Enabling Big Data Analytics For MATLAB Programs Using High Performance Compute Methods

Yun Lu

CE-MS-2018-11

Delft University of Technology

Abstract

In this work, possible solution to allow for scalable MATLAB deployment on big data clusters through Spark without using the official MATLAB toolbox is introduced. Other possible solutions that can be used for accelerating existing MATLAB code including calling modules written by Graphics Processing Unit (GPU), and Python Pool with multi processors are also investigated in this thesis. Among these approaches, Spark solution is achieved by accessing to PySpark through Python. Instead of using distributed computing server of MATLAB that is necessary for the official Spark approach in the newest version, our approach is low-cost, easy to set up, flexible and general enough to handle changes, and enable for scaling up. All the solutions are analyzed for bottlenecks based on their performance in initialization, memory transfer, data conversion and computational throughput. Our analysis shows that initialization & memory transfer for GPU, data conversion for Python/Pyspark when the data input or output have high dimensions can be bottlenecks. For use case analysis, a medical image registration MATLAB application using NCC was accelerated by multiple solutions. The results indicate that GPU and PySpark using cluster have the best performance, which were 5.7x and 7.8x faster than MATLAB with Pool performance. Based on the overall performance of these solutions, a decision tree for the most optimal solution to choose is built for the future research.

TU Delft Delft University of Technology

Quantum & Computer Engineering

Enabling Big Data Analytics for MATLAB Programs Using High Performance Compute Methods

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

Embedded Systems

by

Yun Lu
born in Shanghai, China

Embedded Systems
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Enabling Big Data Analytics for MATLAB Programs Using High Performance Compute Methods

by Yun Lu

Department : Quantum & Computer Engineering
Codenumber : CE-MS-2018-11

Committee Members :

Advisor:	Zaid Al-Ars , CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Claudia Hauff, WIS, TU Delft
Member:	Arjan van Genderen, CE, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

In this digital era, people are exposed to an ever increasing amount and diversity of data, both structured and unstructured. At the same time, tons of new data is generated daily from all kinds of external sources. Camera recording resolution increased from HD, Full HD, to now Ultra HD (4K), which requires new video encoding standards, for example High Efficiency Video Coding (HEVC), which achieves significantly better compression performance for high resolution video but results in extremely high computational complexity [1]. An important application and emerging research area in image processing is object tracking in video surveillance [2]. Tracking a target in a cluttered environment is still challenging. The whole process includes object detection, classification, tracking and identifying the behavior. Another interesting topic that is becoming an active field of research over recent years is image reconstruction, which refers to the process of restoring missing or damaged areas in an image. [3] discusses the latest categories of image reconstruction methods and several applications are illustrated.

Not only is the complexity of image processing increasing, the demands for rapid response or high throughput also become important thanks to the amount of data. Using high-throughput video compression technique as an example, a series of research projects have been done on VLSI architecture design that presents superior performance in parallelism-efficiency for specific algorithms [4] or image compression systems [5].

Among publications related to high-throughput image analysis in recent year, a big part of them were related to medical imaging and biomedicine. Medical images usually have higher dimensionality (usually 2D and 3D) compared to non-medical images, in addition to an even bigger amount of data. Correspondingly, the algorithms used for the analysis in this field are more complicated, making high performance computer aided image processing techniques increasingly important. Some of these algorithms use state-of-the-art pattern recognition technologies for image processing. Jafari et al. [6] show an approach for the detection of brain tumor tissue in magnetic resonance images based on genetic algorithms and support vector machines (SVMs). Sneha et al. [7] present an algorithm for fusing computerized tomography (CT) and magnetic resonance (MR) medical images using biologically inspired spiking neural network. A typical basic image processing system contains the following three stages: acquisition, image processing and description. In the first stage, biomedical images ar acquired by image modalities, such as CT, magnetic resonance tomography (MRT), positron emission tomography (PET), or ultrasound (US). The image processing stage can be further separated into four steps, pre-processing, segmentation, registration, and diagnosis. Image denoising is an important part of pre-processing because medical images encounter noise from various number of sources caused by acquisition, storage and transmission. Pre-processing is

followed by the segmentation step for detection of object boundaries(can be organs, cells, vessels, etc.) and extracting the region of interest. Image registration is one of the key steps in the whole system. It transforms different sets of data into one coordinate system. If the data of two images is acquired by the same type of machine, then image registration is to align images with each other. Or if the images show the same object from different image modalities, time or viewpoints, then image registration can be used to fuse the images together. Using existing databases, and classification algorithms, the final diagnosis can be reached. The last stage of the whole image processing process is the description of diagnosis, using visualization, plain text or tags.

MATLAB is one of the most widely used computational platforms for medical image processing algorithms, which is one of the popular choices of companies. It provides a bunch of functionality, built in tools and toolboxes in different fields. It is easy and quick to develop a runnable application, which also can interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python. According to analytics from iDatalabs, the top two industries that use MATLAB are higher education and computer software. Some other industries that have large segments of MATLAB customers include hospital & health care providers, financial services, aerospace, automotive and medical devices. There are more than 20,000 organizations or companies globally that use MATLAB including Howard Hughes Medical Institute and Facebook. Of all the companies that use MATLAB, 31% are small (<50 employees), 37% are medium-sized and 27% are large (>1000 employees). To keep pace with cutting edge technology, MATLAB also started including Spark scalability features as of their 2017a version. However, MATLAB license is quiet high priced, and toolboxes are charged separately. Running MATLAB on clusters requires a special license, and the price increases with the size of the cluster. The minimum number of nodes in a cluster is 16. Due to the fact that MATLAB is a scripting language, it runs slower than compiled languages. To use multiprocessors in MATLAB to improve performance also needs an extra license [8].

Based on the characteristic of the target users and the algorithms itself, the amount of scalability is different for each solution. For example, scalability is important for solutions created by small companies interested in building a prototype to analyze massive amounts of data quickly. Another example, is represented by the solution for those companies that already use MATLAB and want to explore use cases that involve big data. The former group would prefer a cheap, easy to use and test solution, while the latter one may need a solution that can scale up, have high performance and is easy to maintain.

1.2 Challenges

MATLAB is a closed-source software, the source code is not shared publicly for users to look at or change, just like a black box. The benefit of this closed-source model is that MATLAB has sustained support, reliability and security which is preferred by the research and development departments in companies. However, it also limits the flexibility of MATLAB. This makes it challenging but important to customize the existing MATLAB functions without affecting the core part of the platform. Also, the limited flexibility causes extra effort when integrating MATLAB with clusters or calling other modules. One of the problems will be sharing the variables between MATLAB and those

clusters or processes. MATLAB protects variables it stores and generates from directly being manipulated by hiding the physical memory address of the data from users. Beside this, the success of MATLAB is also due to the ease of programming and due to the fact that it does not require compiling the code.

Recently MATLAB enabled support for Spark. Those who want to update their MATLAB version in order to use this new feature need to renew their license. In addition to the cost of renewing MATLAB, extra cost will be spend on the license that is needed for the cluster. This expense is based on the number of nodes in the cluster, and starts from 5072 euros. There is only one possible configuration for the cluster, all nodes inside the cluster, both workers and the name node, need to install the MATLAB Distributed Computing Server. In addition, for the client node that writes and submits MATLAB applications, the Parallel Computing Toolbox is needed. It is also highly recommends by MathWorks to install all MathWorks products because MATLAB Distributed Computing Server cannot run the jobs whose code requires products that are not installed. It can be more flexible if there is a way to map MATLAB code on a cluster without MATLAB installed on the nodes.

In this thesis, we are aiming to provide solutions to enable low-cost open source tool set to allow for scalable MATLAB deployment on big data clusters through Spark.

Debugging through MATLAB is also an issue. Users should receive the error messages and oversee the ongoing processes in other processes directly from MATLAB. So there is no need for users to switch to another IDE while debugging. Modification of accelerating modules should be also done from MATLAB, this means our solutions should provide interfaces to MATLAB and can be invoked in the same way as built-in function.

Different type of problems will benefit from specific acceleration methods. Also not all situations are suitable for cluster scalability. Other solutions that can be taken into account are multithreads and graphics processing unit (GPU) acceleration. MATLAB also has its own implementations for these solutions included as part of extra toolboxes. In the same way MATLAB integrates Spark, these official solutions also have restrictions in functions or environments and are limited in usage. One of the challenges will be how to sidestep these limitation and still provide improvement in performance.

1.3 Problem definition

In this thesis, we aim at providing some proof-of-concept solutions and implementations that accelerate existing MATLAB code, with usage of multicore processors, GPU and Apache Spark. Furthermore, we aim at comparing these solutions to inspect the bottlenecks by executing a series of micro-kernels. Finally, the solutions are applied to an image registration MATLAB application to observe the influence of these solutions on a specific use-case.

Based on the challenges described in the previous section, we define the following research questions to resolve those challenges:

- Is it possible to easily scale up existing MATLAB code on an Apache Spark cluster using open source tools?

- What applications suitable for a scalable MATLAB implementation, and what is the overhead of scaling MATLAB code on Spark?
- What are the bottlenecks in terms of performance, memory and interconnect bandwidth that limit the speedup gained from Spark scalability?
- What is the advantage of using open source tools to scale up MATLAB code in comparison with the official MATLAB scalability solution? What are the advantages and disadvantages?

1.4 Thesis outline

The remainder of the thesis will cover the following:

- Chapter 2. **Background**. This chapter will present important details regarding the framework, tools and hardware that will be used in this thesis. A brief description is given on the MATLAB environment to be used in the thesis, Spark framework and GPUs.
- Chapter 3. **Use case analysis and alternative solutions**. In this chapter, a medical image analysis algorithm containing the registration process use case is presented and analyzed. This chapter also includes a brief overview of past-to-recent literature for image registration. Then it examines possible solutions for the use case
- Chapter 4. **Implementation and measurements**. This chapter begins by a discussion of the different solutions implementations in practice. Then it investigates the overheads in each type of solutions. It includes time consumption during initialization, memory copy, data conversion and computational throughput.
- Chapter 5. **Discussion**. In this chapter, we discuss the advantages and disadvantages of each solution. A system level analysis is included in the final section of this chapter.
- Chapter 6. **Image analysis use case**. This chapter begins by the implementation of the use case using different solutions. Followed by a description and evaluation of the result.
- Chapter 7. **Conclusions and future research**. In the last chapter, we will provide a summary of the thesis and our solutions, as well as our view on the future. Lastly, the questions in the problem definition part will be answered based on the experiment outcomes.

2 Background

This chapter introduces relevant work in the literature related to the topic of this thesis. First, the chapter starts with discussing the MATLAB platform (from MathWorks) and continues with Apache Spark (used for cluster computing) in order to establish a link between MATLAB and Spark.

2.1 MATLAB environment

MATLAB (matrix laboratory) is a matrix-based language designed for engineering and scientific mathematical computations; it was developed by MathWorks in 1984. MATLAB runs directly from the source scripting files (called m-files), creating the executable code on-the-fly. It has optional add-on toolboxes for a wide range of engineering and scientific applications such as image processing, signal processing, and machine learning. It also provides interfaces to C/C++, Java, .NET, Python, SQL, Hadoop, and Microsoft Excel [9].

2.1.1 History

MATLAB was originally written in C with libraries known as JACKPAC. In 2000, MATLAB was rewritten to use a newer set of libraries for matrix manipulation, LAPACK [10]. LAPACK is a large, multi-author, Fortran library for numerical linear algebra that is designed to exploit level 3 BLAS. BLAS stands for Basic Linear Algebra Subroutines, which are routines that provide standard building blocks for performing basic vector and matrix operations. Level 1 BLAS performs scalar, vector and vector-vector operations, Level 2 BLAS performs matrix-vector operations, and the Level 3 BLAS performs matrix-matrix operations. LAPACK uses block algorithms, which operate on several columns of a matrix at a time. On machines with high-speed cache memory, these block operations can provide a significant speed advantage [11]. MATLAB with LAPACK can offer the opportunity to use multithreading and multiprocessors for additional speed enhancements.

2.1.2 Data types and storage

All numeric variables in MATLAB are stored by default as double-precision floating-point values. Additional data types store text, integer or single-precision values, or a combination of related data in a single variable.

All MATLAB variables are multidimensional arrays. To create an array, MATLAB allocates a contiguous virtual block of memory and store the array in that block. If there is a need to expand the array beyond the available contiguous memory of its original

location, MATLAB has to search for a new contiguous location and copy the contents of the original array, add the new elements and free the original memory. When MATLAB performs an array copy, it only makes a copy of reference unless the content of the array is changed, then MATLAB has to make a copy of the array and modify it.

When the data is too large to fit in memory, MATLAB enables one to create *datastore*. A datastore is an object for reading a single file or a collection of files or data. It allows you to read and process multiple files which have the same structure and formatting as a single entity. When the file is too big, a datastore allows you to read and analyze data from each file in smaller portions that do fit in memory.

The workspace contains variables that you create within (or import into) MATLAB from data files or other programs. Workspace variables do not persist after you exit MATLAB. However, a workspace can be preserved in your current working folder in a compressed file with a .mat extension, called a MAT-file. This is available with a simple "save" command.

2.1.3 Acceleration

There are plenty of ways to accelerating MATLAB algorithms and applications. This varies from translating the scripted MATLAB code to C code or platform specific code, using the Parallel Computing Toolbox to using in-app existing optimized algorithms. Some of the methods like optimizing serial code using preallocation and vectorization or package to C code are still running on the CPU. Since most of the computers or processors nowadays have multicores, MATLAB by default runs some linear algebra and numerical functions and those included in the Image Processing Toolbox multithreaded since Release 2008a. MATLAB programs can also be executed on multiple MATLAB computational engines on a single machine to execute applications in parallel, with the Parallel Computing Toolbox.

However, as a general purpose processor, a CPU is not optimized for particular computations. Sometimes it is more efficient and natural to take advantage of other platforms or hardware for acceleration and scaling up. MATLAB provides interfaces and toolboxes for the following hardware/platform (shown in Figure 2.1):

- Graphics processing Units (GPUs) are designed to do image processing jobs like texture mapping, filtering and translation. MATLAB provides computations on CUDA GPUs (only support NVIDIA GPUs) through the Parallel Computing Toolbox.

- A digital signal processor (DSP) has an architecture optimized for digital signal related operations, especially when processing quickly and repeatedly on a series of data samples. MATLAB uses the DSP System Toolbox to design, simulate and generate code for DSP.

- Field-programmable gate arrays (FPGAs) combines the advantages from general purpose processors (GPPs) and application specific integrated circuits (ASICs). It is more efficient than using GPPs in terms of power and performance and also easier, cheaper and faster than developing in ASICs. With the HDL Coder and

HDL Verifier in MATLAB, one can design and generate target-independent or target-optimized HDL code for FPGAs.

- Some problems are so complicated that they take hours or more of computation. To run these computationally intensive programs simultaneously, MATLAB Distributed Computing Server and Parallel Computing Toolbox are the solutions to scale up MATLAB programs from single computer to all resources in clusters and cloud computing services. MATLAB Distributed Computing Server supports both built-in cluster job scheduler and commonly used third-party schedulers like Microsoft Windows HPC Server and IBM Platform LSF. If one wants to scale up the applications to the cloud, there are multiple options based on factors like maximum number of workers needed or the cloud service provider, which can be checked on the following web page [12].

- MATLAB provides the MATLAB Distributed Computing Server and MATLAB Compiler for processing big data that does not fit in memory of a desktop or cluster. This includes accessing data from Hadoop Distributed File System (HDFS) and running algorithms on Apache Spark or HDFS. However, both Hadoop and Spark are supported only on Linux recently. MATLAB Compiler support for MATLAB objects for Hadoop integration was added in features in version 6.0 (R2015a). The datastore: Tall Arrays which is used to work with out-of-memory data was first introduced in version 6.3 (product R2016b), as well as the support of Spark.

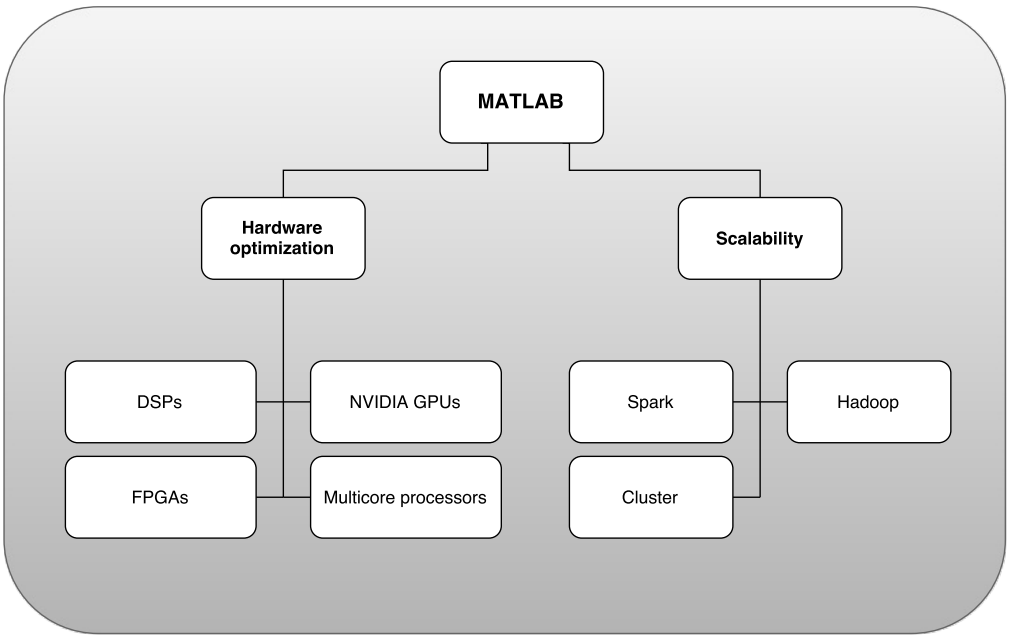


Figure 2.1: MATLAB code acceleration using hardware optimization and scalability

2.1.4 Integration with other languages

MATLAB also provides two-way integration with other programming languages [13], including:

- Calling MATLAB from another language
- Calling libraries written in another language from MATLAB
- Converting MATLAB code to C/C++ code
- Packaging MATLAB programs

Different languages are supported from different MATLAB release version. JAVA integration has been available since Release 12 (MATLAB 6.0). Since Release 12, MATLAB has been always shipped with a bundled JAVA engine (JAVA Virtual Machine or JVM). Python integration was first added in version MATLAB 2014b.

One way to accomplish that is to use MATLAB from within another programming environment via a MATLAB Engine API. Engine programs are standalone programs. These programs communicate with a separate MATLAB process via pipes, on UNIX systems, and through a Microsoft Component Object Model (COM) interface, on Microsoft Windows systems. MATLAB provides a library of functions that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB. When using APIs, one can use MATLAB commands within another programming language without starting a desktop session. Since it only links to a smaller engine library, it can work faster. MATLAB provides Engine APIs for C/C++, Fortran, Java, Python, COM components and applications, as shown in Figure 2.2. It is useful in automating tasks and in applications where MATLAB is an equally privileged, equipotent participant in the application.

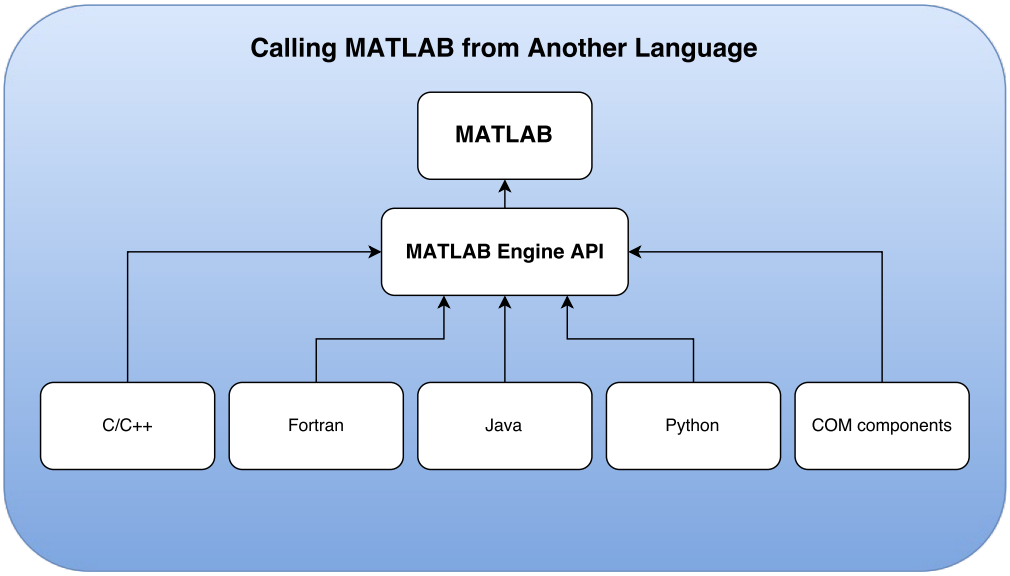


Figure 2.2: MATLAB Engine based approaches

On the other hand, MATLAB can also call functions and objects in another programming language within MALTAB. This is more straight forward compared to using MATLAB from other languages. The way to call functions from different languages is quite consistent. First step is to make sure the path for functions, classes, or objects is available to MATLAB. The next step is to load the library/function/class/objects. After loading them, you can request information about library functions and call them directly from the MATLAB command line. Those imported packages use lazy loading, so they are only loaded into memory when actually needed. When they are no longer need them, they can be unloaded from memory to release that part of the memory. The tricky part, however, occurs in data passing between MATLAB and a function in other languages. For example, automatic-type conversion occurs only when passing data from MATLAB to Java, but only partially on the reverse path. Another important observation to remember is how the data passes between MATLAB and other languages. According to the book *Undocumented Secrets of MATLAB-Java Programming* [14], all data passed from MATLAB to Java except for object references are passed by value, whereas all objects (nonprimitive types) returned from Java are passed by reference. Which means if the Java function wants to modify MATLAB data, the MATLAB data should be encased in a Java reference using MATLAB's javaArray function. MATLAB ships with its own JVM software, which is fully supported by MATLAB. This JVM is only used to enable MATLAB to access Java classes, not to act as an internal Java interpreter. For this reason, some components might not work or some methods (typically used by event callback actions) cannot be directly accessed from MATLAB.

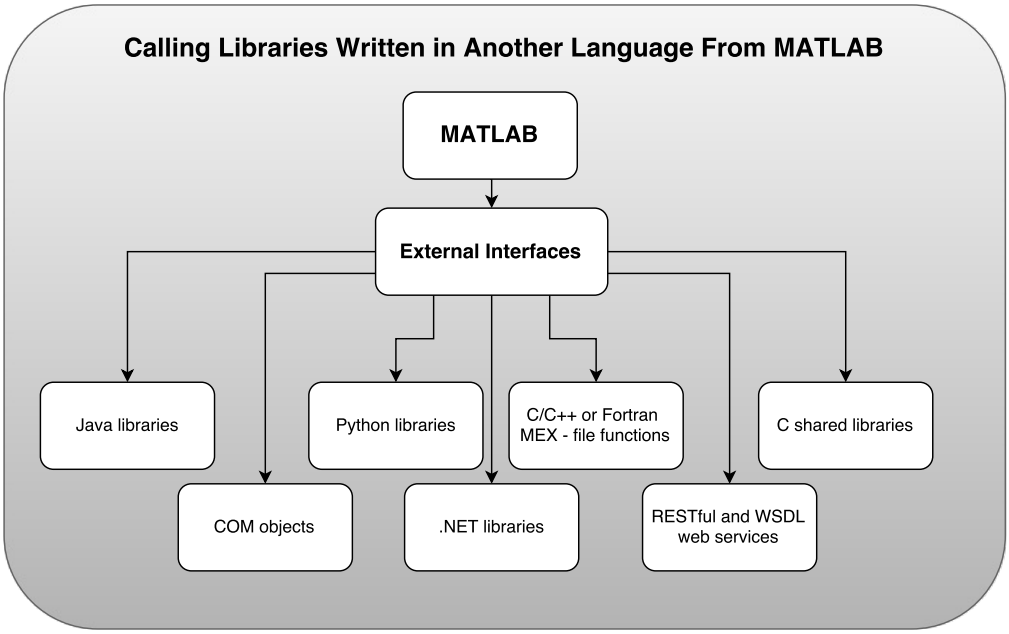


Figure 2.3: Calling Libraries Written in Another Language From MATLAB

For those who want to share the MATLAB functionality royalty-free with people who do not have MATLAB installed, or want to deploy MATLAB as software components

(that can be integrated into web and enterprise systems), it is possible to leverage the power of MATLAB Runtime and MATLAB Compiler SDK, which is shown in Figure 2.4. Before integrate MATLAB functions into external applications, those functions need to be compiled (into a library or an installer) to platform specific shared libraries for the target language using MATLAB Compiler SDK. This compiled code can be installed (with the installer got from last step) or integrated (you get a binary package the target language from last step) to other applications. To run this part of code, one need MATLAB Runtime. The MATLAB Runtime is a standalone set of shared libraries that enables the execution of compiled MATLAB applications or components on computers that do not have MATLAB installed. During the execution, MATLAB Compiler SDK uses APIs to initialize the MATLAB Runtime, load the compiled MATLAB functions into the MATLAB Runtime, and manage data that is passed between the target language code and the MATLAB Runtime.

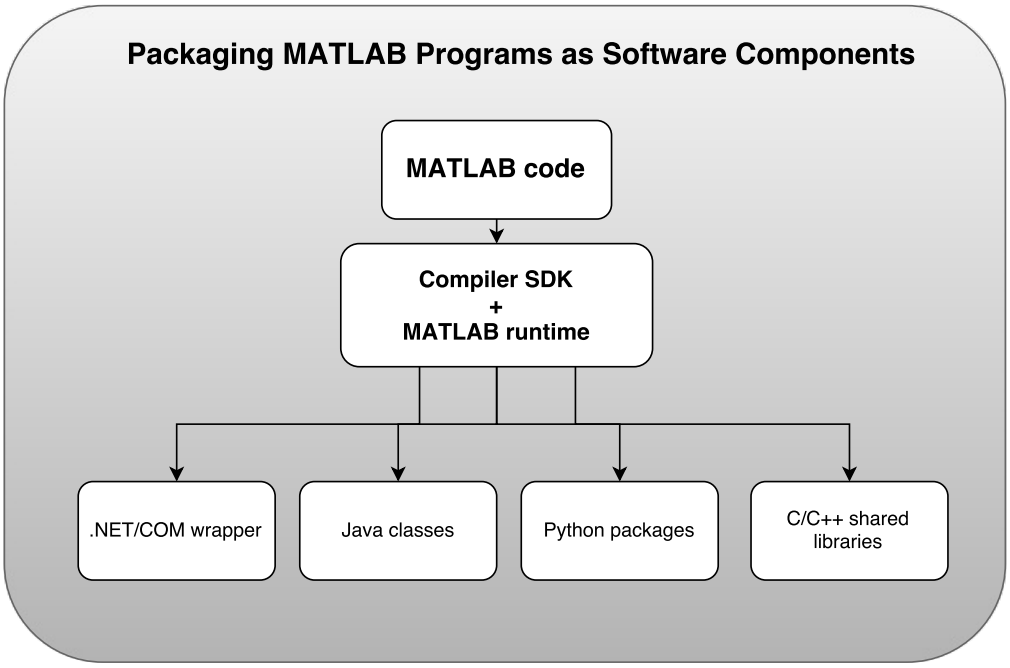


Figure 2.4: Packaging MATLAB Programs as Software Components

For those environments which are not suitable for MATLAB Runtime, such as embedded system, MATLAB also offers MATLAB Coder to generate standalone readable and portable C and C++ code from your MATLAB code (Figure 2.5). The generated code can also be integrated as source code, static libraries, or dynamic libraries.

2.2 Spark framework

Spark is built with Scala and runs on JVM. It is an open-source and generalized cluster computing framework centered on a data structure called the resilient distributed dataset (RDD) for distributed data processing using in-memory data caching.

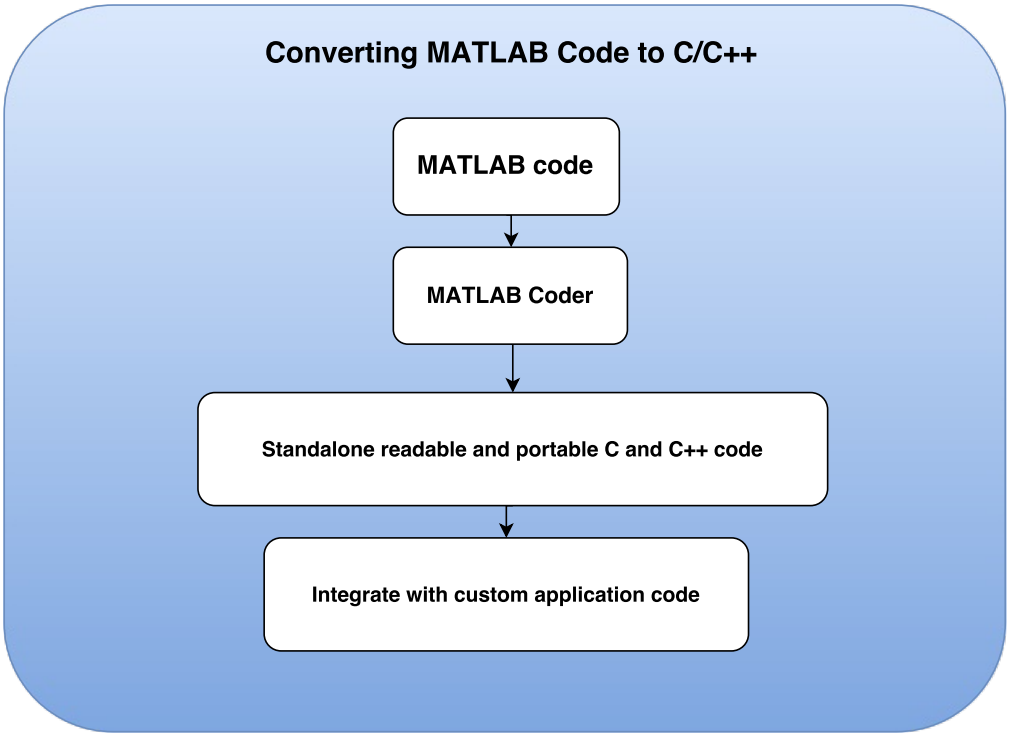


Figure 2.5: Converting MATLAB Code to C/C++

2.2.1 Spark architecture

The architecture of Spark is shown in Figure 2.6. Spark Core is the foundation of the Spark framework, it contains all the basic functionality including distributed tasks dispatching, scheduling, fault recovery, and memory management. Spark Core is exposed to users through application programming interfaces (APIs). Spark (version 2.2.0) provides APIs in Scala, Java, Python, and R. There are also libraries/modules provided by Apache Spark for some specific uses. Those libraries including Streaming for processing real-time data streams, Spark SQL, Datasets, DataFrames for structured data and relational queries, MLlib for machine learning, and GraphX for graph processing. As a cluster computing framework, Apache Spark needs a cluster manager as well as a distributed storage system. Spark has its own simple cluster manager: Standalone, it is a basic cluster manager included with Spark that makes it easy and fast to set up a cluster. It can also be deployed on Hadoop YARN and Apache Mesos. For distributed storage, Aparch Spark provides more flexible choices [15] like Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu or custom storage. Spark also supports a pseudo-distributed local mode which aims for developing and testing, where distributed storage can be replaced by the local file system. In this mode, Spark is run on a single machine with one executor per CPU core.

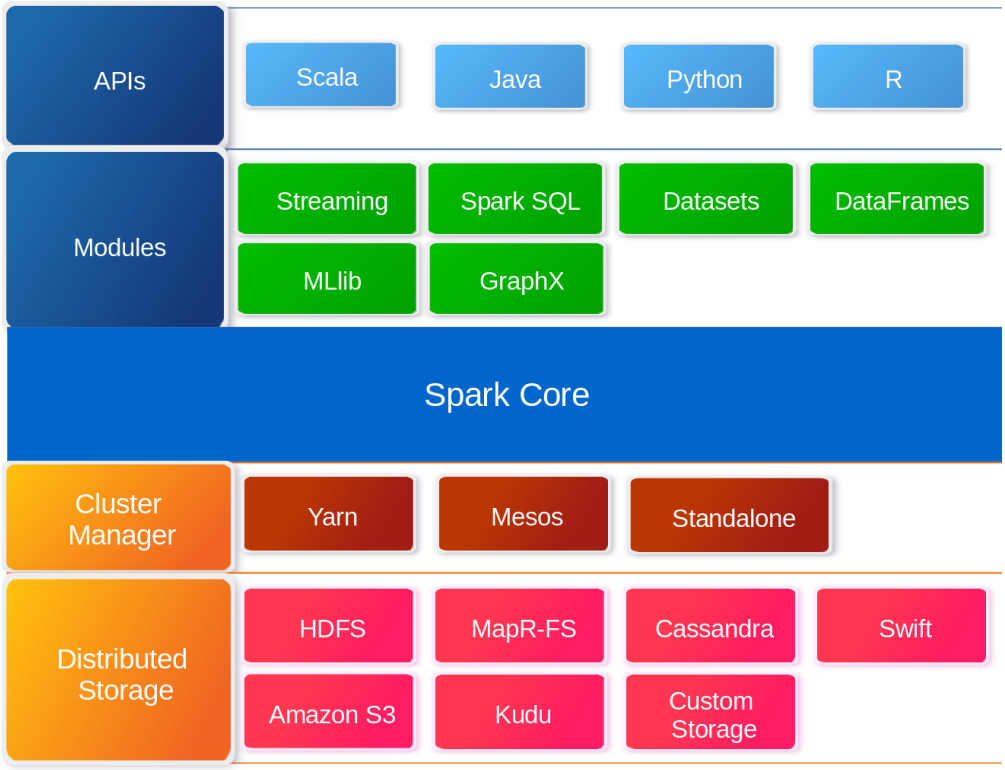


Figure 2.6: Spark architecture

2.2.2 Spark RDD

The most important abstraction in Spark architecture is resilient distributed dataset (RDD) which is the key for in-memory computations on large clusters while being fault-tolerant. RDDs are immutable, partitioned collections of elements that can be operated on in parallel. The data is split into partitions and stored in memory or disk across a cluster of machines. Those partitions are the units of parallelism. There are two general types of operations allowed on an RDD:

- Transformations, that transform (e.g. map, filter, join, etc.) RDDs into new RDDs. Each transformation needs personalization with user-defined behavior (i.e. personalized by a function applied on elements of the collection stored in the RDD)
- Actions, which return a value to the driver program after running a computation on the dataset.

RDD is lazy evaluated, which means the data inside RDD is not available or transformed until an action is executed that triggers the execution. Data stored inside RDDs is stored in memory as much (size) and long (time) as possible, but can also be stored persistently into storage like disk. When designing the computation on Spark, one needs to always consider the balance between the data distribution and data locality, which contributes to the time consumption of parallel computation and data transformation across the network by means of RDD shuffling.

2.2.3 Spark execution

Spark uses a master/slave architecture. It has one central coordinator (driver) that communicates with many distributed workers (executors). The driver and each of the executors run in their own Java processes. The Spark execution process is shown in Figure 2.7.

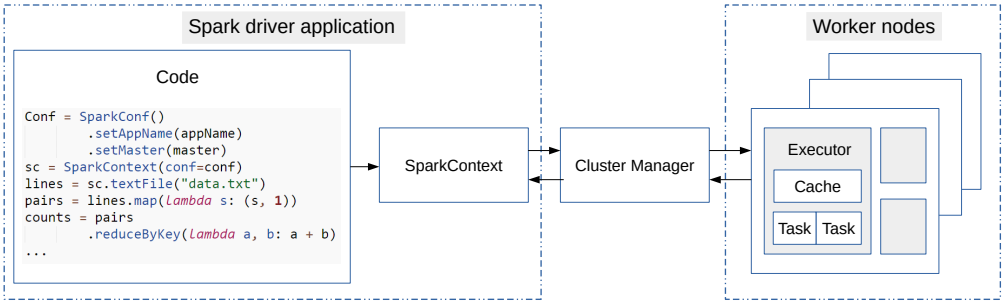


Figure 2.7: Spark execution process

To run Spark on a cluster, the first step of is to create a SparkContext from the Spark driver application. A SparkContext is the core part in Spark that manages the connections to the clusters like Hadoop YARN, Apache Mesos, or standalone, and coordinates running processes on the clusters. In order to create a SparkContext one should first create a SparkConf which contains properties of the SparkContext, such as deployment environment (as master URL), memory size used by executors, and application name. After the SparkContext is created, it can help to get current status of Spark applications, set configuration like default logging level, create distributed entities (e.g. RDDs, accumulators and broadcast variables), access Spark services (e.g. TaskScheduler, BlockManager, ShuffleManager and SchedulerBackends), and run or cancel jobs through DAGScheduler [16].

Once the SparkContext is connected to cluster managers, Spark is able to request resources from cluster managers (currently only CPU and memory are resources that can be requested). It is the cluster manager responsibility to launch Spark executors in the cluster. Executors are worker nodes processes in charge of running individual tasks in a given Spark job. Usually one node in a cluster is one worker, and it holds many executors for many applications (standalone cluster manager, only allows one executor per worker process on each physical machine). The workers are in charge of communicating the cluster manager the availability of their resources. Next, the SparkContext sends the application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, it sends tasks to the executors to run. Executors run the tasks and save the results, or send the result back to the driver.

2.2.4 Spark vs Hadoop

MATLAB Compiler support for Hadoop integration started since R2015a and for Spark since R2016b, both of MATLAB integration required the Linux operating systems. Both Spark and Hadoop provide tools to carry out common big data related tasks. They have

the following main differences:

- Hadoop is a framework which consists of both data storage (HDFS), a distributed processing (YARN), and a processing component called MapReduce.

  Spark is a fast and general computing engine for Hadoop data. It does not have its own distributed storage mechanism, which means that it needs to be run on one of the existing distributed storage systems like HDFS or S3, if a scalable file system is needed. For this reason, it is more fitting to compare Spark with MapReduce rather than with the entire Hadoop system.

- Spark was released 3 years after Hadoop was initially released, it uses different architectural and implementation than Hadoop used. Hash-mased aggregation in Spark is more efficient than sort-based aggregation in MapReduce for word-count like workloads. MapReduces execution model is more efficient for shuffling data than Spark, thus making Sort run faster on Spark [17].

- Spark uses in-memory engine. It can also use disk for reusable data, or data which doesn't fit into memory. While MapReduce is a disk-bound batch processing engine, MapReduce operates in steps between each iteration. First, the results have to be written to the cluster, then the updated data has to be read from the cluster for the next computation. Spark stores most of the data into RAM. This reduces the amount of time for writing and reading, and thus can complete the full processing procedure in near real-time. As a result, for iterative algorithms for machine learning or other computations with subsequent iterations, Spark is a better choice than MapReduce.

- Hadoop and Spark use different fault tolerance strategy. In MapReduce, there is a JobTracker that keeps tracking TaskTrackers by receiving heartbeat periodically. If one TaskTracker is considered to have failed, the JobTracker will reschedule all pending and in progress tasks to other TaskTrarckers.

  Spark uses RDDs. Instead of storing the data, RDDs log the transformations, so called RDD lineage graph. If any partition of an RDD is lost or damaged due to node failures, it will automatically be recomputed using the transformations that originally created it.

To summarize, MapReduce and Spark can both process batch and iterative jobs. MapReduce has better performance for sorting. Spark has more advantages for real-time processing, computations that have several iterations like graph processing, and iterative machine learning algorithms.

2.3 Graphics processing unit

A graphics processing unit (GPU) is a specialized electronic circuit that has massively parallel architecture, this makes them more efficient than general-purpose processor for some compute-intensive algorithms that need to handling multiple tasks in parallel. GPUs are now used for rapid mathematical calculations, generally using in rendering

images. NVIDIA, AMD, Intel and ARM are some of the major companies that make GPUs products.

General-purpose computing on graphics processing units (GPGPU) is a methodology that allows programmers using GPUs to perform general purpose computing, which are traditionally handled by the CPU. Algorithms that are data parallel and throughput intensive are suited to GPGPU.

Comparing with a central processing unit (CPU), a GPU has more cores that can handle multiple threads simultaneously, but with less cache memory and registers, and slower clock speed, which makes it less powerful for individual serial tasks than a CPU. There is a considerable amount of literature on GPU acceleration used in medical field, including image processing [18] and alignment or mapping of genomics sequencing data [19] [20] [21] [22] [23].

Usually GPU-accelerated computing needs the cooperation with CPU, those compute-intensive portions of the application is accelerated on GPU while the remainder of the codes still runs on the CPU.

2.3.1 Compute Unified Device Architecture (CUDA)

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model invented by NVIDIA. It works with all Nvidia GPUs from the G8x series onwards, including GeForce, Quadro and the Tesla line.

It comes with CUDA Toolkit, a complete environment for building scalable GPU-accelerated applications in C, C++, Fortran and Python for CUDA-enabled GPU. It provides libraries, compiler directives, and extensions to industry-standard programming languages (C, C++ and Fortran). It also supports other computational interfaces like OpenCL and OpenGL. The core of CUDA are three abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization.

2.3.2 Execution flow

In a CUDA application, the Host refers to the execution environment that initially invoked CUDA, typically the thread running on a system's CPU processor. And the Device refers to the GPUs. And the piece of data parallel C function that is executed by blocks of threads in device is called a kernel. The CUDA programming model assumes that CUDA threads execute on the physically separate device to the host who runs the C program, so the serial code on the host and the parallel kernel on the GPU can be executed simultaneously. The model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory. Therefore programmer need to manage both of the memory spaces, including allocation, deallocation on both side as well as data transfer between host and device memory. The execution flow of CUDA program is shown in Figure 2.8. During the runtime, the host first copy data from host memory to device memory, then the host instructs the process to GPU, after kernels are finished on GPU, it copies the result from the device back to the host.

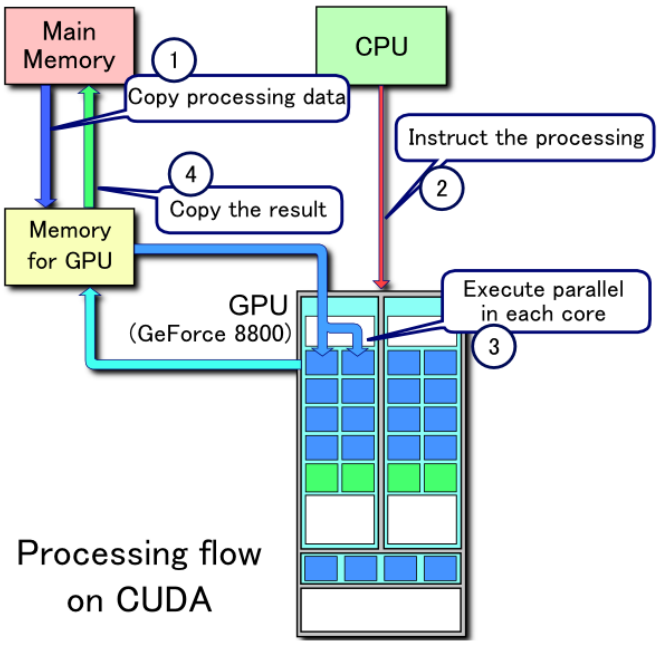


Figure 2.8: CUDA processing flow. From Example of CUDA processing flow, Wikipedia, `https://en.wikipedia.org/wiki/CUDA`, [Online; accessed 24-July-2017]

2.3.3 Thread hierarchy

As we mentioned before, kernel is executed by threads. From programming view, those threads are given a unique thread ID (threadIdx), the thread ID is a three-component vector, so the threads can form a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. Thread blocks also has a three-component vector as its id, can also organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks.

When it comes to hardware, an entire grid is handled by a single GPU chip. Inside the GPU, there are a collection streaming Multiprocessors (SMs), each SM has its own: control units, registers, execution pipelines, caches and a group of CUDA Cores. Each thread defined in the CUDA program, is executed by a core, and each block is executed by one SM (more than one blocks can be executed concurrently when block's memory allows). And the maximum threads that a SM can handles is represented as warp size.

Figure 2.9 shows the thread hierarchy of thread groups of CUDA,

2.3.4 Memory hierarchy

Threads can read and write data from register and different types of memory, different memory are differ in the access latency and scope. The memory hierarchy of CUDA is shown in Figure 2.10.

First all, each thread has its own registers and local memory. The scope of registers is thread local. Access to registers is the fastest, but the number of registers is limited for each block, and the variables in the registers only available during the life as the

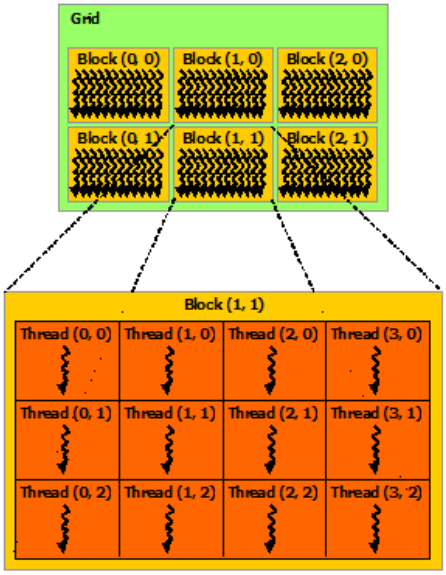


Figure 2.9: Grid of Thread Blocks. From Grid of Thread Blocks, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm`, [Online; accessed 24-July-2017]

thread.

Local memory is used for whatever does not fit into register, it is also private for each thread but has the same access latency as global memory, which means very slow. Variables in the local memory also has the same lifetime as the thread.

Shared memory enables threads belonging to the same block exchange the variables and has low access latency (about 100 times faster than global memory). Shared memory has a lifetime of the block. While using shared memory, one needs to pay attention to the bank conflict, to make sure that no thread can read the part of that shared memory while another thread is writing to it.

Global memory is open to all threads, which makes it has high latency. However it has large memory space for variables. It has the lifetime as the whole application. Global memory can be modified and accessed by host using C-functions like cudaMemcp, cudaMalloc or cudaFree.

Constant memory can also be read from all threads. There is only a limited amount of constant memory can be declared (64KB). And the variable inside the constant memory can not be modified by kernels, therefore, immutable. Also it shares the same memory banks as global memory, so it is still slow, however, it is faster than global memory since variable inside the constant memory is cached and immutable. The lifetime of constant memory is the same as global memory (application).

Texture memory is also read-only memory spaces that can be reached by all threads. It is stored in device memory and cached in texture cache. Comparing with global memory and constant memory, it is optimized for 2D spatial locality,

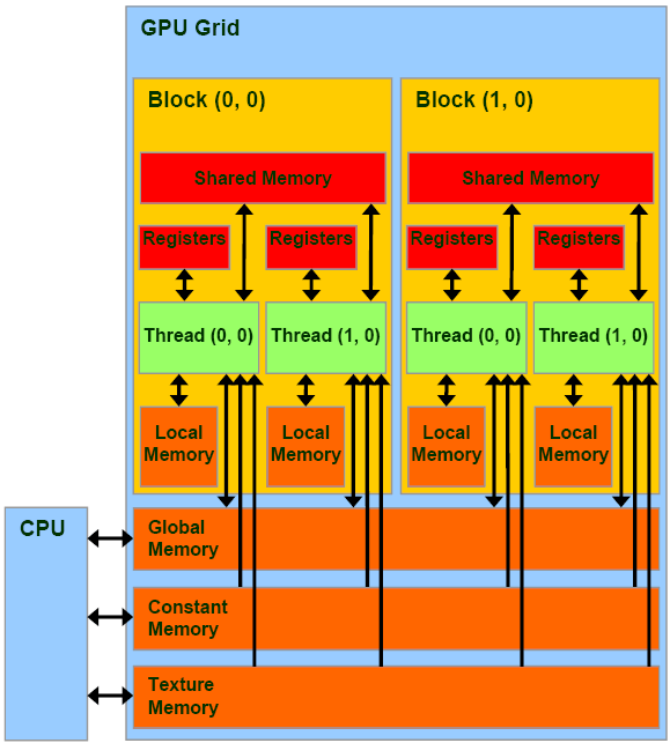


Figure 2.10: CUDA Memory Architecture. From CUDA Memory Model, Jeremiah van Oosten, `https://www.3dgep.com/cuda-memory-model/`, [Online; accessed 24-July-2017]

2.3.5 Synchronization in CUDA

Threads within the same block can access to shared memory and can be explicitly synchronized by calling __syncthreads(). __syncthreads() function acts as a barrier, when it is called, execution will not continue until all the threads have their job finished.

There is no direct function to synchronize between blocks. If one need the data inside global memory share with blocks, one can create own mutexes using the atomic functions provided by CUDA Toolkit.

2.3.6 Integrate with MATLAB

There are several ways to combine both the convenience of using MATLAB and the performance acceleration utilizing GPUs. For example, some popular used image processing functions like fft and filter are already GPU-enabled. There are also two possibilities for more advanced use if one wants to implement their own algorithm on GPU and integrate it to MATLAB. To integrate the GPU kernel into the existing MATLAB software, the code written in C for controlling the GPU has to be adapted. One is to create an executable kernel from CU or PTX (parallel thread execution) files, and run that kernel on a GPU from MATLAB, another one is to run MEX-files with CUDA code and have a single entry point.

In order to use kernel to do GPU accelerate, first need to compile a PTX File from a CU file in command line. With the CU file and PTX file. you can create a CUDAKernel object in MATLAB that can be used to evaluate the kernel. The CUDAKernel functionality was introduced in R2011b in Parallel Computing Toolbox.

Calling MEX-files to achieve higher performance is a more challenging task. A MEX file is a type of file that provides an interface between MATLAB or Octave and functions written in C, C++ or Fortran. When compiled, MEX files are dynamically loaded and allow external functions to be invoked from within MATLAB or Octave as if they were built-in functions. If it is pure CPP or C file, can use mex command in MATLAB to build the MEX file and call it like a MATLAB function. Since we need to build a MEX from CUDA files. We need to use mexcuda command instead of mex command and also modify the code. After we finish the CU coding and debugging, we need to add an entry point for it, known as mexFunction. So that the CUDA code in the MEX-file is conform to the CUDA runtime API. Function mxInitGPU should be called in the beginning of mexFunction to ensure that the MathWorks GPU API is initialized properly.

The mexFunction, like the entry point in most coding language, has a number of inputs and the pointers to inputs as arguments. It also receives a number of expected output and array of pointers to the expected output as arguments. Inside the mexfunction, the number of MEX file input and output arguments as well as the type of these arguments are verified. This gateway routine requires mxArray (which is a C language opaque type of MATLAB arrays) for both input and output parameters. This means that we need to create a mxArray pointer that will later be populated with the result data we get later. The kernel of computational routine has no difference from normal CU file. After defining the entry point for CUDA, it is possible run mexcuda command from MATLAB command line. Mexcuda was introduced in version R2015b. It is an extension of MEX function which can use NVIDIAs nvcc compiler to compile a MEX-file containing the CUDA code.

In order to compile CUDA code, it is necessary to first install the CUDA toolkit version which is consistent with the ToolkitVersion property of the GPUDevice object. By default, MATLAB stores all numeric variables as double-precision floating-point values. It is recommended to cast the numeric variables in the MATLAB side to single-precision using single command, or use double for variables in CU file.

To set the Visual Studio environment for debugging, one needs to specify the option for mexCUDA command as -G, this makes it possible to step through kernel code line by line in one of NVIDIA's debugging applications (NSight or cuda-gdb) by generating debug information for device code. Then open the source CU file in Microsoft Visual Studio and add breakpoints, go to Studio → TOOLS → Attach to Process in Visual Studio, select the running MATLAB process, transport as GPU Debugger, Qualifier as your Debugger set in NVDIA Nsight Monitor, finally from MATLAB, run the MEX-function. The program will run until it hits the breakpoint and pause.

Although using MEX file is more work than using CUDAKernel to integrate CUDA code with MATLAB, there are some reasons for choosing the MEX-file approach.

- MEX-files can interact with host-side libraries, such as the NVIDIA Performance Primitives (NPP) or CUFFT libraries, and can also contain calls from the host to

functions in the CUDA runtime library.

- MEX-files can analyze the size of the input and allocate memory of a different size, or launch grids of a different size, from C or C++ code. In comparison, MATLAB code that calls CUDAKernel objects must preallocate output memory and determine the grid size.

2.4 Link between MATLAB, Spark and native

In order to use MATLAB functions from a C/C++ program, it is possible to use MATLAB as a computation engine. To call C/C++ function from MATLAB, one can build MEX files and use the filename as function name. Such a function can be called from the MATLAB command line as if they were built-in functions.

The situation is similar for Python. MATLAB also provides an engine for Python. Using this engine, one can call any MATLAB function directly and return the results to Python. It is easier to call the Python module from MATLAB than call C functions from MATLAB, since all you need is to include the path of the module to the Python search path. Subsequently, the functions inside the imported module can be used inside MATLAB. There is a third way to link MATLAB with Python, which is connecting the MATLAB Engine for Python to a shared MATLAB session that is already running on the local machine. If there is no running session already active, a new MATLAB session will be started. If you started with converting the MATLAB session being used now to a shared session from a command line in MATLAB, and connect the Python to the session with the same ID of the MATLAB process, the variables can be modified, (new variables added and deleted) from both sides through the MATLAB workspace.

Use case analysis and alternative solutions

3

This chapter begins with an overview of image registration algorithms discussed in the literature. Second part will focus on alternative implementation platforms that have been used in research. This section investigates which platforms the use case can be applied on, and analyzes the previous work that has been done. The last section discusses the comparison of the possible solutions and chooses some of them to implement in the next chapter.

3.1 Image registration algorithms

Image registration has been one of the major areas of research in the medical imaging community for decades in addition to the another area of image segmentation. As an important role in image processing, the task of registration is to align two or more images that are taken differently in time, in angles, or from different modalities. In this area, a broad range of techniques has been developed that can aim for a particular type of data and requirements. Most of the registration processes include algorithms to examine the spatial difference caused by scaling, rotations or movements, and compensate the difference. In this section, some of the widely used image registration algorithms for medical images are discussed.

One of the widely accepted classification of image registration methods is single-modality and multi-modality. The differences between these two classes are whether the images to be registered are acquired by the same type of imaging equipment or different ones; multi-modality image registration is often used in medical imaging. Back to the year 1997, F. Maes et al. [24] reported on a new approach for multi-modality medical image registration by maximizing mutual information.

Image registration techniques can be classified into intensity-based or feature-based methods. Intensity-based registration techniques use intensity patterns as metrics. One example of such metrics is raw pixel values; thus these techniques can reach high accuracy. Feature-based registration techniques, on the other hand, use image features to find correspondence. These features include points, edges, contours, or surfaces. One of the biggest advantage of using features is that they can also be used for multi-modality registration. However, comparing with intensity-based methods, they are less robust, because they usually contain extra stages to define and extract features from images, which can introduce errors for the registration process that cannot be recovered in later stages.

A basic image registration process consists the following four steps: In the first step, features are selected and extracted for feature-based methods or images are transformed (scaling, rotating, shearing, etc.) for intensity-based methods. The second step is measurement. In this step, metrics are used to measure the similarity or differences between

images. Single-modality methods usually compare spatial location of images, and multi-modality methods usually use mutual information to determine statistical dependence or use correlation ratio for functional dependence of images. Some widely used similarity measures [25] [26] [27] include Mutual Information (MI), Cross Correlation (CC), Sum of Squared Difference (SSD) and absolute difference. The third step is optimization. In this step, a spatial transformation model is defined based on the expected relationship between the images, and a serial optimization algorithms are executed to find the optimum parameters that can best align the two images by maximizing the similarity metrics. The final step is image transformation. The sensed image is transformed by mapping transformation functions (rotations, translations, interpolation). For some recursive image registration processes, this step can also include re-sampling for the next iteration.

One of the key feature of the process is similarity metrics which give quantitative evaluation of the similarity between two image or two regions of images. There are no similarity metrics that perform the best for all situations, these metrics are usually appropriated for particular applications.

For intensity-based registration, as we mentioned before, multi-modality medical image registration usually use MI. This is because it is invariant to an arbitrary re-mapping of pixel intensity values, which are caused by different sensors. MI is based on information theory concepts, and make use of a joint probability distribution (trivially extended by Shannon entropy) built from image histogram. It measures how much knowing one image reduces uncertainty about the other, and maximized when two images are aligned. In [28], the author described image similarity measures within a formal mathematical metric framework, and outlined the most popular image (dis)similarity measures that inspect the intensity probability distribution of the images, and reformulation of these measures as metrics. The measures include joint entropy, MI, empirical normalized mutual information (NMI), and symmetric uncertainty coefficient (SUC).

For intensity-based registration of images in the same modality, image similarity measures that use CC, sum of squared intensity differences and ratio image uniformity are commonly used. The classical representative of these intensity-based methods is the normalized CC (NCC) and its modifications. The similarity between window pairs from the sensed and reference images is measured by NCC algorithm, and the maximum is achieved as the corresponding ones. These correlation-like registration methods can align two images with only translation, or even with more complicated deformations like slight rotation and scaling. The main two drawbacks for these methods are the flatness of the similarity measure maxima (occurs when the images are too similar) and high computational complexity [27]. The maxima can be sharpened by preprocessing or by using the edge or vector correlation. Despite these drawbacks, NCC is in use for some applications, due to the easy hardware implementation and potential for real-time applications.

Feature-based registration is recommended for images that contain rich detail and enough distinctive parts. By detecting the correspondence between pairs of points in images, a geometrical transformation can be determined. Mostly used feature types are region features, line feature and point features. To extract these features individually from both images, feature detection methods like Harris Corner Detector (multi-scale

Harris), SIFT (Scale Invariant Features Transform), Canny edge detector or Wavelet based feature detection are used. Once the features are detected, the next step will be matching the correspondence features. In this step, methods used in intensity-based registration can be also applied here like CC and MI methods.

It is also possible to combine both feature-based and intensity-based methods. [29] suggests a method for registration of multi-modal and temporal images of the retina using a combined feature-based and statistics-based method.

For reasons of space, algorithms used in transformation, optimization and resampling are not addressed in this thesis. These algorithms are discussed in [27] and [25]. In general, registration methods are iterative, and the optimizer checks for a stop condition. If there is no stop condition, the optimizer adjusts the transformation matrix to begin the next iteration.

Image registration is still one of the most important steps in medical image processing. It can be used in a series of stages of medical care, diagnosis, treatment, and monitoring disease progression. The applications involving registration include computer-aided diagnosis [30], computer-integrated neurosurgery modeling [31], and surgical navigation and assisted surgery [30].

3.2 Alternative implementation platforms

One of the famous open-source image processing programs is ImageJ. It is a pure Java application released in 1997 that provides extensibility via Java plugins and recordable macros. It can be run as an online applet, a downloadable application, or on any computer with a Java 5 or later virtual machine. The image processing operations can be executed in parallel on multi cores.

Based on ImageJ, a distribution called Fiji was released [32] in 2011. Fiji supports scripting, through appropriate interpreter plugin, scripts can access Fiji's extensive algorithm libraries that implement advanced image analysis techniques in Java. The scripting languages it supports include (Jython, Clojure, Javascript, JRuby and Beanshell). However, many image-analysis solutions available for Fiji can be only used under Fuji and do not have a comparable alternative in other platforms.

In [33], a generic framework for medical image registration is built. This framework is implemented in the C++ programming language. It is specifically built for the MeVisLab rapid prototyping software which is a multi-platform software that includes image processing and visualization modules.

[34] introduced a pure web-based, interactive, extensible, 2D and 3D medical image processing and visualization application. For web-user-interface, a combination of JavaScript with AJAX technology is used. For advanced features like 3D visualization, an advanced version of the client application with a Java Applet is developed. The serve runs the main engine of the software for image processing functionalities. It contains ITK library (a well-known C++ open source library designed to cover many medical image processing routines) and code generator (for generating most of the codes of the higher layers of the application). For developing the visualization algorithm, the VTK library is used, which is a visualization toolkit.

Vemula et al. [35] developed the Hadoop Image Processing Framework, which provides a Hadoop-based library to support large-scale image processing. This solution grouped small image files from image datasets into image bundle file to solve the problem that HDFS has to handle small file storage. The framework uses Java and algorithms are highly parallelized thanks to MapReduce.

In September of 2016, researchers in Southern Illinois University Edwardsville published an open source software package CVIPtools that is available in three variants: a) CVIPtools Graphical User Interface, b) CVIPtools C library and c) CVIPtools MATLAB toolbox [36]. The algorithms code layer is written in standard C. To extend the functionality of CVIPtools to MATLAB, these image processing functions have been ported to MATLAB by writing wrapper functions for CVIPtools library functions using MEX feature of MATLAB. It currently consists of 117 functions covering different areas of image analysis and computer vision such as image geometry, segmentation, edge detection, transform filters, spatial filters, morphological filters, arithmetic and logical operations, and image histogram operations. However, it is not dedicated to medical image processing, and image registration is not included in the toolbox.

Back to earlier years, Mittal et al. addressed in 2008 that FPGAs could be an efficient and promising platform for image processing due to their high computational density, low cost, and optimization [37]. Later in 2013, Chiuchisan [38] implemented the one of the first low-level FPGA-based image processing platforms. This platform is a FPGA-based real-time configurable system consisting of image processing low-level operators that focuses on medical images enhancement, like contrast filter, brightness filter, inverting filter and pseudo-color filter. It used Verilog HDL for reconfigurable architectures.

Beside the above implementation, another widely used platform for image processing is GPU. [39] published in 2013 presents an overview of the work done on GPU accelerated medical image processing, which includes the most commonly used algorithms in medical imaging (image registration, image segmentation and image denoising) and algorithms that are specific to individual modalities. An open source multi-platform library called GpuCV is presented in [40]. It is a framework for image processing and computer vision, which is an extension of Intel OpenCV library.

3.3 Comparison of alternative solutions

As a conclusion of previous work, alternative solutions of image processing platforms are compared in this section.

In the collections of open source toolkits, software platforms or libraries, the most popular languages used are Java and C++. Some of the image processing algorithms on these platforms are run in parallel on multi cores. Depending on the platform and distribution, it is not possible to compare the performances of image processing solutions across multiple platforms.

MATLAB has its own toolbox for image processing, with requires an added license. One way to use the open source libraries mentioned above in combination with MATLAB is to wrap the functions using the MEX feature of MATLAB. The MATLAB with library solution only consists of a few functions that use C++ libraries, and did not offer an interface for plug in or user-defined functions.

These libraries are also used in other platforms like web-based or Hadoop in the algorithm layer. The challenge of web-based solutions is the visualization of 3D images. And the connection of higher layer language used for web service with lower layer language for image processing algorithm. The advantage of Hadoop solution is the scalability. However the learning curve for using Hadoop is one of the drawback. Another problem for using Hadoop is that HDFS is not designed to handle small file storage, like images.

FPGA solution can be powerful and low cost, but is now only used for limited low-level image operations because the coding language limits the possibility of user-defined algorithms.

GPU is a popular and mature solution for medical image processing algorithms. There is also a library that is an extension of OpenCV for easy implementation. However, the performance of this solution highly depends on the hardware and algorithm itself. To extend the existing library with new GPU accelerated functions can be also complex.

Based on this comparison, several solutions are to be considered for testing and evaluation in this thesis, including the MATLAB Toolbox with GPU, MATLAB with Python, and MATLAB with Spark. This is due to the following reasons: MATLAB is a cross platform software that has been widely used in the industry and education It uses a scripting language that is similar to Python. This decreases the difficulty for the user to implement extra functions in Python to extend the library. There is already an integration of C++ with MATLAB, however there is no solution with GPUs yet. Considering the excellent performance of GPUs and the existing library for usage. We also choose it as one of the solutions. The cluster solution provides scalability and fault tolerance features. Although MATLAB includes cluster functions within an extra toolbox, there is no image processing library and the toolbox is at the high end of license cost. Thus we also consider it as a potential solution to figure out how it can be integrated into MATLAB.

Implementation and measurements

4

In the previous chapter, we listed out alternative solutions that have been used for image processing platforms, and chose several solutions from them to test in this thesis. These solutions are the MATLAB Toolbox with GPU, MATLAB with Python and MATLAB with Spark. These solutions will have differences in the way that they are integrated with MATLAB, as well as their performance. For the developers, it will be helpful to estimate the performance and bottlenecks before they decide which acceleration methods to implement for their applications. Based on this, this chapter is concerned with testing the selected solutions and evaluating them using various metrics.

Many situations should be considered when performing the evaluation. We have identified the following metrics as relevant for the evaluation process:

- **Initialization**—This metric evaluates the overhead induced by initialization of calling GPU and Spark from MATLAB. This includes the time for MATLAB to import the libraries needed for running extra modules in other languages, essential packages needed for python to use Spark and the configuration time of the cluster for PySpark.

- **Memory transfer**—This metric describes the overhead used when passing data by value to external functions. Data in MATLAB can be passed either as object handles or by value. In the first way, only the handle is copied, both the original and copied handles refer to the same physical location. In the second way, the actual value is copied to a new address. Because this copied value is independent from the original one, the modification on the copy will not influence the original data. When one uses the javaArray function or creates a Python object using a constructor in MATLAB to encase data, this data is passed by reference.

- **Data conversion**—When passing MATLAB data as arguments to external functions written in another programming language, MATLAB converts the data into types that best represent the data to that language. This can be done either automatically or manually. In a similar way, data transferred back to MATLAB needs to be converted back to a standard MATLAB data type for further computations. We call the time needed to carry out this conversion as the data conversion time.

- **Computational throughput**—For this metric, we compared the computational throughput of GPU, Spark and Python when working with MATLAB. We picked one computationally intensive kernel to represent the computing difference of these solutions.

Aside from the above-mentioned metrics, other factors such as network overhead, disk access time, and number of processing units also influence the computational time; these metrics are not discussed in this thesis.

4.1 Experimental setup

The initialization and computational throughput metrics are measured by time in seconds. Memory transfer and data conversion are measured by throughput in Megabytes per second.

While measuring the time from MATLAB, one can choose any of the following tools: profile, timeit, tic/toc, and cputime. These tools either return CPU time or wall-clock time. CPU time is the time actually spent by the CPU executing the method code; it sums the CPU time across all threads. Wall time is the real-world time elapsed between method entry and method exit. If there are other threads/processes concurrently running on the system, they can affect the results. Among the MATLAB functions, timeit and tic/toc return both wall-clock time. The Timeit function calls the specified function multiple times, and returns the median of the measurements. Tic/toc is usually used when measuring first-time cost or small portion of code as part of a complete function. Cputime returns cpu time while the profile function returns more information like call time. For time measurements in this thesis, we used tic/toc for measuring the first-time cost and either tic/toc or timeit for the rest measurements.

For the purpose of benchmarking those metrics, several use cases have been defined: initialization micro-kernels, data copy micro-kernels, as well as computationally intensive kernels for GPU and Spark. For kernels executed on MATLAB, CUDA and Spark (pseudo-cluster mode on local machine), we use local single node (laptop computer), the setup is presented in Table 4.1. The table also lists the specifications of the GPU used to evaluate kernels that use GPU.

The software and applications used for developing, debugging or executing on the local single node are listed in Table 4.2.

Some kernels were executed on multiple nodes to test the scalable capabilities of deployment on a Spark cluster environment. The hardware and software specifications for the cluster used to carry out these measurements are given in Table 4.3.

4.2 Initialization

In this section, we measure the initialization time needed to start executing functions using two different tool flows. The first tool flow shows the initialization time for executing functions from MATLAB and then accelerated on GPU. The second tool flow shows the initialization time for deploying MATLAB functions through PySpark on a Spark environment.

In order to measure the initialization time for MATLAB and GPU, we invoke a CUDA function, which does nothing but initializing MATLAB GPU library using mexFunction and returning success or failure which make sure that the GPU device is known to MATLAB. The duration for starting the GPU inside the mexFunction and from MATLAB is shown in Figure 4.1a.

It is shown in the figure that the first iteration takes the most amount of time, which represents the warm up time for the GPU. After that, the cost is almost zero for subsequent iterations. This is expected, since the first iteration consumes much time in initialization.

Table 4.1: Local single node setup used for MATLAB, CUDA and local Spark code

Component	*Specification*
Host specifications	
Operating system	Microsoft Windows 8.1
System type	x64-based PC
Processor	Intel(R) Core(TM) i7-4700MQ CPU
# of cores	4
# of threads	8
Processor base frequency	2.40 GHz 2400MHz
Installed physical memory (RAM)	8.00 GB
Cache	6 MB SmartCache
Max memory bandwidth	25.6 GB/s
GPU specifications	
CUDA device number	1
Device	Quadro K1100M
CUDA Driver Version / Runtime Version	7.5/7.5
CUDA Capability Major/Minor version number	3
Total amount of global memory	2048MB
Multiprocessor number	2
CUDA Cores/MP number	2 (Multiprocessors) *192 = 384
GPU Max Clock rate	706 MHz
Memory Clock Rate	1400 Mhz
Memory Bus Width	128 bit
Maximum Texture Dimension Size (x,y,z)	1D=(65536) 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Total amount of constant memory	65536
Total amount of shared memory per block	49152
Total amount of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024

For PySpark, we test the time of creating a SparkContext. Only one SparkContext can be active per JVM. When there is already a SparkContext running, one can access it using the getOrCreate function, or stop it and start a new SparkContext. SparkContext is usually stopped once a job is completed. For testing, we start or get a SparkContext in every iterations. And we measure time for both creating and getting the context from MATLAB and Python, as shown in Figure 4.1b.

Table 4.2: Software and applications setup for local single node

MATLAB deployment	
MATLAB version	64-Bit R2016a
GPU deployment	
CUDA driver	CUDART
CUDA driver version	7.5
CUDA runtime version	7.5
Build/debug tool	Visual Studio 2013
Spark deployment	
Spark version	Spark 2.0.1 built for Hadoop 2.7.3
Scala version	2.11.8
Java version	64-Bit Server VM, Java1.8.0_101
Java vendor	Oracle Corporation
Python	3.5.2
Hadoop version	Hadoop-2.7.3
Build/debug tool	JetBrain PyCharm 2016.2.3(64), Windows Command Line

Table 4.3: Hardware and software specifications for the cluster

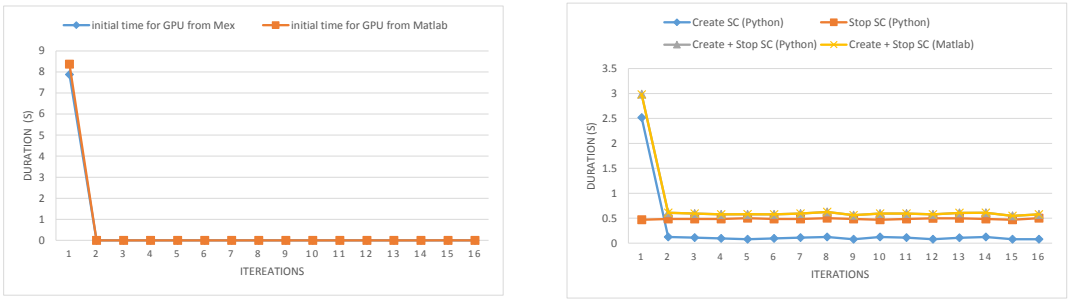
Component	*Specification*
Cluster configuration	4 + 1 nodes & 2 sockets per node
Node model	IBM 8246-L2T
CPU	IBM Power 7 8-cores, 4 threads/core, 3.6GHz, 64 MB L3 Cache
RAM	128 GB
Operating System	CentOS Linux 7
Spark deployment	Spark v2.1.1 with YARN, Hadoop 2.5.1

4.3 Memory copy

To measure the memory transfer, we copy an array and a matrix from MATLAB to GPU and from MATLAB to Python, and measure the transfer throughput (MB/s). We use the array and matrix as examples of frequently used data structures in MATLAB.

4.3.1 MATLAB to GPU

There are at least two ways to copy an array or a matrix from MATLAB to GPU. One way is to use *gpuArray* to copy the array from CPU to GPU in MATLAB and return an object, then pass this gpuArray object to the mexFunction. To copy array from GPU back to CPU, one can then use the *gather* function. The problem is that MATLAB does not have the functionality to clean the memory allocated for GPU, which means one needs to reset the GPU device and clear its memory periodically, which takes 7.1s on average. The other way to copy the array or matrix from MATLAB to the GPU



(a) Measurement of initialization time on GPU per iteration

(b) Measurement of initialization time for PySpark per iteration

Figure 4.1: Initialization time for GPU and PySpark

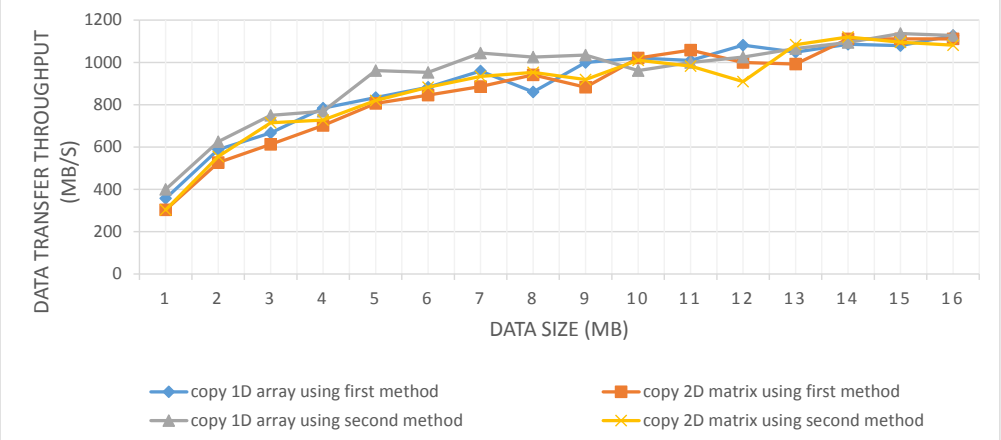
is by directly passing MATLAB 1D or 2D arrays on CPU to mexFunction. Inside this function, the array is copied to GPU using *cudaMalloc* and *cudaMemcpy* (for 1D arrays), or *cudaMallocPitch* and *cudaMemcpy2D* (for 2D arrays). This way of copying data is more difficult to use than the first one, but allows more control on the data, like binding texture on it, and more importantly, it allows the memory on GPU to be easily freed after being used.

The throughput also affected by the format of the data we copy. To show the difference, we use the same total amount of data to be copied as a one-dimensional array and two-dimensional matrices for testing. For example, we compare the throughput between a 1x4 array and a 2x2 matrix. It is easy for MATLAB to create a 2D or higher dimensional array and copy to GPU automatically using the first method. Compared to this MATLAB built-in function, the second method needs to adjust the function from *cudaMemcp* to *cudaMemcp2D* and has to deal with pitchBytes while allocating the memory. The throughput of the data is shown in Figure 4.2a.

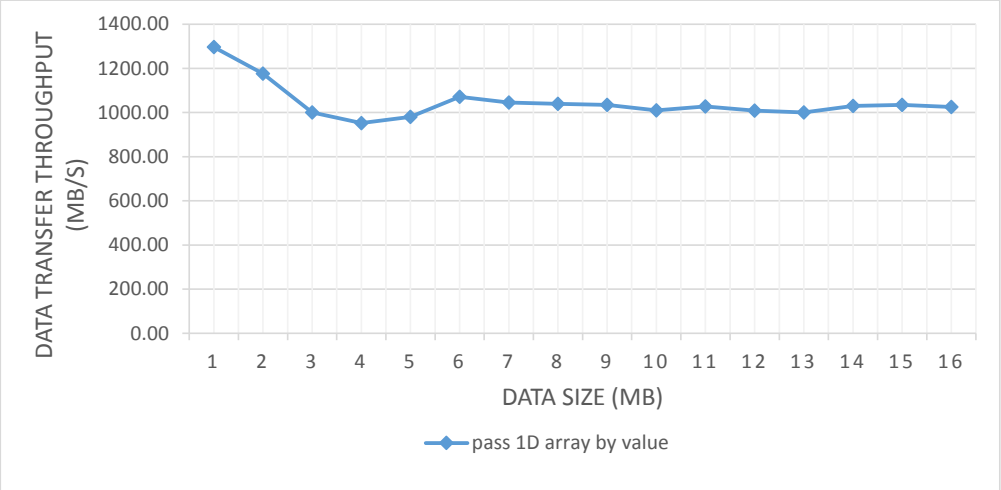
4.3.2 MATLAB to Python

When it comes to data transfer from MATLAB to Python, there are three approaches to perform this transfer. The first approach is to transfer data by handle objects, which are reference types of Python objects in MATLAB. The second approach is to transfer by copying data to Python, and then copy back to MATLAB. This approach works for MATLAB data types that can be mapped to Python data types automatically [41]. Since MATLAB only supports 1xN vector mapping, data transfer of 2D matrices is not supported and it is therefore not discussed for the data copy approach. The third approach is to use the MATLAB engine in Python to handle MATLAB data types. However, calling a Python module that includes the MATLAB engine library from MATLAB itself will cause an initialization error of the MATLAB engine. For this reason, we only consider the first two approaches in this thesis. In the first approach, data is transferred by reference. The transfer time is near to zero. In the second approach, we measure the time to transfer 1D array between MATLAB to Python by directly passing MATLAB arrays as a parameter of the Python function. The result returned from Python is in the form of a Python array handle object, which can be copied and converted to a MATLAB

array using *double* function in MATLAB. The throughput of these two steps together is shown in Figure 4.2b. The figure shows the transfer throughput of the copy for different 1D array sizes. The results indicate that the throughput is relatively constant at around 9 GBytes/second, with a slight increase for smaller array sizes.



(a) Measurement of memory copy throughput for 1D and 2D arrays



(b) Measurement of memory copy throughput for 1D arrays between MATLAB and Python

Figure 4.2: Memory copy kernel

4.4 Data conversion

As we mentioned in the previous section, one way to transfer data between MATLAB and Python is using handle objects. Building a Python handle object is actually a process that passes data to a Python constructor (can be user defined or built in data type like a list or tuple). Although data transfer time is almost zero because it only passes a reference, the data usually needs to be further converted to equivalent MATLAB data

types. In this section, we measure the time for data conversion that is needed before and after transferring 1D and 2D arrays between MATLAB and Python.

In MATLAB, 1D array is casted to *py.list* or *py.tuple* manually. This reference is then passed to a Python function which do nothing and return the same reference to MATLAB. MATLAB then has to use the *cell* function to put the list or tuple into a cell array. Then use *cellfun* to extract the values of the cells and group them into a numerical array again.

Interpreting 2D arrays in MATLAB needs more steps since MATLAB only supports converting 1xN vectors to Python data types. In order to make a 1xN vector that represents a 2D array, we can convert an MxN matrix into a 1xM cell array (using *num2cell* or *mat2cell*), where each cell contains a column of the matrix. After running the Python module (which does not perform any processing and returns the same value back), the result is in the form of a Python Tuple of Python Arrays. The first two steps are the same as those for the 1D array, which are the *cell* and *cellfunc*. After these two steps, the result becomes a cell array of MATLAB double arrays. The final step is using a *cell2mat* function to reform the nested MATLAB array to a matrix again.

Both 1D and 2D array data conversion throughput is shown in Figure 4.3.

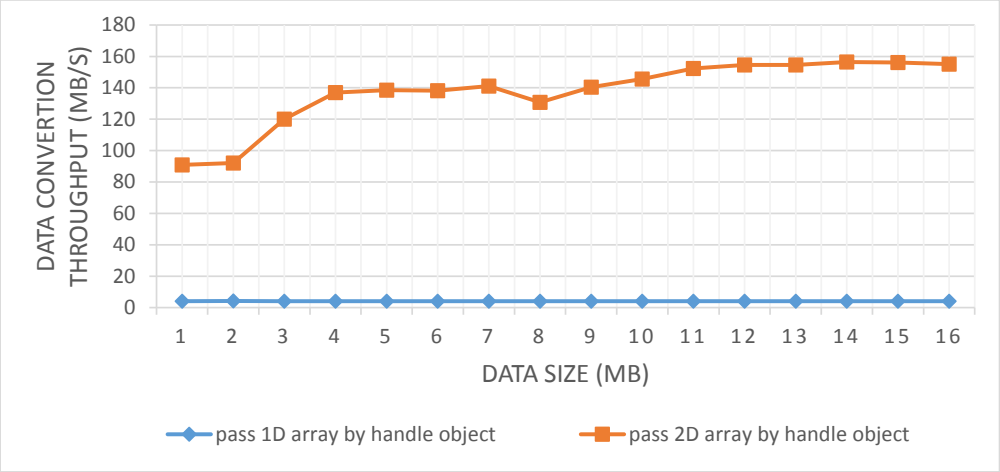


Figure 4.3: Measurement of data conversion throughput for 1D and 2D arrays between MATLAB and Python

4.5 Computational throughput

Computationally intensive kernels should have the following characteristics: little data transfer time, little communication and synchronize time, little data allocate time, and a lot of computation time. To be more specific, little data transfer time means: as little data as possible to transfer between GPU and CPU, and between local and HDFS or RDD. Little communication and synchronize time means: little inter-GPU communication between multiple GPUs, few barrier synchronization in a single block of GPU. Little shuffling and broadcasting in Spark application. Little shuffling means: limited usage of

reduceByKey/groupByKey/sortByKey, repartition, and so on.

Little data allocate time means allocate less space and few times on GPU, and few chances to cache data in disk in Spark.

The computationally intensive kernel we picked finally is to calculate the number of primes less than or equal to x, and this is usually represented as pi(x).

Prime numbers are special in cryptography. Some important algorithms such as RSA make use of the fact that prime factorization of large numbers takes a long time.

Early in the 3rd century BC, an ancient Greek mathematician called Eratosthenes created a simple algorithm called The Sieve of Eratosthenes for finding all prime numbers up to a specified integer. It is a simple and efficient way to find out small prime numbers. However, it is not efficient for larger number. The bit complexity of the algorithm is $O(n(\log n)(\log \log n))$ bit operations with a memory requirement of $O(n)$. This algorithm will need a lot of barrier synchronization. In order to distribute the algorithm and reduce the communication, we design a kernel to test the primality of a single number individually concurrently, and collect the number of prime numbers from all threads/nodes in the last step. The primality test we used was Fermat primality test.

Fermat primality test is based on Fermat's (Little) Theorem: If p is a prime and if a is any integer, then

$$a^p \equiv a \ (mod \ p) \tag{4.1}$$

In particular, if p does not divide a, then

$$a^{p-1} \equiv 1 \ (mod \ p) \tag{4.2}$$

Unlike the Sieve of Eratosthenes, the Fermat primality test is only a probabilistic test to determine whether a number is a probable prime. If a number is prime, it will definitely pass the Fermat primality test, but there are still some composite numbers (called pseudoprimes) that can satisfy the primality condition. The chances of a pseudoprimes is quite low, (21,853 pseudoprimes base two comparing to 1,091,987,405 primes under 25,000,000,000), and the larger n is, the more likely (on average) that a probable prime (PRP) test is correct [42].

To implement this, for each n bigger than 1, we choose a that is also bigger than 1 and calculate a^p *modulo* p. If the result is not a, then p is composite. If the result is a, then p might be prime, and p is called a weak probable prime base a (or just an a-PRP). However, during the implementation, it will cause overflow to store the result of a^{p-1}. The solution for this is to use the multiplication property of modular arithmetic:

$$(a \ * \ b) \ mod \ p = ((a \ mod \ p) * (b \ mod \ p)) \ mod \ p \tag{4.3}$$

We made $a^p \ mod \ p$ into p iterations, where each iteration i does the calculation:

$$(a^i) \ mod \ p = ((a^{i-1} \ mod \ p) * (a \ mod \ p)) \ mod \ p \tag{4.4}$$

the result can be then stored without overflow because it will be always smaller than p.

However there is another problem which is related to workload balance. Large numbers need more iterations to test 4.1 than small numbers. Also, in order to increase the correctness of primilaty, we use a list of prime numbers as bases (a). When a number

is prime or pseudoprimes, it needs to perform the modulo iterations using each base number. On the other hand, those composite numbers that can be detected by base two can skip the rest base numbers. When we try to distribute a list of testing number on a cluster or on GPU, if we use the default partitioner like hash partitioner, the thread or node that gets the chunk with the largest numbers needs the longest time for processing. If we assume each computation of 4.4 costs t time, and there are n base numbers, the chance of a random integer x being prime is about $1/\log(x)$, and we assume that if it is a composite number, it only needs one base number to confirm it is not prime. The estimated time t for primilaty test for a random integer x will be:

$$t = (1 - 1/\log(x)) * x * t + 1/\log(x) * x * t * n = x * t * (1 + (n - 1)/\log(x)) \quad (4.5)$$

Since n is always the same, when the number grows large, the 4.5 is approaching $x * t$. So for the large number in the experiment, we made the sum of the number to test is almost equal for each thread or node. However for small numbers, other partitioning methods will be more efficient.

To profile CUDA applications, we use NVIDIA Visual Profiler that comes with NVIDIA Nsight. It is a cross-platform performance profiling tool that can be integrated into Microsoft Visual Studio or Eclipse. We used the timeline to trace the CUDA activity occurring on both CPU and GPU to make sure that the most time consuming part is the computational part on GPU. Spark also has web UIs to monitor Spark applications visually. We profile the spark kernel locally before we launch it on the cluster.

The result of execution time of this computationally intensive kernel with different approaches are shown in Figure 4.4. The x-axis is x of $pi(x)$, and y-axis is time (s) in log scale. We measure the execution time for the same kernel using only MATLAB, MATLAB with C acceleration, MATLAB with CUDA acceleration, MATLAB calling same functional python module locally (which in the graph is python on local machine), MATLAB calling the same function using PySpark locally using all 8 threads (PySpark local[*] on local machine in the graph), MATLAB calling the same function using PySpark on power7 using all 64 threads (PySpark local[*] Power7 in the graph), as well as MATLAB calling the same function using PySpark on power7 cluster using 4 nodes and all 4*64 = 256 threads (PySpark Yarn Power7 cluster in the graph). For MATLAB and python on the local machine, these two approaches, we only tested till $pi(50000)$, since the time cost became larger and local machine stop working.

From the graph we can see that a pure python implementation is slower than MATLAB, and the difference becomes larger as the number grows. Then comes the PySpark local[*] on the local machine, which uses 8 threads. This solution is faster than MATLAB and pure python approaches, while it is slower than the other approaches. C and CUDA approaches are fast especially when the number is small; C is slower than CUDA when the number increases. PySpark on a single Power7 node and on Power7 clusters is slow in the beginning. PySpark on a single Power7 node is slightly faster than C locally when the number is larger than 55000. PySpark on the Power7 cluster is also slow in the beginning, but the increase in its time consumption is the slowest among all other systems. For larger x numbers and data sets, this implementation represents the fastest solution among all approaches. We further tested this PySpark solution executed on the single Power7 node and on the Power7 cluster for even larger numbers of x. Figure 4.5 is

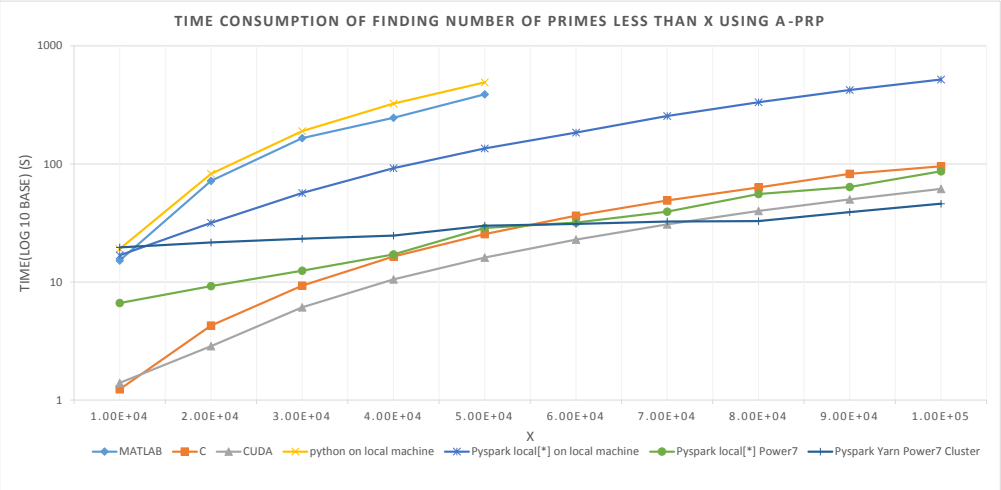


Figure 4.4: Execution time of computationally intensive kernel with different approaches

a continuation of Figure 4.4 for larger x-axis values. shows that the single node spent 4x more time than the cluster solution. This indicates that our PySpark approach enables easy scalability of MATLAB code on a large computation cluster.

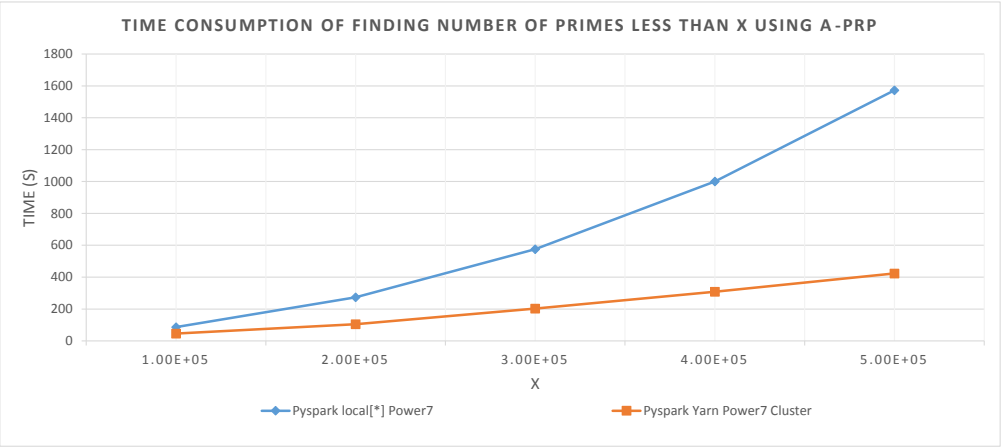


Figure 4.5: Execution time of computationally intensive kernels for larger x values

5 Discussion

In this chapter, we will focus on studying and discussing the performance we got from the previous chapter. Each section aims at one aspect that corresponds to the metrics we measured in previous chapter: Initialization, Memory transfer, Data conversion, Computational throughput. We will use the experience and information we got to explain the results we had. By studying and discussing the result, we can identify the bottlenecks and have more accurate estimation about the performance challenges.

5.1 Initialization

Figure 4.1a shows the measurement of initialization time on GPU. It recorded the initialization time for the first 16 iterations. There are two curves in the this figure. The blue curve shows the initialization time measured from Mex (written in C), and the orange one describes the same time but measured from MATLAB. The initialization time starts with a long duration of about 8.5 seconds at the first iteration, and drops immediately to nearly zero and stays constant starting from the second iterations.

Figure 4.1b illustrates the initialization time for PySpark on the local machine. There are four curves in this figure. The blue curve shows the time duration when a SparkContext is created, measured from Python side. The red curve deals with the time it takes to stop and delete a SparkContext, measured from the Python side. The grey curve is the sum of the previous two stages, from the Python side. The yellow curve is the time duration of creating + stopping the SparkContext measured from the MATLAB side. The yellow curve has a similar trend as the initialization time on GPU. The longest time taken to create and stop a SparkContext is in the first iteration; it takes in total 3 seconds. After which the curve is stable around 0.5 for the following iterations. The blue curve and red curve give more insights about creating and stopping the context. They show that creating a SparkContext contributes to the peak at the first iteration, then it drops to almost 0. This is because only the first time a new Java Platform SE binary process is created and a SparkContext is actually created, for the following iterations, it uses the same context. Although the user-defined instance that creates or gets the SparkContext is stopped by Python automatically, the JVM does not stop running, the new SparkContext object will still bind to this running JVM. However, if one tried to stop the JVM manually by clearing the SparkContext object from MATLAB, the Java SE binary process will be stopped. From MATLAB, it is not possible to create a new SparkContext again, since this returns an error when trying to connect to the Java server when starting java gateway. Also, Spark (including PySpark) is not designed to handle multiple contexts in a single application, which means we cannot have multiple SparkContexts in the same JVM process.

5.2 Memory transfer

Figure 4.2a illustrates the memory copy throughput for 1D and 2D arrays between MATLAB and GPU. The figure describes how the throughput is related with the size of the array. As we mentioned in Chapter 4, there are two different approaches to achieve this. One is to use the built-in function gpuArray provided by MATLAB to copy the array from CPU and GPU, which is represented as the first method. The other one is using the traditional way: cudaMalloc and cudaMemcpy to copy array from CPU to GPU manually. Figure 4.2a shows that all of the four curves have similar start values between 250 and 300, and then grow logarithmically. When the data size is 14 MB, the throughput reaches 1100 MB/s.

Both methods show no difference in the graphs, which means MATLAB implements the gpuArray copy using the same way as the second method. Throughput curves of 2D array and 1D array are also similar. The logarithmic trend is because of the following: for small transfer size, there is a constant overhead, then it increases linearly.

This figure indicates that there is no efficiency difference between the first method and the second method. However, for easy functional program, it is recommended to use the MATLAB built-in GPU array. It is the same in terms of time consumption, but much easier for implementation. The performance of memory transfers between the CPU and GPU depends on many factors, including the size of the transfer and type of system motherboard used.

For memory copy between MATLAB and Python, the throughput starts from a high point around 1300 MB/s, it drops down to 1000 MB/s when the data size is 3 MB, after that, it stays stable.

It is because for the small size of data, the profile is not accurate. After that the curve becomes stable.

5.3 data conversion

Figure 4.3 shows the measurement of data conversion throughput for 1D and 2D arrays between MATLAB and Python. The figure describes the way the throughput changes as a function of data size. The throughput of a 2D array data convention starts with a low point around 90 MB/s, which stays the same value till 2 MB. From 2 MB till 4 MB, the throughput increases till 140 MB/s. The throughput stabilizes at 140 MB/s until the data size reaches 8 MB. The curve drops to 130 MB/s when the data size is 8 MB, and increases gradually again till 160 MB/s at 12 MB. Then the throughput stays the same again till 16 MB.

From the previous chapter, we already know that the common parts of data conversion for both 1D and 2D are *cell* and *cellfunc*. For the 1D array, on average, the *cell* function takes 58% of total time to split all numbers into separate cells, and *cellfun* takes 36.6% which applies double to each cell.

For 2D array, the transition of matrix and cell arrays involves *num2cell* and *cell2mat*. For a matrix A with dimension $m \times n$, the first step *num2cell(A,2)'* will split the contents of A into separate cells and result a 1-by-m cell array, where each cell contains a 1-by-n row of A. After copy back from python, the result is a Python tuple with Python array

inside. Applying *cell* function will turn it into 1-by-m cell array of python array. And because the python array is inside each cell, so we need to use *cellfun* to applies the function (*double*) to the contents of each cell of cell array C. It turns out to be $1 \times m$ cell array and in each cell is 1xn MATLAB double array. The last step is using *cell2mat* to recreate the matrix again.

From profiler in MATLAB, we got the results: The *num2cell* function has a start up time of around 0.22s, after that it increases slowly as m is increase, because it will generate more cells.

cell array that converts Python sequence type to a MATLAB cell array also extends with m, even slower than *num2cell*.

cellfun applies the function to the contents of each cell of cell array, one cell at a time, and without order. The function itself which convert a Python Array to MATLAB array of double (which are same format) does not take any time (constant, less than 0.01). In this case, the cell number affects the time consumption of *cellfun* greatly (about four times as much as *cell* function and three times as much as *num2cell* function) thanks to the iterations, and the *cellfun* time is proportionate to cell number.

cell2mat converts a cell array into an ordinary array. Similar to *num2cell*, also proportional to cell number, and less time consuming comparing to all of the above functions.

So, according to the analysis of all the steps that needed during data conversion. We can say the fewer cells, which is m in a matrix A with dimension $m \times n$,the less time it needs for data conversion involving in the before and after calling Python kernel. Actually, we also tried, for same size of data, the time costs for data conversion of an array that all number in one cell is 280 times faster than the one with each number a cell.

The stable or decrease in the throughput of 2D arrays is a result of increasing in cell number (for example at 8 MB, is because 7MB is a 896x1024 matrix, 8MB is a 1024x1024 matrix) and keep still in the second dimension which is the size of each cell. When the cell number is constant, the throughput increases when cell number increases (2MB to 4MB and 8MB to 14MB).

This figure indicates that we should try to implement our data structures in a way that avoids many cells (like high dimension), so as to avoid applying the same function on each cell one by one to ensure a higher throughput of data conversion.

5.4 Computational throughput

Figure 4.4 illustrates the computational consumption using different approaches: MATLAB, C, Python, Pyspark locally, Pyspark on one Power7 node and Pyspark on a cluster. The computationally intensive algorithm we used is finding the number of primes less than x using A-PRP. The figure describes how time consumption (in log 10 base) is related with the number x.

The graph shows that, for this computationally intensive kernel, a pure Python implementation has the slowest runtime, then is pure MATLAB kernel. Both of these two implementations start with a high point around 16 seconds and end up with 400 seconds when x reaches 50000. The third curve counting from the top is Pyspark on local laptop

using all the threads. Which starts with the same point with 16 seconds, but increase slower than the previous two solutions. For $x = 100000$, the time needed is 519. The third curve is Pyspark on Power7 cluster, as it takes the longest time to finish computation when x is 10000; it uses 20 seconds. However, it increases also the slowest, AS it takes 46 seconds for the largest number. The next curve is local PySpark on one Power7 node. It has similar shape as the one running on local laptop, but faster. It starts with 6.6 seconds for the smallest x and 86 for the largest x. The rest two curves are C and CUDA running on local laptop. They are two fastest solutions for x smaller than 40000, in between, CUDA is faster. Both of them begin with 1.2 seconds, C curve ends up with 95 seconds and CUDA one ends up with 61 seconds at the end.

Python and MATLAB are not fast languages. Both of them are scripting language, so that they do not require an explicit compilation step. Also they use dynamic typing. So it is possible to bind a name to objects of different types during the execution. Both of them are just-in-time compilation. In the case of memory and optimization, MATLAB and Python both offer automatic memory management.

If want use Parallel Computing on a multicore processor in MATLAB, one has to establish a parallel pool of several workers with a Parallel Computing Toolbox license.

Similar to Python, the official Python distribution includes multiprocessing package which is similar to threading module. However, creating multiple processes and exchanging or synchronization between processes using this package is cumbersome. Another solution to parallelization in Python is use Pool package which provides a pool similar to what MATLAB does.

However MATLAB 7.4 (R2007a) from most linear algebra and numerical functions are automatically execute on multiple threads in a single MATLAB session. While in Python, one also needs to explicitly use a package for multithreading. It aims at more advanced programmers.

C is a lower level language, since it is compiled to relatively optimized native code before execution. Compared with MATLAB that uses just-in-time (JIT) compiler, it is faster. It also gives full control of memory allocation. Although MATLAB can automatically execute some linear algebra multi-threads, in order to processed a specific function in parallel, one needs to control it in C-Mex functions directly. Otherwise, MATLAB code is parallelized using multi processors. However, with C, one can multithread the code by using pthreads or OpenMP.

GPU provides highly parallel, multithreaded solutions for various programs. It is most suitable for problems that have large data sets and can use a data-parallel model, and also those problems that have a high arithmetic intensity (more arithmetic operations compared to memory transfer operations).

The limitation of GPU is copying data between host computer and GPU device. Another limitation is the memory access latency because of the limited cache memory and registers, slower clock speed than CPU. But these latencies can be hidden by high calculation throughput because of high data parallelization. GPU or CPU, which one is faster has higher computation throughput largely really depends on the algorithms.

Compared with acceleration using GPU, Spark needs more time to start up, more robust as it can recover from faults automatically. This is the reason why all the curves using PySpark in Figure 4.4 are slow at the beginning. Since PySpark uses as many

Table 5.1: Comparison of the different approaches at the system level

Approaches	**Bottlenecks**			
	Initialization	**Memory transfer**	**Data conversion**	**Computational throughput**
MATLAB				x
C				x (large amount of data)
CUDA	x	x		
Python			x (high dimension)	x
Pyspark			x (high dimension)	x (small amount of data)

worker threads as logical cores on the machines, it is faster than the python solution using single thread.

Both GPU and Spark are working on solving problems in parallel. GPU cores are more designed for image rendering, and it has more simple ALUs comparing to CPU. Spark depends on the type of nodes in the cluster, which can be heterogeneous. Using Spark, one can estimate the time after scale up easily. Storage can be scaled up too. Spark needs more energy, and costs more.

Scalable, high-throughput, fault-tolerant stream processing of live data streams on GPU is under developed and still in a theoretic stage, but it can be done in Spark easily. Spark can integrate with other applications. GPU processors cannot.

Figure 4.5 illustrates the scale up of this computationally intensive kernel on spark cluster when the x continuously increases from 100000 to 500000. According the algorithm itself, the time consumption to calculate $pi(x)$ is proportional to x^2, so the curves in the graph are quadratic function curves. We can see that the same application lasts 1/4 of the time finally when the number of workers increases 4 times.

5.5 System level analysis

Table 5.1 shows a comparison of the different approaches at the system level. MATLAB is not optimized for fast performance. It usually has computational bottlenecks when dealing with complex and large problems.

MATLAB is partly written in C. Therefore, integrating MATLAB with C does not need time for data copying since it only uses pointers. It also does not require time for data initialization. However, as the complexity of the problem increases, writing the program becomes more difficult. The performance of the program is also limited by the number of cores.

Using a GPU is more preferable when solving computationally intensive problems like deep learning and pattern recognition. It has great performance when solving image related problems. However, it is more difficult from the programmer to design the program in such a way that it makes full use of the multiple cores in the GPU. Compared with a CPU, it does not have powerful ALUs, no virtual addressing, and no interrupts. These differences also limit GPU usage, since it cannot be used for general purposes. It can also not perform good to solve real time streaming of input. While it may be possible to use both CPU and GPU to solve different parts of one task, utilizing each of them on the parts where they are most suitable. One should also consider the time

needed for initialization and memory transfer between CPU and GPU.

Python is also not a language optimized for speed. In addition, for a high dimension array, it introduce data conversion bottlenecks when calling Python or PySpark from MATLAB. PySpark, compared to other languages, is easy to use for code parallelization. However since it is designed to work in a cluster, using it for a simple task with a small amount of data will still have a large overhead. Still, if one avoids some time consuming functions (like shuffle) between all RDDs, PySpark can have the best performance.

6 Image analysis use case

In this chapter, a real world image analysis use case is implemented in MATLAB and accelerated through different platforms. The use case we picked is medical image registration. This chapter begins by the overview of this user case. Then, implementations of acceleration using GPU and Spark are represented. Later in this chapter we will include the results and discussion based on the acquired result. Some conclusions are drawn at the end of the chapter.

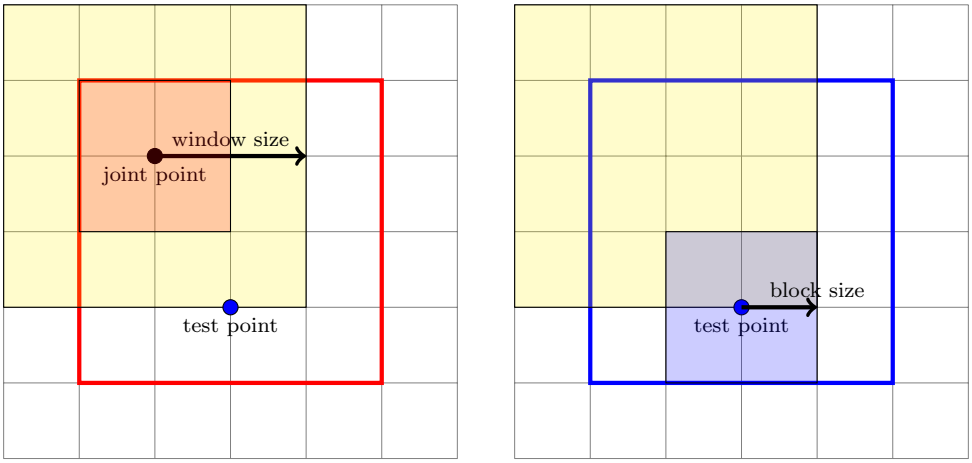
6.1 Image registration

From the image registration algorithms presented in Chapter 3, a single-modality intensity-based method using Normalized Cross Correlation (NCC) was picked to be implemented as our use case. It is a simple, but computationally intensive registration algorithm, which realigns two images after displacement.

The data used in this use case was obtained from MATLAB file exchange [43]. It is a folder which contains MR images of the brain. There are 20 different DICOM files in this folder. One of the MR images is read into the MATLAB workspace as a fixed image (so-called source image) for registration. For the moving images, we use the same image but with displacement. In real life, there is a continuous and an unavoidable movement caused by patients, devices or doctors. We use a range of movement to represent the possible field of movement of the patient. This range is related to the accuracy of devices, operations from doctors, and movement of internal organs. In this chapter, window size or R_{window} represents the maximum of displacement in all directions (two directions for 2D). To simulate the movement, random numbers are generated within $[-R_{window}, R_{window}]$ for each direction as displacement. By shifting fixed images according to the randomly generated numbers in both directions, we can generate countless moving images randomly. The functionality of our use case application is to read in moving images, compare them with the fixed image by applying the image registration algorithm, and finally apply the correction.

Image registration processing starts with segmentation that creates a grid to divide the whole fixed image evenly to small pieces. Based on the size of the image and density of the grid, coordinates of joint points created by each two crossing grid lines are returned. After checking each joint point and deleting those empty points (which do not have color), we can get a reduced list of joint points that may have information.

For each joint point in the reduced list from a fixed image, a window is generated surrounding a joint point (yellow area in both images) which indicates the possible range of displacement of this joint point. In order to find the matching point from moving image within yellow area, for each coordinate inside window of the moving image (test point), we applied NCC to compare regions of images that surrounding joint point and test point



(a) Fixed image (b) Moving image

Figure 6.1: Illustration of block selection in image registration

(indicate as red area in the fixed image and blue area in the moving image); the region size is defined by block size. The NCC algorithm returns the similarity between the joint point and test point. Within all the test points inside the window range, the one that has the highest NCC result is the one matched to the joint point the most. The coordinate difference between joint point and the matching test points is the displacement between original image and moving image at this joint point. The same procedure was carried out on all joint points. Because image registration can be followed by other processing like outline rejection or deformation, knowing only the average displacement from all nodes is not enough. For this reason, for each joint point, we store both the coordinates of the matching test point, and NCC result. This image registration algorithm between two images is computationally intensive, and the time consumption is proportional to the product of number of test points, window size squared, and block size squared.

The next step is to handle a bunch of pictures using the same algorithm or build an image registration stream processing pipeline. In this thesis we use the first solution. While handling a bunch of pictures, there are two different scale of parallelization. The first one is parallelization in image level, which means each image can be processed separately. The other scale is pixel level within each image, since every joint point is independent. We can use either one or combination of both of these two solutions, which depends on the platform.

In the following sections, the parts done in different platforms are described.

6.2 MATLAB implementation

MATLAB implementation can be divided further into two parts. The first part is executed by all the approaches. This part includes image pre-processing part and post-processing part of whole process. Before any image analysis can be applied on the DICOM (a standard for storing and transmitting medical images enabling the integration

of medical imaging devices) images, these images should be converted into MATLAB matrix, or format like PNG or JPEG that can be manipulated by other language such as C or Python. A DICOM data object usually consists of a number of attributes, and these attributes may further contain multiple frames for storing multi dimensional data. DICOM images also use three different data element encoding schemes. These all make transforming a DICOM object a complex job. There are some application available on the Internet to do this job, however they are bulky, installation needed and hard to be integrated with MATLAB. We opted for DICOMread function from MATLAB Image Processing Toolbox because it can be directly used and hide the implementation part from user. DICOMread function returns an M-by-N array for a single-frame gray scale image, or an M-by-N-by-3 array for a single-frame true-color image. Multi-frame images are always transformed to 4-D arrays. The testing data downloaded are 2D MRI images of brains, which are stored in a MATLAB workspace as 256*256 int16 arrays by default after transformation. The post processing part that executed in MATLAB can be differ from image registration strategies. This part can be an outlier rejection using restriction, image translation or rotation, or image display. In order to simplify the experiment, we get the overall displacement by averaging all the difference of inlier matching points of two images. Shift the whole moving image, and display the differences between the shifted moving image and original fixed image.

The other part to be done in MATLAB is the implementation of image registration algorithms on a bunch of images using NCC. Once the algorithm on a single image is complete and tested, we parallelize it on multiple images using multi cores. In order to exploit multi cores from MATLAB, a license for Parallel Computing Toolbox is necessary. This toolbox allows users to create parallel pool on local machine and process images on workers in pool. In MATLAB preferences settings, one can set the preferred number of workers. For the local laptop in this experiment, the worker number is a maximum of 4. The easiest way to modify the registration function from processing single image to a bunch of images is replacing **for-loop** by **parfor-loop**. Corresponding to two parallelization scales, the iterations can be applied on images or on all joint points. Due to the fact that MATLAB does not allow nested parfor loops, it is not possible to combine both types of parallelization. During our testing, we found out both types of parallelization have similar performance. In this thesis, we iterated through images and assign them to workers in the pool.

6.3 GPU implementation

An alternative solution with less time-consuming, though more complicated, is integrating with GPU. It is well known that GPU is superior in image processing. Taking the measurement result from Chapter 4 into consideration, we expected a much better performance than the MATLAB solution.

Once pre-processing was done in MATLAB, joint points coordinates, fixed image and moving image (in 2D arrays) were passed from MATLAB to CUDA using pointers. These parameters were further copied from CPU to GPU, and images were stored in texture memory. We implemented a kernel that computes correlation coefficient of two regions of images in CUDA language. This kernel is executed on GPU with exe-

cution configuration $<<< numBlocks, threadsPerBlock >>>$. Here the *numberBlocks* equals to the number of joint points, and the *threadsPerBlock* is a 2D array of size $(2 * R_{window} + 1, 2 * R_{window} + 1)$. This means, each block takes charge of the similarity comparison of all NCC computations relate to one joint point from the fixed image. Within the block, each thread represents one test point from the moving image within window range. The kernel invocation by thread aims to get the correlation coefficient of the block surrounded by the joint point and the block surrounded by the test point (red and blue area in 6.1). After all threads within one block finishing kernel computation, the correlation coefficients of all points inside the window area (yellow area) are known. As soon as these steps have been carried out, the largest coefficient within the window area as well as the coordinates of corresponding points needed to be figured out. One solution is to maintain a shared largest NCC value and shared struct for point coordinates inside block. Each thread compares the value and replaces the value and struct with its own value if it has a larger NCC value. Here race conditions can occur, which means one thread attempts to compare the currently largest NCC value while another thread is writing. An alternative, though with high overheads is atomic operation. The atomic operation is guaranteed to be performed without interference from other threads. The most suitable built-in one here is atomicMax. However, in our case, comparing NCC value and replacing coordinates needed to be carried out without interference. Two consequent atomic operations together cannot be treated as one non-splittable atomic operation. Therefore, a lock should be used. CUDA library does not provide a ready-to-use mechanism or class for lock. Because it will lower down the computational throughput greatly, which is against the design of GPU. However, user can still implement it by defining a global shared value as a lock, and atomicCAS to lock or atomicExch to unlock. This will sacrifice a lot of performance, because the lock need to be a global parameter, which brings writing and reading overhead. Also, when one thread takes the lock, other threads are keep waiting and do nothing.

A better way to do this is to use parallel reduction. After all threads finish computation. They store the results along with their global coordinates as a structure object inside pre-allocated block shared memory. Within each thread block, a sequential addressing parallel reduction kernel like below (6.1) is executed.

```
for (unsigned int s = maxId1D / 2; s >= 1; s = s / 2)
{
  if ((id1D < s) && ((id1D + s)<maxId1D))
  {
    reduce_point_max(&corr_of_block[id1D], &corr_of_block[id1D + s]);
  }
  __syncthreads();
}
```

Listing 6.1: Parallel reduction

The results got from reduction kernel in each block represent the information of the point that matches with joint point the most. The NCC value and coordinators are further gathered together along with results of other joint points from other blocks to group a final result. This result is transferred back to CPU, and the pointer of which will be passed to MATLAB.

6.4 Python and Spark

As mentioned before, MATLAB does not support Scala and R language. Between Java and Python, we opted for Python and therefore PySpark due to similarity between Python and MATLAB, which can shorten the time for development.

6.4.1 Python implementation

First we implemented the same image registration function for single moving image to fixed one in Python. Coding in Python is almost the same as in MATLAB. The function accepts images, and point list as Python lists from MATLAB, and returns a $P \times 3$ (P is number of list points) Python list, which can be further converted to matrix in MATLAB.

There are at least two ways to scale up one image to batch of image. One is sequentially iterative through all images delivered from MATLAB, the other one is using multiprocessing package in Python to enable multiple processors to work on different images at the same time, similar to the Pool feature in MATLAB.

However, MATLAB returns license manager error -1 and license.dat file error, if a Python module that uses multiprocessing package is called from MATLAB. The license.dat is created during the installation of the license manager on the license server, and is used when installing MATLAB on network clients. The error happens because using the 'multiprocessing' Python module from MATLAB is not a supported workflow.

So although we measure both solutions, with and without multiprocessing module, only the one without Pool and can be actually called directly from MATLAB now. The reason we included both is because this workflow could be fixed in the future or there is a way to bypass the error.

Multiprocessing package supports spawning processes, which allows the Python program to work in parallel. It offers both local and remote concurrency, and because it is process level parallel, it sidesteps the inefficiency caused by Global Interpreter Lock while using multi-threads. It provides two classes of multiprocessing: Pool and Process class. Pool class allocates only executing processes in memory while Process class allocates all the tasks in memory. When there are big amount of images, it is recommended to use pool.

Comparing with MATLAB, Python multiprocessing package provides handy and useful methods to build complex functions. Like *join* method, which blocks the calling thread until the process whose *join* method is called terminates or until the optional timeout occurs. *Lock* method that can be used for atomic operation. It can launch multiple evaluations asynchronously on multiple processors, which is better suited for parallelization. Those methods with *async* in their function name will return an AsyncResult object immediately. By calling *get* method from the result object, the program can retrieve the results whenever is needed. And when the results are ready, it will invoke the callback function. The disadvantage is that the results those *async* functions return are not ordered. So we will not use these functions inside our use case.

6.4.2 PySpark implementation

Depending on the setting, Spark can run with multi-processing and multi-threads. To use Spark, the official way is to use the Distributed Computing Server Toolbox and Compiler Toolbox from MATLAB to access data from HDFS or run algorithms on Spark. This feature can be only run on top of Linux system, and was only included recently.

Starting from Spark 2.0, the entry point to a Spark execution environment like spark shell changed from SparkContext to SparkSession. SparkSession provides a single point of entry to interact with Spark without setting SparkConf, SparkContext or SQLContex. It also makes it easier to use DataFrame and Dataset APIs. However, now DataFrame works better with Scala and very limited with Java or Python. For this reason, in the application, we continue using RDD.

For the PySpark version of application, each image is one record in RDD, and the collection of images are partitioned and distributed across nodes in a cluster. We define a list for storing images and a method that appends image to the list. Each time when the method is invoked in MATLAB, an image is transfered from MATLAB to python, and appended to the list. After all images needed to be processed are stored in the list, from MATLAB, one can start the image registration progress. It will create or reuse a SparkSession with configuration, distribute all the images in distributed storage, and map the image registration function which is the same as the one we used in Python version, on each image. After finishing all stages, results are gathered and transferred back to client node automatically and return to MATLAB. We also tried appending a image directly to an existing RDD or DataFrame by creating a single DataFrame or single RDD for the new image and use the union function to merge the new one with existing one. However, this is a transformation thus it is lazy evaluated, and it takes much more time than parallelize the batch of images in one time.

6.4.2.1 Client mode or cluster mode

According to the official website, there are two deployment modes that can be used to launch Spark applications on YARN. One is cluster mode and the other one is client mode.

In cluster mode, the Spark driver runs inside an application master process, and the client can go away after initiating the application. This mode is suitable for client node that does not have strong computational ability or wants to do something else which may cause bottleneck while running the spark application. It is also good for those clients that are not same physically co-located with cluster workers, because it minimizes the latency caused by network between driver and executors. In client mode, the driver runs directly in the client process, and separated from application master. This mode allows to make full use of cores on the local machine. It is also suitable for running from Spark Shell because input and output of the application is attached to the console directly. It is also recommended for the users to submit applications from a gateway machine that is physically co-located with worker machines, then use client mode.

The use cases of this thesis were only executed in client mode on yarn cluster. However, in real application, both modes can happen. To design an application using yarn cluster mode, one can have a thread on local pc for managing the images. It will send a

signal to another thread after the number of images reaches a threshold, or time runs out. After receiving the signal, another thread will submit these images to HDFS directly or send them to one of the node in side cluster, then initiate and start the application.

6.4.2.2 Configurations on Yarn

While running applications on clusters, configurations and custom properties for the cluster can have big impact on performance. Some example for these basic configurations include number of executors, cores for each executor, and memory allocated for each executor. These configurations or properties can be set when submitting the Spark application through Spark Shell or creating a creating a SparkContext or SparkSession inside the application.

Before doing tests using default or random settings, it is important to estimate the optimal settings with analysis of the data size of the problem and property of client nodes. For example, in this use case, each image (256x256 int32) costs 256 KB for storage. This means, to store 500 images, at least 125 MB of empty space is needed. This information is important while setting Java heap size of Application Master (AM) as well as number of cores on node. When running with client mode, without specifying spark executor cores, yarn will only use one core for each executor by default. The number of cores to use for the YARN AM is also one for client mode. The number of executors is by default set to 2. Since the client node we used for thesis is also a HPC node in yarn cluster, which is the same type as other executor nodes, the setting can be roughly the same.

While processing a image set with 500 images during the experiment, the peak performance occurred when using 8 cores per executor. When the number of images increased to 600 images, which was 150 MB in total, it had better performance with only 2 cores per executor. It is always recommended to use less than 5 cores per executor especially if the application includes frequently read or write from HDFS, because too many cores per executor can affect the read and write throughput to HDFS. However, this number should also not be set too low. With only one core per executor, we will not be able running multiple tasks in the same JVM and gain the speed up. Also, with only one core per executor, broadcasting variables needs to be replicated in each core. After the right amount of cores for each executor is decided, number of executors per node can be calculated by diving total number of cores per node minus one (leave 1 core per node for Yarn daemons) by number of cores per executor. Amount of memory to use per executor process can be figured out by calculation [44]

$$(Mem_{node} - 1024MB)/(N_{exe} + 1) * 0.6 * 0.9$$

Mem_{node} is yarn.nodemanager.resource.memory (in MB) and (N_{exe} represents number of executer per node. Those two decimals numbers inside the equation are default values of spark.storage.memoryFraction and spark.storage.safetyFraction. Number of partition of RDD can be simply set equal to number of cores in the cluster.

6.5 Implementation result

Both hardware and software set up for this user case were the same as those we used in Chapter 4.

6.5.1 Performance analysis

In order to analyze the performance of different approaches, they were set up to process bunches of images with different amount. The result represents the relationship between number of images and execution time. The block size, grid size and window size for image registration algorithm were kept fixed, these parameters were picked by maximizing the performance of MATLAB solution. For each setting (combination of approach and the number of images), the measurements were executed 3 times and toke the average. The settings with same image amount were tested together, so that the processors had similar occupancies when idle. After each experiment, the memory and caches used for storing images and internal variables were freed up partially to avoid memory allocate error caused by lack of space. Preparing for new cache store also helps to improve performance. These free-up procedures were taken charge by each programming language automatically if is possible. Memory assigned for variables in CUDA, C needed to be freed up manually. Python and PySpark have own automatic garbage collection or cache cleaner, at the same time, users can also force garbage collector to release unreferenced memory. MATLAB, had officially no user-defined heap memory until version 7. From version 7, MATLAB has heap both in form of nested functions (closures) and handle objects. Using the *clear* command will synchronously all variables from the current workspace and free up the space by releasing them from system memory.

In Chapter 4 we already discussed about initialization time, memory transfer time and data conversion time for each solution. These extra times are also considered and partly included inside the use cases.

For initialization time, only the start time after warming up was included in measurements. Before measuring any solutions, a small example of data was feed to solutions in order to warm up and load the necessary library. SparkContext in PySpark was destroyed automatically after job is completes and the program exits. If MATLAB is not closed, although new SparkContext was created each time when starting a new setting of test, the initialization time for it after first time warm up was much shorter than the application executing time and therefore can be neglected, according to the conclusion from Chapter 4; this time was included inside overall time measuring. While calling modules written in CUDA and Python, initialization time was also included for these two acceleration measurements after warming up using small data. Comparing with other solutions, MATLAB costs much longer time (20 seconds for starting a pool with 4 workers) to start a pool. After the first time of creating a new pool, the pool is standby and costs no extra time for reusing for a period. After 30 minutes idle time (can be customized in setting), the pool shuts down and is deleted by MATLAB. Then it needs 20 seconds again to create a new pool. Despite the fact that initializing MATLAB pool affects performance more severe than other solutions (especially for real time problems), it can be easily avoided by changing setting or keeping pool busy every 30 minutes.

Because of this reason, we did not include the initialization time of MATLAB pool.

Time consumption for memory transferring was an important part of measurements, this included image copying between CPU and GPU, storing images in node using PySpark, sending images and collecting results between MATLAB and PySpark or Python.

Since we only executed the use case application in client mode on yarn cluster, the overhead caused by network access time was basically related to the cluster structure. It has more stable performance when applying different settings comparing to running with cluster mode. To avoid using cluster mode from outside the cluster, clients can also connect to nodes in the cluster to submit application, command and data, then start work in client mode. In this situation, data transmission is influenced by network traffic, physical location and etc. It would be easier to start MATLAB in one of the nodes inside the yarn cluster and use client mode. Unfortunately, MATLAB can not installed on Power7 machine in the cluster we used. To simulate the case that there is one node with MATLAB installed inside cluster to do the pre-processing part, we generated a bunch of images on one of the node, transformed them into the same format (MATLAB matrix) as output after DICOMread function, and use them as the input for the PySpark image registration algorithm.

For PySpark and Python, the time cost of memory copy between MATLAB and Python, as well as data type conversion for input and results of image processing module can be estimated from the local node. This part was also embodied into the final time consumption.

Figure 6.3 presents the actual results and estimate results for the use case with varieties of accelerating methods. There are in total nine nearly straight curves in this figure. Each curve demonstrates the relation between number of images been processed and time consumption of one approach. Five solid curves represent measurements from the local machine, they are image registration approaches using MATLAB with Pool, Python with Pool, PySpark in pseudo-cluster, Python without Pool, and GPU. Three of the four dashed curves describe measurements executed on yarn cluster or on a single HPC node inside the cluster. These tests include approaches using Python with Pool, PySpark local mode(pseudo-cluster) as well as Yarn client cluster mode. These three test sets are represented in the graph as Cluster-Python+Pool(estimate), Cluster-PySpark(pseudo-cluster)(estimate) and Cluster-PySpark(Yarn)(estimate). There is another dotted curve in the figure called Cluster MATLAB. This curve was created by scaling up the PySpark pseudo-cluster mode curve on cluster, using the same proportion of the time cost of local PySpark pseudo-cluster mode curve to local MATLAB mode.

There are some similarities between the results of our use case and the results showed in Figure 4.4. Among the tests on the local machine, Python without pool solution had the slowest performance. It is followed by MATLAB with pool solution. Local PySpark pseudo-cluster mode performed similar to Python with Pool solution, and required less than half of the time needed by Python without pool to complete the execution. As previously shown in the Figure 4.4, here too, accelerating using GPU is the fastest solution with a big margin. Among all the three dash lines which represent running on HPC single node or HPC cluster, the green one refers to PySpark on Yarn cluster has the best performance. It is slightly faster than using GPU on local machine. The other two curves are still faster than all the solutions except for the one using GPU on

local machine. The order of the performance of these three solutions is the same as the solutions using the same acceleration approaches on the single machine. It is because HPC node in cluster has faster computational power than local machine. The local machine used in this use case has 4 cores; 2 threads per core, that is 8 logical cores in total. Each HPC in the cluster has 16 sockets, one core per socket thus 16 cores. Each core has 4 threads, that is 64 logical cores. It also has 2 Non-uniform memory access (NUMA), processors number 0 to 31 share memory use the first NUMA, and the rest 32 processors use the second one.

If we focus on the first two lines in the Figure 6.3(Local-MATLAB+Pool and Local-Python(no Pool)), Python solution without pool is only two times slower than MATLAB with pool. MATLAB pool can assign work to maximum 4 workers instead of 8 workers, because Parallel Computing Toolbox used for pool considers only real cores, not hyper-threaded cores. From the hardware side, it is because there is usually only one floating point unit per pair of hyperthreads. MATLAB uses floating point almost everywhere, it stores by default all numeric values as double-precision floating point, and has tones of numerical intensive algorithms optimized for floating point. Due to these reasons, it is decided to keep maximum number of computational threads equal to the number of computational core, so that each thread can has a floating point unit. From the software side, whether using hyperthreads can bring benefit or not also depends on the algorithm itself.

Now we take PySpark in pseudo cluster mode and Python with multiprocessors these two curves also into consideration. Despite the fact that they are both multi-threads or multi-processors solutions using the same laptop, the performance is different from each other. Pool in Python is processor-based parallelism, local computer has 4 nodes, as expected, the performance is three times better than Python without pool, and less than two times faster than MATLAB pool solution. PySpark pseudo cluster mode is a non-distributed single-JVM deployment mode. Driver, executors and master are in the same single JVM. It is executed with local[*] configuration, which uses as many threads as the number of processors available to the Java virtual machine (by calling Runtime.getRuntime.availableProcessors()). This means PySpark pseudo cluster mode runs with 8 threads; equals to logical cores. Contrary to expectations that it should be 2 times faster than the Python solution that uses 4 nodes, it is actually slightly slower than the multiprocessor solution. This probably because the image registration application itself can not benefit from hyperthreading. Hyperthreading only gives benefits when the application has delays caused by cache misses, complex control, data hazards, or etc. However the image registration use case implemented is high parallelized, does not contain pending time or conplex control loops. That is why the PySpark with pseudo-cluster mode has similar performance as the processor-based parallelism solution rather than 2 times faster. The reason why the PySpark solution is even slightly slower than the Python solution with multiprocessors, is probably due to the extra time for execution of JVM and the time introduced for spawning of all those execution components needed for PySpark including driver, executors and master. Another factor for execution time difference is the difference in time consumption for data conversion. Converting matrix between Python and MATLAB takes 0.0037 seconds per image on average, while converting matrix between PySpark and MATLAB takes 0.0088 seconds per image on

average, that is nearly 2.5 times in time difference. When the the number of images to be processed increases a lot, data conversion could be a bottleneck for PySpark solution. For example, when applying image registration application on 800 images using PySpark with yarn cluster, 15% of overall time consumption is data conversion.

Although both Python with multiprocessors and MATLAB with pool solutions spawn four workers, the MATLAB solution is much slower than the Python solution. This is because workers assigned by MATLAB run with lower capacity. MATLAB has automatic balance mechanism, after activating all four workers, the total CPU capacity keeps around 70% for this use case. In contrast, when running Python with pool solution, CPU capacity is 100% for most of the time.

When comparing the performance running on a HPC with the one running on the local laptop, running the applications on the HPC take on average half the time needed to run the same applications on the local laptop. Considering the number of physical cores (local computer has 4 cores and HPC node has 16 cores), and the clock speed of cores (local computer cores run with 2.4 GHz process frequency and cores in HPC node run with 3.5 GHz process frequency), it was expected that the application will take less than 1/4 the time using the laptop. However the performance did not reach the expectation. This can be due to the following reasons: first of all, the architecture of processors for these two platforms are different. Local laptop uses x64-bases i7 processor which is based on Haswell architecture, and the HPC is equipped with ppc64 architecture cores. ppc64 is an identifier to refer to the target architecture for applications optimized for 64-bit big-endian PowerPC and Power Architecture processors. The difference in architecture can cause the performance differences. Additionally, each HPC node has 16 sockets and 2 NUMAs, there is a lot latency for communications between CPUs, especially between two CPUs that access to different NUMA. Another interesting finding is accelerating by GPU on the local laptop achieves similar performance to accelerating using PySpark on a 4 nodes cluster. At the same time, these two solutions are around 13 times faster than Python without pool, 6 times faster than MATLAB with Pool, and nearly 4 times faster than Python with Pool on local machine. The performance of GPU usually depends on both warp size and compute capability. And the compute capability decides whether memory transfers and instruction dispatch are grouped by warp or half-warp. In our case, as a high mid range video card, the compute capability of Quadro K1100M type of GPU is 3.0, which means both memory transfer and instruction dispatch are grouped by 32. This sounds like a big advantage in acceleration by parallelization. However based on the following reasons: lower peak FLOPs, weaker ALUs, lower memory bandwidth, smaller caches size per core comparing with CPU, and bottleneck caused by data transferring between CPU and GPU, GPU will not get ideal acceleration (near 32 times faster than Python without pool). GPU is suitable for problems that handle large amount of data with relatively simple operations, but not appropriate for performing complex algorithms, algorithms have interrupts, or algorithms have no data parallelized. GPU is ideal for this use case image registration application. That is why it could be almost as fast as PySpark solution on a cluster with 4 nodes with 16*4=64 physical cores, and only 13 times faster than single thread local computer.

PySpark using cluster is 4 times faster than PySpark with pseudo-cluster at beginning, but gradually turns to be 3 times faster when the data size increases. While scaling

up the cluster, the overhead caused by communication between nodes and heads in the cluster needs also to be considered.

6.5.2 Cost analysis and programming difficulties

Another angle to compare these methods can be from economics perspective, but the cost for building a Spark cluster varies from plan to plan. However, for each solution we used, there is one common part of the overall cost that is unavoidable and can be determined. This cost is the cost for MATLAB licenses. In this part of the thesis, we compare the sum of the cost of licenses for all necessary toolboxes and MATLAB itself for all the solutions.

Price of MATLAB licenses depend on the using purpose which can be one of the standard, education, student or home. Here we compare the price for standard MATLAB licenses. Table 6.1 lists the MATLAB toolboxes required for each solution, along with their perpetual license price. An annual license price for a toolbox usually equals to the price of a perpetual license divided by 2.5.

All the solutions in this table need licenses for MATLAB software and for Image Processing Toolbox. The second one is used to read in DICOM images and store them as matrix in MATLAB. Among all the toolboxes listed in this table, MATLAB Distributed Computing Server toolbox costs the most: it costs more than 5072. This toolbox is necessary for MATLAB programs and Simulink models to be excuted on top of computer clusters, clouds, and grids. This price depends on worker quantity. It starts from 317 per unit for 16 32 worker and requires a minimum quantity of 16 workers.

When running applications on Spark or Hadoop clusters, the official solution requires MATLAB Compiler to create and execute compiled MATLAB applications against clusters. Which is the second most expensive toolbox inside this table. These two costs also make the price for MATLAB with Spark through official way considerable higher comparing to other solutions. In the official solution, user designs a MATLAB application that applies a particular function on data using Spark. Instead of calling Spark functions directly from MATLAB, user needs to create another file to specify Spark properties, to create SparkConf objects, SparkContext objects, and RDD objects, and pass the function handle of the desired function to a flatMap method. Then MATLAB compiler is needed to generate a executable file. The configuration of Spark cluster is similar to directly using Spark through Java or Python, however user can code directly in MATLAB and map the function on data against Spark without rewriting it into other languages. The performance using the official MATLAB Spark solution is expected similar or even slightly better than our solution that running PySpark through Python, because official MATLAB solution probably uses Spark Java API to implement the interface between MATLAB and Spark.

When it comes to programming difficulties, coding in Python is more or less the same as coding in MATLAB. Using multiprocessing package for using pool in Python is also comparable to using pool in MATLAB. PySpark is more difficult to learn and use, it requires the knowledge for both cluster and data storage like HDFS. It also requires experience from users to configure the cluster and to decide which cluster manager to use. The difficult part would be the Spark application optimization, debugging the

Solution	Products that need	Price	In Total
MATLAB with parallel Pool	MATLAB Parallel Computing Toolbox Image Processing Toolbox	2000 1000 1000	4000
MATLAB with GPU	MATLAB Parallel Computing Toolbox Image Processing Toolbox	2000 1000 1000	4000
MATLAB with C or Python	MATLAB Image Processing Toolbox	2000 1000	3000
MATLAB with Hadoop and Spark (official)	MATLAB MATLAB Distributed Computing Server MATLAB Compiler Image Processing Toolbox Parallel Computing Toolbox	2000 >=5072 4000 1000 1000	>=13072
MATLAB with Hadoop and Spark (through Python)	MATLAB Image Processing Toolbox	2000 1000	3000

Table 6.1: Cost for different solutions

application and logging the activities. GPU programming can be a little bit frustrating even with C or CPP programming background. The complicated part is debugging GPU program and the entry point for connecting with MATLAB.

6.5.3 Conclusion

In conclusion, for the image registration use case, the fastest solutions are MATLAB with GPU on single machine and MATLAB accelerated using Spark on the yarn cluster. The solution using GPU costs 1000 euro extra for toolbox license. On the other hand, building a Spark cluster can be even more expensive. The cheapest but also efficient solutions are MATLAB calling Python module using multiprocessors or MATLAB calling PySpark psuedo-cluster mode on the local laptop. They both only require a basic MATLAB license and a license for Image Processing Toolbox, and the whole application can run locally. However the performance of these two solutions is based on the number of processors of local laptop. In order to achieve a better result, a high end computer is necessary. The solution that can be scaled up and also with good performance for a large amount of data is accelerating using Spark. The official way to do it from MATLAB using MATLAB toolboxes is at least 4 times more financially expensive than the proposed solution through the Python module.

The strategy to choose the right solutions is explained in Figure 6.2: If the application needs to process a large amount of data, or may be scaled up in the future, MATLAB with PySpark would be the best choice. This is also true if the solution needs to be robust against power failures. To estimate the needed number of nodes, one can first measure the local mode performance on one of the nodes, compare the performance with what you want to get, and use that to make an estimate of the total number of the required nodes. If the application has data parallel features, especially image processing applications, a

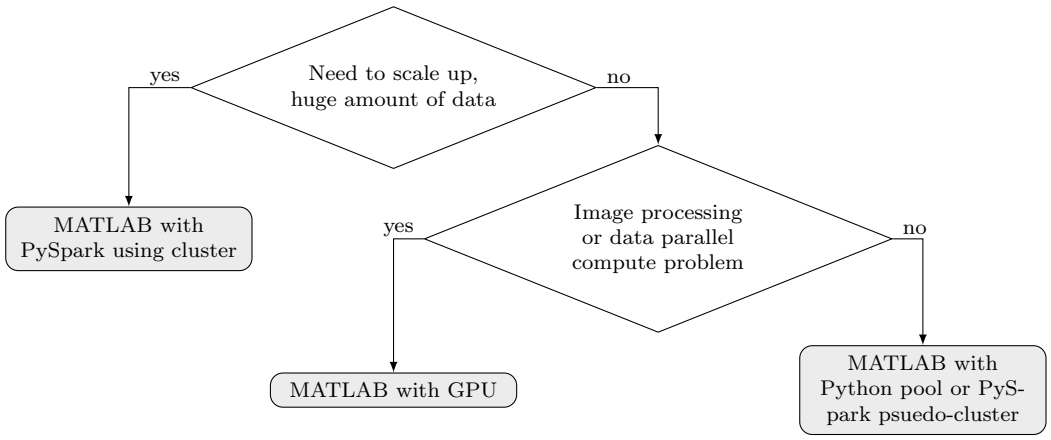


Figure 6.2: Decision tree for solutions

GPU would be a more suitable solution. If it is not ideal to accelerate through GPU, possible alternatives are using Python with multiprocessors or PySpark psuedo-cluster locally. These two methods speed up the application by exploiting multi-processors.

6.6 Further improvement

In this section, we list some improvements that can help increase performance.

As mentioned in the beginning of this chapter, it was not possible to call the Python module using pool from MATLAB due to limitations in the workflow. However, there is a feasible way to bypass this problem according to MATLAB experts. Using this method, we can achieve performance that is superior to the PySpark psuedo-cluster mode.

One of the features that can be included in the future is to automate the workflow with pipelines. This is useful for streaming data or data that changes quickly. For such a scenario, one can look into cudaStream library for GPU, sklearn package for Python, and streaming module for PySpark.

For the solution that uses GPU, some extra acceleration can be gained by parallelization using both GPU and CPU to hide the bottleneck of GPU-CPU data copy. For example, at the same time an image is being processed on GPU, the next image is copied from CPU to GPU. The sequential addressing parallel reduction algorithm used in the use case can be modified again since only half of the threads were used in the first iteration.

For the PySpark solution, we still need to find out the way to store data from MATLAB directly to RDD or HDFS without using an extra toolbox. It is also worth trying to combine C or CUDA with Spark. One feasible way that already proved possible is wrapping C file to .so file, and submit this file (with the flag –py-files) with PySpark file that invoke the C function together. This can gain appreciable acceleration. Using a cluster solution with C was 2.5 times faster than using only a Spark cluster (167 seconds compared to 423 seconds).

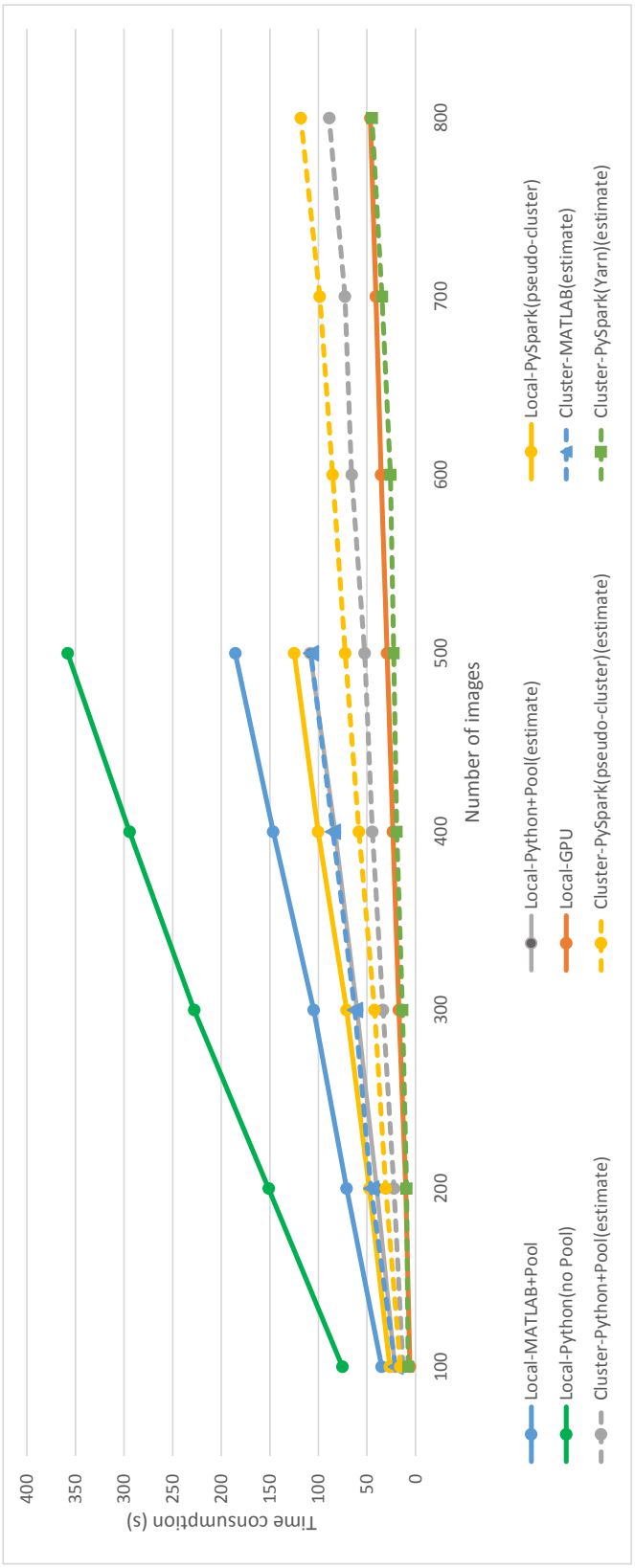


Figure 6.3: Image registration time consumption testing on both local pc and cluster

Conclusions and future research

7

7.1 Conclusions

This thesis has investigated possible solutions to accelerating existing MATLAB code, by calling modules written by GPU, Spark and Python Pool with multi processors. These solutions have been analyzed for bottlenecks based on their performance in initialization, memory transfer, data conversion and computational throughput. The solutions were compared with each other and with MATLAB official solutions which include MATLAB Pool and MATLAB cluster.

MATLAB is one of the most popular choices for commercial and educational organizations. It provides a bunch of functionality and toolboxes, which makes it convenient to build a prototype. It is widely used in industries including higher education, computer software and hospital & health care. At the same time, MATLAB keeps updating and releasing toolboxes for the latest features like Spark support. However, MATLAB also has the following problems: First, except for matrix related computations, MATLAB applications are slow compared with other languages like Java or C. Second, MATLAB provides extra useful functions by adding extra toolboxes at an extra cost, or updating to new version. For teaching purposes or building prototypes in small companies, this is not necessary. Also, advanced programming in MATLAB like using Spark and calling C, can be still complex for users. We considered to design several ways to improve MATLAB code and compare the final performance with original MATLAB solutions.

In this direction, we have proposed several ways that can be used for accelerating the MATLAB application. Respectively, MATLAB integrated with GPU, C, Python and PySpark. All these approaches are compiled or wrapped so that they can be invoked from MATLAB directly to replace the original function. Among these approaches, we allow access to PySpark through Python instead of using the newest MATLAB version and distributed computing server that are necessary for the official Spark approach. Four metrics are tested on these solutions to identify bottlenecks. These are initialization, memory transfer, data conversion and computational throughput. Our measurements demonstrated that initialization & memory transfer for GPU, data conversion for Python/Pyspark when the data input or output have high dimensions can be bottlenecks. For testing the computational throughput, a computationally intensive kernel (finding the number of primes less than x using A-PRP) was used. When input data size was small, Pyspark had large overhead, C and CUDA were the best choices, respectively 4.9x and 7.6x faster than MATLAB. When the data size increased, MATLAB using Pool has almost the same performance as PySpark running in psueudo-cluster mode. PySpark on a cluster had the best performance, which can be 12x faster than MATLAB with Pool. Locally, the CUDA solution was still the best, which was 8.8x faster.

Based on the results obtained, a medical image registration MATLAB application

using NCC was accelerated by multiple solutions. This implementation further indicates the overall performance of these solutions when it comes to real medical image processing application. Considering this use case, GPU and PySpark using cluster had the best performance, which were 5.7x and 7.8x faster than MATLAB with Pool performance, respectively. A decision tree for the most optimal solution to chose was obtained based on the results, which can be used for choosing the right accelerated kernel in future platform implementations. One possible application of our research would be a library that can provide acceleration for MATLAB. Such a library would detect the type of the MATLAB code and application, and automatically choose the best solution under that circumstance.

Returning to the questions posed at the beginning of this thesis, it is now possible to state the following answers:

- It is possible to scale up existing MATLAB code on an Apache Spark, with little effort, using open source tools. The extra work needed here is to rewrite the code part that needed to be executed on Spark in Python.

- In general the applications that can benefit from scalable cluster are suitable for a scalable MATLAB implementation. The overhead of scaling MATLAB code on Spark will be the initialization time to start the cluster and the data conversion between MATLAB and PySpark. The initialization time from MATLAB on local laptop machine to start pseudo-cluster PySpark with all available threads takes 3 seconds, while starting a pool with same amount of threads using MATLAB Pool is 20 seconds.

- The bottlenecks that limit the speedup gained from Spark scalability is the data conversion when the input or output data are more than one dimension, and the network connection between local node and the cluster when they are not physically co-located.

- One the advantages of using open source tools to scale up MATLAB code in comparison with the official MATLAB scalability solution is reducing the cost for the license. More importantly is the flexibility of the code that run on the cluster. The official MATLAB scalability solution uses MATLAB compiler to compile the code into an executable file, which is difficult for further acceleration. It also limited the possibility to use libraries. However, with open source tools, we can further improve the performance for example using GPU or C on cluster. One of the disadvantage will be rewriting the code into Python and PySpark, and the extra piece of code to convert the data type between MATLAB and Python.

7.2 Future perspectives

To future our research, we intend to implement a wrapper outside the existing CUDA library so that they can be directly invoked in MATLAB. Taken the similarity of MATLAB and Python code, another research will be carried on which focus on the automatic Python code generator. This can be further used in PySpark code generation, after

setting up the cluster configuration by the user manually. To minimize the time consumption of data conversion in Python and PySpark solution, further work needs to be done in order to share the data between Python and MATLAB more efficiently. [45]

Bibliography

[1] R. Wei, R. Xie, L. Zhang, and L. Song, "Fast depth decision with enlarged coding block sizes for hevc intra coding of 4k ultra-hd video," in *2015 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct 2015, pp. 1–6.

[2] S. Ojha and S. Sakhare, "Image processing techniques for object tracking in video surveillance- a survey," in *2015 International Conference on Pervasive Computing (ICPC)*, Jan 2015, pp. 1–6.

[3] C. Guillemot and O. L. Meur, "Image inpainting : Overview and recent advances," *IEEE Signal Processing Magazine*, vol. 31, no. 1, pp. 127–144, Jan 2014.

[4] T. H. Tsai, Y. H. Lee, and Y. Y. Lee, "Design and analysis of high-throughput lossless image compression engine using vlsi-oriented felics algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 1, pp. 39–52, Jan 2010.

[5] A. K. Gupta, S. Nooshabadi, D. Taubman, and M. Dyer, "Realizing low-cost high-throughput general-purpose block encoder for jpeg2000," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 7, pp. 843–858, July 2006.

[6] M. Jafari and R. Shafaghi, "A hybrid approach for automatic tumor detection of brain mri using support vector machine and genetic algorithm," vol. 1, pp. 1–8, 12 2012.

[7] S. Singh, D. Gupta, R. Anand, and V. Kumar, "Nonsubsampled shearlet based ct and mr medical image fusion using biologically inspired spiking neural network," *Biomedical Signal Processing and Control*, vol. 18, pp. 91 – 101, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1746809414001943

[8] "Companies using matlab," https://idatalabs.com/tech/products/matlab, [Online; accessed 19-April-2018].

[9] "Matlab features," https://nl.mathworks.com/products/matlab/features.html, [Online; accessed 29-June-2017].

[10] "Matlab — wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=MATLAB&oldid=787262126, [Online; accessed 29-June-2017].

[11] "Matlab incorporates lapack," http://nl.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html, [Online; accessed 03-July-2017].

[12] "Parallel computing on the cloud with matlab," https://nl.mathworks.com/products/parallel-computing/parallel-computing-on-the-cloud.html, [Online; accessed 10-July-2017].

[13] "Using matlab with other programming languages," https://nl.mathworks.com/solutions/matlab-and-other-programming-languages.html, [Online; accessed 05-July-2017].

[14] Y. M. Altman, *Undocumented Secrets of MATLAB-Java Programming*, 1st ed. Chapman & Hall/CRC, 2011.

[15] "Apache spark — wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Apache_Spark&oldid=786787628, [Online; accessed 03-July-2017].

[16] "Spark clustered," https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-cluster.html, [Online; accessed 21-July-2017].

[17] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.

[18] P. Bhosale, M. Staring, Z. Al-Ars, and F. Berendsen, "Gpu-based stochastic-gradient optimization for non-rigid medical image registration in time-critical applications," in *Proc. SPIE Medical Imaging Conference*, Houston (TX), United States, February 2018.

[19] S. Ren, K. Bertels, and Z. Al-Ars, "Efficient acceleration of the pair-hmms forward algorithm for gatk haplotypecaller on gpus," *Evolutionary Bioinformatics*, vol. 14, March 2018.

[20] N. Ahmed, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Gpu accelerated api for alignment of genomics sequencing data," in *Proc. IEEE International Conference on Bioinformatics and Biomedicine*, Kansas City, USA, November 2017.

[21] S. Ren, K. Bertels, and Z. Al-Ars, "Gpu-accelerated gatk haplotypecaller with load-balanced multi-process optimization," in *Proc. 17th annual IEEE International Conference on BioInformatics and BioEngineering*, Washington DC, USA, October 2017.

[22] ——, "Exploration of alternative gpu implementations of the pair-hmms forward algorithm," in *Proc. 3rd International Workshop on High Performance Computing on Bioinformatics*, Shenzhen, China, December 2016.

[23] E. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "Gpu-accelerated bwa-mem genomic mapping algorithm using adaptive load balancing," in *Proc. 29th International Conference on Architecture of Computing Systems*, Nuremberg, Germany, April 2016, pp. 130–142.

[24] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens, "Multimodality image registration by maximization of mutual information," *IEEE Transactions on Medical Imaging*, vol. 16, no. 2, pp. 187–198, April 1997.

[25] D. L. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes, "Medical image registration," *Physics in medicine & biology*, vol. 46, no. 3, p. R1, 2001.

[26] M. V. Wyawahare, P. M. Patil, H. K. Abhyankar *et al.*, "Image registration techniques: an overview," *International Journal of Signal Processing, Image Processing and Pattern Recognition*, vol. 2, no. 3, pp. 11–28, 2009.

[27] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and vision computing*, vol. 21, no. 11, pp. 977–1000, 2003.

[28] A. Melbourne, G. Ridgway, and D. J. Hawkes, "Image similarity metrics in image registration," vol. 7623, 03 2010.

[29] M. Xu and Y. Sun, "Registration of multimodal and temporal images of the retina using a combined feature-based and statistics-based method," in *2012 5th International Conference on BioMedical Engineering and Informatics*, Oct 2012, pp. 353–357.

[30] X. Huang, J. Ren, G. Guiraudon, D. Boughner, and T. M. Peters, "Rapid dynamic image registration of the beating heart for diagnosis and surgical navigation," *IEEE transactions on medical imaging*, vol. 28, no. 11, pp. 1802–1814, 2009.

[31] K. Miller, A. Wittek, G. Joldes, A. Horton, T. Dutta-Roy, J. Berger, and L. Morriss, "Modelling brain deformations for computer-integrated neurosurgery," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 1, pp. 117–138, 2010.

[32] J. Schindelin, I. Arganda-Carreras, E. Frise, V. Kaynig, M. Longair, T. Pietzsch, S. Preibisch, C. Rueden, S. Saalfeld, B. Schmid *et al.*, "Fiji: an open-source platform for biological-image analysis," *Nature methods*, vol. 9, no. 7, p. 676, 2012.

[33] T. Boehler, D. van Straaten, S. Wirtz, and H.-O. Peitgen, "A robust and extendible framework for medical image registration focused on rapid clinical application deployment," *Computers in biology and medicine*, vol. 41, no. 6, pp. 340–349, 2011.

[34] S. E. Mahmoudi, A. Akhondi-Asl, R. Rahmani, S. Faghih-Roohi, V. Taimouri, A. Sabouri, and H. Soltanian-Zadeh, "Web-based interactive 2d/3d medical image processing and visualization software," *computer methods and programs in biomedicine*, vol. 98, no. 2, pp. 172–182, 2010.

[35] S. Vemula and C. Crick, "Hadoop image processing framework," in *Big Data (Big-Data Congress), 2015 IEEE International Congress on.* IEEE, 2015, pp. 506–513.

[36] N. L. R. D. D. J. M. J. S. Deependra K. Mishra, Scott E. Umbaugh, "Image processing and pattern recognition with cviptools matlab toolbox: automatic creation of masks for veterinary thermographic images," pp. 9971 – 9971 – 11, 2016. [Online]. Available: https://doi.org/10.1117/12.2238183

[37] S. Mittal, S. Gupta, and S. Dasgupta, "Fpga: An efficient and promising platform for real-time image processing applications," in *National Conference On Research and Development In Hardware Systems (CSI-RDHS)*, 2008.

[38] I. Chiuchisan, "A new fpga-based real-time configurable system for medical image processing," in *E-Health and Bioengineering Conference (EHB), 2013*. IEEE, 2013, pp. 1–4.

[39] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu–past, present and future," *Medical image analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

[40] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision," in *Proceedings of the 16th ACM international conference on Multimedia*. ACM, 2008, pp. 1089–1092.

[41] "Mathworks — pass data to python," http://nl.mathworks.com/help/matlab/matlab_external/passing-data-to-python.html, [Online; accessed 28-September-2017].

[42] "2.2: Fermat, probable-primality and pseudoprimes - probable primes," http://primes.utm.edu/prove/prove2_2.html, [Online; accessed 27-November-2017].

[43] "Dicom example files," https://nl.mathworks.com/matlabcentral/fileexchange/2762-dicom-example-files, [Online; accessed 26-April-2018].

[44] "Use the right level of parallelism," https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/sparksqlshufflepartitions_draft.html, [Online; accessed 26-April-2018].

[45] J. Debayle and B. Presles, "Rigid image registration by general adaptive neighborhood matching," *Pattern Recognition*, vol. 55, pp. 45 – 57, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320316000455