GPU integration for Deep Learning on YARN

ROBIN ANDERSSON
GPU integration for Deep Learning on YARN

Robin Andersson

Master of Science Thesis

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

21 August 2017

Examiner: Jim Dowling
Abstract

In recent years, there have been many advancements in the field of machine learning and adoption of its techniques in many industries. Deep learning is a sub-field of machine learning that is largely attributed to many of recent innovative applications such as autonomous driving systems.

Training a deep learning model is a computationally intensive task that in many cases is inefficient on a CPU. Dramatically faster training can be achieved by making use of one or more GPUs, coupled with the need to train more complex models on increasingly larger datasets, training on a CPU is not sufficient.

Hops Hadoop is an Hadoop distribution that aims to make Hadoop more scalable, by migrating the meta-data of YARN and HDFS to MySQL NDB. Hops is currently making efforts to support distributed TensorFlow. However, GPUs are not currently managed natively by YARN, therefore, in Hadoop, GPUs cannot be scheduled to applications. That is, there is no support for isolating GPUs to applications and managing access to applications.

This thesis presents an architecture for scheduling and isolating GPUs-as-a-resource for Hops Hadoop. In particular, the work is constrained to supporting YARN’s most popular scheduler, the Capacity Scheduler. The architecture is implemented and verified based on a set of test cases. The architecture is evaluated in a quantitative approach by measuring the performance overhead of supporting GPUs by running a set of experiments.

The solution piggybacks GPUs during capacity calculation in the sense that GPUs are not included as part of the capacity calculation. The Node Manager makes use of Cgroups to provide exclusive isolation of GPUs to a container. A GPUAllocator component is implemented that keeps an in-memory state of currently available GPU devices and those which are currently allocated locally on the Node Manager.

The performance tests concluded that the YARN resource model can be extended with GPUs, and that the overhead is negligible.

With our contribution of extending Hops Hadoop to support GPUs as a resource, we are enabling deep learning on Big Data, and making a first step towards support for distributed deep learning.
Sammanfattning

De senaste åren har många framsteg gjorts inom maskininlärning och användning av dess tekniker inom många olika industrier. Djupinlärning är ett delområde inom maskininlärning som är hänförlig till många av de senaste innovativa applikationerna såsom system för autonom bilkörning.

Att träna en djupinlärningsmodell är en beräkningsmässigt intensiv uppgift som i många fall är ineffektivt på endast en processor. Dramatiskt snabbare träning är möjlig genom att använda en eller flera grafikkort, kopplat med behov att träna mer komplexa modeller med större datamängder, är det inte hållbart att endast träna på en processor.

Hops Hadoop är en Hadoop distribution med målet att göra Hadoop mer skalbart, genom att migrera metadata från YARN och HDFS till MySQL NDB. Hops utför i nuläget ett arbete med att stödja distribuerad TensorFlow. För närvarande finns inget stöd för grafikkort som en resurs i YARN, därmed, i Hadoop, så kan grafikkort inte schemaläggas för applikationer. Mer specifikt, det finns inget stöd för att isolera grafikkort för applikationer och erbjuda som en resurs.


Lösningen tar inte hänsyn till grafikkort som en del av kapacitetsberäkningen. Node Manager komponenten använder Cgroups för att utföra isolering av grafikkort. En GPUAllocator komponent har implementerats som håller ett tillstånd över vilka grafikkort som allokertas och vilka som är lediga på Node Manager.

Experimenten konkluderar att YARN kan stödja grafikkort som en resurs och att tidspåverkan för detta är försumbart.

Detta arbete med att stödja grafikkort som en resurs på Hops Hadoop, gör det möjligt att utföra djupinlärning på stora datamängder, och är ett första steg mot stöd för distribuerad djupinlärning.
Acknowledgements

Firstly, I would like to thank my examiner Jim Dowling, Associate Professor at KTH ICT school, for help during the thesis process. Secondly, I would like to thank my supervisor at Hops Hadoop, Gautier Berthou and the members of the Hops team for valuable discussions and for their efforts in helping me get familiar with the development environment.
# Contents

1 Introduction ............................................. 1
   1.1 Problem description ................................... 3
   1.2 Goals .................................................. 3
   1.3 Boundaries ........................................... 3
   1.4 Hypothesis ............................................ 4
   1.5 Ethical, social and environmental aspects ............... 4

2 Background ............................................. 7
   2.1 Linux Cgroups ........................................ 7
      2.1.1 Hierarchical model ................................ 7
      2.1.2 Device subsystem ................................... 8
   2.2 NVIDIA GPU ........................................... 9
      2.2.1 CPU vs GPU in machine learning .................... 9
      2.2.2 Management library ................................ 9
      2.2.3 GPU device numbers ................................. 10
      2.2.4 Drivers ............................................ 10
   2.3 Cluster Management .................................. 10
   2.4 Hadoop-YARN architectural overview .................... 11
      2.4.1 Client ............................................. 12
      2.4.2 Application master ................................ 12
      2.4.3 Resource manager .................................. 13
         2.4.3.1 Capacity scheduler .............................. 13
         2.4.3.2 ResourceTrackerService ........................ 15
      2.4.4 Node manager ..................................... 15
         2.4.4.1 NodeStatusUpdater .............................. 16
         2.4.4.2 ContainerManager ............................... 16
         2.4.4.3 CgroupsLCEResourcesHandler .................... 17
         2.4.4.4 LinuxContainerExecutor ......................... 17
         2.4.4.5 Recovery ....................................... 18
   2.5 Related work .......................................... 18
      2.5.1 Mesos .............................................. 18
2.5.2 GPU allocation ........................................ 19

3 Method .................................................. 21
3.1 Research approach .................................... 21
3.2 Data Collection & Analysis ............................ 21

4 Design & Implementation .................................. 23
4.1 Resource constructor ................................... 23
4.2 Resource Manager components ....................... 23
  4.2.1 Configuration ...................................... 23
  4.2.2 DominantResourceCalculatorGPU .................. 24
4.3 GPU Management components ......................... 25
  4.3.1 Abstraction Layer .................................. 25
  4.3.2 NVIDIA Implementation ............................ 25
4.4 Node Manager components .............................. 26
  4.4.1 Configuration ...................................... 26
  4.4.2 NodeStatusUpdaterImpl ............................ 26
  4.4.3 GPUAllocator ...................................... 26
    4.4.3.1 Data-structures ............................... 27
    4.4.3.2 Initialization .................................. 27
    4.4.3.3 Allocation ...................................... 27
  4.4.4 CgroupsLCEResourcesHandlerGPU .................. 28
    4.4.4.1 Setting up the devices hierarchy ............... 28
    4.4.4.2 Isolation ....................................... 29
4.4.5 Recovery ........................................... 29

5 Analysis ............................................... 31
5.1 Validation ............................................ 31
5.2 Experimental Testbed & Measurement Tools ............ 31
  5.2.1 NM startup and registration with RM ............... 31
  5.2.2 NM launching containers .......................... 32
5.3 Configuration ......................................... 32
  5.3.1 Default ........................................... 32
  5.3.2 DefaultCgroup ..................................... 32
  5.3.3 CgroupsGPU ....................................... 33
5.4 Experimental Results ................................ 33
  5.4.1 NM startup and registration with RM ............... 33
  5.4.2 NM launching containers .......................... 34
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Distributed TensorFlow on YARN</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Execution of MapReduce jobs on YARN</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Components in the Resource Manager</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of multiple queue configuration</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Components in the Node Manager</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Mesos architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Mesos containerizer GPU allocation components</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Time to start NM in relation to configuration</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution time of preExecute method in relation to configuration</td>
<td>35</td>
</tr>
<tr>
<td>A.1</td>
<td>NodeStatusUpdaterImpl deciding number of GPUs to register with RM</td>
<td>45</td>
</tr>
<tr>
<td>A.2</td>
<td>GPUAllocator initialization</td>
<td>46</td>
</tr>
<tr>
<td>A.3</td>
<td>Initialization of Cgroup hierarchy</td>
<td>47</td>
</tr>
<tr>
<td>A.4</td>
<td>Contents devices.list file in the hops-yarn hierarchy</td>
<td>48</td>
</tr>
<tr>
<td>A.5</td>
<td>Cgroup devices controller and hops-yarn hierarchy</td>
<td>48</td>
</tr>
<tr>
<td>A.6</td>
<td>Interaction for allocation and isolation of GPU</td>
<td>49</td>
</tr>
<tr>
<td>A.7</td>
<td>Interactions for recovering GPU allocations for GPUAllocator</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables

A.1 NvidiaManagementLibrary implementation overview ............. 44
A.2 Overview of tests run ................................................ 51
List of Listings

4.1 New Resource constructor ................. 23
4.2 Old Resource constructor ................. 23
4.3 Per-container GPU maximum/minimum allocation configuration . 24
5.1 Default configuration ................. 32
5.2 Standard Cgroup configuration ................. 32
5.3 CPU and GPU Cgroup configuration ................. 33
List of Acronyms and Abbreviations

YARN  Yet Another Resource Negotiator
HDFS  Hadoop Distributed File System
RM    Resource Manager
NM    Node Manager
AM    Application Master
RPC   Remote Procedure Call
CPU   Central Processing Unit
GPU   Graphics Processing Unit
TPU   Tensor Processing Unit
FPGA  Field-Programmable Gate Array
Cgroups Linux Control Groups
Chapter 1

Introduction

In recent years there have been many significant advancements [1][2][3] in the field of machine learning [4]. New applications making use of machine learning techniques can be seen almost daily, breaking the barrier of what was previously thought of as fiction. Much of the recent progress in the field can be attributed to a sub-field of machine learning called deep learning [5]. Deep learning is composed of algorithms inspired by the structure and function of the brain called artificial neural networks. In deep learning, deep neural networks are trained that make use of multiple layers, which makes it possible to learn non-linear data patterns. An impressive list of applications making use of deep learning models include self-driving cars [6], medical diagnosis system outperforming doctors [7] and AI beating the world champion in Go[8].

Producing a machine learning model is a non-trivial process that needs to account for several criteria [9] for the model to generalize beyond the examples found in the training set. These criteria include, amongst other, selecting an appropriate machine learning technique and fine-tuning parameters, training and testing on a representative dataset, accounting for overfitting and normalizing of training samples. Furthermore, the training phase to produce the machine learning model may take a long time, potentially days or even weeks. This is particularly true for deep learning models, since they are typically large, complex and may require many epochs to converge. In conclusion, it is often a time-consuming process.

How long it takes to train specific a machine learning model depends on factors such as the number of training samples, epochs, model complexity and computational power available. The only factor here that should not impact the accuracy of the resulting model is the computational power available for training.

So how can we increase the computational power for training machine learning models? In principle there are two approaches, assuming that the training can be parallelized. Scaling vertically and scaling horizontally.
means that the training is run on a single machine and resources are added to the machine capability to increase performance. For machine learning applications, scaling vertically would mean making use of specialized hardware such as GPU, FPGA or TPU that have been shown to provide significant speed-ups when training machine learning models such as neural networks. The most widespread of the three for machine learning applications are GPUs. A GPU is used to run highly computation-intensive and parallelizable operations during training, such as matrix multiplication, while the remaining operations are run on the CPU. Scaling horizontally is done by distributing the training over multiple machines, and by increasing the number of machines taking part in the computation, increases the computational power. For machine learning applications, this is also an efficient solution to reduce the training time. However, high-bandwidth data transfer protocols such as RDMA [10] should be installed in the cluster for best performance. Since a Machine can only house up to a fixed number of GPUs, the future solution for efficient machine learning training is likely to scale horizontally to distribute the training, and scale vertically by adding more specialized machine learning hardware. A set of machines used for processing may be referred to as a cluster, and is typically managed by software that handles scheduling, execution of jobs, resource localization and failure scenarios, such as Hadoop [11].

Hops Hadoop [12] is an Hadoop distribution that migrates the metadata from HDFS and YARN into a MySQL NDB Cluster. By storing the metadata in NDB, Hops is able to support multiple stateless Name Nodes, achieving 16x performance gains over Hadoop HDFS [13].

HopsWorks [14] is the front-end for Hops. It introduces the abstraction of projects and datasets and provides users a graphical interface to interact with services such as Spark, Flink, Kafka, HDFS and YARN. There are currently efforts being made to incorporate distributed TensorFlow for the platform with the goal to make it the go-to platform for machine learning. Figure 1.1 shows an overview of the anticipated result.
1.1 Problem description

YARN does not support resources other than memory and CPU, meaning that training may potentially be a time-consuming process since no GPU or other specialized hardware may be used. The contribution of this thesis will be a solution for supporting GPU on YARN and benchmarking the overhead for supporting GPU.

1.2 Goals

The purpose of this work is to demonstrate that supporting GPU as a schedulable and isolable resource in YARN for Hops Hadoop is possible, and that it comes with a negligible overhead.

1.3 Boundaries

The solution for GPU support will only be for Hops Hadoop, specifically it will only consider the Capacity Scheduler when configured to use a single queue since Hops does not make use of multiple queues.

There exists several GPU manufacturers but this thesis only takes NVIDIA GPUs into account since they are currently the biggest manufacturer. The design will take steps towards providing a generic interface for different manufacturers but will only be tested for NVIDIA.

Several GPU allocation policies are possible. One approach is GPU sharing, where multiple containers may use the same GPU concurrently. Another approach is GPU timeslicing, where access to the GPU is given as a round-robin timeslice.
for concurrently running containers. The allocation policy for this work will be container exclusive. Meaning that a container is allocated GPU(s) exclusively, so at any given time a GPU may only be accessible by at most one container, for the duration of that container’s execution.

There exists certain topological advantages such as NVLink, which means that some GPUs residing on the same machine are interconnected with high-bandwidth, enabling fast GPU-to-GPU communication. This work will not consider such topological benefits.

The architecture will be based on Hops Hadoop 2.7.3, and will not consider any future changes to the architecture of Hops Hadoop or Hadoop.

GPU support will only be tested on Red Hat Enterprise Linux 7.2 since it is the Linux-distribution Hops Hadoop is using.

1.4 Hypothesis

The hypothesis set by the thesis is that it is possible to support GPU as a schedulable and isolable resource in YARN, under the boundaries set in section 1.3, and that it comes with a negligible overhead.

1.5 Ethical, social and environmental aspects

From an ethical point of view, making it possible to use GPUs for machine learning applications causes no potential issues by itself, but rather what types of activity the models will be used for. Machine learning itself does not impose constraints on what type of data may be processed or how ethical the use of the resulting model would be. One of the typically portrayed unethical machine learning applications is artificial intelligence. Although no system can replicate the human mind, systems exist which can outperform humans in well-defined use-cases. It should be noted that just as GPUs can accelerate machine learning applications which causes harm, they can also be used for to combat unethical activity such as anomaly detection for credit-card frauds.

From an environmental point of view, GPUs may be used to further advancements of modeling and predicting climate change. Furthermore, a hybrid CPU+GPU environment have been found to be more energy efficient than only using a CPU, in fact multiple orders of magnitude more energy efficient. [15] Worth noting is that these comparisons typically compare workloads optimized for GPU, and puts power consumption in relation to processing time.

Machine learning applications can provide social benefits on different levels. For example on an individual level, allowing doctors to make more accurate
diagnosis. Also allowing hospitals to make better utilization of their staff since less doctors may be used for consultation. On a societal level, more accurate diagnosis should lead to a healthier population, making it possible for more individuals to contribute to the society.
Chapter 2

Background

This section will describe necessary background information for the reader to comprehend the thesis. Firstly there is a section about Linux Cgroups, a resource isolation mechanism in the Linux kernel. Secondly, a section introducing NVIDIA GPU and an entail of how GPUs may be identified on a system and what software is necessary for processes to access them. The next section constitutes the largest part of the background and describes YARN components and entities used to manage cluster resources, and how Cgroups are being used for resource isolation. Finally, a section about the recently added GPU support in Apache Mesos in related work.

2.1 Linux Cgroups

Linux Cgroups [16] are a Linux kernel feature that allow processes to be organized into hierarchies which can be limited in terms of various types of resource usages. Linux Cgroups is composed of several so called subsystems, also referred to as controllers, each with a specific purpose to control different resources types. There are many subsystems, for example memory to limit memory usage of a process, cpu to limit cpu usage and blkio which limits access time when using block devices such as hard disk drives.

2.1.1 Hierarchical model

The Linux Process Model and Linux Cgroups are similar in the way that they are structured. Cgroups subsystems are organized hierarchically, just like processes, and each child Cgroup created under a parent Cgroup inheirit the same resource limitations as their parents. All processes on Linux are child processes of the same parent: the init process, that is created at boot time and is forked to create child
processes, which in turn fork their own child processes. Since all Linux processes
descend from a common parent, the hierarchical model can be illustrated as a tree
of processes, which is not the case with Cgroups. Cgroups hierarchy is composed
of potentially multiple unconnected trees. A single tree would not be enough,
since each hierarchy may have one or more subsystems attached to it, depending
on what resources should be limited.

The Cgroup hierarchy directory containing all the subsystems is created and
mounted at boot time as a virtual filesystem. A common location in several
Linux distributions is `/sys/fs/cgroup`. Under the root directory, a directory for each
Cgroup subsystem can be found, in which directories for hierarchies attached to
specific subsystems are created.

### 2.1.2 Device subsystem

The device subsystem [17] is used when the processes in the Cgroups hierarchy
should be restricted in terms of which devices, or rather more specifically, which
device numbers it may access on the system. To be able to enforce device access,
each Cgroup keeps a whitelist of which device numbers processes attached to
the Cgroup may access. The device subsystem hierarchical model imposes the
constraint that a child Cgroup can only access exactly the same or a subset of
accessible device numbers of the parent Cgroup. Cgroup exposes an interface for
users to define which processes should be attached to Cgroups and what device
numbers may be accessed, using a set of control files.

`devices.list` lists what device numbers are currently accessible in the Cgroup.
An entry in this whitelist consists of four parts. The device type, which may be
either ‘c’ for char device, ‘b’ for block device or ‘a’ for all devices. It is then
followed by a major and minor device number, separated by a colon, such as
‘1:2’. The major or minor device number can be replaced with a ‘*’ to match all
major or minor device numbers. Finally, the device access is defined, which is
a composition of ‘r’ for read access, ‘w’ for write access and ‘m’ for being able
to mknod. When the parent Cgroup of the hierarchy is created the file contains
a single entry of the form ‘a *:* rwm’, meaning that access to all devices is by
default allowed. The file should never be explicitly modified, as this is done by
the kernel, when writing to `devices.allow` and `devices.deny`

`devices.allow` is used to add entries to the Cgroup `devices.list`, which is done
by writing device access entries to the file.

`devices.deny` is used to remove entries in the Cgroup `devices.list`, which is
done by writing device access entries which should be removed.

`tasks` is a list of tasks, which have been attached to the Cgroup and is therefore
limited by the device access constraints defined in the Cgroup. Attaching a process
to the Cgroup can be done by writing the PID to this file.
2.2 NVIDIA GPU

Graphics Processing Unit, or commonly known as GPU is a specialized electronic circuit, traditionally used to perform computation-intensive and parallelizable computations to create frames for display devices, such as monitors. GPUs are now used for more than creating frames, its highly parallelizable computational model, large memory buffer and high-bandwidth makes it ideal for certain algorithms and processing large data sets.

2.2.1 CPU vs GPU in machine learning

Machine learning is one of the fields in computer science that benefit the most from using GPUs. Training a machine learning model, such as a neural network can result in dramatic speed-ups when certain operations are run on a GPU instead of the CPU. The reason is that a neural network typically requires processing large data sets and running computation-intensive, highly parallelizable operations such as matrix multiplication. It is also possible to make use of multiple GPUs concurrently, allowing for even more speed-ups.

2.2.2 Management library

NVIDIA Management Library [18] is a C library that exposes an API for monitoring and managing the state of NVIDIA GPUs. The library is named libnvidia-ml.so and is installed along with the NVIDIA driver. It is intended for use with third party applications. There are two types of operations, those that modify the state of the GPUs and those who extract the state. Queries to extract state involve:

- Identification: Static and dynamic information relating to identification of the GPU, such as querying what bus the GPU resides on and the device id on the bus. The minor device number can be retrieved for all GPUs on the system, the major device number is always set to 195.

- Power management: Current information about power usage and limits.

- Clocks/PState: Maximum and current clock speeds and performance state of the GPU.

- GPU utilization: GPU utilization rates, both for the computational part in terms of processing and buffer utilization.

- Active processes: A list of processes currently running on the GPU and how much memory each process is using in the buffer.
• ECC error counts: Number of internal errors handled by error correcting
code during processing, either since last boot or during the whole life of the
GPU.

Queries to modify state involve:
• ECC mode: Enable and disable ECC.
• ECC reset: Clear single and double bit ECC error counts.
• Compute mode: Control whether compute processes can run on the GPU
and whether they run exclusively or concurrently with other compute
processes.
• Persistence mode: Control whether the NVIDIA driver stays loaded when
no active clients are connected to the GPU.

2.2.3 GPU device numbers

A device number is used to uniquely identify a particular device on the system,
therefore a GPU can be identified as a combination of a major and minor device
number. The major device number is used in the kernel to identify the driver for
the device, whereas the minor device number is used by the driver to determine
which device file is being referenced. For NVIDIA GPUs, the major device
number is always set to 195 and the minor device number corresponds to which
GPU is being referenced, starting from 0 up to $N-1$ if there are $N$ GPUs present
on the machine. Each NVIDIA GPU present on the system is exposed as a separate
device file in /dev, and listed as e.g. /dev/nvidia0 for the GPU with minor device
number 0.

2.2.4 Drivers

For the operating system to be able to communicate with a device, a device driver
needs to be used. In the case of NVIDIA GPUs there are three device files which
needs to be accessible by the process utilizing the GPU. These are /dev/nvidia-
uvm, /dev/nvidia-uvm-tools and /dev/nvidia-ctl.

2.3 Cluster Management

A cluster is a collection of machines that are connected directly or in-directly
and is typically abstracted to the degree that it may be thought of as a single
system. Clusters are needed to provide sufficient processing power and storage
to meet organizations requirements. Managing clusters requires sophisticated software to handle all the issues that are inherent of distributed systems such as machine failure and replication of data. A typical division of software in a cluster is resource management and file system. The resource management handles the allocation of resources and executions of jobs on the cluster, and the file system provides an interface to get access to, create, read and write files for running applications.

2.4 Hadoop-YARN architectural overview

Hadoop YARN [19] is a collection of entities that provide resource management in a Hadoop cluster. Hadoop YARN was developed for the second generation of Hadoop with the goal of separating the programming model from the resource management. The first generation was developed to leverage an ecosystem where MapReduce was the only supported framework, one of biggest drawbacks of the first generation of Hadoop. YARN provides a generic interface where frameworks only need to implement an application specific component that interacts with YARN to execute jobs in the cluster. Furthermore, different types of data processing engines such as real-time streaming, batch-processing and relational queries are also supported.

There are three different types of control entity in the YARN architecture: a Resource Manager, a Node Manager and an Application Master.

The Resource Manager is the central unit of control in the cluster and exposes APIs for users to submit jobs that are to be executed on the cluster. There is only a single active instance of the RM at any time, and it keeps a global view of the cluster. It is also the entity that makes scheduling decisions, deciding on which machines in the cluster an application should be executed.

The Node Manager is run on every machine in the cluster that should be used for execution of applications. It is responsible for launching tasks on the machine and manages the container life-cycle. A container is an abstraction for a unit of work in a job. Other responsibilities for the NM include authentication of leases for containers, resource management of the machine, garbage collection and some auxiliary services.

The Application Master is the head of a particular job, it is also the first container that starts for any given job. The AM will request resources in the cluster for execution of its particular job. Furthermore, it manages the lifecycle phases of the application, such as the flow of the execution, e.g. in the case of Map Reduce; running mappers before reducers.

Figure 2.1 depicts the flow of interactions for running MapReduce jobs on YARN.
2.4.1 Client

The client is used to submit an application to the RM. It is typically done by creating a YarnClient object, which is then initialized with parameters for the specific job. After the YarnClient is initialized the client sets up the Application Context, prepare the first container which is the Application Master, and submits the job to the RM.

2.4.2 Application master

After the job has been submitted to the RM by the client, the AM is started on one of the NM. The sequence of interactions the AM performs when launching an application are: communicate with the RM to request containers that satisfy requirements such as resources, and after receiving those containers communicate with one or more NM to launch the containers to execute the job. The AM communicates asynchronously, during communication with RM an AMRMClientAsync object is used to request containers, and an event handler for callbacks AMRMClientAsync.CallbackHandler is used to receive the containers. For communication with the NM(s), an NMClientAsync object is used to start and monitor the execution of the job, in the same fashion calls are received through NMClientAsync.CallbackHandler. Running containers report back their status to the AM, and the AM is responsible for the lifecycle of the application such as
2.4. HADOOP-YARN ARCHITECTURAL OVERVIEW

handling failed containers.

2.4.3 Resource manager

The Resource Manager is a central unit of control that essentially is the heart of resource management in the cluster. The RM arbitrates the cluster in terms of available resources and mainly serves the purpose of allocating containers on Node Managers to execute submitted jobs. It keeps a global view of all Node Managers and in doing so can make optimal scheduling decisions. The Resource Manager is a pure scheduler in the sense that it does not manage the state of the running application. The AM is responsible for receiving status updates and taking decisions. This separation of responsibility means that the scheduler is able to scale better and support different frameworks. The Resource Manager supports different types of scheduling policies, of which only one may be active at any time. Figure 2.2 depicts an overview of components in the RM.

![Resource Manager Components](image)

Figure 2.2: Components in the Resource Manager [21]

2.4.3.1 Capacity scheduler

The YarnScheduler in figure 2.2 component corresponds to one of three pluggable scheduling policies that may be used by the Resource Manager. The schedulers are FifoScheduler, FairScheduler and CapacityScheduler. The FifoScheduler and FairScheduler will not be explained as they are not considered for this thesis.
The CapacityScheduler is designed to support the idea of running applications in a shared and multi-tentant cluster where it is important to support capacity guarantees between different groups of users (organizations) while maximizing the utilization of the cluster. Traditionally, each organization would have their own cluster with sufficient compute capabilities to meet their SLAs, but it leads to underutilization during non-peak hours. The CapacityScheduler enables sharing of resources between organizations, so that when resources are underutilized by one organization they are free to use by others. The benefits are that it provides elasticity and optimal cluster utilization. In addition, there are not only limits on an organizational level that limit how much resources the whole organization may use, but also on an per-user level.

An abstraction introduced by the CapacityScheduler are queues. Queues can be thought of as a subset of all the capacity in the cluster. The root queue is the main queue, and matches the capacity to that of the entire cluster capacity. Under the root queue, it is possible to define hierarchies of queues, where the first level of the hierarchy is referred to as parent queue and the sub-queues are referred to as leaf queues. This model makes it possible to define complex capacity constraints in an organization.

To exemplify, an organization may consist of multiple departments such as support, engineering and marketing. Each of these departments may further be divided into several divisions. Figure 2.3 depicts how the queues can be configured in an organization.

![Figure 2.3: Example of multiple queue configuration [22]](image-url)
2.4. HADOOP-YARN ARCHITECTURAL OVERVIEW

2.4.3.1 ResourceCalculator

The CapacityScheduler measures capacity as a percentage, and there are currently only two ways to measure capacity in the cluster. The first approach is to only consider memory as the capacity measurement in the cluster. Which would mean that given a total cluster capacity of 100GB, 10GB would be considered as 10% of the capacity. To use this approach the CapacityScheduler should be configured to use the DefaultResourceCalculator.

The second approach is to consider both memory and CPU and pick the dominant fraction. The solution to this is based on Dominant Resource Fairness [23]. To exemplify, assume a total cluster capacity of 100GB and 8 virtual cores. 10GB and 2 virtual cores would amount to 25% since 2 out of 8 is larger than the fraction 10 out of 100. To use this approach the CapacityScheduler should be configured to use DominantResourceCalculator.

In addition to measuring capacity, both ResourceCalculator classes make decisions such as if a particular container can be allocated on a specific NM.

2.4.3.2 ResourceTrackerService

The ResourceTrackerService is a component that interacts with NMs using RPC for registration and forwarding heartbeats from the NM to the YarnScheduler. It is also used to check liveliness of NM and status of containers that are running is have been requested or is running on the NM.

2.4.4 Node manager

The Node Manager is run on each of the machines in the cluster. In the slave/master architecture they fit the part of the slave node. Figure 2.4 gives an overview of the components in the NM.
2.4.4.1 NodeStatusUpdater

When starting the NM, this component registers with the RM and offers a subset of the available resources in terms of memory and virtual cores that may be allocated by the RM on the machine. To determine how large fraction of the resources should be offered, the NM keeps two configuration parameters [25]. *yarn.nodemanager.resource.memory-mb* determines the physical memory in terms of MB. *yarn.nodemanager.resource.cpu-vcores* determines the number of virtual cores. After registration, subsequent communication with the RM is to send heartbeats which piggyback any updates related to containers. These updates may involve new containers started by AMs, running container statuses and containers which have completed using the ResourceTrackerService.

2.4.4.2 ContainerManager

The ContainerManager is the most central component of the NM. It is composed of a number of subcomponents that each perform a necessary task to run containers on the machine.
2.4.2.1 RPC Server

The RPC server corresponds to the component that the AMs communicate with in order to start containers that have been allocated by the RM or to kill existing ones. Each request is authorized using the ContainerTokensSecretManager.

2.4.2.2 Resource Localization Service

Resource localization component handles downloading and organizing files that are needed by the container. It also enforces access control on the downloaded files to ensure other containers are not able to interfere.

2.4.2.3 Containers Launcher

The Containers Launcher component is responsible for preparing and launching containers. It maintains a pool for threads to prepare and launch containers and perform cleanups of containers.

2.4.3 CgroupsLCEResourcesHandler

CgroupsLCEResourcesHandler is used by the NM enforce cpu isolation for containers using Cgroups. CgroupsLCEResourcesHandler exposes two methods, `preExecute(ContainerId, Resource)` and `postExecute(ContainerId)`. `preExecute` is used to create the child Cgroup for the container in the cpu subsystem and writes to the `cpu.shares` to enforce cpu isolation. `postExecute(ContainerId)` is used to delete the container Cgroup from the cpu subsystem.

2.4.4 LinuxContainerExecutor

The LinuxContainerExecutor is used when the NM is configured to use resource isolation through Cgroups, using the CgroupsLCEResourcesHandler component. LinuxContainerExecutor is responsible for launching containers and makes use of an external executable `container-executor` for this purpose. Before launching a container, LinuxContainerExecutor calls CgroupsLCEResourcesHandler `preExecute` to create the child Cgroup for the container in the cpu subsystem and initialize the subsystem. Afterwards, when launching the container using the `container-executor` executable, the PID of the launched container is written to the `tasks` file of the Cgroup, which effectively attaches the container PID to the Cgroup. When the container has completed, crashed or is killed, the `postExecute` method is called and the Cgroup is removed.
2.4.4.5 Recovery

NM supports restarts, a feature which means that the NM may be restarted without affecting running containers. Since each container runs as a separate process and is not in any way attached to the NM. Consider for example a case where all resources have been allocated on the NM, but an upgrade needs to be made. Simply killing all the containers and stopping the NM would not be ideal for the users running jobs. Furthermore, when the NM restarts it needs to re-instantiate the state about currently running containers and containers that need to be started.

For this purpose, the NM uses LevelDB [26], to store metadata about running and requested containers. When the NM is restarted, it reattaches itself to the containers by monitoring the exit code of running containers and launches requested containers.

2.5 Related work

2.5.1 Mesos

Apache Mesos [27] is a resource manager which recently integrated GPU as a native resource. Mesos provide three base abstractions, the Mesos master, the Mesos agent and framework scheduler. Figure 2.5 illustrates the overlying architecture.

The Mesos Agent provides the same abstraction as a Node Manager in YARN. They both run on each slave node on which jobs should be executed and manage the functionality required to run tasks, such as resource isolation and localization of files.

The Mesos Master keeps an overview of all the Mesos Agents and determines the free resources available on each slave node, and sends resource allocation offers to frameworks based on an allocation policy.

The frameworks have their own scheduler component that may accept or decline an offer by the Mesos Master to allocate resources on a set of machines.
2.5. RELATED WORK

2.5.2 GPU allocation

Mesos recently incorporated GPU as a first class resource, providing the same resource abstraction as memory and CPU, a simple scalar value. Each GPU is simply thought of as a number, without taking any account for the GPU compute capability, or topology in which the GPU reside.

Figure 2.6 illustrates the architecture of the Mesos Agent GPU allocation and isolation components. Since this work will not consider GPU support for Docker on YARN the relevant components in the figure are the Mesos Containerizer, Isolator API and Nvidia GPU Allocator.
The Mesos Containerizer is an API to instantiate the container environment for the container to be run on the Mesos Agent. In order to setup the environment for the container to isolate resources like CPU, memory and GPU, Mesos leverages an Isolator API. For resource isolation on Linux, the Isolator API makes use of Linux Cgroups. To isolate GPU(s), the device subsystem is used to restrict access to device numbers. Mesos instantiates a hierarchy with a fixed set of device numbers, a child Cgroup is created under the hierarchy for each container, and device access is modified so that the container is only given access to the set of GPUs that was allocated for the container.
Chapter 3

Method

3.1 Research approach

The research methodology in this thesis follows the quantitative research method. The thesis is divided into several deliverables. The first deliverable consists of producing a design of how GPU may be integrated under the current resource model. The second deliverable is the software implementation of the design. The final deliverable is to provide an evaluation of the overhead of scheduling and isolating GPUs. The goal is to show that it is feasible to allow GPU scheduling under the current resource model, and show that the overhead is negligible. Arguably the overhead should not be significant, but is still deemed a relevant contribution to the work. The implementation is verified by extending the existing Apache Hadoop test suits and additional tests for the added components.

Due to time limitations to complete the thesis, the solution will not be a general solution for Apache Hadoop. There exists several boundaries as mentioned in section 1.3. The solution will however provide valuable insights for how a a general solution would be implemented.

3.2 Data Collection & Analysis

Experiments were conducted to collect data that in order to evaluate the overhead of scheduling GPUs. The overhead can be broken down into two different scenarios. Firstly, the overhead of starting a Node Manager and initializing subsystems. Secondly, the overhead on the Node Manager when allocating and isolating a container with GPU.
Chapter 4
Design & Implementation

4.1 Resource constructor

An additional Resource constructor, seen in listing 4.1 has been added. The previous constructor, seen in listing 4.2, only specified memory and virtual cores.

Listing 4.1: New Resource constructor
Resource.newInstance(int memory, int vcores, int gpus)

Listing 4.2: Old Resource constructor
Resource.newInstance(int memory, int vcores)

So effectively, GPUs are expressed as a quantity in the same way as memory and virtual cores. The interpretation of the gpus field is that the number corresponds to the number of physical GPU cards. Which suits the resource model of performing exclusive allocations of GPUs for containers.

The old constructor is left unchanged, so in effect there are two ways to specify the resource depending on whether the Resource should include GPUs or not. The old constructor should still exist, otherwise compatibility will break with other frameworks such as Apache Spark. Specifying the constructor without GPUs sets the field to 0.

4.2 Resource Manager components

4.2.1 Configuration

The Capacity Scheduler uses configuration that specifies the minimum and maximum memory and virtual core allocation for a container. The same type
of configuration should be added for GPU, to enable explicit configuration of how many GPUs a container may allocate. See listing 4.3 for new configuration.

Listing 4.3: Per-container GPU maximum/minimum allocation configuration

\[
\text{yarn.schedulers.minimum-\text{allocation-gpus}}
\]

\[
\text{yarn.schedulers.maximum-\text{allocation-gpus}}
\]

\(\text{yarn.schedulers.minimum-allocation-gpus}\) default value is 0, which is highly recommended since otherwise the first container for the AM, which does not require GPU, will not be allocated.

\(\text{yarn.schedulers.minimum-allocation-gpus}\) default value is 8. It should be set to some value greater than 0, such as the largest number of GPU(s) present on a machine in the cluster.

4.2.2 DominantResourceCalculatorGPU

As mentioned in the boundaries, this solution will not support multiple queues. Therefore the complexity of the Capacity Scheduler is reduced drastically, since it is the main feature of it. Instead, only the root queue is supported. Supporting GPUs with multiple queues would mean incorporating GPUs as part of the capacity calculation. There are two issues with that.

Firstly, GPUs are typically a sparse resource, they are expensive and it is not possible to fit many on a single machine machine. Therefore, if a user may only allocate 10% of the cluster resources, and there are 10 GPUs available on the cluster in total, allocating one would mean the user would not be able to request more resources. Ideally, it should be possible to allocate more than that.

Secondly, an underlying assumption when considering memory and virtual cores is that they are both critical for containers. A task is not able to run without memory or virtual cores, a task can execute without a GPU. This leads to the issue that if there is a single GPU in the cluster, and it is allocated, it is as if the whole capacity of the cluster is allocated to applications, even though the whole cluster may still be un-utilized except for the container using the GPU.

A new component DominantResourceCalculatorGPU extends the current DominantResourceCalculator to account for GPU. A piggyback solution for GPUs have been implemented, meaning that the component calculates capacity only based on memory and virtual cores, exactly the same as DominantResourceCalculator.

\(\text{computeAvailableContainers(Resource available, Resource required)}\) is the only important method that needs to be extended. It calculates the number of containers with a Resource \textit{required} that can fit in a Resource capability \textit{available}. During a scheduling cycle, the scheduler loops through pending ResourceRequests submitted by AMs, and attempts to allocate a container to a NM. The operation which is used to determine whether or not a particular
container may be allocated to a NM, depends on if the ResourceCalculator `computeAvailableContainers` method returns a positive integer or not. The method is extended to make sure that in addition to memory and virtual cores, also GPUs can "fit" on the NM. Since the solution only makes use of the root queue, and not multiple queues the provided solution for scheduling is sufficient.

### 4.3 GPU Management components

#### 4.3.1 Abstraction Layer

An abstraction layer, `hops-gpu-management` [30] is defined to provide a common interface, `GPUManagementLibrary`. A generalized interface is needed to be able to support multiple GPU manufacturers in the future. The abstraction layer defines the following methods.

```java
public boolean initialize();
public boolean shutDown();
public int getNumGPUs();
public String queryMandatoryDrivers();
public String queryAvailableGPUs(int configuredGPUs);
```

- `initialize()` method performs operations to initialize the state of the library and load necessary drivers. This action is needed by the Nvidia Management Library, and will likely be needed in the future by other GPU manufacturers.
- `shutDown()` method performs operations for cleanup after using the library. This action is needed by the Nvidia Management Library to release any allocated GPU resources.
- `getNumGPUs()` method performs operations for discovering the total number of GPUs residing on the system.
- `queryMandatoryDrivers()` method queries the operating system or the management library for the GPU manufacturer to discover the device numbers corresponding to the drivers.
- `queryAvailableGPUs(int configuredGPUs)` method performs operations for discovering device numbers for the GPUs that should be used for scheduling. The configured number of GPUs sets an upper limit of how many GPUs should be offered by the NM and returns the device numbers corresponding to those GPUs. The semantics behind `configuredGPUs` is explained in 4.4.2.

#### 4.3.2 NVIDIA Implementation

An implementation of the `GPUManagementLibrary`, called `NvidiaManagementLibrary` [31] is produced for NVIDIA GPU. The implementation JNI calls to query the C-
api of the Nvidia Management Library, in combination with system calls. Figure A.1 shows an overview of interface method and the implementation.

## 4.4 Node Manager components

### 4.4.1 Configuration

When `yarn.nodemanager.resource.gpus.enabled` is set to true, the NM initializes the GPUAllocator and Cgroups device hierarchy, as described in section 4.4.3 and 4.4.4.

`yarn.nodemanager.resource.gpus` sets an upper limit of how many GPUs the NM will at-most offer to the RM. Since there is no guarantee that the configured value matches the actual number of GPUs on the machine, it is necessary to query the system to detect GPU(s). Section 4.4.2 describes the mechanism of deciding how many GPU(s) the NM will offer up for scheduling.

### 4.4.2 NodeStatusUpdaterImpl

The NodeStatusUpdaterImpl which reports available resources to the RM have been extended to also take into account GPUs. When registering with the RM, the component sends the resource capability, that is how much memory, number of virtual cores and GPUs that should be offered for scheduling. `yarn.nodemanager.resource.gpus` sets an upper limit of how many GPU(s) the NM will at-most offered to the RM. Since there is no guarantee that the configured value matches the actual number of GPU(s) on the machine, it is necessary to query the system to detect GPU(s). For this purpose, NodeManagerHardwareUtils and LinuxResourceCalculatorPlugin have been extended. Figure A.1 shows the interactions between the components. The NodeStatusUpdaterImpl invokes the `getNodeGPU(yarnConfiguration)` method with the configured value of `yarn.nodemanager.resource.gpus`. The method then gets an instance of the LinuxResourceCalculator which interacts with NvidiaManagementLibrary to find out the number of GPU(s) on the machine. NodeManagerHardwareUtils then returns the min between how many GPU(s) were configured, and how many are available on the machine. This is how the at-most semantic is achieved.

### 4.4.3 GPUAllocator

The new component GPUAllocator initialize and maintain data-structures that keep track of which GPUs have been allocated to which container, and GPUs that are currently not allocated. To manipulate the state it exposes an API that provides
an allocate(ContainerId, numGPUs) and release(ContainerId) method. It is important to separate the concept of allocation and isolation. The GPUAllocator component only performs the allocation in terms of updating internal state of the NM to reflect the GPU(s) allocated for running containers.

4.4.3.1 Data-structures

There are four different data-structures which the GPUAllocator keeps. The data-structures identifies a GPU or a driver as a device number 195:N, encapsulated by a Device class, where N is the minor device number of the NVIDIA GPU. The major and minor device number for the drivers are also encapsulated in a Device object.

- availableGPUs Set<Device>: A set of GPU(s) on the machine that have been configured to be used by the NM, and are currently not allocated for any container.
- totalGPUs Set<Device>: A set of all the GPU(s) on the machine which may or may not be configured to be used by the NM.
- mandatoryDrivers Set<Device>: A set of drivers that each container being allocated GPU(s) needs access to.
- containerGPUAllocationMapping Map <ContainerId, Set<Device>>: A mapping between containerId and a set of allocated GPUs.

4.4.3.2 Initialization

When the GPUAllocator is created, it creates the data-structures, and a subsequent call to the initialize method is made to initialize the state. The performs a series of method invocations to the NodeManagerHardwareUtils and NvidiaManagementLibrary to initialize the data-structures. Figure A.2 shows the interaction between the GPUAllocator and NvidiaManagementLibrary during initialization of the component. Message 1.1 getNodeGPUs() have been simplified, see figure A.1 for the full method sequence.

4.4.3.3 Allocation

The allocate(ContainerId, numGPUs) method takes two parameters, a ContainerId uniquely identifying the container, and the number of GPUs to allocate for the container. Internally, the method selects the requested number of GPUs, numGPUs from availableGPUs data-structure and creates a mapping <ContainerId, Set<Device>> to reflect the allocation. The return value of the allocate method
is not intuitive, since it is the device numbers of all the GPU devices present on the machine, that have not been allocated for the container, the reason for this is explained in section 4.4.4.2. The release(ContainerId) method removes the <ContainerId, Set<Device>> mapping and adds the Device objects back into the availableGPUs data-structure.

4.4.4 CgroupsLCEResourcesHandlerGPU

The CgroupsLCEResourcesHandlerGPU component is an extension of component CgroupsLCEResourcesHandler that adds support for the Cgroups devices subsystem. The component initializes a Cgroup hierarchy in the device subsystem. Furthermore, before a container is launched, it creates a child Cgroup under the hierarchy, one for each container, and enforces GPU device access as described in section 4.4.4.2.

4.4.4.1 Setting up the devices hierarchy

When the NM is started, the devices subsystem hierarchy needs to be initialized to reflect the set of device numbers that any container may at-most be given access to. Preparing the hierarchy may be broken down into several steps.

**Step 1.** the hierarchy for the devices subsystem needs to be created. Assume Cgroups is mounted at /sys/fs/cgroup, then the devices subsystem is located at /sys/fs/cgroup/devices. Creating the hierarchy can be done by issuing a mkdir command in the devices subsystem, which upon creation initializes the hierarchy with the devices control files. See figure A.5 for an overview of the directory structure after creating the hierarchy, assuming the hierarchy is named hops-yarn.

**Step 2.** Upon creation of the hierarchy. The devices.list file in the hierarchy contains a single entry a *:* rwm. Since each child Cgroup created under the hierarchy inherits the parent device access, the hierarchy should be initialized with a predefined set of devices number. The first step is to remove the default entry in devices.list, this is done by writing a *:* rwm to devices.deny. After the default entry is removed, the next step is to add explicit device number access to the hierarchy.

**Step 3.** The set of device numbers that should be added to the hierarchy devices.list, is a default list of device numbers, the GPU drivers device numbers and the device numbers corresponding to each GPU on the machine. The default list of devices number is a list of the bare-minimum device numbers a container should be given access to on a Linux system, as identified by Mesos [32]. The device numbers for the GPUs and drivers present on the system are retrieved by the component from the GPUAllocator. After collecting the device numbers, the component writes each device number entry to devices.allow file.
Figure A.3 shows the interaction between CgroupsLCEResourcesHandlerGPU and GPUAllocator. Figure A.4 shows the contents of the hierarchy whitelist.

4.4.4.2 Isolation

If a new child Cgroup is created under the hierarchy defined in figure A.5 and if the hierarchy devices.list file contains the contents in figure A.4, then all containers would effectively have the same device access. In order to provide GPU isolation, only the container which have been allocated one or more GPUs should have exclusive access to the allocated GPU(s). Therefore the device access of the Cgroup corresponding to a container that been allocated GPUs, needs to remove all device entries corresponding to GPUs that it has not been allocated. For this purpose, the GPUAllocator allocate method returns the list of all device entries for GPUs that should be denied access to, any GPU that have been allocated, will not be included in the list, and therefore will still be given access to.

When a container is being launched, the LinuxContainerExecutor will enforce resource isolation by calling the preExecute(ContainerId, Resource) method in the component. preExecute(ContainerId) in turn calls setupLimits(ContainerId, Resource) method which creates the Cgroup for the container, both for the cpu and devices subsystem. In the case of restricting device access for GPUs, the method extracts the requested number of GPUs in the Resource object and proceeds with an allocate(ContainerId, numGPUs) call to the GPUAllocator component that returns the GPUs to which access should be denied. The device numbers are then written to the devices.deny file of the container Cgroup, and thereby removes access to GPUs for which the container has not been allocated.

However, it is not sufficient to just create the Cgroup with the correct device access for the container. To attach the container to the Cgroup, the PID of the container process must be written to the tasks file of the child Cgroup. This is done after the preExecute method, during the actual launch of the container, when the LinuxContainerExecutor launches the container.

When the container has completed its task, a clean up in performed, and the LinuxContainerExecutor calls the postExecute method, which in turns calls clearLimits that deletes the Cgroup from the hierarchy, and also removes the allocation in the GPUAllocator by calling the release(ContainerId) method. See figure A.6 for an overview of the interaction with devices subsystem during GPU allocation.

4.4.5 Recovery

Since GPUAllocator keeps an in-memory state to keep track of GPU allocations for containers, it will be problematic if the NM needs to do a restart. If the state is
not restored, the GPUAllocator will simply assume that all GPUs are available for new containers to use after restart since the ContainerId-GPU mapping is gone. Solving this issue requires that the GPUAllocator is restored to the original state after restart.

The NM makes use of LevelDB, which makes it possible to persistently store key-value pairs. The metadata stored in LevelDB includes the Resource capability of the container. So in effect, what is stored in LevelDB does not need to be modified. However, it is not sufficient simply to know that a container makes use of GPUs, since the GPUAllocator needs to know exactly which GPUs, in terms of device numbers, is allocated for the container. The natural solution to this problem is to simply read the Cgroup for each container. Since the ContainerId of the container stored in LevelDB corresponds directly to the name of the Cgroup for the container, the CgroupsLCEResourcesHandlerGPU reads each Cgroup for containers that make use of 1 or more GPUs and reinitialize the GPUAllocator based on GPU device numbers written to *devices.list*. See figure A.7 for the interaction of the components during recovery.
Chapter 5

Analysis

5.1 Validation

The implemented solution needs to be validated so that it is possible to claim it is correct. The tests must be deterministic and produce the same result each time they are run. Furthermore, the functionality must conforms to what was specified in section 1.3.

Therefore the test suit consists of two parts, firstly a set of tests that were added to test functionality in added components such as the GPUAllocator, secondly extending certain Apache Hadoop tests. All tests are written in JUnit [33]. Table A.2 shows an overview of the tests that were run to validate the architecture.

5.2 Experimental Testbed & Measurement Tools

Benchmarking of the RM have not been included in the evaluation since the changes are not considered significant enough to have any affect on the performance. To measure the overhead of scheduling GPUs, two scenarios were identified on the NM. Each scenario is run 10 times for three different configurations.

5.2.1 NM startup and registration with RM

During startup there are several new invocations and state that need to be initialized. Firstly, the NodeStatusUpdaterImpl needs query NvidiaManagementLibrary to discover how many GPUs are present and select the min or the GPUs present on the machines and the configured value of GPUs. Secondly, the GPUAllocator needs to be initialized which requires a series of calls to the NvidiaManagementLibrary. Thirdly, the Cgroup devices hierarchy needs to be initialized.
5.2.2 NM launching containers

This scenario measures the time it takes to execute the \textit{preExecute} method which enforces resource isolation through Cgroups. The method encapsulates the series of method calls needed to isolate GPUs as depicted in A.6. Therefore it is a suitable experiment to measure the overhead.

5.3 Configuration

The different configurations used in the testing scenarios are referred to as \textit{Default}, \textit{DefaultCgroup} and \textit{CgroupsGPU}.

5.3.1 Default

The \textit{Default} configuration corresponds to how the NM is configured when running containers without Cgroups. When Cgroups is not used, there is no CPU or GPU isolation and container’s are simply spawned as unix processes. See listing 5.1 for configuration.

Listing 5.1: Default configuration

```
<property>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>DefaultContainerExecutor</value>
</property>
<property>
  <name>yarn.nodemanager.linux-container-executor.resources-handler.class</name>
  <value>DefaultLCEResourcesHandler</value>
</property>
```

5.3.2 DefaultCgroup

The \textit{DefaultCgroup} configuration corresponds to how the NM is configured when running containers with Cgroups only for CPU. When this configuration is used, CPU isolation is enabled for containers which means that CgroupsLCEResourcesHandler initializes and writes to the Cgroup subsystem for CPU. See listing 5.2 for configuration.

Listing 5.2: Standard Cgroup configuration

```
<property>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>LinuxContainerExecutor</value>
</property>
```
5.4 Experimental Results

5.4.1 NM startup and registration with RM

Figure 5.1 shows the time in milliseconds for starting the Node Manager. The overhead is quite noticeable, as there is approximately 600-700 more milliseconds when running CgroupsGPU configuration compared to DefaultCgroup.

The overhead is not surprising since when the NM starts, it initializes the NvidiaManagementLibrary, which in turn loads NVIDIA drivers. Furthermore, several queries to the NvidiaManagementLibrary are needed to instantiate the GPUAllocator component and to report Resource capability to the RM through the NodeStatusUpdaterImpl. Although the overhead is noticeable, it is only run

5.3.3 CgroupsGPU

The CgroupsGPU configuration corresponds to how the NM is configured when running containers with Cgroups only for CPU. When this configuration is used, CPU and GPU isolation is enabled for containers which means that CgroupsLCEResourcesHandlerGPU initializes and writes to the Cgroup subsystem for CPU and devices, as described in the implementation. See listing 5.3 for configuration.

Listing 5.3: CPU and GPU Cgroup configuration

```
<property>
    <name>yarn.nodemanager.container-executor.class</name>
    <value>LinuxContainerExecutor</value>
</property>
<property>
    <name>yarn.nodemanager.linux-container-executor.resources-handler.class</name>
    <value>CgroupsLCEResourcesHandlerGPU</value>
</property>
<property>
    <name>yarn.nodemanager.resource.gpus.enabled</name>
    <value>true</value>
</property>
<property>
    <name>yarn.nodemanager.resource.gpus</name>
    <value>1</value>
</property>
```
once and subsequent interactions such as GPU allocation will not be costly since there is no need to interact with the NvidiaManagementLibrary, as all the state is located in-memory.

Figure 5.1: Time to start NM in relation to configuration

![Bar chart showing time to start Node Manager](chart.png)

**5.4.2 NM launching containers**

Figure 5.2 shows the time in milliseconds for executing the preExecute method, effectively also showing the additional overhead for allocating GPUs. It should be noted that the overhead is barely noticeable seeing as the scale is in milliseconds. Therefore the overhead can be considered negligible for enforcing GPU isolation for a container.
Figure 5.2: Execution time of preExecute method in relation to configuration
Chapter 6

Conclusions

The previously stated hypothesis is true, the YARN resource model can serve GPU under the current resource model, given the boundaries set by this work. The overhead of serving GPUs has so far been shown to be negligible, although all experiments have not been executed yet. At any rate it is important to put the overhead in perspective. Utilizing GPUs may lead to days, weeks or even months faster training and an overhead in the area of milliseconds is well worth it. To implement a general solution for using GPUs in the Capacity Scheduler

6.1 Future Work

6.1.1 Multiple queue support

The presented solution only supports one queue, the root queue. Future work could involve implementing support for multiple queues while making use of GPUs. Desirable properties of multiple queues could be to restrict GPUs so that they may only be allocated to a particular queue, for example only the queue corresponding to the data science division may make use of GPUs. Which is not possible under the current design of the Capacity Scheduler.

6.1.2 Look beyond NVIDIA GPU

NVIDIA is the biggest manufacturer for GPU and there are no indications this will change in the near future, even though google are making GPU which they have no made publically available hardware for yet. AMD have gotten traction with new Ryzen CPUs and may release their new line of GPU soon. Therefore consideration could be made to support or being to extend the abstraction layer and implementation new GPU manufacturer implementations.
6.1.3 GPU sharing policies

More GPU allocation policies could be implemented. Enabling time-slicing or GPU sharing may be beneficial for certain workloads and interest in the machine learning community should be examined to see which approach is more useful.

6.1.4 Fair Scheduler and Fifo Scheduler support

Support for GPU using the Fair Scheduler and Fifo Scheduler should be implemented, to satisfy more users needs for how they want to schedule GPUs.

6.1.5 Take advantage of NM GPU topology

A mechanism to co-schedule GPUs being scheduled on the same machine to be allocated in such that topological advantages can be exploited. For example if two GPUs are scheduled on the same NM for one container, then it should be allocated in such a way that the two GPUs selected for allocation should be connected with, for example, NVLink.
Bibliography


[26] “Github - google/leveldb: Leveledb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values,” https://github.com/google/leveldb, accessed: 2017-08-06.


[29] “Gpu support in mesos - google slides,” https://docs.google.com/presentation/d/1Y1IUIWV6g1HzD1wYIYXy6AmbfnczWfjvvmqqpeDFBic/edit#slide=id.p, accessed: 2017-08-19.


# Appendix A

## Illustrations

<table>
<thead>
<tr>
<th>Method</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize()</td>
<td>The libnvidia-ml.so library needs to be initialized to instantiate the internal state of the library. The initialized method implementation consists of a call to <code>nvmlInit()</code> that initializes the library.</td>
</tr>
<tr>
<td>shutDown()</td>
<td>When the libnvidia-ml.so has been initialized, and all the queries to the library have been performed, the library needs to be shutdown. The <code>shutDown</code> method does this by calling <code>nvmlShutdown()</code>.</td>
</tr>
<tr>
<td>getNumGPUs()</td>
<td>To query the number of GPU(s) present on the system, this method performs a call to <code>nvmlDeviceGetCount()</code>.</td>
</tr>
<tr>
<td>queryMandatoryDrivers()</td>
<td>Finding the device numbers for the drivers means finding out the device number for <code>dev/nvidiactl</code>, <code>dev/nvidia-uvm</code> and <code>dev/nvidia-uvm-tools</code>. This is done by doing a <code>stat</code> call and then extracting the minor and major device from the <code>st_rdev</code> field.</td>
</tr>
<tr>
<td>queryAvailableGPUs(int configuredGPUs)</td>
<td>This method queries the system to find out how many GPUs exist using <code>nvmlDeviceGetCount()</code>, and then proceeds to get a handle using <code>nvmlDeviceGetHandleByIndex()</code>, the handle can then be used with <code>nvmlDeviceGetMinorNumber</code> to find the minor device number. The <code>configuredGPUs</code> parameter is used to determine an upper limit for many device numbers should be returned. If three GPUs are detected, with device number 195:0, 195:1 and 195:2, and <code>configuredGPUs</code> is set to 2, only 195:0 and 195:1 will be used for scheduling.</td>
</tr>
</tbody>
</table>

Table A.1: NvidiaManagementLibrary implementation overview
Figure A.1: NodeStatusUpdaterImpl deciding number of GPUs to register with RM
Figure A.2: GPUAllocator initialization
Figure A.3: Initialization of Cgroup hierarchy
Figure A.4: Contents devices.list file in the hops-yarn hierarchy

Figure A.5: Cgroup devices controller and hops-yarn hierarchy
Figure A.6: Interaction for allocation and isolation of GPU
Figure A.7: Interactions for recovering GPU allocations for GPUAllocator
<table>
<thead>
<tr>
<th>Test Class</th>
<th>Tests</th>
<th>Test Results</th>
</tr>
</thead>
</table>
| TestGPUAllocator                               | testGPUAllocation  
testGPUAllocatorRecovery  
testExceedingGPUResource  
testZeroGPUAllocation              | Succeeded    |
| TestCgroupsLCEResourcesHandlerGPU             | testGPUAllocation  
testGPUAllocatorRecovery  
testExceedingGPUResource  
testZeroGPUAllocation              | Succeeded    |
| TestResourceManager                            | testResourceAllocation  | Succeeded    |
| TestResourceTrackerService                     | testNodeRegistrationSuccess  
testDistributedNodeRegistrationSuccess  
testNodeRegistrationWithMinimumAllocations | Succeeded    |
| TestResources                                  | testFitsIn  
testComponentwiseMin                                                                 | Succeeded    |
| TestCapacityScheduler                          | testCapacityScheduler                                               | Succeeded    |
| TestNodeStatusUpdater                          | testNMRegistration                                               | Succeeded    |
| TestLinuxResourceCalculatorPlugin             | testGetNodeGPUs                                                  | Succeeded    |

Table A.2: Overview of tests run