear Mixture Model* (LMM) fitting error analysis and *Inter-Band Spectral Information* (IBSI) analysis.
A complete schema of VD estimation techniques is reported in Figure 4.1.
The *Intrinsic Dimensionality* (IDIM) is another well-known concept which has been defined as the minimal number of parameters to characterize data [42]. Of course, there are several differences between the IDIM and VD concepts. First, VD uses a NP detection approach while ID is based on an eigenanalysis.
Second, the IDIM value is a single fixed number since it is completely determined by data. In this way the IDIM value is independent from the application.

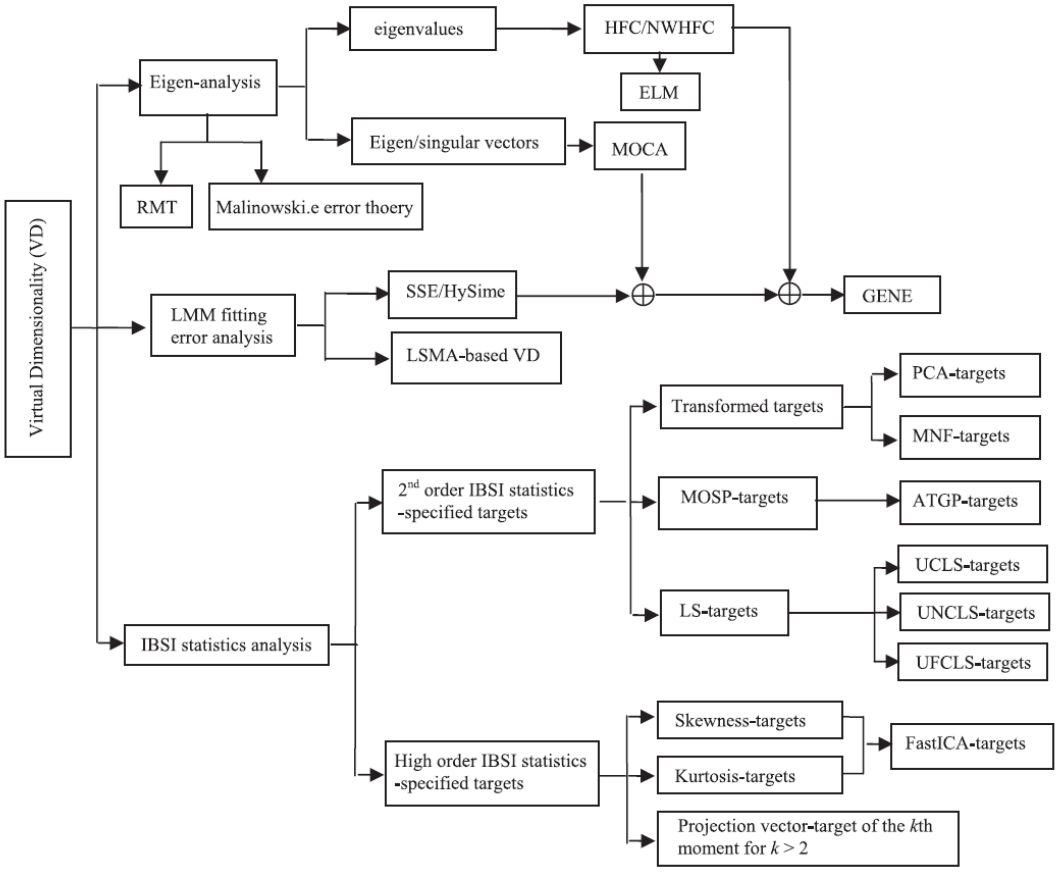


Figure 4.1: Schema of different VD estimation algorithms [22].

On the other hand, the VD value changes with the false alarm probability and is determined by signal sources under hypothesis specified by eigenvalues, eigenvectors or targets generated by different applications.
Third, since IDIM is based on eigenanalysis, it uses only 2-order statistic while targets, used as signal source for VD, can be characterized by high order statistics.
Finally, IDIM usually requires a noise estimation and Gaussian assumption which is generally not true in hyperspectral images. Instead, VD uses targets specified by applications as signal sources and adopts the NP detector to determine if the signal sources are true target signals or noise.
Thus, considering the selection of signal source applicable to a given application, VD works better than IDIM in hyperspectral data exploitation.
The VD value, obtained with the presented techniques, can be threated in several applications as the number of endmembers, the number of dimensions to keep after dimensionality reduction, the number of bands to be selected for band selection, etc.
In particular, the HFC method has been deeply studied and developed exploiting several different architectures.

4.1 Harsanyi–Farrand–Chang (HFC) Algorithm

A commonly used approach to characterize data is to perform an eigenanalysis computing the eigenvalues and eigenvectors of the covariance and correlations matrices both for data and noise.
Instead of dividing the noise and signals eigenvalues in two sets as it is done in several methods, the HFC algorithm considers the signal-noise decomposition step as a binary hypothesis tesing problem. In this way, it is possible to use the *Neyman-Pearson Detectors* (NPD) for determining how many signal sources is present in the data, subject to a false alarm probability.
The HFC method does not make any assumption about LMM or Gaussianity but it requires that the hyperspectral signatures are deterministic signal sources and the noise is white with zero mean.
Let denote a hyperspectral pixel with $x = [r^1, \ldots, r^L]^T$ where r^i is the reflectance value acquired on the i-th band. The hyperspectral image is given by $X = [x^1, \ldots, x^N]$ where N is the number of pixels.
The first step of HFC algorithm is the computation of sample data covariance (K) and correlation (R) matrices.
The K matrix is computed as:

$$K = \frac{(X - \bar{X})\,(X - \bar{X})^T}{N - 1} \tag{4.1}$$

where $\bar{X}$ is the sample mean vector.
The R matrix is evaluated as follows:

$$R = \frac{X\,X^T}{N} \tag{4.2}$$

Let $\hat{\lambda_1} \geq \ldots \geq \hat{\lambda_L}$ and $\lambda_1 \geq \ldots \geq \lambda_L$ be the eigenvalues set of R and K matrices, respectively.
Assuming that the signal sources have positive energies and indicating the noise variance in the l-th band as $\sigma^2_{n_l}$, the following statements hold:

$$\hat{\lambda_l} - \lambda_l > 0 \qquad \text{for} \quad l = 1, \ldots, VD \tag{4.3}$$

$$\hat{\lambda_l} - \lambda_l = 0 \qquad \text{for} \quad l = VD + 1, \ldots, L \tag{4.4}$$

where

$$\hat{\lambda_l} = \lambda_l + \sigma^2_{n_l} \qquad \text{for} \quad l = 1, \ldots, VD$$

$$\hat{\lambda_l} = \lambda_l = \sigma^2_{n_l} \qquad \text{for} \quad l = VD + 1, \ldots, L$$

In order to determine VD, the problem outlined by Eq. 4.3 and 4.4 can be formulated as a binary hypothesis problem:

$$\begin{aligned} H_0 : \quad & z_l = \hat{\lambda}_l - \lambda_l = 0 \\ & \qquad\qquad \text{for} \quad l = 1, \ldots, L \\ H1 : \quad & z_l = \hat{\lambda}_l - \lambda_l > 0 \end{aligned} \tag{4.5}$$

where the alternative hypothesis H_1 represents the case in which the eigenvalue of R matrix is greater than its corresponding eigenvalue of covariance matrix while the null hypothesis H_0 represents the case in which both eigenvalues are equal.
This means that when H_1 is true (i.e. H_0 fails) for a certain band l, there is a signal source which contributes to the eigenvalues of R matrix in addition to noise in the l-th band.
As pointed out in [43], each pair of eigenvalues $\hat{\lambda}_l$ and λ_l under hypothesis H_0 and H_1 can be modeled as random variables with conditional probability densities given by:

$$p_0(z_l) = p(z_l|H_0) \cong N(0, \sigma^2_{z_l}) \qquad \text{for} \quad l = 1, \ldots, L \tag{4.6}$$

$$p_1(z_l) = p(z_l|H_1) \cong N(\mu_l, \sigma^2_{z_l}) \qquad \text{for} \quad l = 1, \ldots, L \tag{4.7}$$

where μ_l is an unknown constant and the variance $\sigma^2_{z_l}$ is computed as:

$$\sigma^2_{z_l} = Var[\hat{\lambda}_l - \lambda_l] = \sigma^2_{\hat{\lambda}_l} + \sigma^2_{\lambda_l} - 2\, Cov(\hat{\lambda}_l, \lambda_l) \qquad \text{for} \quad l = 1, \ldots, L \tag{4.8}$$

where $\sigma^2_{\hat{\lambda}_l}$ and $\sigma^2_{\lambda_l}$ are given by [43]:

$$\sigma^2_{\hat{\lambda}_l} = Var[\hat{\lambda}_l] \cong \frac{2\hat{\lambda}_l{}^2}{N} \tag{4.9}$$

$$\sigma^2_{\lambda_l} = Var[\hat{\lambda}_l] \cong \frac{2\lambda_l^2}{N} \tag{4.10}$$

Applying the Schwarz's inequality in Eq. 4.8:

$$Cov(\hat{\lambda}_l, \lambda_l) \leq \sqrt{Var[\hat{\lambda}_l]\, Var[\lambda_l]} \cong \frac{2}{N}(\hat{\lambda}_l\, \lambda_l) \tag{4.11}$$

the following statement holds:

$$Cov(\hat{\lambda}_l, \lambda_l) \to 0 \qquad \text{as} \quad N \to \infty$$

which implies that:

$$\sigma^2_{z_l} \cong \sigma^2_{\hat{\lambda}_l} + \sigma^2_{\lambda_l} \cong \frac{2\hat{\lambda}_l{}^2}{N} + \frac{2\lambda_l^2}{N} \qquad \text{as} \quad N \to \infty \tag{4.12}$$

Finally, from (4.5), Eq. 4.6 and 4.7, it is possible to define the false alarm probability (P_F) and detection power (P_D):

$$P_F = \int_{-\infty}^{\tau} p_0(z)\, dz \tag{4.13}$$

$$P_D = \int_{-\infty}^{\tau} p_1(z)\, dz \tag{4.14}$$

A NPD for the hypothesis problem in Eq. 4.5 can be obtained by maximizing P_D subject to P_F which has been fixed at a given significance value α. This value allows to find the threshold value τ_l for the following decision rule which defines the NPD:

$$\delta^{NP}(z_l) = \begin{cases} 1 & \text{if} \quad \Lambda(z_l) > \tau_l \\ 1 \text{ with probability } k & \text{if} \quad \Lambda(z_l) = \tau_l \\ 0 & \text{if} \quad \Lambda(z_l) < \tau_l \end{cases} \tag{4.15}$$

where the likelihood ratio test $\Lambda(z_l)$ is given by:

$$\Lambda(z_l) = \frac{p_1(z_l)}{p_0(z_l)}$$

According to (4.15), if $\hat{\lambda}_l - \lambda_l > \tau_l$, the test is failed. This means that there is a signal source which contributes to the R matrix eigenvalue in the l-th band.
It should be pointed out that the binary hypothesis problem (4.5) has to be solved for each hyperspectral band. Therefore, the threshold τ_l must be estimated for each pair of $\hat{\lambda}_l - \lambda_l$ assuming a fixed value for α.
Using (4.15), the VD value can be obtained as follows:

$$VD_{HFC}(P_{F_\alpha}) = \sum_{l=1}^{L} \delta^{NP}(z_l) \tag{4.16}$$

where P_{F_α} is the false alarm probability at given value α. To summarize, the HFC method pseudocode is reported below.

Algorithm 1 - HFC method

Require: $X = [x^1, \ldots, x^N], P_{f_\alpha}$

Compute R matrix
Compute K matrix
$\lambda^R = eig(R)$ {compute eigenvalues of R}
$\lambda^K = eig(K)$ {compute eigenvalues of K}
$dim = 0$ {initialize the number of endmembers}
for l:=1 to L **do**
  $\sigma_l \cong \sqrt{\frac{2\hat{\lambda_l}^2}{N} + \frac{2\lambda_l^2}{N}}$
  Solve $P_{f_\alpha} = \frac{1}{\sigma_l\sqrt{2\pi}} \int_{-\infty}^{\tau} e^{-\frac{z_l^2}{2\sigma_l^2}} dz_l$ to find τ
  $diff = \lambda_l^R - \lambda_l^K$
  if $diff > x$ **then**
    $dim = dim + 1$
  end if
end for
return dim

The analysis of the HFC method starts with the inspection of Matlab code contained in the Endmember Induction Algorithms Toolbox [44]. The first step is the acquisition of the input hyperspectral image and the choice of different α values for the false alarm probability.

```
% Read Hyperspectral Image
filename = 'image_file';
delimiterIn = ' ';
headerlinesIn = 1;
mat = importdata(filename, delimiterIn, headerlinesIn);

data = mat.data;

%% data size
[nvariables nsamples] = size(data);

%Set alpha values
alpha = [10^(-3) 10^(-4) 10^(-5)];
```

The image dataset and the method used to store each hyperspectral image are the same as presented before (See Section 3.2).
The second step involves the computation of R matrix and covariance matrix K together with their eigenvalues.

```
14 %% R and covariance matrix K
15 R = (data*data')/nsamples;
16 K = cov(data');
17
18 %% Eigenvalues
19 lr = sort(eig(R),'descend');
20 lcov = sort(eig(K),'descend');
```

The last step is made up of the binary hypothesis problem solution in order to estimate the number of spectrally distinct signatures.

```
21 %% Differences and variances
22 diff = lr - lcov;
23 std = sqrt(2*(power(lr,2)+power(lcov,2))/nsamples);
24
25 %% Hypothesis Test
26 p = size(alfa,2);
27 vd = zeros(1,p);
28 for i=1:p
29    tau = -norminv(alfa(i),zeros(nvariables,1),std);
30    vd(i) = sum(diff > tau);
31 end
```

It is important to highlight the method used for the estimation of τ (line 29).
The *Cumulative Distribution Function* (CDF) of a continuous random variable X can be written as the integral of its *probability density function* (PDF) f_X:

$$F_X(x) = \int_{-\infty}^{x} f_X(t)\,dt = p \tag{4.17}$$

This function takes an input x and returns the probability value p in the interval $[0,1]$. The inverse of CDF, also called *quantile function*, finds the x value such that:

$$F^{-1}(p) = x \tag{4.18}$$

This means that, by using the inverse of CDF, it is possible to obtain the x value which solves Eq. 4.17.

Considering the HFC method, the PDF is a Gaussian distribution and, for this reason, the τ value is computed through the Matlab function *norminv* which evaluates the normal inverse CDF.
Starting from the Matlab version, a serial C version has been developed.
The eigenvalues of R and K matrices have been computed exploiting the functions *tred2* and *tqli*, previously described, where all the instructions for eigenvectors computation have been removed from both functions.
The other non trivial step to implement is the computation of the normal inverse CDF.
In order to tackle this problem, it is possible to express the quantile function of the standard normal distribution, called *probit* function, in terms of the *inverse error* function (*erf*):

$$\Phi^{-1} = \sqrt{2}\,\mathrm{erf}^{-1}(2p-1) \qquad p \in (0,1) \tag{4.19}$$

For a normal random variable with mean μ and variance σ^2, the inverse CDF is computed as follows:

$$F^{-1}(p) = \mu + \sigma\Phi^{-1}(p) = \mu + \sigma\sqrt{2}\,\mathrm{erf}^{-1}(2p-1) \qquad p \in (0,1) \tag{4.20}$$

This is the formula that is used for the computation of the inverse CDF inside the HFC algorithm remembering that the considered Gaussian function has the following properties:

$$\mu_l = 0$$
$$\sigma_l \cong \sqrt{\frac{2\hat{\lambda_l}^2}{N} + \frac{2\lambda_l^2}{N}}$$

Now the problem is how to compute the $erf^{-1}()$ function. In this case, it is possible to use an approximation which is based on the *erf* function present inside the C standard *math* library.
After that the development of the serial C version, an exhaustive code profiling and analysis have been conducted using both synthetic and real hyperspectral images. In this way, it has been possible to discover that the most computationally intensive parts were the computation of R and covariance matrices which takes around 90% of total execution time.
It is important to notice that all of those computations can be parallelized and the high data dimensionality suggests exploiting a massively parallel processor such as the GPU. Moreover, since the R and covariance matrices are of small dimensions ($Bands \times Bands$), the amount of data on which the next steps of the algorithm will work is not high. This means that the use of GPU for extracting eigenvalues could degrade the performance. For this reason, it has been decided to transfer the obtained GPU result (R and covariance matrices) back to the CPU and terminates the execution in a serial way.
The precision of the implementation is another critical issue that has been taken into account. In particular, due to the analysis of the algorithm results, it has been discovered that a single precision floating point arithmetic is not sufficient for a correct VD calculation. Indeed, single precision floating point is suitable only for small images until 50 MB, but there is an underestimation of the VD with bigger images.
Starting from the serial C version, a new algorithm implementation has been carried out

exploiting the GPU technologies through CUDA programming model.
As said before, the steps which should be developed in parallel involve the computation of R and covariance matrices. Since both the matrices evaluation requires the input hyperspectral image, it must be transferred to the device memory.
After that, both matrices calculation can be performed through the cuBLAS library by using the *cublasDgemm()* routine. Since both the matrices are symmetric, it is not necessary to apply any transposition and the result can be directly used even if it is stored in column-major format.
However, it should be underlined that the covariance matrix computation requires a preprocessing step before applying the cuBLAS routine: it is necessary to remove the mean from each band of the input hyperspectral image.
In order to do so, it is necessary to iterate over the total number of bands and perform a sum of all the samples for each band through the cuBLAS routine *cublasDasum()*. This function takes an input array, computes the sum of absolute values contained in the array and allows to store the result in the host or device memory by using the function *cublasSetPointerMode()*. For the considered case, the results have been stored inside the GPU memory in order to avoid unnecessary data transfers.
After that, a first custom CUDA kernel divides the sum of absolute values by the number of pixels and a second one carries out the computation of the zero-mean image.
Figure 4.2 depicts the steps of the CUDA implementation pointing out memory transfers.

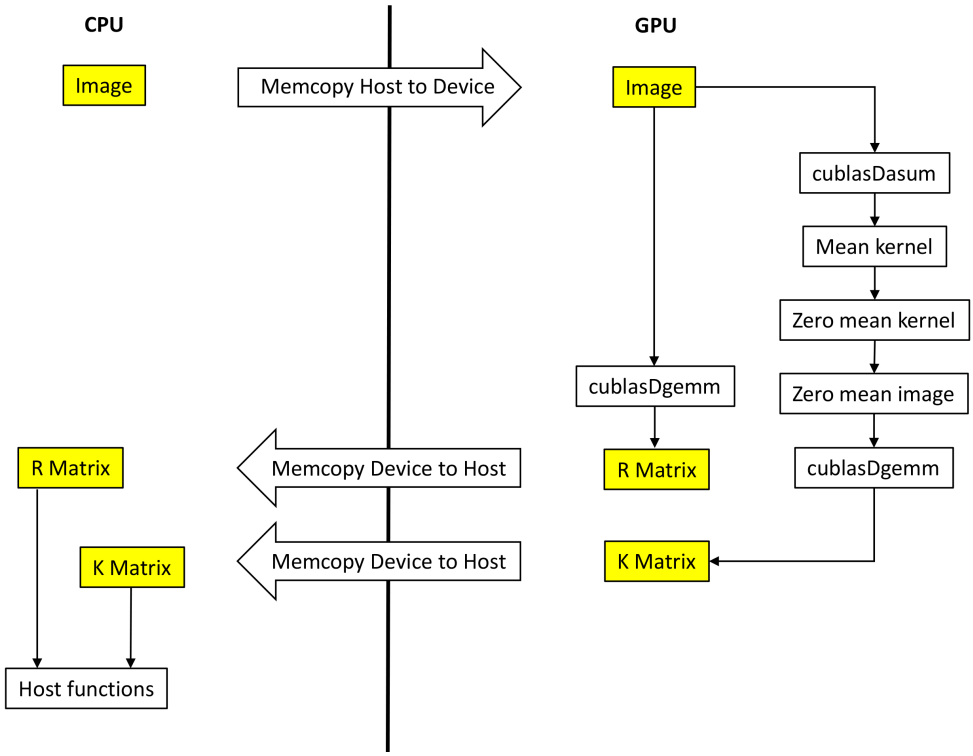


Figure 4.2: Schema of different VD estimation algorithms. Data and operations are in yellow and white boxes, respectively

Finally, two parallel versions have been carried out through the Python language in order to evaluate the performance even using an object oriented and popular programming language. The first one is based on the *NumPy* [45] and *SciPy* [46] packages which exploit multi cores execution, while the second one used NVIDIA GPUs through the *PyCUDA* package [47].
NumPy is a very well-known package for scientific computing with Python. It has several features such as:

- a N-dimensional array object
- tools for integrating C/C++ and Fortran code
- linear algebra, Fourier transform and random number capabilities.

Moreover, NumPy does not require any external linear algebra library to be installed but, if provided, it can use external libraries for building an application. Several *LAPACK* libraries such as *ATLAS, MKL* or the *Accelerate/vecLib* can be used. All these libraries contains *BLAS* highly optimized routines for specific processor types.
The HFC algorithm steps can be implemented exploiting the functions provided by *NumPy* package. In particular, the *R* matrix can be computed through the *numpy.dot()* function which performs matrices multiplication.
On the other hand, for the computation of the covariance matrix, *NumPy* provides the *numpy.cov()* function which automatically evaluates the covariance matrix of the input image.
Finally, the eigenvalues of the *R* and covariance matrices can be obtained through the *numpy.linalg.eigvalsh()* function which computes the eigenvalues of a general matrix given in input.
The last step involves the computation of the inverse CDF which cannot be performed through *NumPy*. For this reason, it is necessary to use the *SciPy* package.
Scipy is a collection of mathematical algorithms and functions based on the Python *NumPy* package. In particular, the *norm.ppf()* function inside the *SciPy stats* module has been used to evaluate the inverse of CDF.
It should be highlighted that by using pre-built functions, the code development time is reduced.
The Python code for the HFC algorithm is reported below.

```
1 import numpy as np
2 from scipy.stats import norm
3
4 # Set alpha values and read input hyperspectral image
5 alpha = [10^(-3) 10^(-4) 10^(-5)]
6 him = read_hyperspectral_image()
7
8 # Compute R and K matrices
```

```
 9  R = np.zeros((bands, bands))
10  K = np.zeros((bands, bands))
11
12  R = np.dot(him,him.T)
13  R = np.divide(R, him.shape[1])
14
15  K = np.cov(img)
16
17  # Evaluate eigenvalues of R and K matrices
18  eigR = np.linalg.eigvalsh(R)
19  eigK = np.linalg.eigvalsh(K)
20
21  # Compute estimation of variance
22  std = np.sqrt(2*(np.power(eigR,2)+np.power(eigK,2))/him.shape[1])
23
24  # Solve hypothesis test problem
25  diff = eigCorr - eigCov
26  vd = []
27  for item in alpha:
28          tau = -norm.ppf(item, np.zeros((R.shape[0])), std)
29          vd.append(sum(diff>tau))
```

The hyperspectral image is stored inside a NumPy array (*him*) while the output VD values for each alpha is stored in Python list (*vd*).
The second Python parallel version of the HFC algorithm is based on *PyCUDA* package which provides access to the CUDA driver API.
For instance, the *SourceModule()* class from the *PyCUDA compiler* module allows to define CUDA code source that during the execution will be compiled using the *nvcc* compiler for a GPU with the specified compute capability. In particular, since the *Reikna* library provides highly optimized matrix multiplication implementation, it has been used for the development of the HFC algorithm.
The *Reikna* package is built on top of *PyCUDA* adopting the so-called *CLUDA* as abstraction layer [48].
First of all, it is necessary to create a *Thread* object which will be used to manage all the elaborations:

```
1  import reikna.cluda as cluda
2
3  api = cluda.cuda_api()
4  thr = api.Thread.create()
```

The variable *thr* will contain the created *Thread* object which will be used to manage the data transfer and code compilation.
The data can be allocated through the *thr.array()* function which returns an *Array* object.
The data can be copied to the GPU memory through the *thr.to_device()* function.
Finally, the result can be copied back to the host through the *get()* function which returns a *NumPy* array with the contents of the allocated *Array* object on the GPU.
The most computationally expensive part, which is the computation of R and K matrices, can be performed through the *MatrixMul()* defined in the *Reikna linalg* module.
This function performs the matrix multiplication but before to be used, it has to be compiled as shown in the code below.

```
1 import reikna.cluda as cluda
2 from reikna.linalg import MatrixMul
3
4 api = cluda.cuda_api()
5 thr = api.Thread.create()
6
7 dot = MatrixMul(dev_him, dev_him, out_arr = dev_R, transposed_b=True)
8 dotc = dot.compile(self.thr)
9 dotc(dev_R, dev_him, dev_him)
```

where *dev_him* and *dev_R* variables contain the hyperspectral input image and the result of matrix product, respectively. Both variables must be allocated on the device memory.
The matrix product is performed through the *MatrixMul()* function. It is necessary to use the *transposed_b* clause since a matrix has to be multiplied by its transpose.
Concerning the computation of the covariance matrix, it should be pointed out that it requires as input a zero-mean image.
The zero-mean image has been computed with a custom CUDA kernel which has been compiled and loaded onto the device with the *SourceModule()* function provided by the PyCUDA package.
An example of how it works is reported below.

```
1 import pycuda.autoinit
2 from pycuda.compiler import SourceModule
3
4 mod = SourceModule("""
5         __global__ void find_zero_mean_image(double *mat,
6                                              double *mean,
7                                              int row, int col)
8         {
9                 const int i = blockDim.y*blockIdx.y + threadIdx.y;
```

```
10                  const int j = blockDim.x*blockIdx.x + threadIdx.x;
11
12                  if(i < row && j < col){
13                          mat[i*col + j] = mat[i*col + j] - (mean[i]);
14                  }
15          }
16  """)
17
18  # Compile the code and load onto the device
19  find_zero_mean_image = mod.get_function("find_zero_mean_image")
20  find_zero_mean_image(dev_him, dev_mean, row, col, block, grid)
```

The CUDA kernel source code is fed into the constructor of a *pycuda.compiler.SourceModule* which will be compiled and put onto the device through the *get_function()* routine.
After that, the CUDA kernel can be called specifying the required parameters together with the block and grid dimensions.
The final part of the HFC algorithm, which involves the eigenvalues computation and the solution of the hypothesis test, has been performed using the *NumPy* package as it was done for the Python version.
After that all the different versions have been developed, the correctness and the execution times of each of them have been evaluated through the hyperspectral image dataset.
The HFC algorithm execution times with real hyperspectral images are reported in Table 4.1.

Table 4.1: HFC algorithm processing time of serial, CUDA, Python and PyCUDA version using real images dataset

Image	**Size [MB]**	**Serial [s]**	**CUDA [s]**	**Python [s]**	**PyCuda [s]**
Hydice	2.64	0.197	0.023	0.032	0.103
Indian pines	17.64	1.596	0.062	0.270	0.133
Cuprite	34.24	2.605	0.062	0.499	0.180
PaviaU	81.49	3.401	0.077	0.624	0.244
Salinas	94.94	8.639	0.165	1.304	0.296
Botswana	209.00	13.628	0.200	2.419	0.474
KSC	211.06	16.734	0.233	2.642	0.510
WTC	268.63	27.071	0.358	4.878	0.671
PaviaC	398.44	22.542	0.345	3.431	0.802
Indian_pines-EW	952.25	97.29	2.74	16.693	3.240
Indian_pines-NS	1379.94	141.184	7.631	26.485	9.182

In order to obtain a better understanding of the different versions performance, it is possible to plot the processing times using the logarithmic scale as shown in Figure 4.3. It is possible to notice the absence of a linear relationship between the execution time and the hyperspectral

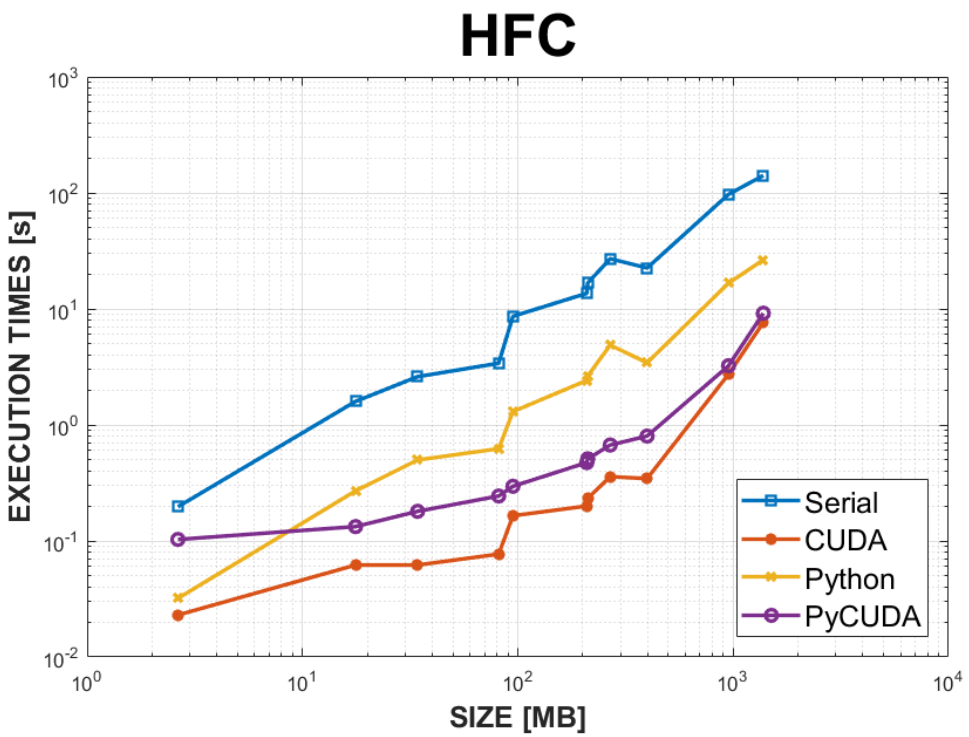


Figure 4.3: HFC processing times plot on logarithmic scale

image size. This is due to the fact that the computation time of the R and K matrices is strictly dependent from the shape of the hyperspectral image as well as the next algorithm steps depends on the number of bands of the input data.
An example is given by the WTC and PaviaC hyperspectral images. The CUDA and Python versions show a lower computation time for Pavia than WTC even if the former has a bigger size than the latter. On the other hand, PyCUDA shows the opposite behavior and this can be due to the different optimization of the *Reikna* package.
Another point, which should be underlined, is that the *PyCUDA* implementation achieves higher processing times than the CUDA one since the *Reikna* package is not well optimized as the cuBLAS library. Moreover, PyCUDA has an higher abstraction level than CUDA.
Concerning the Hydice hyperspectral image, a performance comparison between Python and PyCUDA version has to be carried out. This is the only case where the Python version is able to outperform the PyCUDA one and this is due to the image dimension. Indeed, the Hydice is the smallest image and, for this reason, the time required to copy it to the device memory and compile the CUDA kernel code has an higher impact. This conclusion can be derived through a code profiling using NVIDIA Visual Profiler. The characterization of the different activities performed by the GPU has highlighted that the memory transfers took a time that ranges from 30% to 10% of the total execution time while the effective computation took from 70% to 90%. Therefore, memory transfers have a bigger impact when considering smaller images since the time needed for the data transfers is close to the computation time. As said before, the CUDA

implementation outperforms all the other versions. It is important to underline that the CUDA version is capable to satisfy the real-time constraints by processing the WTC hyperspectral image in about 0.358 s which is far below the 5 s deadline.
To the best of author's knowledge, the most recent works about real-time estimation of the endmembers number, which can be found in the literature, are [49–51].
In [50], an FPGA-based implementation of HFC algorithm has been proposed. The authors used a *Xilinx Virtex 7 XC7VX690T FPGA* working at 75 MHz. Their experiments involved the Cuprite and WTC hyperspectral images which took a processing time of about 1.64 s and 4.26 s, respectively.
The proposed CUDA implementation outperforms the FPGA one since it guarantees a processing time of 0.062 s and 0.358 s for the same images.
It is important to notice that NVIDIA Tesla K40 and Xilinx Virtex 7 are high-end devices but the former has been released in 2013 while the latter in 2011.
Concerning the Virtex 7 implementation, the description of the optimization steps performed on the FPGA are reported in [50]. Since the GPU architecture is different from the FPGA one, it has been necessary to perform a different optimizations in the presented HFC implementation.
Concerning the WTC image, notice that although the GPU and FPGA versions are able to satisfy the real-time constraints, the total execution time of the latter is very close to 5 s. This represents a critical issue since the estimation of endmembers number is only a step of the whole hyperspectral unmixing chain.
Regarding the work proposed in [49], the estimation of endmembers number has been performed using HySime algorithm which has been developed exploiting different technologies such as multi-core CPUs, DSPs and GPUs. The best performance has been obtained using the *Automatically Tuned Linear Algebra Software* (ATLAS) library which allowed to reach an execution time of 0.549 s on the Cuprite image.
The proposed CUDA implementation is about 8 times faster. This is not only due to the fact that the HFC algorithm has a lower computational complexity, but also to the optimization carried out in order to achieve real-time performance i.e. cuBLAS library and the eigenvalues computation method.
Finally, in the work reported in [51], the authors presented a VD implementation on an *NVIDIA GTX 580* which had an execution time of 0.43 s and 1.07 s using the Cuprite and WTC images, respectively.
In both cases, the CUDA implementation presented before reaches better performance. Moreover, it is important to highlight that the authors of [51] adopted single precision floating point arithmetic while, in this work, the double precision floating point arithmetic was adopted in order to provide correct results even using high dimensional images.
A final comment concerns the whole hyperspectral unmixing chain.
The analysis of the HFC algorithm highlighted that one of the most computationally intensive part is the computation of the covariance matrix. Whenever it is decided to apply the SNR-based criterion in the second step of the hyperspectral unmixing chain, the same matrix is required together with the computation of its eigenvalues and eigenvectors. In this case, it

is possible to combine the two algorithms avoiding to perform the same calculation twice, saving memory space and reducing the total execution time.

5 Unknown Class Problem in Image Classification

Deep Neural Networks (DNNs) [52] employ a set of machine learning algorithms designed to create a high-level representation of input data by learning from examples.
Over the recent years, these methods have demonstrated outstanding results in many different tasks, such as automatic speech recognition, natural language processing or image classification.
The success of DNNs has not only attracted significant attention from academia, but has also generated a continuously growing demand from industry for DNN based applications [53, 54].
However, the exceptional performance of DNNs comes at the cost of high computational complexity and memory requirements, which makes their deployment and real-time inference on embedded edge devices a challenging task [55, 56]. Moreover, industrial utilization often poses additional constraints on cost, power consumption or available resources and time to market for the development of any new system.
To address the aforementioned challenges, several specialized hardware architectures have been proposed, developed and released recently on the market by different manufacturers [57, 58].
The main drawback of these solutions is that they are still not widely available and their usage requires expertise, which is frequently not present. Therefore, general purpose processors still remain a viable alternative for executing DNNs on embedded platforms.
In this work, we consider Convolutional Neural Networks (CNNs), a specialized type of DNN that expects data with specific temporal or spatial structure as input, such as time-series or images. These constraints allow the encoding of certain specific properties into the network architecture, making CNNs particularly suitable for applications like image classification [59], object detection [60] or scene labeling [61].
The presented work focuses on an image classification task developed at Swiss Federal Institute of Technology (EPFL, Lausanne) in collaboration with a company. This is the reason why, the application details cannot be provided due to copyright reason, hence only general results will be reported.
Concerning image classification, the CNN network is trained to recognize a pre-defined set of images. However, once the CNN has been deployed on the embedded system, it could happen

that a new image, which has not been used during the training, is acquired by the device. In this case, the input image must be classified as "unknown".
In order to tackle this problem, solution based on the *dropout at test time* technique has been exploited.

5.1 Neural Networks

The building block of *Neural Networks* (NN) is the artificial neuron, a simplified mathematical model of the biological neural, which is illustrated in Figure 5.1.

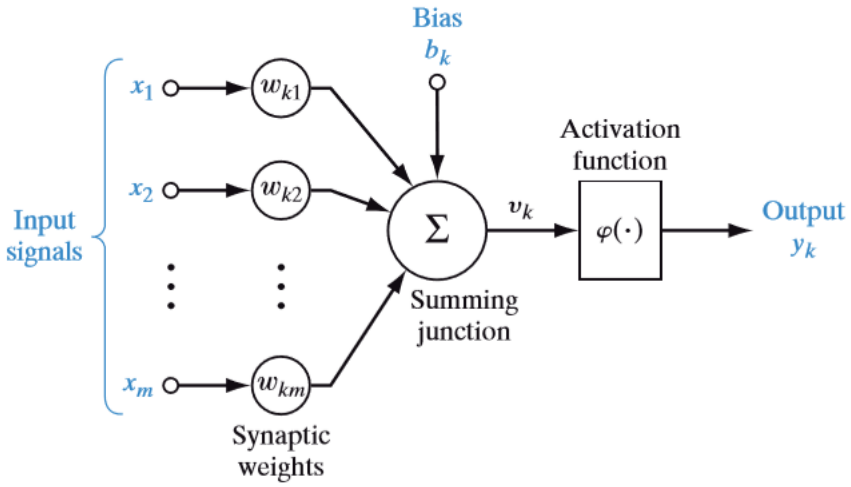


Figure 5.1: Model of artificial neuron [62]

The output y_k of the artificial neuron is obtained by applying a non-linear activation function ($\varphi(\cdot)$) to the dot product between the input and weight arrays followed by an additive bias (b_k) array.
The weights (w_k) and the bias are the neuron parameters which can be tuned in order to obtain the desired behavior.
In early NN applications, the *sigmoid* and *tanh* non-linear functions have been most frequently used. However, recently new functions have been adopted to increase the performance. One of them is the *Rectified Linear Unit* (ReLU) which became one of the most common choice for the activation function also due to its computationally efficiency. It simply sets all the negative values to 0 and, for this reason, it can be quickly performed on most of the hardware platforms.
Other alternatives to ReLU are, for instance, *Leaky ReLU* [63] and *Maxout* functions [64].
All the aforementioned activation functions are depicted in Figure 5.2.
NN are made up of an artificial neurons collection; neurons are connected to each other in a directed acyclic graph, forming the so-called *feed-forward* network.
Generally, the neurons are organized in *fully-connected layers* where each neuron in a layer is connected to every neurons in the consecutive one, but no connections exist between

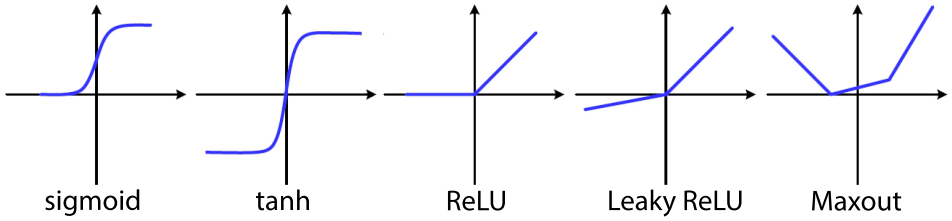


Figure 5.2: Typical activation functions used in NNs

elements of the same layer.
The layers whose input is the output of a previous layer are called *hidden layers*.
The general architecture of a NN is depicted in Figure 5.3.

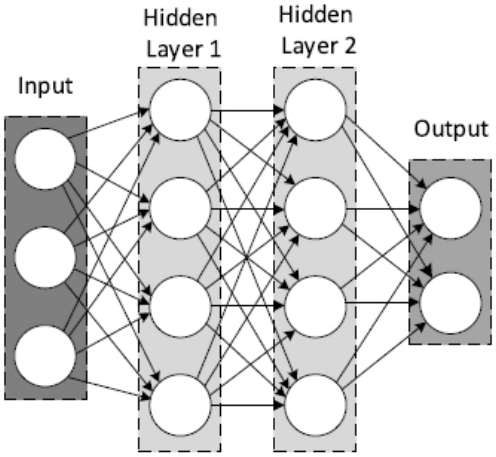


Figure 5.3: General architecture of NNs

This structure allows to compute the output from each layer as matrix-vector operations:

$$y = Wx + b \tag{5.1}$$

where W and b are the weight vectors and bias values of the neurons in the considered layer, respectively.
The design of a NN for a specific application involves the choice of the number of layers and the amount of neurons in each of them. Since increasing the number and size of the layers allows the NN to learn more complex functions at the cost of higher computational complexity, the NN design is a problem strictly related to the specific application.
An important issue to consider while dimensioning NN layers is that too high capacity network may not be able to generalize but will focus on the specific characteristics of the training dataset. This phenomenon is called *overfitting* and it can be reduced not only by adjusting the network size, but also with other methods such as enlarging the training dataset or by using regularization method: *batch normalization* [65] or *dropout* [66].

An example of the overfitting phenomenon is reported in Figure 5.4.

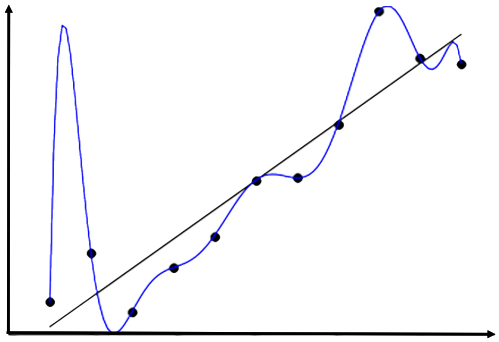


Figure 5.4: Illustration of overfitting phenomenon. Black dots represent the training data example.

Although the blue line fits perfectly the data, the black one can be expected to generalize better.
On the other hand, too small network can be hard to train with local methods [67].
It is important to underline that the weights and biases of the NN must not be defined by hand, but their values are estimated through an iterative learning process called *training*.
The majority of machine learning applications uses the *supervised learning* approach in which labeled training data are given in input to the NN and the output is compared against the desired one using a *loss function*.
The goal of the training process is to minimize the loss function by usually using a *Gradient Descent* based optimization method [52].
The gradient of the loss function with respect to the trainable parameters can be estimated at a given point by propagating the computed error through the network in backward direction. Then, the network parameters are adjusted proportionally to the gradient scaled by a number, called *learning rate*, in order to minimize the loss function.
All these steps are repeated until the output error goes below a predefined threshold.
The performance of the trained network is periodically verified through a validation dataset containing samples not seen by the NN during the training process. If the prediction on unseen samples is not accurate, the model and learning parameters have to be adjusted until satisfactory results are obtained.

5.2 Convolutional Neural Networks

A CNN is a specialized NN that expects data with specific temporal or spatial structure.
The standard NN explained before, were fully-connected. For this reason, this type of NN can be difficult to be exploited for task with high input data sizes such as images.
For instance, let's consider an image classification NN working on 128-by-128 pixel RGB images. The first hidden layer would have $128 \times 128 \times 3 \approx 50,000$ weight parameters.

The design of NN able to solve an image classification task would require several layers increasing the number of total NN parameters and possibly leading to overfitting.
CNN are made up of *Convolutional* layers which solve this problem by exploiting the concept of *local connectivity* and *parameter sharing*.
The local connectivity concerns the property of natural images in which pixels that are close to each other tend to be highly correlated while pixels far apart have very low correlation. It is possible to take advantage of this characteristic by making each neuron dependent on a small and spatially localized subset (*receptive field*) of the layer's input. In this way, the number of network parameters and computations are significantly reduced by capturing anyway an higher level semantics about the inputs.
The network structure is further simplified adding the concept of *parameter sharing* to the *local connectivity* one. This means that the weight and bias parameters are shared across a group of neurons within the same layer allowing the forward pass to be executed as a spatial convolution.
Due to these properties, the CNN weight coefficients are often identified as *filters* or *convolutional kernels*.
The most commonly used filter sizes are: 1×1, 3×3, 5×5 and 7×7.
Concerning filters larger than 1×1, the input images are often zero-padded to avoid an output dimension reduction.
The output of the group of neurons, which share the same weights, is called *feature map*.
The main aim of convolutional layers is to extract feature from their inputs which can be used to classify a given image.
Another type of layer which is often adopted after the convolutional one is the *pooling layer*. It is used to decrease the spatial dimension of data throughout the network in order to reduce the total number of computations. This is achieved by replacing the values of a block with only one as it happens with the *max pooling* or *average pooling* operations which replace block of data with their maximum or average, respectively.
An example of these two operations is reported in Figure 5.5.

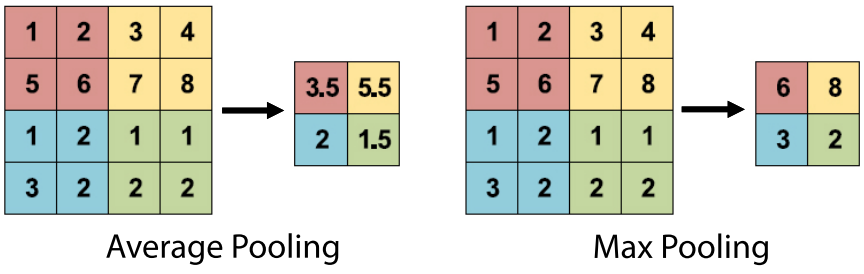


Figure 5.5: Average and Max pooling

Typically, the final part of a CNN network is built through *fully-connected* layers which are used to perform the classification of the input image.
To summarize, the CNN network can be divided in two parts (*feature extraction* and *classifier*) made up using the different types of layers as shown in Figure 5.6.

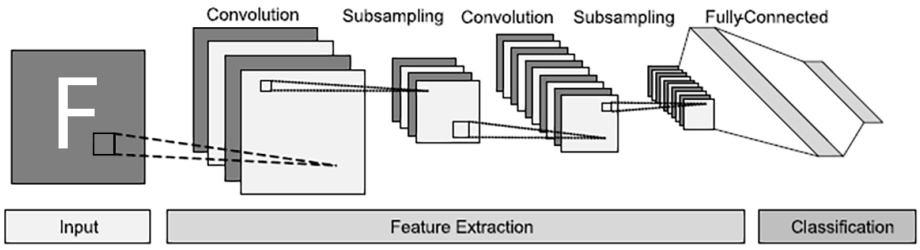


Figure 5.6: Example of CNN architecture

It is important to point out that the overfitting can be present in deep CNN.
In order to tackle this problem, the *dropout* technique can be exploited.
Its key idea is to randomly drop with probability p during the training of all the connections from one neuron to the next layer and multiply all the other ones by $\frac{1}{p}$.
On the other hand, during the validation phase all the connections will be restored. In this way the generalization capability of CNN is improved.
In CNN, dropout is applied on the fully-connected layers usually with $p = 0.5$.
An example of dropout technique is reported in Figure 5.7.

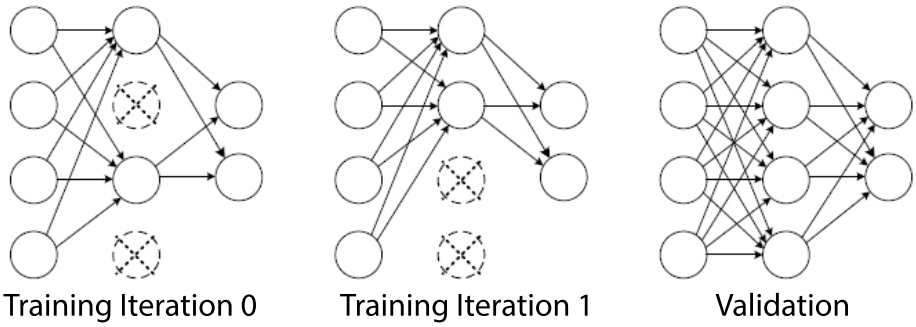


Figure 5.7: Example of dropout application

5.3 Network Architecture and Development

The CNN architecture used for the studied image classification task is reported in Table 5.1.
The network expects a $128 \times 128 \times 1$ gray scale image as its input and performs a 15-way classification. The structure is made up of 5 basic layers: three convolutional layers followed by two fully-connected ones.
Each convolutional layer uses kernels of size 3×3 with a stride of 1 and no padding and produces as outputs 32 feature maps.
The ReLU non-linearity and a max pooling operation are applied to the output of the convolutional layer.
The output of the feature extractor part is given in input to the 2-layer fully-connected classifier.

Table 5.1: CNN architecture

	Layer Type	Input size	Parameters
1	Convolution	1x128x128	320
2	ReLU	1x126x126	-
3	Max-pooling	1x126x126	-
4	Convolution	1x63x63	9248
5	ReLU	1x61x61	-
6	Max-pooling	1x61x61	-
7	Convolution	1x30x30	9248
8	ReLU	1x28x28	-
9	Max-pooling	1x28x28	-
10	Linear	6272	401408
11	ReLU	64	-
12	Linear	64	975

The hidden layer is built using 64 neurons followed by the ReLU activation function.
Finally, the probability of belonging to each class is computed using the *Softmax* function. This means that for each value of the second fully-connected layer, the following formula is applied:

$$\text{softmax}(y_j) = \frac{e^{y_j}}{\sum_{k=1}^{k=NC} e^{y_k}} \qquad \text{for} \quad j = 1, \ldots, NC \tag{5.2}$$

where y_j is the j-th output element of the network and NC identifies the total number of classes.
The presented CNN has been trained using Torch [68] and several data augmentation techniques such as rotation, random change of contrast and sharpness have been used to increase the variability of the training data. Moreover, the dropout technique has been applied on the first fully-connected layer in order to reduce the overfitting problem.
Once the training has been completed, it is possible to export the values of all the network weights and biases in a header file in order to perform the image classification on an embedded device.
The target embedded platform chosen is an Allwinner R16 quad-core ARM Cortex-A7 working at 1.2 GHz and equipped with 512 MB of DDR3 RAM, 8 GB of eMMC memory and OpenWrt Linux as operating system.
Starting from the Torch model, three network versions have been developed on the embedded platform. One is based on C programming language while the second and third ones use the C++ programming language to exploits the *Arm Compute Library* (ACL) [69], which contains a collection of highly optimized software routines for ARM Cortex-A processors, and TensorFlow [70], one of the most widespread DL framework.
It should be highlighted that the training, C and C++ ACL network implementations are not subject of this work.

Concerning the TensorFlow version, it has been necessary to create a Python script in order to load the network parameters, estimated with Torch, inside the TensorFlow model.
After having tested its correct functioning, TensorFlow allows to save the network model and all parameters values by storing everything in a protocol buffer file.
Then, the TensorFlow C++ API provides a set of routines to restore the saved model and perform the inference on the embedded device.
In order to execute the code on the Allwinner board, it is necessary to cross-compile the code for the specific target and, for this reason, the *bazel* [71] build tool has been exploited. By setting the proper configuration file, it is possible to use both the local compiler and the *Linaro toolchain* [72] which is designed for ARM processor.
Once the compilation has been carried out, the executable file can be run on the embedded device which uses the TensorFlow saved model to perform the image classification.
Another important feature provided by the TensorFlow C++ API is the possibility to profile the application as shown in Figure 5.8.

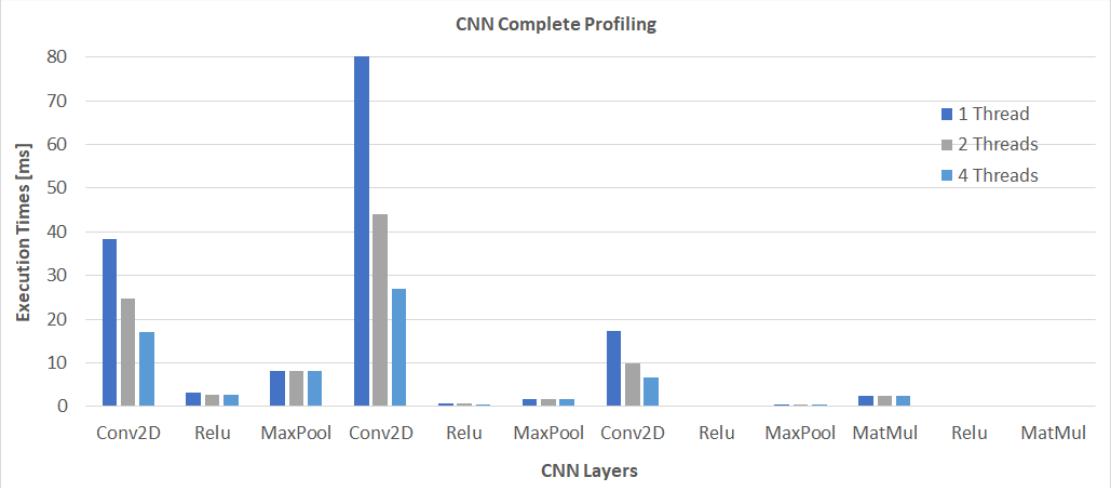


Figure 5.8: CNN complete profiling with different threads number.

The CNN complete profiling has been performed changing the number of threads. As expected, the convolutional layers takes ≈98% of total execution time.
The execution times of the TensorFlow C++ API version has been compared with the C and C++ ACL ones. The comparison is illustrated in Figure 5.9.
The different versions comparison shows that, in general, TensorFlow performances are very close to the ones of the ACL optimized version for any number of threads.
These results prove that, for the considered case study, the automatic generation of a CNN solver through TensorFlow can be used to develop neural network on embedded device at a lower development cost compared to the C and ACL versions.

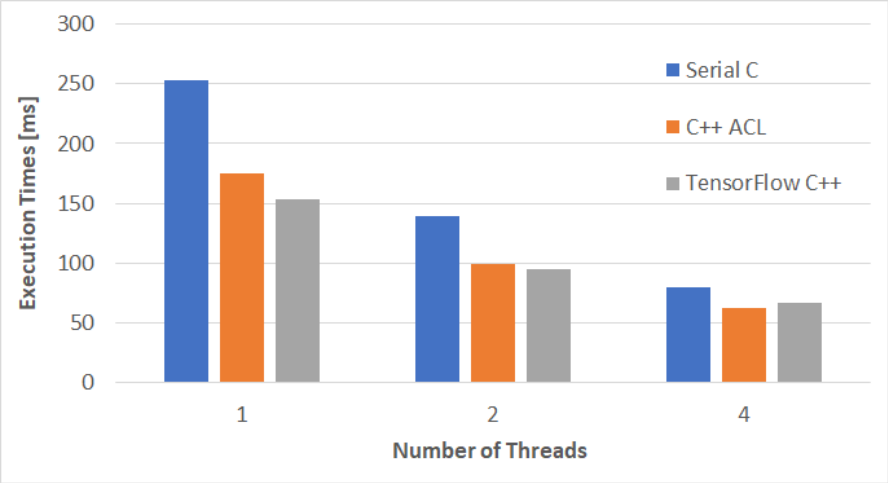


Figure 5.9: Comparison of different implementations.

5.4 Unknown Class Problem

As previously said, one of the main problems in image classification is the identification of new images, which have not been seen during training, as *unknown*.
Moreover, it is important to underline that the unknown set is open making impossible the training of the network for classifying the unknown images in a proper new class.
Thus, even if the image is unknown, it will be classified as one of the known classes during the inference as shown in Figure 5.10.

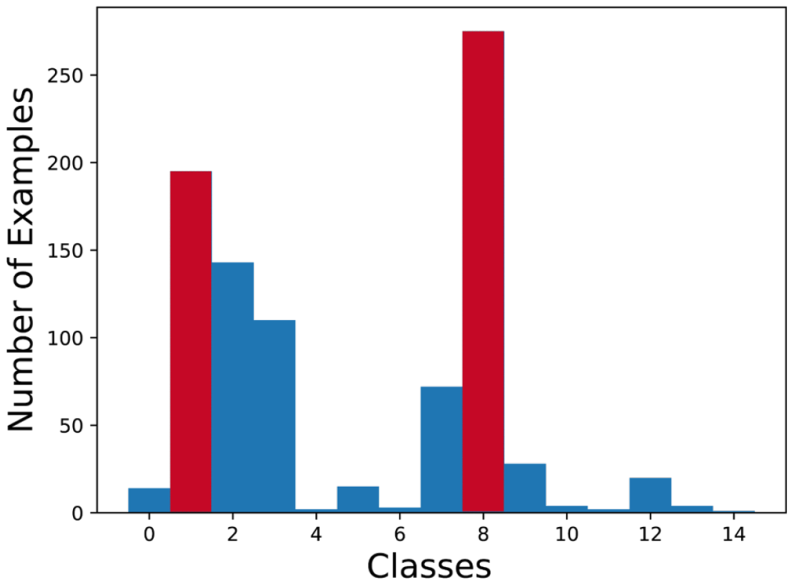


Figure 5.10: Classification of unknown images.

The unknown dataset is made up of 888 images which have been given in input to the CNN

network. Most of those images have been identified as class 1 and 8 (highlighted with red box) and, for this reason, the next analyses will be focused on these classes.
First of all, the output values of the last fully-connected layer, before the *softmax* function, has been analyzed for the class 8 and the unknown class, as depicted in Figure 5.11.

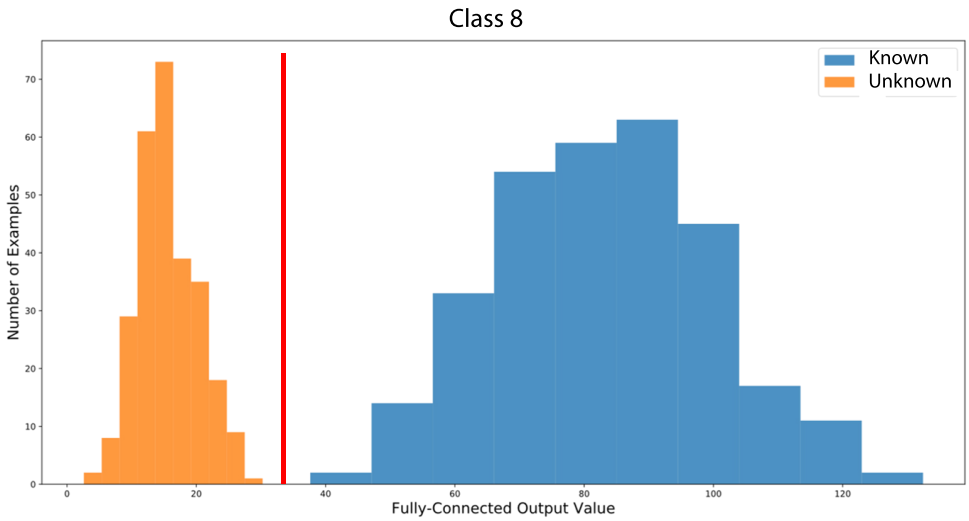


Figure 5.11: Classification of unknown images.

This analysis highlighted there is a clear separation between the unknown and the known class. This means that by setting a threshold on the output value of the last fully-connected layer, it would be possible to distinguish among the two classes.
However, the same plot can be obtained for the class 1 which is shown in Figure 5.12.

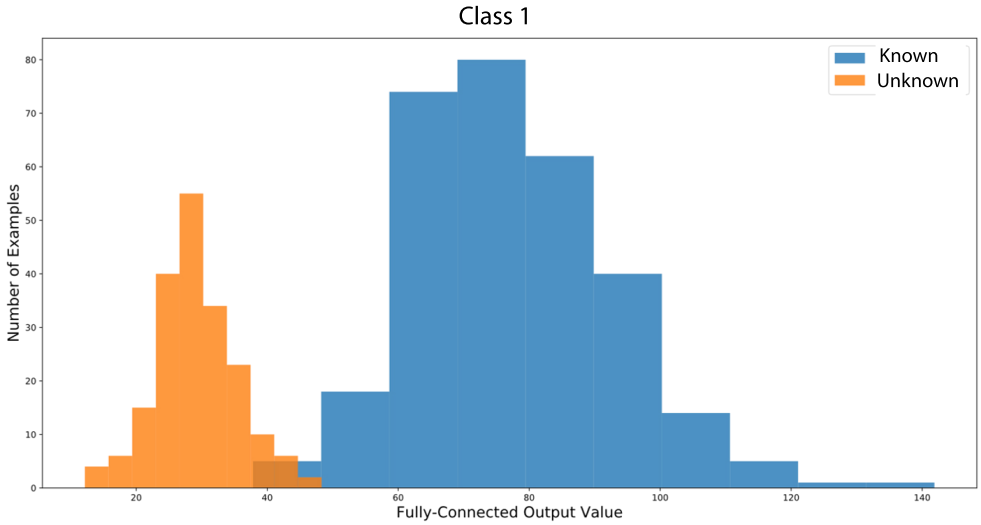


Figure 5.12: Classification of unknown images.

Concerning class 1, it is impossible to properly distinguish among the known and unknown

class. For this reason, it is necessary to exploit a new strategy: the *dropout at test time*.
The dropout technique has been developed to prevent the network from overfitting during the training. However, it can be also exploited during the inference phase in order to check the classification confidence of each image. This strategy is called *dropout at test time* and, for the considered case, the dropout is applied to the first fully-connected layer.
A block diagram of the procedure is illustrated in Figure 5.13.

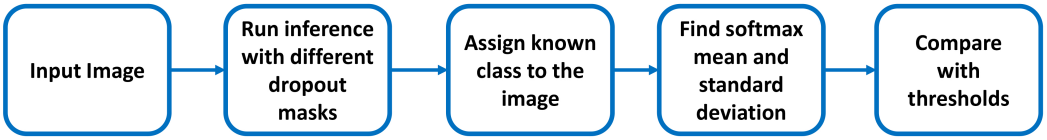
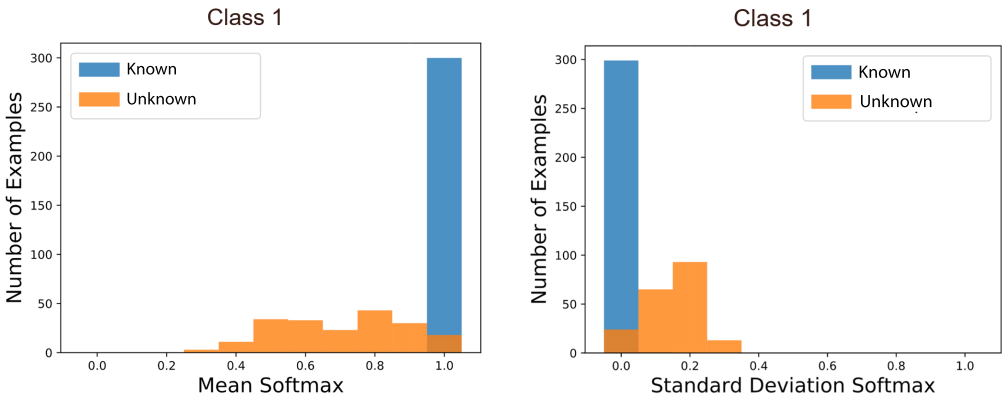


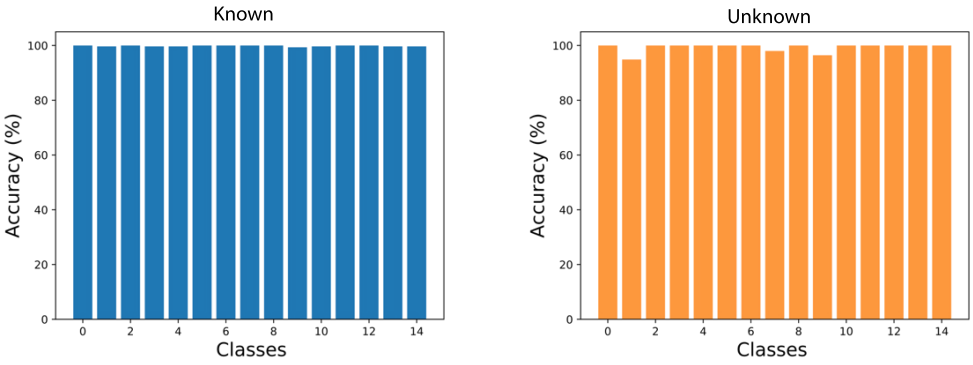
Figure 5.13: Dropout at test time block diagram.

First of all, the input image is given in input to the CNN network.
After that, the inference phase is executed with different dropout masks and each of the predicted output class is kept.
Then, the input image is assigned to a known class which has the maximum number of occurrences during the previous step.
Finally, the mean and standard deviation of the softmax values of the predicted class, obtained through the different dropout masks, are computed. If those values do not satisfy thresholds, which have been estimated for each known class, the input image is classified as unknown.
Concerning the execution time of this procedure, it has to be remembered that the dropout is applied only on the first fully-connected layer of the CNN. This means that is is possible to apply the convolutional layers just one time and iterates only over the last network layers. Moreover, as pointed out in Figure 5.8, since these layers take only ≈ 2 % of the total execution, running them several times will not affect the CNN performance.
Thanks to the dropout at test time technique, the number of unknown images detected increases, but there are still several images which cannot be rejected as reported in Figure 5.14.
There are still some unknown images which will be classified as class 1.
However, it should be pointed out that the aim of this process is to avoid the classification of a known class as unknown. Thus, it is better to set thresholds on the softmax mean and standard deviation which avoids any misclassification of the known class even if some unknown images will be classified in the wrong way.
The results of *dropout at test time* procedure applied to each known class are depicted in Figure 5.15.
Considering class 1 in Figure 5.15(b), around the 95% of unknown images, which have been identified as class 1, will be correctly classified as unknown keeping an accuracy around 99% for the known images.
Thus, it is possible to recognize most of the unknown images without affecting the classification of the known ones.
It should be highlighted that, as shown in Figure 5.10, for some known classes only few unknown images fall inside them.



(a) (b)

Figure 5.14: Softmax mean (a) and standard deviation (b) of known class 1.



(a) (b)

Figure 5.15: Results for the known (a) and unknown (b) images.

In order to tackle this problem, several solutions can be exploited such as increasing the unknown images dataset so that there will be a more equity distribution of unknown examples among the considered classes or training the CNN removing one of the known classes and using it as the unknown dataset.

6 Embedded Real-Time Fall Detection

Among older adults, unintentionally falls are the most common cause of nonfatal trauma-related hospital admission and the leading motive of fatal injuries.
As pointed out in [73], more than 25% of people aged over 65 years old falls every year. This number increases to 32%-42% for those over 70.
Another important point is that 30%-50% of people living in long-term care institutions fall each year with almost half of them having recurrent falls. The falls lead to 20%-30% of mild to severe injuries and 40% of death injuries. Concerning 65 years old people in 2004, the average cost of a single hospitalization for fall-related injuries amounted to $17,483 in the US with a total cost forecast of $240 billion by 2040.
Elderly people are not the only group affected by unintentionally fault. Indeed, any person with some sort of fragility presents the same issue.
In this scenario, an automated monitoring system, which detects occurring falls and sends remote notifications, can increase the level of care provided to people with fragilities.
The designed system is a wearable device that can be put on elderly people for a continuous monitoring. The device is equipped with a *Micro-Controller Unit* (MCU) which manages sensor data acquisition, executes fall detection algorithm in real-time and issue alert notifications toward a remote system so that timely aid can be provided.
Furthermore, the overall wearable device for fall detection should fulfill the following requirements:

- Be constantly connected to a remote, cloud-based system that provides notification at the caregivers
- Be as lightweight and small as possible in order to be more conformable to wear
- Be running on battery as long as possible without recharging since the monitoring must be performed 24×7

These requirements have to be carefully taken into account during software development.
First of all, the processing of sensor data must be performed on the device itself since their

continuous transmission would involve a dramatic increase in power consumption [74]. Thus, the algorithms for real-time fall detection should be explicitly developed for low power MCUs. It should be pointed out that the automated fall detection using data coming from sensors, such as three-axial accelerometers, is considered an open research problem [75]. In this field, machine and *Deep Learning* (DL) techniques have demonstrated an extreme potential [74, 76].

6.1 Wearable Devices

Wearable devices are designed for monitoring purposes. They can be worn by an user under, with or on top of clothing and are made up of electronic components such as sensors and processing units. These devices are better than smartphones which have several limitations such as the fact that the sensors are managed by the operating system possible running multiple applications [77].
Concerning fall detection, several wearable devices have been developed in the literature.
In [78], the proposed system uses accelerometers and gyroscopes which are connected to a board through Zigbee transceiver. The board has an *Intel PXA255* processor which performs the processing.
In [79], a three-axial accelerometer, an MCU and a Bluetooth module have been attached to a jacket and connected through stretchable conductive nylon. Another system in [80] is made up of accelerometers and FPGA for signal processing.
An head-worn device has been developed in [81] and it contains an accelerometer and a barometer connected to a *TI MSP430* MCU.
It has to be underlined that all the fall detection methods presented before are executed on the devices themselves by using an estimated threshold which is applied to either data signals or statistical indicators.
More advanced techniques are adopted in other approaches at the cost of performing the computation on a remote workstation.
An example is given by the approach described in [82] where the *Sensing Health with Intelligence, Modularity, Mobility and Experimental Reusability* (SHIMMER) integrated platform has been adopted [83]. In this case, the accelerometer signals are remotely processed through *Support Vector Machine* (SVM) algorithm and the results are compared to the ones obtained with *k-Nearest Neighbor* (KNN) and *Complex Trees* methods.
A similar approach is presented in [76] where the off-line elaboration is performed on signals acquired by accelerometers and gyroscopes.
In these cases, the fall detection analysis is based on machine learning techniques.
Nowadays, apart from SHIMMER platform, other *Commercial Off-The-Shelf* (COTS) products are emerging. Among those, the *STMicroelectronics* produces the *SensorTile*, a powerful device in terms of memory, computational capabilities and power consumption using only a chip area of 182.25 mm^2.
The SensorTile together with its expansion board is shown in Figure 6.1. The expansion board is used to power the SensorTile with a 100 mAh Li-Ion battery and for programming and

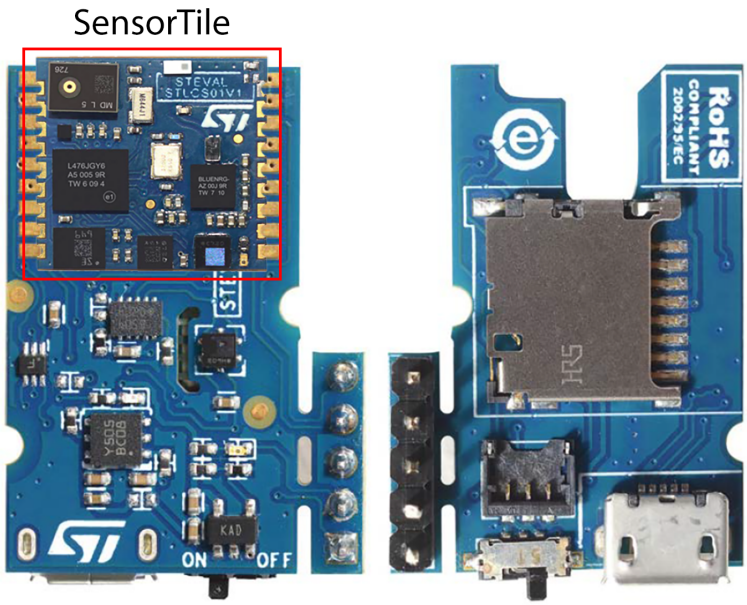


Figure 6.1: SensorTile and expansion board photo [84]

debugging through the SWD connector. Moreover, it provides a microSD card socket.
All the sensors are designed on the SensorTile itself as shown in the block diagram below. The

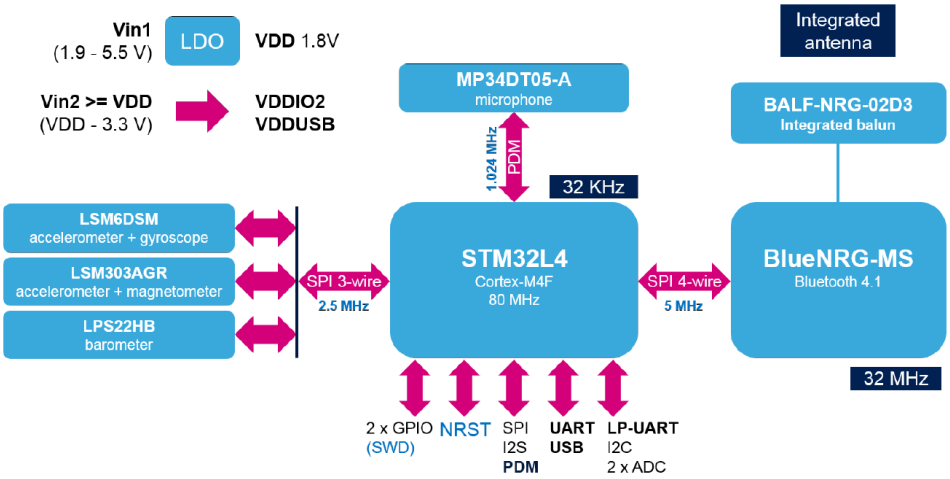


Figure 6.2: SensorTile block diagram [84]

SensorTile main components are:

- STM32L476JG: 32-bit ultra-low-power MCU
- LSM6DSM: 3D accelerometer and 3D gyroscope
- LSM303AGR: ultra-low power 3D accelerometer and 3D magnetometer

- LPS22HB: 260-1260 hPa absolute digital output barometer
- MP34DT05-A: 64 dB SNR digital MEMS microphone
- BlueNRG-MS: Bluetooth low energy network processor
- BALF-NRG-02D3: 50 Ω balun with integrated harmonics filter
- LD39115J18R: 150 mA low quiescent current low noise LDO 1.8 V.

In particular, the on board MCU is an *ARM CORTEX-M4F* with FPU which fully supports single-precision data instructions and implements a full set of Digital Signal Processing (DSP) ones. The MCU is also equipped with 1 MB of flash memory and 128 KB of SRAM.
In [85], a description of a wearable device for human activity recognition with the SensorTile board is reported.

6.2 Deep Learning for Fall Detection

Nowadays, DL is one of the most adopted paradigm by the scientific community.
DL techniques can perform task in several different fields from playing board games to estimate a world poverty map [7].
Recurrent Neural Networks (RNN) are a particular form of *Artificial Neural Network* (ANN) [52] in which a part of the output is fed back to the input. This behavior is illustrated in Figure 6.3 on the left.

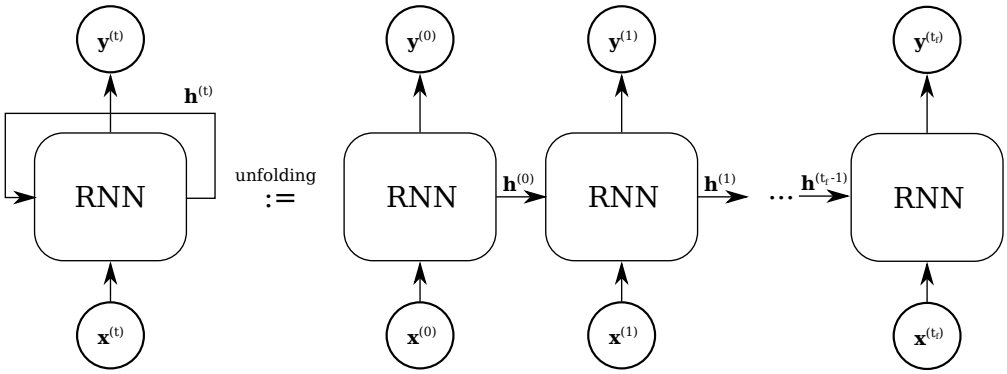


Figure 6.3: On the left, a single RNN cell which is temporally unfolded for training [86]

The RNN structure can be described by the following equation:

$$y^{(t)} = wg(Wx^{(t)} + Uh^{(t-1)} + b) + c \tag{6.1}$$

where

$$h^{(t)} = g(Wx^{(t)} + Uh^{(t-1)} + b)$$

In Eq. 6.1, $x^{(t)}$ and $y^{(t)}$ are the input and output at time t, respectively; g is a non linear function; $h^{(t)}$ is usually identified as the *hidden state* of the network [52]; w, W, U, b and c are the parameters to be estimated via *stochastic gradient descent* using a training dataset of input sequences.
The use of recurrence inside the RNN makes this technique suited for analyzing time series of signals. However, this type of network is very difficult to train.
In order to tackle this problem, the *temporal unfolding* technique is used as shown in Figure 6.3 on the right.
This is the typical training technique for RNN and it is based on the fact that input sequence of pre-defined length is fed into the unfolded network.
As said before, the RNNs are mainly used for processing time series of signals which are the typical outputs from accelerometer sensors.
In this case, the input signal has to be scanned with a *sliding window* whose size has to match the level of unfolding defined for training (ww).
The aim of training procedure is to estimate the optimal values for the network parameters which guarantee an output value closest to the expected one.
Once the training is complete, a new input sequence is given to the RNN in order to recognize a specific pattern. This process, called *inference*, applies the RNN to each input window (w_i) extracted from the input stream using a *sliding window* of width ww.
As shown in Figure 6.4, the slide is typically performed at intervals (s) of constant length called *stride*.

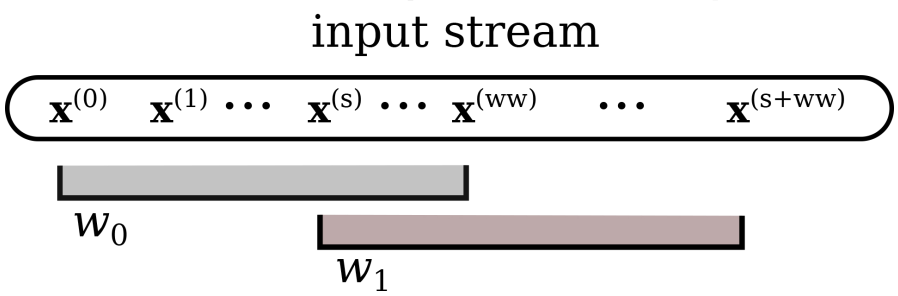


Figure 6.4: Creation of input windows w_i using a sliding window with width ww and stride s ($s < ww$) from a generic input stream [86]

One of the main weaknesses of RNN is the so-called *vanishing gradient problem*. Since the network is unfolded for the training, during the back-propagation phase, which is used to calculate the gradient of loss function with respect to the parameters, the gradient tends to be close to zero as moving backward in the network. Once the gradient becomes zero, the network stops to learn.
In order to avoid this issue, a special type of RNN, capable of learning the long-term temporal dependencies, has been designed: *Long Short-Term Memory* (LSTM) [87, 88].
The LSTM cell schema has been schematized in Figure 6.5.
At each time step t, the cell is fed with the concatenation of $x^{(t)}$ and $h^{(t-1)}$.
The input is spread over four *gates*: *cell gate* (c), *input gate* (i), *output gate* (o) and *forget gate*

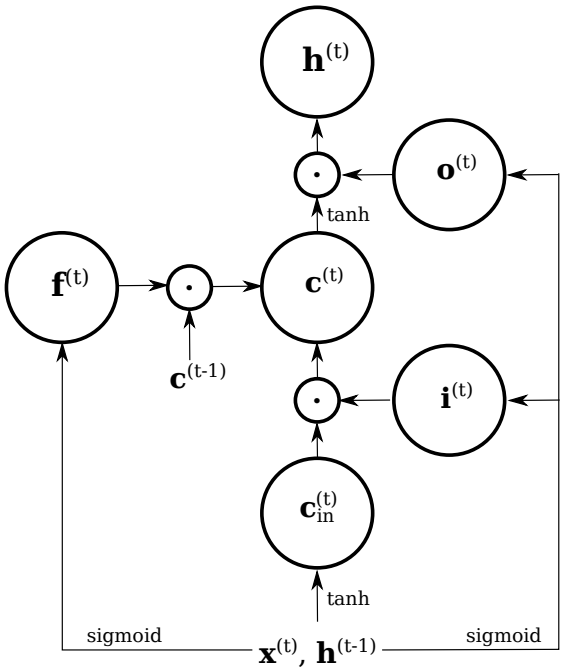


Figure 6.5: LSTM cell schema. Each circle identifies the equations 6.2-6.7. Each arrow pointing to a circle and dots represents addition and vector/matrix multiplications, respectively. [86]

(f). This is the main difference between LSTM and RNN.
Each gate can be modeled as Eq. 6.1 using a parameter matrix W, a bias vector b and applying the *sigmoid* or *tanh* as non-linear function.
A single LSTM cell is described by the following equations:

$$C_{in}^{(t)} = \tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_{C_{in}}) \tag{6.2}$$
$$i^{(t)} = \text{sigmoid}(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \tag{6.3}$$
$$o^{(t)} = \text{sigmoid}(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o + b_{forget}) \tag{6.4}$$
$$f^{(t)} = \text{sigmoid}(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f) \tag{6.5}$$
$$c^{(t)} = f^{(t)}c^{(t-1)} + i^{(t)}C_{in} \tag{6.6}$$
$$h^{(t)} = o^{(t)}\tanh(c^{(t)}) \tag{6.7}$$

where:

- W_{xc}, W_{hc}, W_{xi}, W_{hi}, W_{xo}, W_{ho}, W_{xf}, $W_{hf} \in \mathbb{R}^{LS \times LS}$
- b, $x^{(t)}$, $h^{(t)}$, $C_{in}^{(t)}$, $i^{(t)}$, $o^{(t)}$, $f^{(t)}$, $c^{(t)} \in \mathbb{R}^{LS}$

LS is called the LSTM *size*, an *hyperparameter* defined during the design which is constant among all the LSTM cells. The term b_{forget} in Eq. 6.4, called *forget bias*, is typically set equal to 1 and it has been added to improve the LSTM network performance, as reported in [89].

A fall detection system, designed using DL techniques, have been presented in [90] and it is made up of two parts: *Convolutional Neural Network* (CNN) and LSTM.
The former identifies spatial information from the data and is one of the most used tools in image processing [52]. Thus, the CNN extracts features from the sensor signals while the LSTM discovers a temporal relationship among the features. However, this system has only been developed off-line without designing a version for a small embedded device.
Due to the high computationally complexity of DL algorithms, only recently they started to be developed on embedded devices. For instance, Google released *TensorFlow Lite*, a lightweight version of *TensorFlow* [70], which is one of the most widespread DL framework, for the development on mobile and embedded devices.
Nevertheless, the limited computational power and memory on embedded devices require a proper design of the DL algorithms. Moreover, the algorithm has to be adapted to the considered MCU in order to achieve the desired performance.
It is important to notice that, for the moment, only the inference phase is developed on the embedded system while the training is still performed off-line with a high computational power device.
In literature, several techniques have been presented in order to make feasible the DL algorithm development on embedded system.
A first strategy is to prune the training model by keeping only the parameters used for the inference phase [70].
Other sophisticated methodologies can be found in literature such as fixed-point representation [91], weight clustering [92], weight approximation [93] and parameter compression [94].
The fixed-point solution, for instance, allows to reduce the memory footprint at the cost of loosing accuracy. From the computational point of view, it is necessary to distinguish between linear and non-linear function. Using a small number of bits, addition and multiplication can be performed in much more efficient way in fixed-point.
On the other hand, the fixed-point computation of non-linear function such as *tanh* or *sigmoid* leads to a loss of precision with respect to the floating point implementation.
The fixed-point version of the *ReLU* non-linear function does not suffer the loss of precision, but it can be inserted in LSTM networks only with special provision [95].
Another strategy for reducing memory footprint is the *integer quantization* which could be based on 8-bit integers [91].
First, it is necessary to define a range of values for all the parameters and intermediate variables. Then, all the float values are translated into such range and the new values are used for the computation. In most practical approaches, non-linear functions are still computed using floating point values avoiding loss of precision.
Concerning integer quantization, it is even possible to find works which adopts ultra low precision weights using only two $(-1/+1)$ or three $(-1/0/1)$ values representation [96, 97]. However, these approaches require a complete network re-design and can lead to a high accuracy loss.

6.3 Network Architecture

The design of DL architecture for the presented fall detection system has been conducted taking into account the development on relatively resource-constrained device, such as the SensorTile, satisfying real-time constraints.
The block schema of the adopted DL network is reported in Figure 6.6.

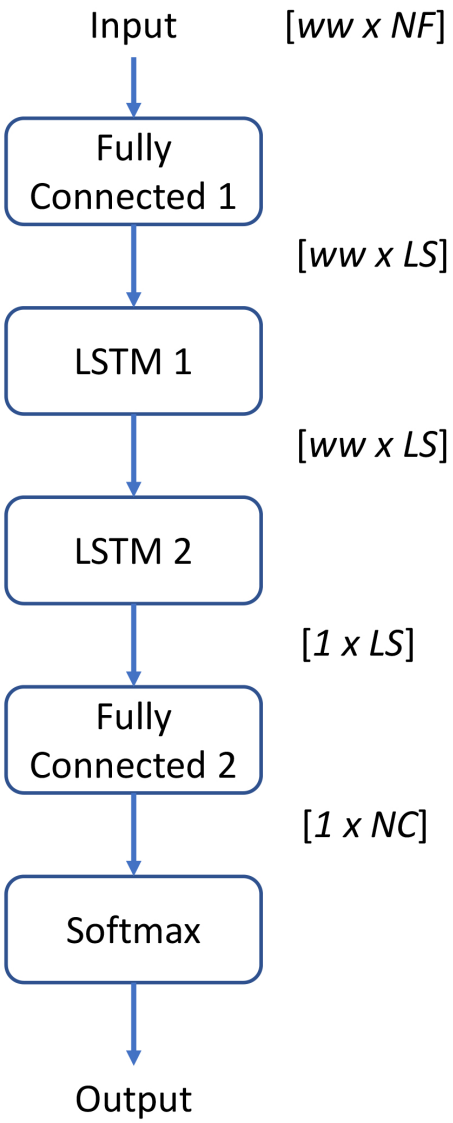


Figure 6.6: Model architecture of the proposed solution.

The input size depends on the number of features (*N_Features*) chosen and the size of windows (*ww*) to be fed into the network. For the considered case, the *N_Features* is set equal to 3 (i.e. the accelerometer values) and *ww* to 100 (the network is fed with a time series of 1 s sampled at 100 Hz).

The output size depends on the number of classes that for the case studied are:

- FALL: event of fall
- ALERT: event of interest which may or may not lead to a fall
- BKG: all the activities not related to a fall such as walking, jumping, etc.

Concerning the network structure, it is based on two LSTM cells stacked on top of other.
Each cell has an inner dimension LS of 32 units.
The input are pre-elaborated through a fully connected layer (*Fully Connected 1*) which can be modeled by the following equation:

$$X = \text{ReLu}(X_{in}W_{hidden} + b_{hidden}) \tag{6.8}$$

where $X_{in} \in \mathbb{R}^{WW \times NF}$ is the input network, $W_{hidden} \in \mathbb{R}^{NF \times LS}$ and $b_{hidden} \in \mathbb{R}^{LS}$ are the parameters to be estimated.
The output of the second LSTM cell is fed into another fully connected layer (*Fully Connected 2*) which performs the following operation:

$$X_{out} = ZW_{output} + b_{output} \tag{6.9}$$

where $Z \in \mathbb{R}^{1 \times LS}$ is the output of LSTM 2, $W_{output} \in \mathbb{R}^{LS \times NC}$ and $b_{hidden} \in \mathbb{R}^{NC}$ are the parameters to be estimated.
Finally, in order to extract the probability for each class, the *Softmax* function is applied to X_{out} as follows:

$$\text{softmax}(x_{out_j}) = \frac{e^{x_{out_j}}}{\sum_{k=1}^{k=NC} e^{x_{out_j}}} \qquad \text{for} \quad j = 1, \ldots, NC \tag{6.10}$$

The dataset which has been chosen for training and testing the proposed fall detection system is the SisFall [98]. This dataset includes 38 identities: 15 elderly and 23 young subjects doing 34 different activities in a controlled scenario with several retries.
A total of 4510 complete recordings has been obtained.
A very important aspect is the recording method. Indeed, instead of using smartphones as it happens for several datasets, for SisFall a properly designed embedded device has been used. It has been attached on the body as a belt buckle and it acquired data from a 3D accelerometer and a gyroscope.
The description of the training phase is not a topic of this thesis, but only the inference phase, performed on the embedded device, will be threated.

6.4 Memory Occupancy, Computation Power and Power Consumption Metrics

A set of general metrics for evaluating the requirements in terms memory occupancy, computing power and power consumption for the development of an LSTM cell on a generic MCU

have been derived.
Due to the limited memory footprint on embedded device, one of the most important aspect in the LSTM cell development is the design of data storage layout. Indeed, it has a great impact on performance computation and memory occupancy.
Thus, through a suitable data storage layout, the operations involved in Eq. 6.2-6.7 can be performed in a more optimized way.
The chosen layout is depicted in Figure 6.7.

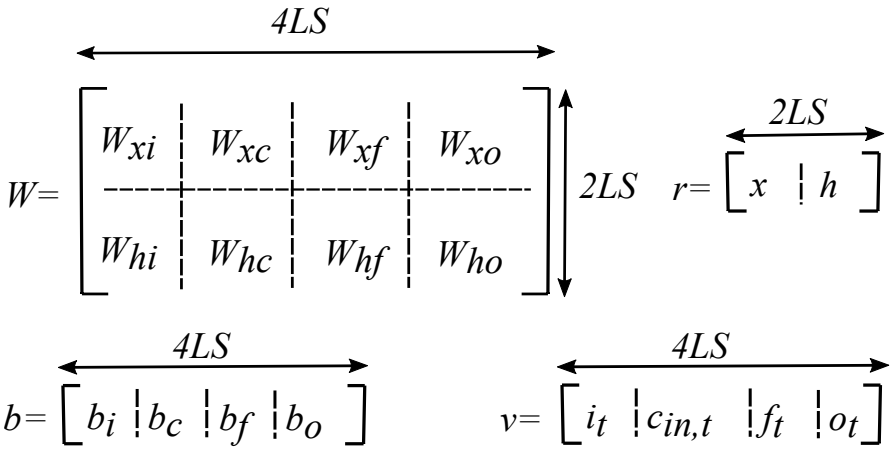


Figure 6.7: Schema of data layout storage [86]

As shown in the picture, all the matrices and biases, required to compute i_t, $c_{in,t}$, f_t and o_t, are stored in a unique weight matrix W and bias array b. Moreover, the two cell inputs (x and h) are packed in the r array. In this way, the Equations 6.2-6.5 can be evaluated with just one matrix multiplication followed by an addition. The results are stored in the v array as shown below:

$$v = W \cdot r + b \tag{6.11}$$

After that, the term b_{forget} is added to the part of the v array representing the o_t component. Then, the *sigmoid* and *tanh* functions are applied element-wise to each part of the v array on the basis of its position.
Finally, the c_t expression is computed using a temporary array of dimension LS.
It is important to underline that v and r arrays can be reuse for the following LSTM cells while W matrix and b array are private to each cell.
This means that for each LSTM cell added to the network, the memory footprint will increase consequently.
The memory occupancy required by an LSTM cell can be estimated as follows:

$$Size_{Net} = In \cdot Size_{In} + \sum_{i=1}^{N_W} W_i \cdot Size_{W,i} + \sum_{j=1}^{N_b} b_j \cdot Size_{b,j} + \\ + \sum_{k=1}^{N_T} Temp_k \cdot Size_{Temp,k} + Out \cdot Size_{Out} \tag{6.12}$$

where In is the total number of input values, $Size_{In}$ is the size in bytes of the input values, W_i is the element number of the i-th W matrix, represented using $Size_{W,i}$ bytes.
Moreover, b_j is the number of elements of j-th b array; the number of temporary k arrays are taken into account by the term $Temp_k$, together with their dimension $Size_{Temp,k}$ in bytes.
Finally, Out is the number of output elements, represented using $Size_{Out}$ bytes. By applying Eq. 6.12, it is possible to estimate the size of LSTM network ($Size_{Net}$).
Using the results obtained so far, the memory occupancy metric is:

$$MCU_{RAM} > Size_{Net} \tag{6.13}$$

where MCU_{RAM} is the RAM memory size of the considered MCU and $Size_{Net}$ is the value obtained through Eq. 6.12.
It should be underlined that the design of the metric has been performed by taking into account only the RAM memory.
Indeed, from the computational point of view, the use of the flash memory reduces the performance and makes more difficult to respect the real-time constraint.
Concerning the evaluation of the computation complexity metric, this is more difficult to estimate since the numerical representation affects the definition of the metric.
In this work, the IEEE single precision floating point arithmetic has been adopted.
However, even if it has been decided to use either fixed point format or integer quantization, the metric can be derived in the same way as it is done for the 32 bits floating point case.
In order to estimate the computational complexity of an LSTM cell, the Eqs. 6.2-6.7, 6.8 and 6.7 have to be properly analyzed.
In particular, it is necessary to estimate the computational complexity of the elementary operation present in those Equations.
Let's start with the sum of matrices.
Consider $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$. Their sum $C \in \mathbb{R}^{m \times n}$ is defined as:

$$c_{i,j} = a_{i,j} + b_{i,j} \qquad i = 1, \ldots, m \quad \text{and} \quad j = 1, \ldots, n \tag{6.14}$$

Therefore, each output element needs a single sum to be computed. Thus, the total number of operations to be performed is equal to mn and the computational complexity is $\mathcal{O}(mn)$.
Concerning the product of matrix, let's consider $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$. Their product $C \in \mathbb{R}^{m \times n}$ is defined as:

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j} \tag{6.15}$$

therefore, each product element needs p multiplications and $p-1$ additions to be computed. Thus, the total number of operations to be performed is equal to $(2p-1)mn$ and the computational complexity is $\mathcal{O}(pmn)$.
Considering the non-linear functions, it is important to notice that the number of operations needed to compute the *tanh* function is strictly related to the adopted numerical precision. In this case, IEEE single precision floating point arithmetic is considered.

An important property used for evaluating this function is:

$$\tanh(-x) = -\tanh(x)$$

Thus, it is possible to use thresholds only on positive numbers.
The *tanh* function is evaluated as follows:

Algorithm 2 - Tanh Function

if $0 \leq x < 2^{-12}$ **then**
  $\tanh(x) \leftarrow x$
else if $2^{-12} \leq x < 1$ **then**
  $\tanh(x) \leftarrow \frac{-\text{expm1f}(-2x)}{\text{expm1f}(-2x)+2}$
else if $1 \leq x < 9$ **then**
  $\tanh(x) \leftarrow 1 - \frac{2}{\text{expm1f}(2x)+2}$
else if $x \geq 9$ **then**
  $\tanh(x) \leftarrow 1$
end if

The $expm1f(x)$ function is an approximation of $e^x - 1$ which requires 15 operations, while each division takes 4 operations to be evaluated.
Therefore, in the worst case, the *tanh* function requires 22 floating point operations (FLOP).
As for the Hyperbolic tangent, the computational complexity of the sigmoid is related to the numerical precision.
The *sigmoid* function is evaluated as follows:

Algorithm 3 - Sigmoid Function

if $x \geq 0$ **then**
  $\text{sigmoid}(x) \leftarrow \frac{1}{1+\text{expf}(-x)}$
else if $x < 0$ **then**
  $\text{sigmoid}(x) \leftarrow \frac{\text{expf}(x)}{1+\text{expf}(x)}$
end if

The $expf(x)$ function is an approximation of e^x and takes 10 FLOP. Thus, the total number of single precision floating point arithmetic is 15.
Consider the expression of $C_{in}^{(t)}$ in Eq. 6.2, the product $W_{xc}x^{(t)}$ requires, being a matrix-matrix multiplication, $(2LS-1)LS = 2LS^2 - LS$ floating point operations (FLOP).
The product $W_{hc}h_{t-1}$ requires the same number of FLOP, while the sum between these two products and $b_{C_{in}}$ requires $2LS$ FLOP.
The total number of *tanh* operations is equal to LS. Hence, the FLOP required for this compu-

tation is equal to $\approx 22LS$.
The total number of FLOP needed to compute $C_{in}^{(t)}$ is estimated as:

$$2LS^2 - LS + 2LS^2 - LS + 2LS + 22LS \approx 4LS^2 + 22LS \tag{6.16}$$

The $i^{(t)}$ and $o^{(t)}$ computations are similar to the previous one, with the difference that the tanh function is replace with the $sigmoid$, which requires approximately 15 FLOP. Also in these cases the $sigmoid$ is evaluated LS times. The total number of FLOP for each equation is given by:

$$2LS^2 - LS + 2LS^2 - LS + 2Ls + 15LS \approx 4LS^2 + 15LS \tag{6.17}$$

Concerning the f_t function evaluation, it is also necessary to consider the b_{forget} term which requires LS sums.
In this case, the number of FLOP is approximately:

$$4LS^2 + 16LS \tag{6.18}$$

Finally, $c^{(t)}$ requires $3LS$ FLOP (each element needs 2 multiplications and a sum), while $h^{(t)}$ takes $\approx 23LS$ FLOP since each element need a multiplication and a tanh.
Thus, a single LSTM cell takes a number of FLOP which is approximately:

$$4LS^2 + 22Ls + 4LS^2 + 15LS + 4LS^2 + 15LS + 4LS^2 + 16LS + 3LS + 23LS \approx 16LS^2 + 94LS \tag{6.19}$$

Moreover, it is important to notice that each cell performs these operation ww times.
It is possible to affirm that the total number of FLOP of a single LSTM cell is approximately:

$$ww(16LS^2 + 94LS) \tag{6.20}$$

The FLOP of the fully-connected layer and the $ReLU$ function depends mainly from the matrix-matrix multiplication and matrix addition. Considering their dimensionality, the FLOP are:

$$(2NF - 1)wwLS + LSww = 2NF \cdot ww \cdot LS \tag{6.21}$$

Finally, the output fully-connected layer and the $softmax$ function requires the computation of $ZW_{output} + b_{output}$, which accounts for $(2LS-1)NC$ (multiplication) and NC (sum) FLOP. The exponential is evaluated NC times, so it accounts for $\approx 10NC$ FLOP, while the division is performed NC times taking $4NC$ FLOP.
The total number of FLOP is given by:

$$2LSNC - NC + NC + 10NC + 4NC = 2LS \cdot NC + 14NC \tag{6.22}$$

It is possible to compute the total number of FLOP needed by a network with N LSTM cell

using this formula:

$$\underbrace{2NF \cdot ww \cdot LS}_{ReLU} + N \cdot \underbrace{ww(16LS^2 + 94LS)}_{LSTM} + \underbrace{2LS \cdot NC + 14NC}_{softmax} \tag{6.23}$$

with a consequent computational complexity of $\mathcal{O}(N \cdot ww \cdot LS^2)$. The metric related to processing time under real-time constraint can be written as:

$$\frac{\mathsf{Net}_{\mathsf{FLOP}}}{\mathsf{MCU}_{\mathsf{FLOPS}}} < \frac{ww}{f_s} \tag{6.24}$$

where $\mathsf{Net}_{\mathsf{FLOP}}$ is the number of floating point instructions of the network, $\mathsf{MCU}_{\mathsf{FLOPS}}$ is the number of floating point instructions per second (FLOPS) performed by the MCU and f_s is the sampling frequency used for acquiring the samples.
The final metric concerns the power consumption.
The battery life can be estimated using the following equation:

$$Time = \frac{B_Q}{I_{abs}}. \tag{6.25}$$

where B_Q is the capacity of the battery expressed in mAh and I_{abs} is the total current absorbed by the MCU.

6.5 Inference Module on Embedded Device

The inference phase of the LSTM network has been developed on the SensorTile device using the memory layout storage described before.
For the target application, the precision losses due to pruning or quantization can not be tolerated so the single precision floating point arithmetic has been adopted.
In addiction, considering ARM MCU, it is possible to exploit the CMSIS library which contains highly optimized linear algebra operations for single floating point values [99].
In particular, in order to use CMSIS matrix operations, it is necessary to store the variables in a $arm\_matrix\_instance\_f32$ structure which is defined as follows:

```
1 struct arm_matrix_instance_f32{
2   uint16_t numRows;
3   uint16_t numCols;
4   float32_t *pData;
5 }
```

In this way, it is possible to perform the multiplication between two matrices through the $arm\_mat\_mult\_f32()$ routine.
Concerning the array operations, the *arm_add_f32()* function has been exploited to perform

the element-wise sum of two arrays. It does not require to store the array in particular structures.
The MCU working frequency has been set equal to 80 MHz which is the maximum allowed.
The pseudocode of the LSTM network implementation is reported below:

Algorithm 4 - LSTM network

```
1: Initialize network
2: while true do
3:    Read input ww
4:    for i:=1 to ww do
5:       Apply fully connected with ReLU layer
6:       for j:=1 to N do
7:          Compute LSTM cell {evaluate eq. 6.2-6.7}
8:       end for
9:    end for
10:   Apply fully connected with softmax layer
11: end while
```

Line 1 identifies the RAM memory allocation together with the loading of trained weights which were initially stored in the flash memory.
In Line 2, the *while* loop is repeated forever in order to process the incoming signals by caching windows of *ww* width.
The outer *for* loop in line 4 provides as input to the LSTM network the values acquired from the accelerometer.
Each input is pre-elaborated through the fully connected layer (Line 5) and given in input to the cascade of LSTM cells on which the inner for loop (Line 6) iterates.
Finally, each class probability is estimated through the *softmax* function (Line 10).
In order to validate the correctness of the developed inference phase, the results of the embedded solution have been compared to the ones obtained through the TensorFlow version on a workstation.
The tests have been performed on several input sequences considering a window width and a number of features equal to 100 and 3, respectively.
The *Mean Squared Error* (MSE) between the outputs of the two implementation is about 10^{-7} and this proves the correctness of the inference implementation on the embedded device.
The final step is the evaluation of the different metrics previously described.
Concerning the memory occupancy of the network, the floating point version takes about 82 KB and the total RAM memory is 128 KB.
By using Eq. 6.13, it is possible to check that the metric is satisfied.
For what concerns the computational complexity metric, the following assumptions have

been made:

- MCU working frequency set to 80 MHz
- $ww = 100$ (1 sec at sampling frequency of 100 Hz)
- $LS = 32$
- $NF = 3$
- $NC = 3$

As said before, the estimation of the MCU_{FLOPS} in Eq. 6.24 strictly depends on the adopted MCU.
For the ARM Cortex-M4 MCU, the MCU_FLOPS can be estimated by considering that the involved instructions are assembly VMLA (multiply and accumulate, 3 clock cycles), VADD (addition, 1 clock cycle), VMULT (multiplication, 1 clock cycle), VDIV (division, 14 clock cycles).
Therefore, considering the operations involved in the network, it is possible to estimate the mean duration of an operation in 7 clock cycles.
By knowing the mean duration of the instructions and the working frequency of the MCU, it is possible to compute the $\mathsf{MCU}_{\mathsf{FLOPS}}$:

$$\mathsf{MCU}_{\mathsf{FLOPS}} = \frac{80MHz}{7} \approx 11.4 \cdot 10^6 \,\mathsf{FLOPS} \tag{6.26}$$

On the other hand Net_{FLOP} can be computed substituting the considered values of the network parameters inside the Eq. 6.23.
By applying Eq. 6.24:

$$\frac{3897843}{11.4 \cdot 10^6} < \frac{100}{100} \rightarrow 0.342 < 1. \tag{6.27}$$

Therefore, the floating point version of the network is real-time compliant and the estimated execution time is very close to the one obtained on the SensorTile which is around 0.3 sec.
Finally, the power consumption metric has been computed.
Since the Eq. 6.25 requires the estimation of the absorbed current from the MCU, it has been computed through the *STM32CubeMX Power Consumption Calculator*. The power consumption of the Bluetooth module has not been taken into account since it is strictly dependent on the adopted communication protocol.
On the other hand, the current absorbed by the accelerometer can be neglected since, according to the specification, it is an order of magnitude below the one required by the MCU.
Under these assumption, the total current estimated is about 5 mA.
Thus, by considering that the SensorTile battery has a capacity of 100 mAh, it is possible to apply the Eq. 6.25:

$$Time = \frac{100mAh}{5mA} \tag{6.28}$$

It can be concluded that the device could stay active for 20 hours without recharging.
However, it is important to underline that this battery lifetime estimation has been conducted considering the MCU continuously working.
As said before, each input sequence is ready after 1 sec but the total execution time is 0.3 sec. This means that, after having completed the inference phase, it could be possible to put the MCU in sleeping mode until a new input sequence is ready. In this way, it would be possible to reduce the power consumption increasing the autonomy of the wearable device.

Conclusion

This thesis explored the application of new technologies and suitable solutions in real cases featuring requirements, from time to time, of high performance computing, real-time processing, low power consumption, embedding & wearability.
In particular, the work performed is related to three projects.
The first was in collaboration with the *Hyperspectral Computing Laboratory* of Estremadura University with the aim of carrying out a system for the real-time elaboration of hyperspectral images.
The second was in collaboration with *Swiss Federal Institute of Technology* (EPFL, Lausanne) for the implementation of a CNN on an embedded system through TensorFlow framework and the analysis of the unknown class problem in image recognition.
The third was in collaboration with *Computer Vision Laboratory* of University of Pavia and in partnership with *STMicroelectronics* for the development of a real-time fall detection system devoted to the elderly people.
For what concerns the first project, the performance improvements, which can be achieved through parallel architectures, have been shown. In particular, those architectures have been applied to the hyperspectral imaging field.
Several algorithms versions have been developed using programming language like C and Python exploiting the OpenMP API and CUDA framework for multi-core and many-core device, respectively.
It has been shown that the CUDA version of the VD based on HFC method, used to extract the number of pure materials in an hyperspectral image, outperforms all the other implementations. This algorithm version is not only able to satisfy the real-time constraint by keeping its execution time below 5 s, but it also allows to perform the same computation 75 times faster than the serial C version.
Considering the BS techniques, used to reduce the dimensionality of the image cube, the CUDA version of the SNR-based criterion provides a speedup of 40 compared to the serial C one outperforming even all the other implementations.
However, GPU technology is not always the best solution as it happens for the ID-based criterion. In this case, the low computationally complexity of the algorithm together with its structure do not allow to exploit the CUDA potentialities.
Future works involve the development of the whole hyperspectral unmixing chain exploiting different technologies and still being able to satisfy the real-time constraint.

Recently, in other fields such as the machine learning, after the development on a workstation exploiting parallel architectures, an implementation on small embedded systems follows. This is what happens in the second considered project that concerns the development of a CNN for image recognition, trained using high performance workstations, on an embedded device through the TensorFlow framework. After the cross-compilation of the C++ code for the embedded device, it is possible to run the trained TensorFlow network keeping an execution time nearly the same as with properly designed versions based on optimized libraries such as ACL.
Moreover, a solution to the classification of unknown images has been presented and tested on the real embedded device.
Possible improvements concern the acquisition of new unknown images and the application of other techniques such as the *OneClass Support Vector Machine* (OneClassSVM) method which has been designed for outliers detection.
Finally, the third project is based on the development of a real-time fall detection system for elderly people. A RNN, trained on a workstation exploiting GPU architectures, has been developed on the SensorTile, a small low power embedded device by STMicroelectronics. In particular, it is possible to acquire 100 three-axis accelerometer values in 1 s and to perform the classification of the input sequence in 0.3 s satisfying the real-time constraint.
Moreover, a set of general metrics for evaluating the network requirements in terms of memory occupancy, computing power and power consumption have been derived.
In the next steps, the Bluetooth communication will be developed and, in order to save power, the MCU will be put in sleep after the inference phase until a new input sequence will be ready.
On one hand, the presented research paves the way for the development of a framework which can be provided to the scientific community in order to perform hyperspectral images analysis in real-time exploiting GPU architectures.
On the other hand, the research has been focused on the development from GPU architectures on embedded devices. In particular, it has been shown that, for the considered case, the cross-compilation of TensorFlow for ARM architecture guarantees performance comparable to custom designed solutions. However, this process cannot be performed for small embedded devices such as the SensorTile. Therefore, the TensorFlow model has to be properly developed in order to satisfy the proposed metrics.
Nevertheless, the work performed shows a certain maturity level of the presented technologies and their compliance and suitability with the requirements of demanding and diffused applications. The proposed implementations are very efficient and promising, so as to bode well that, accordingly to the technological evolution trend, this is the right way to deal with heavy computational problems.

A OpenMP API

OpenMP is an API designed for multi-core, shared memory systems [100]. It is made up of compiler directives, library routines and environmental variables which allows to divide the execution among cores present in modern CPUs.
As shown in Figure A.1, there are two types of shared memory systems: *Uniform Memory Access* (UMA) and *Non-Uniform Memory Acces* (NUMA).

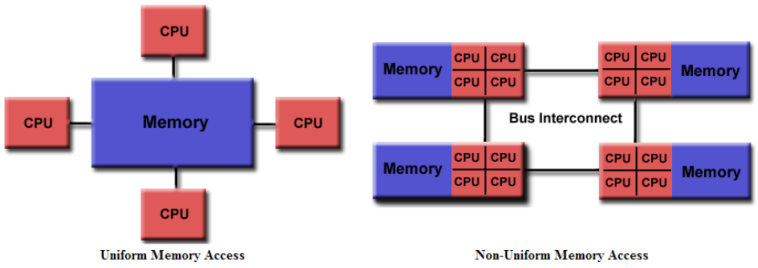


Figure A.1: UMA model (left) and NUMA model (right) [101]

In the UMA model, the memory latency is independent from the CPUs since they share the physical memory. On the other hand, in the NUMA model each processor can access its own local memory faster than non-local one.
As illustrated in Figure A.2, OpenMP uses the fork-join model of parallel execution.

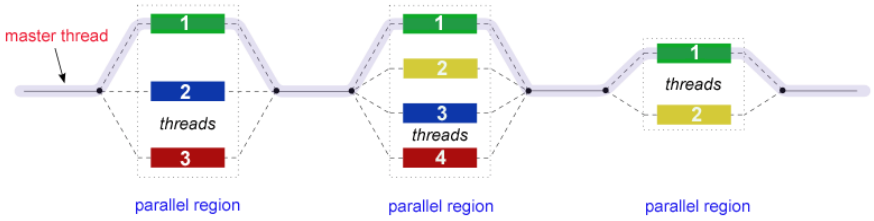


Figure A.2: Fork-join model [101]

All OpenMP programs begin with the *master thread* which executes the code sequentially

until a *parallel region* construct is encountered. At this point, the *master thread* generates a team of parallel threads (*fork*) so that the work can be split among them. When all the threads complete their task, they synchronize (*join*) and the control of the execution returns to the *master thread*.
As far as a C/C++ program is concerned, it is necessary to include the *omp.h* header file in order to use the OpenMP API.
The portion of the code, which has to be parallelized, must be indicated through *#pragma* compiler directive together with proper clauses.
For instance, it is possible to decide which variables will be shared among threads (*shared* clause) and which ones will be private to each thread (*private* clause).
A typical example of compiler directive, used to parallelize for loop, is reported below:

```
#pragma omp parallel for num_threads(3) private(var1) shared(var2) schedule (static)
```

where *num_threads* specify the number of threads inside the parallel region, *private* and *shared* represent the private and shared variables, respectively, and *schedule* which defines how loop iterations are divided into contiguous non-empty subset (chunk) and how these chunks are distributed among the threads. For instance, using the static scheduling implies that the total number of loop iterations are equally divided among the threads.

B cuBLAS Library

The *cuBLAS* library is an implementation of the Basic Linear Algebra Subprograms (BLAS) on top of NVIDIA CUDA runtime [102].
In order to use the cuBLAS library, all the necessary data must be allocated on the GPU memory and the header file *cublas_v2.h* must be included. Then, after having called the desired functions, the results are copied back to the host.
The cuBLAS library adopts the column-major storage and 1-based indexing in order to guarantee a maximum compatibility with existing Fortran code. Since C/C++ uses the row-major storage, manipulations or transpositions have to be applied on the data in order to read the result in the correct way.
The two different storing strategies are reported in Figure B.1. Before calling a cuBLAS function,

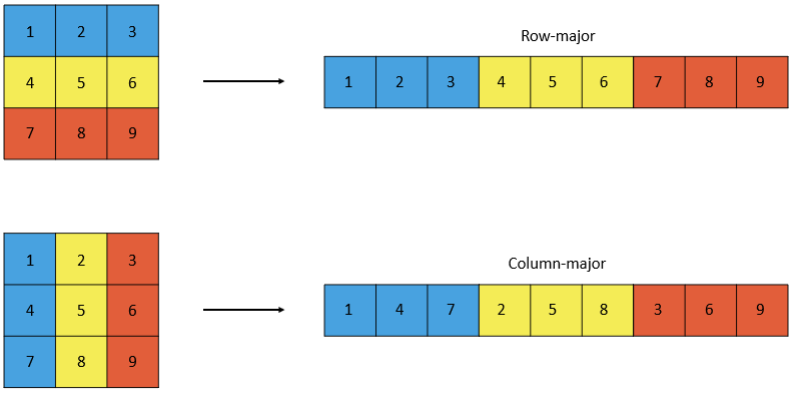


Figure B.1: Row-major storage and column-major storage

it is required to initialize a *handle* to the cuBLAS library context by calling the *cublasCreate()* function. This *handle* must be passed to each cuBLAS function.
At the end of program execution, all the resources assigned to the cuBLAS library context must be released through the *cublasDestroy()* function.
Some cuBLAS functions requires or returns scalar parameters which can be stored on the host or on the device. The data transfer will be automatically managed by cuBLAS library setting

the library pointer mode equals to *CUBLAS_POINTER_MODE_HOST* or *CUBLAS_POINTER_MODE_DEVICE*.
The cuBLAS functions return a *cublasStatus_t* error state which contains the status of the operation. All the functions can be divided in three levels:

- *Level 1*: scalar and vector operations
- *Level 2*: matrix-vector operations
- *Level 3*: matrix-matrix operations

In the presented work, the *cublasDgemm()* function has been exploited.
The function prototype is the following:

```
cublasStatus_t cublasDgemm(cublasHandle_t handle,
                    cublasOperation_t transa,cublasOperation_t transb,
                    int m, int n, int k, const double *alpha,
                    const double *A, int lda,
                    const double *B, int ldb, const duble *beta,
                    double *C, int ldc)
```

This function performs matrix-matrix multiplication solving the following equation:

$$C = \alpha op(A) op(B) + \beta op(C) \tag{B.1}$$

where *A*, *B* and *C* are matrices stored in column-major format with dimension *op(A)* $m \times k$, *op(B)* $k \times n$ and *op(C)* $m \times n$, respectively, while α and β are scalars.
The *op()* function allows to perform matrix transformations.
Considering for instance the matrix *A*, the following operations can be applied:

$$op(A) = \begin{cases} A & \text{if} \quad transa == CUBLAS\_OP\_N \\ A^T & \text{if} \quad transa == CUBLAS\_OP\_T \\ A^H & \text{if} \quad transa == CUBLAS\_OP\_C \end{cases}$$

where A^H is the conjugate transpose matrix.
Another cuBLAS function which has been used in this work is the *cublasDasum*().
The function prototype is the following:

```
cublasStatus_t cublasDasum(cublasHandle_t handle, int n,
                           const double *x,
                           int incx, double *result)
```

This function computes the sum of the absolute values of the elements inside the array *x*. Hence, the operation performed is the following:

$$res = \sum_{i=1}^{n} |Im(x[j])| + |Re(x[j])| \qquad \text{where} \quad j = 1 + (i-1) * incx \tag{B.2}$$

Bibliography

[1] J. Diaz-Montes, Y. Xie, I. Rodero, J. Zola, B. Ganapathysubramanian, and M. Parashar, "Federated computing for the masses–aggregating resources to tackle large-scale engineering problems," *Computing in Science Engineering*, vol. 16, pp. 62–72, July 2014.

[2] E. Torti, G. Danese, F. Leporati, and A. Plaza, "A hybrid cpu–gpu real-time hyperspectral unmixing chain," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, pp. 945–951, Feb. 2016.

[3] A. Barberis, G. Danese, F. Leporati, A. Plaza, and E. Torti, "Real-time implementation of the vertex component analysis algorithm on gpus," *IEEE Geoscience and Remote Sensing Letters*, vol. 10, pp. 251–255, Mar. 2013.

[4] G. Danese, M. Giachero, F. Leporati, and N. Nazzicari, "An embedded multi-core biometric identification system," *Microprocess. Microsyst.*, vol. 35, pp. 510–521, July 2011.

[5] E. Torti, D. Koliopoulos, M. Matraxia, G. Danese, and F. Leporati, "Custom fpga processing for real-time fetal ecg extraction and identification," *Comput. Biol. Med.*, vol. 80, pp. 30–38, Jan. 2017.

[6] T. Lillesand, R. Kiefer, and J. Chipman, *Remote Sensing and Image Interpretation*. Wiley, 2015.

[7] N. Jean, M. Burke, M. Xie, W. M. Davis, D. B. Lobell, and S. Ermon, "Combining satellite imagery and machine learning to predict poverty," *Science*, vol. 353, no. 6301, pp. 790–794, 2016.

[8] E. Torti, A. Fontanella, G. Florimbi, F. Leporati, H. Fabelo, S. Ortega, and G. Marrero Callico, "Acceleration of brain cancer detection algorithms during surgery procedures using gpus," *Microprocessors and Microsystems*, vol. 61, 2018.

[9] J. Plaza, A. J Plaza, and C. Barra, "Multi-channel morphological profiles for classification of hyperspectral images using support vector machines," *Sensors (Basel, Switzerland)*, vol. 9, pp. 196–218, Jan. 2009.

[10] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis, M. R. Olah, and O. Williams, "Imaging

spectroscopy and the airborne visible/infrared imaging spectrometer (aviris)," *Remote Sensing of Environment*, vol. 65, no. 3, pp. 227 – 248, 1998.

[11] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 2012.

[12] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, pp. 54–62, Sept. 2005.

[13] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2018.

[14] NVIDIA, "Kepler GK110 whitepaper," 2012.

[15] C.-I. Chang and S. Wang, "Constrained band selection for hyperspectral imagery," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 44, pp. 1575–1585, June 2006.

[16] P. W. MAUSEL, "Optimum band selection for supervised classification of multispectral data," *Photogramm Eng Remote Sens*, vol. 56, pp. 55–60, 1990.

[17] J. R. P. Stephen D. Stearns, Bruce E. Wilson, "Dimensionality reduction by optimal band selection for pixel classification of hyperspectral imagery," *Proc. SPIE*, vol. 2028, pp. 2028 – 2028 – 10, 1993.

[18] C.-I. Chang, Q. Du, T.-L. Sun, and M. L. G. Althouse, "A joint band prioritization and band-decorrelation approach to band selection for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 37, pp. 2631–2641, Nov. 1999.

[19] M. Gong, M. Zhang, and Y. Yuan, "Unsupervised band selection based on evolutionary multiobjective optimization for hyperspectral images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 54, pp. 544–557, Jan. 2016.

[20] K. Sun, X. Geng, L. Ji, and Y. Lu, "A new band selection method for hyperspectral image based on data quality," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 7, pp. 2697–2703, June 2014.

[21] S. Jia, G. Tang, J. Zhu, and Q. Li, "A novel ranking-based clustering approach for hyperspectral band selection," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 54, pp. 88–102, Jan. 2016.

[22] C. Chang, "A review of virtual dimensionality for hyperspectral imagery," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, pp. 1285–1305, Apr. 2018.

[23] C.-I. C. Su Wang, "Band prioritization for hyperspectral imagery," *Proc. SPIE*, vol. 6302, pp. 6302 – 6302 – 12, 2006.

[24] a. C.-I. J.C.Harsanyi, W.Farrand, "Detection of subpixel spectral signatures in hyperspectral image sequences," *Proc. Amer. Soc. Photogramm. Remote Sens. Ann. Meeting*, pp. 236–247, 1994.

[25] J. M. Bioucas-Dias and J. M. P. Nascimento, "Hyperspectral subspace identification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 46, pp. 2435–2445, Aug. 2008.

[26] J. M. P. Nascimento and J. M. B. Dias, "Vertex component analysis: a fast algorithm to unmix hyperspectral data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, pp. 898–910, Apr. 2005.

[27] D. Heinz, C. . Chang, and M. L. G. Althouse, "Fully constrained least-squares based linear unmixing [hyperspectral image classification]," in *IEEE 1999 International Geoscience and Remote Sensing Symposium. IGARSS'99 (Cat. No.99CH36293)*, vol. 2, pp. 1401–1403 vol.2, June 1999.

[28] T.-M. Tu, C.-H. Chen, J.-L. Wu, and C.-I. Chang, "A fast two-stage classification method for high-dimensional remote sensing data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 36, pp. 182–191, Jan. 1998.

[29] A. A. Green, M. Berman, P. Switzer, and M. D. Craig, "A transformation for ordering multispectral data in terms of image quality with implications for noise removal," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 26, pp. 65–74, Jan. 1988.

[30] J. B. Lee, A. S. Woodyatt, and M. Berman, "Enhancement of high spectral resolution remote-sensing data by a noise-adjusted principal components transform," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 28, pp. 295–304, May 1990.

[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes in c: The art of scientific computing. second edition," 1992.

[32] A. Fontanella, E. Marenzi, E. Torti, G. Danese, A. Plaza, and F. Leporati, "A suite of parallel algorithms for efficient band selection from hyperspectral images," *Journal of Real-Time Image Processing*, Mar. 2018.

[33] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. New York, NY, USA: Wiley-Interscience, 2006.

[34] C.-I. Chang, "An information-theoretic approach to spectral variability, similarity, and discrimination for hyperspectral image analysis," *IEEE Transactions on Information Theory*, vol. 46, pp. 1927–1932, Aug. 2000.

[35] N. Keshava and J. F. Mustard, "Spectral unmixing," *IEEE Signal Processing Magazine*, vol. 19, pp. 44–57, Jan. 2002.

[36] C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Norwell, MA, USA: Kluwer/Plenum, 2003.

[37] C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 42, no. 3, pp. 608–619, 2004.

[38] J. Bioucas-Dias and J. M. P. Nascimento, "Estimation of signal subspace on hyperspectral data," *Proc. SPIE*, vol. 5982, Oct. 2005.

[39] O. Kuybeda, D. Malah, and M. Barzohar, "Rank estimation and redundancy reduction of high-dimensional noisy signals with preservation of rare vectors," *IEEE Transactions on Signal Processing*, vol. 55, pp. 5579–5592, Dec. 2007.

[40] C. Chang, W. Xiong, W. Liu, M. Chang, C. Wu, and C. C. Chen, "Linear spectral mixture analysis based approaches to estimation of virtual dimensionality in hyperspectral imagery," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 48, pp. 3960–3979, Nov. 2010.

[41] C. Chang, W. Xiong, H. Chen, and J. Chai, "Maximum orthogonal subspace projection approach to estimating the number of spectral signal sources in hyperspectral imagery," *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, pp. 504–520, June 2011.

[42] K. Fukunaga, "15 intrinsic dimensionality extraction," in *Classification Pattern Recognition and Reduction of Dimensionality*, vol. 2 of *Handbook of Statistics*, pp. 347 – 360, Elsevier, 1982.

[43] T. Anderson, *An Introduction to Multivariate Statistical Analysis.* Wiley Series in Probability and Statistics, Wiley, 2003.

[44] http://www.ehu.eus/ccwintco/index.php/EndmemberInduction_Algorithms.

[45] http://www.numpy.org/.

[46] https://www.scipy.org/.

[47] https://documen.tician.de/pycuda/.

[48] https://pypi.org/project/reikna/.

[49] E. Torti, M. Acquistapace, G. Danese, F. Leporati, and A. Plaza, "Real-time identification of hyperspectral subspaces," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 7, pp. 2680–2687, June 2014.

[50] C. Gonzalez, S. Lopez, D. Mozos, and R. Sarmiento, "A novel fpga-based architecture for the estimation of the virtual dimensionality in remotely sensed hyperspectral images," *Journal of Real-Time Image Processing*, vol. 15, pp. 297–308, Aug. 2018.

[51] S. Sánchez and A. Plaza, "Fast determination of the number of endmembers for real-time hyperspectral unmixing on gpus," *Journal of Real-Time Image Processing*, vol. 9, pp. 397–405, Sept. 2014.

[52] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[53] N. Zeghidour, N. Usunier, I. Kokkinos, T. Schatz, G. Synnaeve, and E. Dupoux, "Learning filterbanks from raw speech for phone recognition," *CoRR*, 2018.

[54] Computer Vision Machine Learning Team, "An On-device Deep Neural Network for Face Detection," *Apple Machine Learning Journal*, vol. 1, 2018.

[55] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *CoRR*, 2014.

[56] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation," *CoRR*, 2016.

[57] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on Vision Processing Unit for Mobile Applications," *IEEE Micro*, vol. 35, pp. 56–66, Mar. 2015.

[58] "GAP8, the Industry's Lowest Power IoT Application Processor."

[59] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[60] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[61] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, pp. 664–676, Apr. 2017.

[62] S. Haykin, *Neural Networks: A Comprehensive Foundation (3rd Edition)*. 2009.

[63] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[64] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. III–1319–III–1327, JMLR.org, 2013.

[65] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[66] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[67] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surface of multilayer networks," *CoRR*, vol. abs/1412.0233, 2014.

[68] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[69] https://developer.arm.com/technologies/compute library.

[70] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[71] https://bazel.build/.

[72] https://www.linaro.org/latest/downloads/.

[73] W. H. Organization., *WHO global report on falls prevention in older age*. World Health Organization Geneva, 2008.

[74] X. Fafoutis *et al.*, "Extending the Battery Lifetime of Wearable Sensors with Embedded Machine Learning," IEEE, 2018.

[75] N. Pannurat *et al.*, "Automatic Fall Monitoring: A Review," *Sensors*, vol. 14, pp. 12900–12936, jul 2014.

[76] M. Hemmatpour *et al.*, "A neural network model based on co-occurrence matrix for fall prediction," in *Wireless Mobile Communication and Healthcare*, pp. 241–248, Springer, Cham, Nov. 2017.

[77] R. Igual *et al.*, "Challenges, issues and trends in fall detection systems," *BioMedical Engineering OnLine*, vol. 12, p. 66, July 2013.

[78] M. Nyan *et al.*, "A wearable system for pre-impact fall detection," *Journal of Biomechanics*, vol. 41, pp. 3475–3481, Dec. 2008.

[79] S. Jung *et al.*, "Wearable Fall Detector using Integrated Sensors and Energy Devices," *Scientific Reports*, vol. 5, p. 17081, Dec. 2015.

[80] S. Abdelhedi *et al.*, "Design and implementation of a fall detection system on a Zynq board," in *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–7, IEEE, Nov. 2016.

[81] G. Cola, M. Avvenuti, P. Piazza, and A. Vecchio, "Fall detection using a head-worn barometer," in *Wireless Mobile Communication and Healthcare* (P. Perego, G. Andreoni, and G. Rizzo, eds.), pp. 217–224, Cham: Springer International Publishing, 2017.

[82] F. Hossain *et al.*, "A direction-sensitive fall detection system using single 3D accelerometer and learning classifier," in *2016 International Conference on Medical Engineering, Health Informatics and Technology (MediTec)*, pp. 1–6, IEEE, Dec. 2016.

[83] A. Nicosia *et al.*, "Efficient light harvesting for accurate neural classification of human activities," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–4, Jan. 2018.

[84] STMicroelectronics, *SensorTile development kit*, 7 2018. Rev. 7.

[85] A. Nicosia, D. Pau, D. Giacalone, E. Plebani, A. Bosco, and A. Iacchetti, "Efficient light harvesting for accurate neural classification of human activities," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–4, Jan. 2018.

[86] E. Torti, A. Fontanella, M. Musci, N. Blago, D. Pau, F. Leporati, and M. Piastra, "Embedded real-time fall detection with deep learning on wearable devices," *21st Euromicro Conference on Digital System Design (DSD)*, Prague, Czech Republic, 2018.

[87] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[88] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Comput.*, vol. 12, pp. 2451–2471, Oct. 2000.

[89] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 2342–2350, JMLR.org, 2015.

[90] F. J. Ordóñez and D. Roggen, "Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition," *Sensors*, vol. 16, no. 1, 2016.

[91] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1131–1135, Apr. 2015.

[92] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," *CoRR*, vol. abs/1412.6115, 2014.

[93] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," *CoRR*, vol. abs/1306.0543, 2013.

[94] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[95] S. S. Talathi and A. Vartak, "Improving performance of recurrent neural network with relu nonlinearity," *CoRR*, vol. abs/1511.03771, 2015.

[96] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or –1," *CoRR*, vol. abs/1602.02830, 2016.

[97] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights +1, 0, and –1," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, Oct. 2014.

[98] A. Sucerquia, J. D. López, and J. F. Vargas-Bonilla, "Sisfall: A fall and movement dataset," *Sensors*, vol. 17, no. 1, 2017.

[99] https://developer.arm.com/embedded/cmsis.

[100] OpenMP Architecture Review Board, “OpenMP application program interface version 2.0,” May 2002.

[101] https://computing.llnl.gov/tutorials/openMP/.

[102] NVIDIA, *CUBLAS Library User Guide*, v9.2 ed., 2018.