A FRAMEWORK FOR MANAGING
SHARED ACCELERATORS IN
HETEROGENEOUS ENVIRONMENTS

Eoghan O’Neill
Master of Engineering
25th August 2015

QUEEN’S UNIVERSITY BELFAST

A thesis submitted to the
Faculty of Engineering and Physical Sciences
School of Electronics, Electrical Engineering and Computer Science
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
Abstract

Heterogeneous processing technologies provide opportunities for new levels of flexibility and performance in applications. Current utilisation of such technologies has focused on HPC workloads and often couples applications to specific device targets through low level code. While this approach can produce optimal performance for specialised applications on a fixed hardware platform, it requires significant expertise to code and is not portable.

This work presents an approach that reduces development overheads for high-level application developers and provides a framework to allow applications to dynamically target heterogeneous devices.

A task based approach to application development is presented that decouples high-level code from device specific logic and also removes any need to manually manage devices from within applications. The SHEPARD framework facilitates this through a central repository of device specific implementations, and couples these with execution time estimates to effectively allocate tasks among shared heterogeneous processors.

The approach presented in this work is applied to various In-Memory Database scenarios including analytic workloads. Experiments show how a single task call can enable applications to execute code on multiple devices, and managing such tasks among a limited set of devices can allow multiple user workloads to share resources effectively, while improving workload performance.
Publications


Acknowledgements

Completing a PhD is a challenging experience that has provided me with many new skills and allowed me to learn and contribute to some of the most exciting technologies in the industry. This would not have been possible without the advice and support of the many people I have worked with throughout this process.

I wish to thank my supervisors, Peter Kilpatrick, Peter Milligan, Dimitrios Nikolopoulos and Paul Sage, for the valuable feedback and advice they have given me. In particular, I wish to thank Peter Kilpatrick for freely offering so much of his time over the course of this work. I also must thank both Peter Kilpatrick and Peter Milligan for the motivating discussions we had early in my PhD which helped give me the energy and drive to undertake this work.

I would also like to thank SAP for providing the time and resources that have allowed me to research with both cutting edge hardware and software. Special thanks goes to my colleague John McGlone, especially for all the work we have done in the area of acceleration and the HARNESS EU project. John is fond of pushing me to present and disseminate my work often, and despite my initial reluctance, this has provided me with valuable
and rewarding experience.

Finally, I must thank all my friends and family that have helped keep me sane and always encouraged me. My family Peter, Jacqueline, Kelley, Patrick, John and Sadie. Friends, Barry and Emma for blowing off steam at lunch; Matthew for his supreme comedy talent; Richard “Slick” Rea for gaming sessions that were the perfect escape and his wife Brittany for putting up with them and even cooking; and Joanne for her support and for keeping everything ticking at home even at the most stressful times.
# Contents

Abstract i

Publications ii

Acknowledgements iii

Contents v

## I Introduction & Background 1

### 1 Introduction 2

1.1 Dawn of Mainstream Parallel Computing 3
1.2 Evolution of Compute Technologies 4
1.3 Grid and Cloud 6
1.4 Challenges 8
1.5 Vision 11
1.6 Contributions 14
1.7 Thesis Outline 16

### 2 Background 19
2.1 Existing Approaches .................................. 21
  2.1.1 Languages ........................................ 22
  2.1.2 API .............................................. 24
  2.1.3 Libraries ......................................... 26
  2.1.4 Compilers & Code Transformation ............... 27
  2.1.5 Auto-Tuning ...................................... 28
  2.1.6 Run-time Frameworks ............................. 29
  2.1.7 Operating Systems ................................. 31
  2.1.8 Characterisation & Benchmarks .................. 32
  2.1.9 Scheduling ....................................... 35
  2.1.10 Virtualisation .................................. 37
  2.1.11 Cloud ......................................... 39
  2.1.12 Databases ...................................... 42
  2.1.13 Energy ......................................... 44
  2.1.14 Security ....................................... 45
  2.2 Summary ........................................... 46

II The SHEPARD Framework 49

3 Enabling Acceleration Through Managed Tasks 50
  3.1 High-Level Concept .................................. 50
    3.1.1 Managed Tasks .................................. 51
      3.1.1.1 Sourcing Implementations .................. 54
      3.1.1.2 Making Decisions ............................ 55
      3.1.1.3 Task Costs ................................ 56
4.1 Existing Limitations .............................................. 95
4.2 Managed Plug-Ins ................................................. 96
  4.2.1 Summary: Plug-In Tasks Vs OpenCL Composed Tasks 102
  4.2.2 Repository extensions ...................................... 104
4.3 Enhanced Framework Summary .................................. 105
4.4 Introducing the Management Runtime ........................... 108
  4.4.1 Need for a Management Runtime......................... 108
  4.4.2 Device Abstraction ....................................... 110
  4.4.3 Runtime Task Allocation .................................. 112
4.5 Experiments ...................................................... 117
  4.5.1 Hardware .................................................. 117
  4.5.2 Dataset .................................................... 118
  4.5.3 SHEPARD versus Native ................................... 120
  4.5.4 Available Implementations ................................. 122
  4.5.5 Scheduling Comparison .................................... 124
4.6 Conclusion ...................................................... 128
  4.6.1 Limitations ............................................... 129

III  Application of SHEPARD to the SAP HANA
IMDB  ................................................................. 131

5  Application of SHEPARD to In-Memory Database Analytics132
  5.1 Acceleration and IMDBs ........................................ 133
  5.2 Managed Analytic Queries on SAP HANA ...................... 136
    5.2.1 Analytics on HANA via AFL ............................. 136
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2 Application of SHEPARD within the HANA IMDB</td>
<td>138</td>
</tr>
<tr>
<td>5.2.3 SHEPARD Run-time Task Allocation</td>
<td>140</td>
</tr>
<tr>
<td>5.3 Context of Acceleration in AFL Queries</td>
<td>141</td>
</tr>
<tr>
<td>5.3.1 Studied Analytic Queries</td>
<td>142</td>
</tr>
<tr>
<td>5.3.1.1 K-Means</td>
<td>143</td>
</tr>
<tr>
<td>5.3.1.2 K-Nearest Neighbours (KNN)</td>
<td>143</td>
</tr>
<tr>
<td>5.3.1.3 Multiple Linear Regression</td>
<td>144</td>
</tr>
<tr>
<td>5.3.2 Datasets</td>
<td>145</td>
</tr>
<tr>
<td>5.3.3 Hardware Platform</td>
<td>145</td>
</tr>
<tr>
<td>5.4 Analytic Query Performance</td>
<td>146</td>
</tr>
<tr>
<td>5.5 Experiments</td>
<td>149</td>
</tr>
<tr>
<td>5.5.1 Baseline - SAP-H</td>
<td>150</td>
</tr>
<tr>
<td>5.5.2 Multi-User Experiments</td>
<td>151</td>
</tr>
<tr>
<td>5.5.3 10 Users - Large Datasets</td>
<td>153</td>
</tr>
<tr>
<td>5.5.4 5 Users - Mixed Queries</td>
<td>157</td>
</tr>
<tr>
<td>5.5.4.1 Query Performance Distribution</td>
<td>159</td>
</tr>
<tr>
<td>5.5.5 100 Users - Mixed Queries</td>
<td>163</td>
</tr>
<tr>
<td>5.5.5.1 Query Performance Distribution</td>
<td>166</td>
</tr>
<tr>
<td>5.6 Conclusions</td>
<td>170</td>
</tr>
</tbody>
</table>

**IV Conclusions & Future Work** 172

**6 Conclusions** 173

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Outcome Summary</td>
<td>174</td>
</tr>
<tr>
<td>6.2 Challenges Revisited</td>
<td>175</td>
</tr>
</tbody>
</table>
6.3 Discussion & Future Work . . . . . . . . . . . . . . . . . . . . 183

V Appendix 188

A OpenCL Wrapper Library Code Comparison 189

Bibliography 213
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Device Development Trade-offs</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Composing applications with tasks</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Application Creation</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>OpenCL Platform View</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>SHEPARD overview</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>SHEPARD Interaction Diagram</td>
<td>76</td>
</tr>
<tr>
<td>3.5</td>
<td>Dictionary Encoding of In-Memory Table Data</td>
<td>79</td>
</tr>
<tr>
<td>3.6</td>
<td>Delta Merge Operation</td>
<td>79</td>
</tr>
<tr>
<td>3.7</td>
<td>Column merge time per device</td>
<td>86</td>
</tr>
<tr>
<td>3.8</td>
<td>Execution time difference between native and managed execution</td>
<td>86</td>
</tr>
<tr>
<td>3.9</td>
<td>Delta Merge Table Execution Time per Task Allocation</td>
<td>87</td>
</tr>
<tr>
<td>3.10</td>
<td>Delta Merge Table Column Scaling</td>
<td>88</td>
</tr>
<tr>
<td>3.11</td>
<td>Delta Merge Table Column Mixes - Average Task Time</td>
<td>93</td>
</tr>
<tr>
<td>3.12</td>
<td>Delta Merge Table Column Mixes - Time to complete all tasks</td>
<td>93</td>
</tr>
<tr>
<td>4.1</td>
<td>SHEPARD Framework</td>
<td>107</td>
</tr>
<tr>
<td>4.2</td>
<td>Virtual Device Abstraction</td>
<td>111</td>
</tr>
</tbody>
</table>
4.3 SHEPARD Task Allocation Strategy . . . . . . . . . . . . . . 113
4.4 Managed Plug-in Interaction Diagram . . . . . . . . . . . . . 116
4.5 Comparison of Native Delta Merge Versus SHEPARD . . . . 121
4.6 Delta Merge Total Execution Time per Allocation Strategy . . 127

5.1 Managed analytic query process using SHEPARD and AFL . . 139
5.2 Native Query - Proportion of Time Spent in each Query Phase 147
5.3 Accelerated Query - Proportion of Time Spent in each Query
    Phase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
5.4 10 Users, Times Speed-up Versus Native Execution Time . . . 154
5.5 10 Users, SAP-H, Times Slower with Additional Workloads . . 156
5.6 5 Users, Times Speed-up Versus Native Execution Time . . . . 159
5.7 5 Users - Per Query Execution Time Versus Single User Time 159
5.8 Example Query Time-line Chart . . . . . . . . . . . . . . . . . . 161
5.9 5 Users - Breakdown of SHEPARD Managed Query Performance
    [top]Device Queue Lengths
    [bottom]Query Execution Time (Algorithm Phase) . . . . . . 163
5.10 5 Users - Breakdown of Native Query Performance . . . . . . 163
5.11 100 Users, Times Speed-up Versus Native Execution Time . . 165
5.12 100 Users - Per Query Execution Time Versus Single User Time
5.13 100 Users - Breakdown of SHEPARD Query Performance
    [top]Device Queue Lengths
    [bottom]Query Execution Time (Algorithm Phase) . . . . . . 169
5.14 100 Users - Breakdown of Native Query Performance . . . . . 171
List of Tables

3.1 Hardware Platform ........................................ 85
3.2 Experiment Scenarios ........................................ 88
4.1 Hardware Platform ........................................ 118
4.2 Sample Table .............................................. 119
4.3 Simulated Workloads ........................................ 121
4.4 Available Device Implementations .......................... 123
4.5 Static allocation of tasks ................................... 126
5.1 Algorithm Datasets (Rows x Columns) .................... 145
5.2 Experimental Hardware Platform ......................... 146
5.3 SAP-H - 5 Users, Times Slower with additional workloads 158
5.4 SAP-H - 100 Users, Times Slower with additional workloads 164
Part I

Introduction & Background
Chapter 1

Introduction

Heterogeneous computing provides opportunities for significant and sustainable performance improvements in many applications. New technologies, such as GPUs and FPGAs, open up new ways of creating applications and allow new sets of problems to be tackled efficiently [1]. However, while these new technologies afford many benefits, they also come with their own challenges which limit their uptake and hinder widespread adoption.

When targeting heterogeneous technologies, application developers are currently required to provide device specific code for each device they wish to use, and to develop a work placement or scheduling strategy at design time. The work of this thesis aims to address these challenges by decoupling device specific code from the main application and dynamically allocating work to available devices at run-time.
1.1 Dawn of Mainstream Parallel Computing

Before 2005, when multi-core processors became mainstream, most processors were single core and exhibited ever increasing clock frequencies with each new generation. As such, the majority of applications were logically written in a sequential manner. This afforded two main benefits for applications, the first being relative ease of implementation. The second benefit was the property of “free” speed up with each new generation of processor, as the performance of sequential code could scale well with processor frequency without having to recode the application.

Unfortunately, the scaling of processor clock frequencies had its limits, primarily due to physical barriers of power density required to maintain ever faster clock frequencies [2, 3]. This signalled the end of the “free” speed-up cycle that most sequential, legacy codes enjoyed. Intel’s co-founder Gordon Moore observed that “the number of transistors incorporated in a chip will approximately double every 24 months”. This observation became known as Moore’s Law and has helped drive processor advancement. Given the industry’s desire to chase Moore’s Law, but now lacking the ability to scale the performance of a single processor core, chip manufacturers turned instead to putting more cores on each die. This allowed applications to utilise multiple threads to execute their workload in parallel, meaning work could be completed faster. However, legacy codes are typically not written to take advantage of multiple cores as performance was derived mainly from clock speeds.

The result of the shift toward more cores meant that many applications
now had to be rewritten to take advantage of this new paradigm, and scale with cores instead of frequency. This is not an easy task, however, as multi-threaded applications are typically more complex to create, and many algorithms written to perform well sequentially do not easily map to parallel threads. Indeed, Amdahl’s law (1.1) describes the fact that, due to limitations in parts of an application being sequential, there is a limit to the speed up of adding extra cores.

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]  

(1.1)

where: \(S\) = achievable speed-up, \(N\) = number of cores, and \(P\) = portion of the application that can be executed in parallel.

As a result of this, the gains of many core, while still attractive, present a number of new challenges developers must overcome to achieve good performance for their applications. Given the large base of existing legacy codes, the industry as a whole has a significant challenge to overcome with regard to updating or rewriting existing code bases.

1.2 Evolution of Compute Technologies

In many systems, CPUs serve as the only processor available to applications. Thus, CPUs are not optimised to any one particular type of workload as they must be able to deal with all workloads in the system. For many applications this is sufficient; however, other applications, in particular those in the big data or HPC domains, require higher levels of performance that
1.2 Evolution of Compute Technologies

can only be achieved with more specialised processors. This has resulted in the emergence of other processing technologies, in particular accelerators that primarily exist as co-processors, typically connected via PCIe bus to the host CPU processor. Popular acceleration technologies include the GPU, FPGA and Many-Integrated-Core Architecture. Applications that utilise these accelerators typically offload some of their tasks to these processors with results being returned to the host CPU for further or final processing.

The Graphics Processing Unit (GPU), as the name suggests, was originally optimised for processing of 2D and 3D image data. This resulted in an architecture that exhibits high memory bandwidth and a large number of simple cores which gives excellent SIMD performance [4].

Field Programmable Gate Arrays (FPGAs) offer programmable hardware logic, allowing developers to create a specialised processor that fits exactly their algorithmic needs. This can result in high performing and energy efficient implementations [1]. However, they require very low level tools to program, leading to long and expensive development cycles.

The Many-Integrated-Core (MIC) architecture, used in Intel Xeon Phi accelerators, offers around 60+ x86 cores with large vector units. This allows them to give good SIMD and MIMD performance [5], thereby trading off raw compute power for flexibility.

Figure 1.1 presents a general overview of the trade-offs of employing existing technologies. While this figure is intended merely as a guideline, the general trend is that more specialised compute devices can offer improved performance at the cost of increased development cost, complexity and loss of portability.
Many of the aforementioned technologies come with proprietary tools and languages, and while these tools ease development on their respective platforms, they lock development into that platform, meaning applications need to be recoded to take advantage of other devices. In addition to this platform lock-in, the code is often optimised at a much lower level. Given that many of these technologies are evolving rapidly, the consequence of low level optimisation means that codes need to be constantly updated to maintain their high performance as architectures change.

1.3 Grid and Cloud

Beyond the processor technologies, the environment in which applications are run is changing.

Traditionally applications are run on on-premise hardware owned by the application user. In this scenario, with knowledge of popular hardware or applications, hardware could be procured to match the business needs. The
business would then own and host the servers and manage their maintenance and performance. Therefore in this case, the business can invest in the dedicated hardware required for the applications they need and remain fixed for a period of time.

For academic institutions, Grid computing has long been used to manage running applications on shared and often distributed hardware. Focusing on large batch jobs, particularly in the HPC domain, booking of time is usually employed to share resources between multiple users [6]. With Grid computing, the resources available on the platform are known to the developers who can then fine tune their applications to make best use of the available resources. Applications also have the benefit of running in isolation for their allotted time period.

The cloud, however, has disrupted the traditional hardware environment by extending the Grid approach to create hosted environments in which clients rent resources on which to run their applications [6]. There exist a number of large cloud providers including Amazon Web Services [7] and Microsoft Azure [8].

The major benefit of the cloud is its flexibility, allowing tenants to alter their resource provision to extend or reduce their costs or application performance. While this is clearly desirable from a cost-performance standpoint for the application user, it introduces a degree of challenge for applications as available resources can change each time the application runs. Introduce accelerators into this environment and the challenge becomes significant.

While the ability to use accelerators, such as GPUs, is present in some cloud environments, for example the G2 instances provided by Amazon [9],
it is typically limited to coupling a GPU to a single VM instance. Therefore a degree of flexibility is removed from the cloud environment, effectively creating a fixed hosted platform as there is a hardware dependency from the application. Clearly there is room for much more flexibility, but due to the overheads of developing for parallel technologies, applications are ill equipped to take advantage of a flexible pool of heterogeneous processors. Additionally, cloud providers often give performance guarantees in the form of Service Level Agreements (SLAs) which is possible through CPU resource provision, but not currently with accelerators as applications are unable to dynamically make use of them.

1.4 Challenges

From what has already been discussed it is clear to see the wealth of opportunities heterogeneous technologies provide to applications to maximise performance or energy efficiency. The challenge then, becomes one of enabling developers to allow their applications to take advantage of these opportunities.

1. Hardware Coupling and Code Dependency

When creating code for heterogeneous processors, such as accelerators, the developer often needs to choose a technology and decide which parts of the program to offload to the additional processors. This creates a fixed hardware dependency in the application, and assumes a certain performance characteristic of the target hardware, resulting in a static work allocation. This can lead to sub-optimal application performance and
1.4 Challenges

applications that are not portable.

Heterogeneous technologies are most commonly applied to improve application performance, and this has been the area most researched. As a consequence, the hardware dependency issue has not been adequately addressed in existing research.

2. Overhead of Coding for Heterogeneous Platforms

It is not feasible for developers to become experts in all processing technologies that are emerging. For example CUDA for NVIDIA GPUs [10] employs an SIMD approach, FPGAs typically employ pipelined data-flow models and multi-core CPUs use technologies such as OpenMP [11] or explicit POSIX threads.

Given that these technologies are constantly evolving, code may need re-optimised for each new generation. Therefore, the high-level application developer may find themselves required to learn how to create algorithms for every available processing technology.

One technology, OpenCL [12], has emerged to try to address this issue. However, OpenCL kernels are written at a lower level and their performance is not portable between different architectures, requiring multiple kernels to be written. Developers still need to decide in their code which device to place work onto. However, OpenCL is supported by many device vendors such as Intel, NVIDIA, AMD and Altera, [13, 14, 15, 16], which covers CPU, GPU, Xeon PHI and FPGA technologies. Therefore, OpenCL can provide a promising foundation for further work.

Existing work to reduce development overheads has focused on automatic
targeting of portions of an application to specific devices such as GPUs via code annotation or transformation, of loops in particular. This enables applications to send work to a specific device with little additional effort, but introduces a hardware dependency while also imposing a static work allocation strategy.

3. Diverse Nature of Performance on Devices
Another significant challenge is using these devices in an appropriate manner. Due to the diverse performance characteristics of different hardware architectures, applications need to be able to make sensible decisions when executing work such that they avoid creating bottlenecks by running work on processors which will perform poorly for a given workload. Additionally, applications need knowledge of the current workloads placed on accelerators in order to effectively load balance work among all available processors and avoid over provisioning a single resource while others idle. Much work has been done to characterise workloads in acceleration technologies, but it has been mostly focused on optimisation of HPC workloads, where a single application aims to maximise utilisation of all resources.

4. Shared Use of Accelerators
Typically, accelerators have been used in HPC environments where an application can have sole access to it. This will not be the case in many everyday environments and especially not in multi-tenant environments. Therefore, applications cannot assume sole ownership of accelerators and must manage their execution based on the availability of these resources.
1.4 Challenges

Some work has been done to investigate shared use of acceleration technologies through virtualisation, but this often lacks a global task management mechanism and requires additional management code from the developer to target the devices explicitly.

5. Application of Managed Acceleration to Enterprise Applications

Much work in the area of heterogeneous processing has focused on optimising a single application to maximise utilisation of dedicated resources, often consuming very large datasets or involving distributed resources.

Enterprise applications differ in that they often have to satisfy requests from multiple users, often in an ad-hoc fashion, as opposed to the batch nature of grids and HPC computing. Additionally, a number of enterprise applications may have to share resources as they operate within the same node or platform.

Therefore, integrating any management framework into an enterprise environment is a significant challenge that will be tackled in this work by targeting real-world use cases based on the SAP HANA In-Memory Database.

In summary, for developers, the challenge is to allow applications to be written at a high level, decoupled from low level knowledge of the hardware. For applications, the challenge is to allow them to make use of any heterogeneous processing hardware on the platform they are deployed to, and in shared environments, dynamically execute work on these resources depending on their suitability and availability.
1.5 Vision

This section will outline the vision of the thesis in the context of the challenges that have been identified. In essence, this work aims to create a managed environment that allows applications to target work to heterogeneous processors dynamically. To facilitate this, developers should compose their applications with tasks, whose implementations and device allocation is managed by a runtime which has knowledge of the task performance and the current workload already placed on the available devices.

![Figure 1.2: Composing applications with tasks](image)

Note that not all application workload will need to be targeted to heterogeneous processors. In many circumstances, applications will contain a few tasks or functions where most of the execution time is spent. Therefore, most of an application may remain unchanged while these compute intensive functions should be delegated to accelerated managed tasks, as illustrated in Figure 1.2. These functions should be managed separately to the main ap-
plication, allowing such tasks to be load balanced across available resources. Therefore, for most applications, much of the trivial and management work can be sufficiently accommodated on the CPU; while more compute intensive tasks will need to be executed using, for example, vectorised CPU, GPU or FPGA implementations.

To support this, two separate types of developer, the application developer and the expert developer are identified.

**Application Developer:** The application developer should only be concerned with the high level logic and not device specific code. In this sense, the application developer should be able to compose the application in the form of tasks that need to be completed. How these tasks are implemented on each device, or where they will execute at runtime should not be their concern.

**Expert Developer:** The expert developer is concerned with device specific implementations of tasks. The expert developer can focus on optimising code without the overhead of developing a full application.

Given these two development classifications there needs to be some mechanism to connect the high-level, task based logic from the application developer, to the device specific implementations of the expert developers.

This work, therefore, proposes a run-time manager that receives tasks from applications and decides on the application’s behalf, which implementation to use and on which device to place that task.

To facilitate this, the run-time manager needs knowledge of the devices and implementations available, as well as a mechanism to measure the per-
formance of an implementation on a supported device. This information will allow intelligent task placement decisions to be made and avoid any single device becoming a system bottleneck when multiple applications wish to execute tasks.

To this end, this work uses the concept of a task cost. A task cost is a measure of some aspect of its execution, whether that is execution time, energy consumption or monetary cost in the context of a cloud for instance. Thus, a task cost can be thought of as a means to inform allocation and scheduling decisions. In this work, the focus is on execution time and therefore, cost is measured in time.

Task costs can be evaluated at run-time using the parameters supplied by the application. Using this cost information can then enable intelligent placement decisions by estimating the performance of a task on the available devices and by factoring in the amount of work previously allocated to each device. Factoring in the work that is already allocated is done by recording the tasks and their associated costs when they are allocated to a device. Using costs to create these two pieces of information allows the device with the lowest overall task turnaround time to be chosen.

Given the presence of task implementations and costs, there needs to be a central location to source all of this information. Therefore, this work also proposes a platform repository that stores all the information needed for applications to call and execute tasks on a given platform. This repository should store information on the devices available, the task implementations that support these devices, and the task costs associated with such implementations.
1.6 Contributions

Using such a repository to store and make implementations available to applications introduces a great deal of flexibility when composing and deploying applications. Device specific implementations do not need to be developed in-house, they can be sourced externally, whether that is from a contractor, device vendor, cloud provider or 3rd party marketplace. This in turn creates a market for expert developers to sell their implementations to. With the degree of competition in academia for accelerating tasks, this approach could provide a link between the academic and business worlds.

1.6 Contributions

From the challenges identified in the area of heterogeneous computing and the vision outlined, the objective of this work is to create an approach to enable the use of heterogeneous processors in a manageable way.

With this objective, the contributions of this thesis relate to how programs are written and how work generated by applications is allocated among available heterogeneous processors.

The contributions are:

1. A task based programming approach to decouple high level application code from device specific implementations, allowing application developers to focus solely on application logic. This helps to decouple applications from devices by removing device specific code (challenge 1), as well as reducing development overhead on the application programmer (challenge 2).
2. A central platform repository that stores device information, device specific implementations and performance information for both devices and implementations. This concept provides a flexible way for applications to consume device specific tasks and for device developers to monetise their work. By storing device specific implementations separate to the high level application and allowing them to be loaded at run-time avoids the need to include device specific code and removes the need to manually manage devices (challenge 1). Storing this information separately also removes the need for the application developer to have low-level, device specific coding expertise (challenge 2).

3. A cost based approach to selectively execute task implementations on devices. Using the cost concept, different allocation goals can be set, such as lower execution time, minimal energy consumption, or reducing monetary cost of rented resources. This helps to address the diversity of hardware processing capabilities and allows applications to intelligently allocate work to devices (challenge 3). This also avoids burdening the developer with deriving a task allocation strategy in their high-level application code as the runtime takes responsibility for allocating tasks (challenge 2).

4. A framework that allows applications to dynamically call, load and execute task implementations at runtime by utilising both the repository and cost approaches. The framework uses costs and monitors device workloads to place tasks appropriately and accommodate multiple applications on a limited set of devices (challenge 4).
5. A study of managed accelerators in a real in-memory database. By applying this work to a real-world, multi-user environment, this thesis demonstrates the ability for heterogeneous processors to benefit large, complex application scenarios, rather than focusing on self-contained HPC applications.

1.7 Thesis Outline

In chapter 2 this thesis discusses some of the work that has already been done in this area and the current state of the art.

Chapter 3 introduces the framework, called SHEPARD, that is responsible for allowing applications to dynamically execute developer defined tasks that use OpenCL kernels. In this chapter the framework repository is detailed, with motivation for the need for the information contained and how it is used to facilitate task execution. This chapter describes how a compilation pass is used to insert information from the repository into the application and retrieve implementations without requiring the application developer to do so. Chapter 3 also discusses a costing approach for making intelligent decisions on task placement with respect to which device should be used to execute requested tasks. Experiments in this chapter focus on a real-world motivated use case to demonstrate the ability of the framework to enable an application developer to easily decompose a process into a set of tasks that can be transparently distributed among multiple heterogeneous devices.

Chapter 4 presents an extension based on a critical assessment of the initial work in chapter 3. In particular, this chapter focuses on two main
improvements: the creation of a runtime that facilitates allocation of tasks and decisions at run-time, and the creation of pre-compiled plug-ins that allow non-OpenCL tasks to be called from applications and more complex tasks to be called simply from user code. These tasks are loaded at run-time and can be updated externally to calling applications. The enhancements in this chapter allow implementations and cost information to be updated without having to recompile the calling applications. Experiments revisit the use case of chapter 3 and introduce an FPGA implementation.

Chapter 5 presents the application of this work to a multi-user in-memory database (IMDB) using real algorithms contained in an analytic library supplied with the database. Three algorithms are implemented for accelerators in this chapter to provide a range of workloads. This chapter focuses on the ability of the framework to manage devices in a multi-user environment where devices must be shared among competing processes. Experiments focus on assessing the ability of the framework to enable an IMDB to offload tasks from many users across a number of accelerators and thereby improve response times and reduce overall system load on the shared CPU resource. An OLAP\textsuperscript{1} benchmark is used to provide a baseline workload on the system and additional user requests are performed which make calls to perform analytic tasks. This chapter demonstrates the ability of the framework to execute tasks on all available devices, and by using the costing approach, balance workload effectively, resulting in faster response times for each user.

Chapter 6 closes the thesis with an assessment of the work presented and a discussion of further work that can be addressed in the future.

\textsuperscript{1}OLAP - On-Line Analytic Processing
Chapter 2

Background

Use of heterogeneous computing is expanding rapidly and is set to become the norm. Most major hardware vendors are committing to the heterogeneous view and are actively creating and evolving new types of hardware. Intel, known for its dominance in the CPU space, are embracing many core with the many integrated core (MIC) architecture[17], commercially available in Xeon Phi accelerator cards. NVIDIA have continually evolved the GPU, but now they too are moving to a more heterogeneous era via chip-sets with ARM and GPU cores, such as their Tegra offerings [18]. AMD too has developed what it terms an accelerated processing unit (APU), a single chip with CPU and GPU elements combined; a move also embraced by Intel since their Sandy/Ivy-bridge chips[19].

The need to exploit these technologies has never been greater, and pressure to do so will only increase over time. The expectation is that chips will emerge with a number of different available cores, each designed to meet a specific demand. Failure to exploit this will inevitably lead to sub-optimal
application performance, and wasted resources. As such, the problem of heterogeneity no longer applies only to the HPC market. The advances in architectures mentioned above give every indication that all modern platforms, from mobile devices to enterprise servers, will feature at the very least CPU and GPU elements. Therefore, every application will have to be applied to the heterogeneous space. Consequently, there is the need for a means by which application developers can easily exploit multiple processor architectures. Typically, application developers have exploited compiler optimisations; however the speed of change towards heterogeneous computing will quickly overwhelm what any single compiler can do. Thus, there is a need for developers to adequately express their applications in a parallel manner to expose opportunities for co-processing on many processors. Therefore, a system is required that can allow application developers to create code that can perform their required tasks, but can also orchestrate the application effectively over any available processing resources. This would allow an application developer to focus on the logic of the system, and not worry about low level architectures.

In addition, the emergence of the cloud as the future home of most of the worlds computation means that targeting a static hardware environment will soon not only be undesirable, but sub-optimal. The future cloud market sees a number of layers forming where multiple companies will compete. These layers are likely to be, IaaS, PaaS, SaaS [6]. Infrastructure as a Service (IaaS) enables multiple providers to compete to supply the hardware to perform the required processing in the cloud. This means that software will have available multiple sources of hardware and can pick and choose which
hardware resources can best meet their needs. Thus, where now it may be
desirable to exploit multiple types of architecture, in the future it will be a
requirement. In a mature competitive market, infrastructure providers will
be able to provide access to the latest advancements in hardware, including
CPU, FPGA and GPU technologies, among others. The ability to manage
these resources will be paramount to keeping any business competitive, and
its code base maintainable.

Although there has been much work on scheduling across cloud and grid
systems, this has tended to operate on a job based system and there has
been a focus on memory management across nodes. Cloud orchestration,
in particular, has seen work focused on virtual machine provisioning across
the landscape. While these are important challenges, there are new chal-
lenges unique to heterogeneous platforms that have not yet been addressed
adequately. The main challenge is that of orchestrating execution among
various processing resources with differing architectures, memory spaces and
processing capabilities. These challenges are not trivial, and it is vital that
they are addressed if the true potential of heterogeneous processing is to be
realised. When addressing these challenges, it is also important to avoid
placing significant additional burden on the application developers.

2.1 Existing Approaches

Though turbulent and continually evolving, the area of heterogeneous com-
puting has seen much research investment. The major hardware vendors
have all created tools to enable developers to avail of their products. The
most prominent examples are CUDA from NVIDIA[10] and the Parallel Studio from Intel[20]. These represent the most mature tools available in the marketplace and provide the developer with multiple tools to identify performance problems in their code and optimise effectively. The major problem with these tools is not their ability to aid the developer; indeed the compilers from both companies are able to aggressively optimise code effectively, but rather the requirement that the produced code target their platforms. In an environment as unstable as the current hardware market, being static should no longer be necessary, nor is it desirable.

2.1.1 Languages

There are a number of mature languages available for CPU based programming; many have also been continually extended over time such as C and FORTRAN. Not all of these languages were not created with parallel programming in mind and instead employ APIs or libraries such as POSIX or OpenMP to enable parallel programming.

The benefit of using and extending existing languages is that large code bases do not need to be “thrown away” but instead can be retrofitted with the use of libraries and extensions. There is also the added benefit of familiarity with the language resulting from large communities of developers already equipped with the necessary skills, which minimises training.

The major benefits of this approach are also the biggest flaws. The fact that developers can retrofit parallel changes means that the code is not written in a purely parallel way. This can lead to problems maintaining the code
as various locking and barrier mechanisms have to be added, which if not well documented leads to code that is easy to break and hard to debug. This also perpetuates the use of out-dated legacy code, which further inhibits future progression to more advanced technologies.

One language, Lime, discussed in work by Duback et al. [21], aims to introduce targeting of code to GPUs by extending the Java language. In this language, developers create tasks, which are an equivalent computational unit to an OpenCL kernel. The Lime compiler then generates optimised OpenCL kernels and Java byte code versions of these tasks and can allocate work to either the GPU or CPU. The results of this work show that the kernels created achieve comparative performance to hand-tuned OpenCL kernels and the applications studied achieve overall speed-up over native Java alone. However, the work assumes that the GPU is the superior device for dense computational kernels and will place the most demanding workloads there. Additionally, the compiler also assumes that the application has sole ownership of the GPU and no consideration is given to other workloads that may also compete to use the GPU. Despite these limitations, the goal of LIME is to enable application developers to operate using high level code and decouple application development from hardware specifics. Removing the burden of device management helps maintain application portability and reduces the developmental burden on the programmers.
2.1 Existing Approaches

2.1.2 API

Application Program Interfaces (APIs) provide a means to introduce parallelism to code in a controlled way. APIs such as OpenMP\cite{11} and OpenACC\cite{22} allow regions of code, especially loops, to be marked as parallel. The compiler then makes the necessary transformations to enable parallel execution of the code. Such APIs provide tremendous benefit as they allow parallel regions to be expressed quickly with minimal coding while providing good speed-ups on available hardware.

The main issue with such APIs is that the code base remains in a sequential form. The parallel nature of the code is not well expressed and therefore harder to break down onto multiple devices.

Current implementations of these technologies also create a static assignment of work to a single device. OpenMP typically creates a thread per core for the CPU, unless explicitly directed otherwise. OpenMP can also be used with the Intel Xeon Phi, as shown in a study by Cramer et al.\cite{23}, which comes with the added requirement for the developer to run the code natively on the accelerator, or in an offload mode where data is copied between the accelerator and host.

OpenACC, on the other-hand, aims to create parallel code to target an attached accelerator specifically, such as a GPU. Therefore, each time the code runs, it will always execute on the same device, regardless of other existing workloads or the actual performance of the target device. In reality, depending on device model and the current demand on the device, performance will change. Therefore, it is preferable to dynamically assign work to
such devices, rather than in a static manner.

Work by Li et al. [24] attempts to create an OpenMP implementation that can execute on either the CPU or GPU depending on the code region. While promising, they note that some regions cannot be translated to run on the GPU due to the way they are expressed in sequential code, such as recursion or use of the OpenMP work sharing directives; this highlights the major drawbacks of attempting to apply sequential codes to parallel technologies.

Additionally, different implementations of APIs such as OpenACC can yield different levels of performance for the same code, as evidenced in a study by Grillo et al.[25]. Some implementations may lack the ability to utilise all the features of a device, such as shared memory on GPUs, and therefore applications dependent on the API will not be able to leverage fully the power available.

OpenCL [12] attempts to standardise the co-processor development model in a similar manner to CUDA which targets GPUs. OpenCL abstracts devices into a consistent model using distinct memory spaces and executes work on these devices via compute kernels ordered using device queues. However, OpenCL on its own is quite low-level, requiring the developer to manage device discovery and determine where to place workload. Additionally, creating kernels often requires low level management of memory spaces and configuration of thread groups to create optimal performance. All these extra parameters place significant burden on the developer. Nevertheless, OpenCL supports numerous devices and architectures by popular vendors including NVIDIA, AMD, Intel, Altera and more. Therefore, OpenCL is a good candidate to base further work upon and has been used in many of the works
2.1 Existing Approaches

discussed in this chapter.

2.1.3 Libraries

Libraries provide a clean way to exploit parallel devices without exposing the developer to excessive amounts of resource management. One such example is the “Thrust”\[26\] library from NVIDIA, which aims to provide easy to use abstractions to perform tasks in parallel over arrays of data.

Libraries benefit the developer as most of the management is abstracted away leaving a simple means of producing parallel code.

The downside is that such abstractions force the developer into a certain way of coding. Where code is not easily expressed in terms of the library calls provided then it becomes difficult to use.

Other libraries have emerged that target specific domains, such as cuBLAS\[27\], MAGMA and PLASMA\[28\] libraries which target linear algebra. These libraries aim for optimal performance of the target algorithms on certain devices, such as GPU for cuBLAS or CPU for PLASMA, or seek to maximise utilisation of fixed CPU+GPU environments for MAGMA. The assumption present in these libraries, however, is that the application has sole ownership of the system resources and is free to fully occupy all devices. Once again, the libraries are tuned for fixed hardware configurations and cannot adapt to other hardware platforms.
2.1 Existing Approaches

2.1.4 Compilers & Code Transformation

Compilers offer many advantages to the developer by optimising code offline. Some compilers even allow extended pre-processor commands that result in auto-transforming serial code into parallel code. The compiler from PGI [29] allows the developer to mark parallel loops which are compiled to run on NVIDIA GPUs or multi-core CPUs. The GPU Ocelot[30] project from the Georgia Institute of Technology also allows NVIDIA PTX code to be transformed to run on multi-core CPUs. OpenMPC [31] takes the popular OpenMP codes and transforms them to target GPUs with just over a 10% performance penalty when compared to hand written CUDA.

Other work includes MCUDA [32] which takes CUDA code and generates parallel C code for CPU. The intent here is that CUDA code is written in a highly parallel manner and therefore, given this structure, the CPU can also benefit from the same parallel pattern. However, the introduction of OpenCL has superseded this work, as OpenCL allows for single kernel codes to operate on any supported devices. Consequently, the SWAN [33] and CU2CL [34] projects, aim to take CUDA code and convert it into portable OpenCL code. This is useful as much work done to optimise applications for NVIDIA GPUs can now take advantage of numerous OpenCL devices.

These auto-transforming projects allow free conversion and free parallel speed-up on existing codes. The drawback is that they are not optimal and they will not easily translate to new technologies without extensive reworking. This means that the developer either has to recode their application for the new hardware manually, or wait for support from the 3rd party compiler.
2.1 Existing Approaches

The LLVM project[35] provides a portable intermediate language representation for applications. The LLVM project enables a multitude of front-ends to be created that share a common backend. Using this common intermediate representation, multiple passes can be performed to transform or optimise the code before further compiling it to a native target or running it on a run-time interpreted environment. Work has shown that the LLVM compiler infrastructure, in combination with C/C++ front-ends, such as Clang, performs as well, if not better, than more established compilers like GCC[36]. LLVM has also gained much interest and contribution from large companies such as Intel and NVIDIA and, therefore, offers great potential.

2.1.5 Auto-Tuning

The parallel nature of new heterogeneous technologies means that there are a number of parameters that developers must choose when launching workloads on these devices. Technologies such as CUDA and OpenCL allow kernels to be launched with certain thread group properties which dictate how the work is split across the cores available. As a result, different launch parameters can yield different performance results on different architectures, or even different device models which employ variants of the same architecture. As a result, there is an added overhead on development whereby the best launch configurations must be derived.

To combat this, works by Spafford et al. [37] and Dokulil et al. [38] have explored auto-tuning OpenCL and the HyPHI library parameters, respec-
2.1 Existing Approaches

In [37], the authors investigate tuning OpenCL parameters on GPU kernel launches to different GPU models. They show how different devices can achieve best performance with different configurations, and by auto-tuning, the performance portability of OpenCL can be improved. Their focus is, however, on optimising application performance through auto-tuning rather than system wide orchestration. The authors of [38] auto-tune the HyPHI library, which takes Intel Intel Threading Building Blocks (TBB) algorithms and allows them to run on the Intel Xeon Phi accelerator. They tune the batches of work that co-execute on the accelerator. Depending on how much faster the accelerated implementation is than that of the host CPU, larger or smaller batches of work should be sent to the accelerator. Using black-box tuning with negligible overhead the authors demonstrate that tuning of such parameters on-line can quickly yield positive results with minimal overhead.

The work discussed shows promise that auto-tuning can be effective across a number of technologies including GPU and Intel Xeon PHI, freeing the developer from the burden of deriving the best configurations while not imposing significant overhead on the system.

2.1.6 Run-time Frameworks

There are many mature language frameworks that use common runtimes such as .Net which uses the Common Language Runtime (CLR) from Microsoft and Java and the JVM from Oracle. The use of these runtime layers allows multiple languages to be used that cater to different needs, yet still compile to
2.1 Existing Approaches

A single portable code. For example, F# allows for functional programming while C# caters for object oriented application development. Both these languages compile to the CLR and as such can employ the same optimisations despite differing language semantics.

Frameworks provide a level of abstraction to the developer which removes the complexity of low level optimisation. The framework then assumes the responsibility of optimising the execution of the application. In the heterogeneous space, some frameworks, such as Merge [39] and Mars [40], enforce a map-reduce structure and the developer must provide an implementation for each device they wish to use. In return, the framework can dynamically execute parts of an application on different devices. This idea is expanded by Rafique et al. [41] to continually score task performance to determine the best device to use. The Harmony[42] framework uses a similar knowledge-based approach without enforcing the map-reduce structure. Augonnet et al [43, 44] demonstrate how their framework, StarPU, can minimise, pre-fetch and overlap transfers to decrease overall execution time. In the Anthill[45, 46, 47] framework the developer describes applications as executable tasks (filters) and communication between them (streams). The framework then schedules the tasks and manages memory management. Published results state that this approach can achieve 78% scaling efficiency in multi-core environments.

The Qilin framework by Luk et al. [48] provides an array based API to the developer. The array based operations are then backed up by device specific implementations using CUDA and Intel TBB libraries to provide implementations for the GPU and CPU, respectively. Qilin shares work between the CPU and GPU dynamically by executing a portion of the task on each device
to estimate performance. While this approach is shown to work well for two devices, there is no indication of how this will scale, especially when more than just CPU and GPU resources are used. Additionally, this approach assumes that the application has sole access to all processing resources and does not accommodate multiple simultaneous workloads.

Existing works discussed in this section demonstrate that frameworks can allow developers to create applications quickly and still perform well in heterogeneous environments. However, much work has typically focused on CPU+GPU systems and many have used CUDA which limits the ability to leverage other types of devices. Additionally, for many of the frameworks, the developer also needs to provide the device specific implementations and there is no decoupling of the application programmer from device specific code. However, the existing works demonstrate that it is viable to offload management tasks from the developer and maintain good performance.

2.1.7 Operating Systems

Operating systems too must face the challenge of heterogeneous platforms. The primary research in this area comes from the Barrelfish OS project from ETH Zurich [49, 50]. The aim of this project is to build an OS from the ground up designed to run on all available devices. The OS works by having each device run its own driver kernel, capable of scheduling threads. Communication is then achieved via message passing between devices to maintain system state. The benefit of this is that each driver kernel will be optimised to schedule threads on its device and this concern is removed from the de-
developer. The developer must alter the code to work via message passing to achieve parallel execution. Applications within a heterogeneous OS would have better knowledge of available resources but must still be able to operate optimally by choosing to run on the most appropriate resources.

The MOSIX [51, 52] operating system provides transparent management for entire clusters. Using the Virtual OpenCL (VCL) layer that the operating system provides, applications can see all available OpenCL devices in the cluster without the need to employ explicit message passing such as MPI. This affords the application developer the ability to distribute work across many nodes, but still places responsibility on the developer to identify and orchestrate workload allocation. The developer is tasked with distributing and offloading work across accelerators in all cases, resulting in fixed workload allocation.

While promising, work on making operating systems operate across heterogeneous technologies still requires much more research before it will attract widespread and commercial use. However, the more heterogeneous technologies advance, the greater the need for OS management of such devices will become.

2.1.8 Characterisation & Benchmarks

Benchmarks and characterisation tools allow the limits of heterogeneous platforms to be identified and understood. Such limits can be viewed as bottlenecks at run-time or performance penalties resulting from accessing remote memory for example.
2.1 Existing Approaches

While there are a number of benchmarks available for traditional CPU based systems, for heterogeneous systems, such benchmarks are much less mature. The Scalable Heterogeneous Computing (SHOC) benchmark [53] offers different levels of assessment, including low level measurement of memory and PCIe bus bandwidth as well as maximum FLOPS per device. The benchmark also assesses simple operations, such as sorting, and more complex OpenCL kernels. The Parboil [54] and Rodinia [55, 56] benchmarks provide access to algorithms that cover popular application domains, attempting to cover a broad range of scenarios to give a full picture of device performance.

Characterisation allows for the generation of insightful feedback or predictive estimates. This is shown in work by Chen et al. [57] which matches cores based on their architectural properties to predicted workload resource demands, in particular using Instruction Level Parallelism (ILP), branch prediction and data locality.

Becchi et al. [58] schedule tasks to either a GPU or CPU using input size to estimate transfer and execution times. NUMA contention in a multi-GPU platform was characterised by Spafford et al. [59] demonstrating potential increases in device utilisation with multiple applications.

He et al. [60] investigate a number of scheduling approaches focusing on queue based load balancing. They demonstrate the value of using additional offline information to aid in JIT based execution on a heterogeneous system.

The Quincy [61] distributed scheduling system characterises the costs of data locality in a large distributed system and demonstrates the possible gains in characterising applications and pursuing an approach of minimum-cost over predicted congestion and execution times. The Quincy use case
is focused on data locality for distributed tasks and is able to reduce data transfers across a cluster by a factor 3.9 of in their experiments.

Albayrak et al. [62] profile kernels in terms of execution time and memory transfers to and from the device. Using this information they use a greedy algorithm to estimate the best device mapping for a set of kernels from an application. In their work they discount transfer times when the data already resides on the target device from a prior kernel. They compare their mapping results to a commercial mixed-integer programming technique, which is used to determine the optimal mapping, and conclude that the greedy algorithm is sufficient to achieve optimal mapping in most cases.

Che et al. [63] present a qualitative examination of development and performance of applications on CPU, GPU and FPGA technologies. In their study they note the various degrees of effort required to target each technology and the performance achieved by applications on each device. Their work reveals that each technology offers different advantages that makes heterogeneous computing compelling, but the diverse costs and efforts to target code to these technologies introduces significant development overheads.

Characterisation allows intelligent orchestration decisions to be made based on observed or derived system behaviours. It also allows bottleneck scenarios to be avoided where resource demands can be calculated or estimated. There is a strong case for characterising a system and its applications, even at run-time. For applications that wish to target heterogeneous platforms, some knowledge of the device and performance of the tasks they will run is necessary in order to adequately allocate workload in a sensible manner. The main concern is the overhead that can be introduced, which should
be minimised.

2.1.9 Scheduling

Heterogeneous processing often requires a scheduling strategy to achieve good performance on a set of tasks by placing work across available devices. For applications that use technologies such as CUDA and statically assign tasks to the GPU, for example, there is no opportunity to make use of idle resources, or alter allocation if the GPU becomes the bottleneck. Therefore, much research has been done to discover methods of placing work across different processing devices in heterogeneous platforms.

Wang et al. [64] create a speed-up based OpenCL kernel allocation strategy. They use a single queue, ordered by estimated kernel speed-up and input size, and push the smaller and lowest speed-up kernels to the CPU and highest to the GPU. To generate speed-up estimates, their system examines static kernel properties, such as number of instructions, and run-time parameters, such as global work size. Their results show good improvements on achieved turnaround times for a work queue however, the work is limited to OpenCL applications running on systems with a single GPU and CPU.

Grewe et al. [65] examine OpenCL kernel partitioning over a fixed CPU with integrated GPU package. They note that multiple OpenCL kernels can be sent to GPU at once, but only a single kernel can execute at a time, leading to large queuing delays when the GPU is under high contention. Using a benchmark on the fixed compute resources, they generate a partition model for OpenCL kernels based on static kernel features such as global memory
accesses and number of work items. They then measure average delay at run-time to account for device contention. Using these pieces of information, a kernel partition ratio is created.

Binotto et al. [66] use linear equations to describe task performance on GPU and CPU respectively and update these at run-time, maintaining a history or actual observed task execution times. Using this model combined with task history, they can improve applications by dynamically assigning batches of tasks among the CPU or GPU versus static assignment. An alternative approach by Joselli et al. [67] proposes continuous sampling of a portion of the work on CPU or GPU to determine which device should process the next batch. The authors rationale for this approach is to avoid situations where other processes start on the device and therefore degrade performance. By sampling regularly, the application can re-assign work at run-time.

Other work by Shelepov et al. [68] has embedded meta-data into program binaries that can be supplied to a run-time scheduler which can then make informed allocation decisions to available devices. The meta-data relates to architectural traits of the task that determine how sensitive its performance is to changes in clock frequency, for example. This allows the scheduler to place work across heterogeneous cores to maximise performance.

All of the works discussed emphasise the need for a dynamic scheduling strategy, showing that static assignment to a single device, such as a GPU, is often sub-optimal and can be bettered by executing work on all processors. This of course requires management and an appropriate scheduling strategy given that the best device to use may change depending on factors such as
2.1 Existing Approaches

input size.

2.1.10 Virtualisation

Emerging processing technologies such as FPGAs and GPUs do not natively support virtualisation, common in CPUs. As a result, many virtual machines cannot access attached PCIe accelerators, or at best, can use pass-through to tie a PCIe device to exactly one virtual machine instance. The outcome is that such devices cannot be shared between multiple virtual machines, limiting their use in hosted environments. Walters et al. [69] survey the performance of pass-through on popular hypervisors and show that close to native performance is achievable. The authors also note that using Linux containers, as opposed to virtual machines, achieves the best performance. Linux containers, in contrast to virtual machines, share a single kernel with the host and therefore can all access attached PCIe devices. This comes at the cost of security as there is then no management placed on these devices and each guest OS can run code directly on attached devices.

Some work has been done to address the lack of robust hardware supported virtualisation, particularly on GPUs. Common to many of these approaches is a mechanism whereby CUDA calls from the guest is communicated to the host which maintains the state of the GPU execution and sends results back to the guest VM process. Shi et al. [70] created vCUDA, which intercepts the CUDA API and requires a separated memory space to be created on the host OS to manage GPU states and requests. Each GPU is managed via a single thread, but multiple CUDA applications from differ-
ent VMs can make requests to the same device via its thread. In this case, multiple requests are serialised and kernels are executed one at a time. One issue is that memory reserved by an application is held until the application frees it. Therefore, with the limited memory available on GPUs, this can become an issue. This approach is also able to suspend and migrate VM instances by waiting until no kernels are executing on the GPU and saving the CUDA state on the host OS. The results of the vCUDA approach show good performance when compared to native execution. However, the solution is tied specifically to CUDA and the authors note that GPUs support a number of APIs, such as OpenCL and OpenGL. Therefore, virtualising all these technologies, especially as both the software and hardware are evolving rapidly, can quickly become infeasible. Other projects such as rCUDA [71] and gVirtuS [72] use a similar approach to allow VMs to access remote GPUs, presenting them as local to the application, although changes to the application code are required to support this. GVim [73] is another example that modifies Xen VMs with a custom kernel module inserted into the guest OS to enable transparent access for CUDA application to a host GPU. GVim employs a credit based system to load balance work between competing VMs.

Shadowfax by Merritt et al. [74] allows CUDA codes to be executed on local and remote GPU devices. Their work creates a virtual GPU (vGPU) which is attached to a physical GPU. CUDA calls are made against these vGPU devices and queued for execution. For remote execution, a server thread is created on the target node to receive and execute the CUDA calls. Shadowfax allows sharing of GPU devices by allowing multiple vGPU devices to be attached to a single physical GPU. While promising, this work requires
that the application is written using CUDA, and in a way that allows it to scale to many GPUs. Therefore, the application developer is still burdened with significant development overhead and a hardware dependency.

NVIDIA has moved towards limited hardware supported virtualisation in its GRID environment [75]. Currently, this is limited to two GPU models which allow a number of users to share a single card which contains a number of GPU packages.

Much of the work in the area of virtualisation is promising and paves the way for better accommodation of heterogeneous technologies such as GPUs in hosted cloud environments. Unfortunately, many approaches are restricted to CUDA, thereby limiting devices an application can use and still requiring the application developer to create task allocation strategies over available devices.

2.1.11 Cloud

The emergence of the cloud has created a number of opportunities within the industry. For cloud providers there are opportunities to share hardware platforms among many clients, typically via virtualisation technologies. This allows for high levels of hardware utilisation. Cloud providers also benefit from economies of scale, by building clouds of large numbers of commodity hardware, they can acquire resources at rates much lower than smaller businesses can for their on-premise servers.

For clients, they have the benefit of flexibility and scale, able to adjust the level of resources they use to their needs, without having to make capital
investments or maintain hardware. Clients can also benefit from SLAs \(^1\) that guarantee certain levels of performance.

One drawback of the current state of cloud computing is that applications assume a virtualised environment, whereby at most, the memory or CPU cores available may change. Therefore, there is the assumption that heterogeneity is limited in such environments. While some cloud providers do offer some heterogeneous processing opportunities [9, 76, 77], in the form of fixed GPU nodes, they are not flexible and intended for applications that specifically and statically target GPU devices, thus undermining the philosophy of the cloud.

Much research has been done to better accommodate the use of heterogeneous processing technologies within hosted clouds and distributed environments. Technologies such as rCUDA [71] and Virtual CL (VCL) [51, 78] attempt to allow developers to query for devices that exist on different nodes and execute code on these devices as if they were local. For example, Barak et al. [78] create a transparent layer for OpenCL and OpenMP that removes the complexity of having to use implementations such as MPI to achieve the same goal of using remote devices. To do this the system provides queuing, scheduling and scatter-gather services to the developer. Though the developer must still manually handle the orchestration of the devices, the ease of access helps reduce the overall complexity of programming the application.

These technologies, however, rely on the developer to manually control requesting multiple resources and allocating work to provisioned devices. As many applications may typically target work to a single device, they will

\(^1\) Service Level Agreements
not automatically scale if additional GPU resources are added. Instead, the developer must write their code in such a way as to accommodate such elasticity.

Work by Ravi et al. [79], extends the gVirtuS project to allow GPU sharing. Most GPUs do not allow kernels from different processes to co-run on the device, but do allow co-execution of kernels from the same process context. To address this, the authors run application kernels within a virtual shared context. Of course, this introduces security risks as kernels from different applications now share the same GPU context. Despite this limitation, which may be addressed in future hardware revisions, the authors demonstrate the ability to improve GPU throughput by judiciously co-locating kernels, such as pairing kernels that can both execute within the thread limit of the device.

Lee et al. [80] look at scheduling accelerator nodes only when required and allocating jobs which achieve the most speed-up to these nodes. The Rain infrastructure by Sengupta et al. [81] also looks at shared GPU allocation by intercepting CUDA device allocation calls and scheduling CUDA kernels based on least attained GPU service, whereby applications with lesser GPU usage are prioritised to attain fairness. Ravi et al. [82] also investigate alternative scheduling techniques of OpenCL jobs between CPU and GPU devices. In their work they describe a relative speed-up schedule that places jobs on either GPU or CPU queues based on relative performance. When a device queue is empty they either take a job from another device queue or wait, based on whether an aggressive or conservative policy is used. If CPU execution time is available, then they schedule the shortest jobs first. They demonstrate that these schedules can beat blind round-robin and approach
the performance of ideal schedules. The authors also look at multi-node scheduling, whereby an application can scale to use any number of GPU or CPU devices. The authors assume that applications have been written by the developer specifically to facilitate this. Their scheduling policy allows the provision of the number of nodes and types of resources to be altered, thereby improving throughput for the studied applications.

Clearly, the existing work has demonstrated not only the viability of using heterogeneous processing in the cloud space, but also that sharing of such devices can improve device utilisation while still improving performance of co-hosted applications. Additionally, while most work is either tied specifically to CUDA, or requires manual effort from the developer, it still highlights how elasticity in that cloud can allow applications to improve performance by dynamically making use of acceleration technologies when they are available.

2.1.12 Databases

Another area where heterogeneous computing has been studied is in databases. Databases allow a number of users to interact with large data stores, taking control of the management of user queries and scheduling and allocating work on shared hardware. Recent advances have seen the emergence of in-memory databases (IMDBs) which remove a major bottleneck, disk access. IMDBs, such as SAP HANA [83], aim to offer orders of magnitude faster query times, particularly for analytic queries. Therefore, databases now strive to offer users results in real-time on complex and predictive analysis of database data.
2.1 Existing Approaches

To facilitate this drive to improve performance, many authors have taken the inevitable step of introducing acceleration technologies into the database, and this has even seen commercially available GPU based solutions such as the massively parallel database MAPD [84].

Many studies have looked at implementing and optimising specific database workloads on devices such as GPUs and FPGAs. This has included GPU implementations of table sorting [85], SELECT queries [86], compression [87] and merge operations [88] which has yielded significant speed-up over their CPU counterparts. Sukhwani et al. [89] have also shown how FPGAs can not only improve the performance of analytic queries, but reduce the load on the host CPU by offloading such work.

Work by Breß et al. [90, 91, 92] looks at selectively executing database workloads on either CPU or GPU devices. The workloads studied include common queries such as sort and scan, and using performance estimation of user queries, the authors create a system which can dynamically allocate work between the CPU and GPU. Their system monitors actual performance of the queries on the chosen devices and can improve future decisions based on measured performance.

Other work has investigated JIT\(^1\) compilation of SQL queries into OpenCL for the GPU and vectorised code for the CPU [93]. The authors utilise a map-reduce style pattern referred to as “compute / accumulate” to execute partitions of a table query on CPU or GPU devices. Using this approach the authors can improve certain OLAP queries and demonstrate that OpenCL codes written for GPU can compete with CPU specific vectorised code.

\(^1\)Just in Time
Many of the approaches which seek to introduce heterogeneous processing into the database domain focus on optimisation of fixed hardware platforms and assume that queries have immediate access to available devices and do not worry about competing workloads. These assumptions mean that there is no accommodation of simultaneous workloads and do not allow for different hardware configurations that may include non-GPU technologies. Additionally, since databases have to deal with multiple users all issuing queries, the benefit of offloading work to accelerators is also an important aspect worth investigating.

2.1.13 Energy

One area where heterogeneous processing offers great potential is that of energy efficient computing. Different architectures specialise in different types of computation. FPGAs allow algorithms to be implemented in hardware, offering incredibly power efficient performance, at the cost of long development times and lack of generality. GPUs offer very good performance per watt ratios on numerical matrix and vector operations when compared to the CPU. Brodtkord et al. [1] compared a number of processing technologies in terms of gigaflops per watt, revealing that the CPU was less efficient than a GPU, while the FPGA was the most power efficient. While the energy efficiency of different models will vary, this gives a good general impression of the potential for improving energy efficiency of applications by judiciously employing heterogeneous processing where appropriate.

Some work has been done on investigating power measurement and fre-
quency scaling on GPU that has yielded promising results. GPUWattch by Leng et al. [94] showed how dynamic frequency scaling on GPU can result in power savings of over 14% at the cost of only 3% of performance. However their power models average an absolute error of 9.9% and 13.4% for a consumer and enterprise class model of GPU, respectively. While Burtscher et al. [95] study a high-end enterprise GPU and discover that the accuracy of the hardware sensor on the GPU is not sufficiently accurate, indicating that while GPUs can enable power savings within a system, the current tools available are not yet mature enough to fully understand where savings can be made.

2.1.14 Security

New processing technologies often take time to mature and gain many of the equivalent features of enterprise grade CPUs. For example, when GPUs were first being used for general purpose computation they did not feature error-correcting code (ECC) memory. Security is another important aspect, given that a number of processes may now be placing sensitive information on these devices, no longer just display data. Studies by Di Pietro et al [96] and Maurice et al. [97], revealed the that GPUs suffer from information leakage, whereby one process can access the data of another process on the same device. The authors were able to exploit this vulnerability in the GPU cards using standard CUDA calls in native and virtualised contexts. Such vulnerabilities show that GPU architectures and software may not yet be fully mature enough for shared usage involving sensitive data without some
form of strict management.

2.2 Summary

Heterogeneous computing is a vibrant area of research with many works covering a diverse range of challenges. The relatively recent explosion of the application of heterogeneous processing technologies coupled with their rate of evolution means that to leverage the power available in hardware platforms is a significant challenge.

Much of the work discussed in this chapter has shown promising results in the areas of optimising applications for heterogeneous environments. However, from this discussion a number of key challenges and shortcomings can be identified that still need to be addressed.

Hardware Dependency in Applications

Much of the work discussed has involved fixed hardware targets, particularly GPUs, and often using CUDA. Such a dependency means that applications are not portable to other devices. Another common thread is that many of the solutions assume that an accelerator is the best device to use for dense computation, as evidenced by many compilers and code transformers that target such code specifically to the GPU. In reality, the performance of devices will vary depending on a number of factors such as device model and task inputs, and so a static assignment to any particular device may not be appropriate in many cases.
2.2 Summary

**Burden on Developer to Provide Device Specific Implementations**

Though much work has been done to allow applications to execute tasks on different devices, particularly through frameworks, the developer is often required to provide device specific code, such as CUDA or OpenCL kernels, for the frameworks to use. This places a significant burden on the application developer which is accustomed to working at a much higher level. Instead, what is needed is a means by which application developers can gain access to such implementations that can be provided externally and used within the application, rather than requiring the application developer to write them. This leads to the requirement of an external repository of implementations that developers can call from their application.

**Allocating Work to Available Devices**

Some of the frameworks and virtualisation technologies discussed provide the ability for an application to make use of many heterogeneous devices, even remote devices. For many of these technologies, the onus is still on the developer to write their code in such a way that it can scale to these devices. In this case the assumption is often that the application should target a GPU, and the developer should write their code to scale to many GPUs if available. However, in modern systems the performance of different GPU models will vary, and there may be other processing devices such as Intel Xeon Phis present. Therefore, developers cannot be expected to code for every eventuality and possible device configuration. Instead, applications should be able to be composed of tasks that can be allocated to different devices in a dynamic manner. As a consequence, such allocation decisions should not be
created by the developer, but instead by a run-time management process.

**Sharing Devices among Applications**

Although useful, particularly in the HPC domain, the focus of much work to date has been on optimising a single task or application to maximise utilisation of heterogeneous resources. However, for interactive, business and cloud based applications, the likelihood is that no single application may claim ownership of processing resources. In reality, often many users interact with a system to perform tasks and these tasks should all be able to make use of the available resources. Consequently, a single application cannot determine the best allocation of work alone, as it has no knowledge of what other applications may be doing. Instead, what is required is a run-time management process that has knowledge of tasks from all applications, and therefore a full view of tasks to be performed, which can then determine the best allocation of tasks to devices.
Part II

The SHEPARD Framework
Chapter 3

Enabling Acceleration Through Managed Tasks

This chapter will elaborate on the vision presented in the introductory chapter with concrete details of implementation and motivation. Firstly, it will expand on and discuss the high-level concepts outlined in the introductory chapter. Next, the implementation steps will be presented, detailing how the various aspects of the approach are achieved. The chapter ends with a case study, results and evaluation of the current state of the approach.

3.1 High-Level Concept

Development for heterogeneous technologies comes with significant overheads. Developers must define where and when parts of the application should use available processors and often must use proprietary tools to do so. This can create a hardware dependency within an application, preventing it from
being portable. What is required is a method of allowing applications to remain portable, but still take advantage of extra processing technologies where present, without placing this burden on the developer.

To address this challenge, this thesis proposes that application development should be decoupled from the hardware by removing the need to target specific devices for accelerated workloads through the use of managed tasks.

3.1.1 Managed Tasks

In this work, managed tasks are defined as computationally intensive functions or portions of an application that would benefit from acceleration, whether that is from vectorisation on the CPU or offload to an attached device, such as a GPU. These tasks should be managed in the sense that the application should be allowed to simply call the task and have the answer returned, but should not care which implementation is used to compute the result, or which device is used to process the task. Therefore, the implication is that some mechanism exists that allows an implementation to be loaded and executed at run-time.

Managed tasks are particularly useful for common or popular algorithms or application functionality such as merge and sort operations or analytic processes such as K-Means which may be used in a number of applications. Separating this logic from the application and managing multiple implementations separately allows code to be reused.

This concept builds on the established idea of library software. Using libraries allows applications to share a single code base, thereby providing
additional functionality which developers do not need to re-implement in every application they write. Whereas library software must be called directly and often provides a single implementation, managed tasks allow for the provision of a number of device specific implementations from which the most suitable will be selected at run-time. Additionally, managed tasks provide intelligent load balancing of tasks among devices by dynamically deciding which implementation can best process a task given the performance of individual implementations.

This approach to creating applications imposes a minor change to the core development of an application. Most of the application can be created as it normally would, but where certain parts of the application are identified as candidates for acceleration, instead of manually producing device specific code, a managed task is called. This simplifies the development, and introduces flexibility into the application as it now has options on how it executes at run-time, rather than a static coupling to a fixed hardware set.

This decomposition of an application into managed tasks also facilitates the identification of two classes of developer, each charged with different aspects of application creation as depicted in Figure 3.1.
Application Developer

The application developer should be concerned with the high level creation of an application. This means that the application developer should be concerned with the flow of an application and which tasks it needs to perform to achieve its job. Therefore, the application developer should not be concerned with the specific hardware used to execute each task an application needs to complete.

For the most part, the application developer can create typical C++ code, and where available, make calls to accelerated implementations that are provided externally. This simplifies application creation, and also keeps high level code portable, and maintainable as there are not device specific codes or optimisations included.

Expert Developer

The presence of managed tasks necessitates the existence of device specific implementations that are made available to the application to perform that task.

Generally, accelerated implementations need to be created by an expert with knowledge of the hardware and associated tools. The difference in this case is that these task implementations are external to the application, meaning that they can be updated or added to after deployment of the application.
3.1 High-Level Concept

3.1.1.1 Sourcing Implementations

The presence of this managed task model affords a great deal of flexibility when creating an application. For example, an initial naive implementation can be provided, and later, can be optimised and updated. As the task is external to the application, the core application does not need to be recoded.

Of course, this ability for applications to call and dynamically load implementations means that not all code must be completed by an in-house expert developer. In fact, this approach to composing applications means that implementations can be provided from third-parties also. Thus, the concept of an expert developer can be split into a number of different sources.

**Expert** A developer employed or contracted to create an accelerated implementation for a specific device.

**Vendor** Device vendors often supply basic function implementations with their SDKs or provide optimised libraries for their devices.

**Cloud** Cloud providers can now have the option to licence or provide optimised implementations for the hardware present in their cloud as an incentive to encourage customers to use their services.

**Third Party Marketplace** Virtual market places that cater for specific devices or specific application domains could be created to supply implementations to applications.

When the concept of an “expert” is decomposed into these sources it becomes clear that there are a lot of opportunities to source external implementations.
3.1 High-Level Concept

3.1.1.2 Making Decisions

Having the ability to call accelerated tasks, wherever they may be sourced from, is only one part of the solution. Given a set of implementations and available processing resources, a decision must be made at run-time as to which device should be chosen to execute the task.

Given that the application will be unaware of which devices and implementations will be available on the platforms it is deployed to, it cannot assume any particular device preference for any given task. This may be the case in, for example, CUDA [10] development where the application will specifically target a GPU. However, even in this scenario, there are no guarantees on the capabilities of the GPU that is present on the target platform. Just like CPUs, different GPU models exhibit very different performance properties, thus a given assumption on task placement may be invalid if a particularly weak GPU is present.

It is clear that some knowledge of device performance is required when deciding where to place a managed task at run-time. This may lead to the conclusion that devices simply need to be benchmarked, with their performance measured in flops or some other metric. While this may work in many cases, it is not a guarantee of performance for all implementations.

As discussed previously, the architectures of heterogeneous devices vary greatly. As such, raw computing numbers do not give a true picture of performance for all algorithms. For example, GPU performance suffers greatly when there is a high degree of execution branching, or low levels of memory access coalescence [98]. As a result, for most tasks, the maximum perfor-
3.1 High-Level Concept

Performance on any given device will not be achievable and so performance mileage will vary between device types and even between different models of the same device type [99][56][4][100]. Additionally as more research is carried out to discover the strengths and weakness of these technologies, many of the performance assumptions are changing such as GPUs being tens to hundreds of times faster than CPUs as evidenced in a paper by Lee et al. [101].

Therefore, it is proposed that decisions should be made based on implementation performance for a particular device, rather than on general device performance.

3.1.1.3 Task Costs

In order to measure this performance, the concept of "Task Costs" is introduced here.

Task costs are a measure of some aspect of an implementation’s performance on which the applications running within a platform wish to optimise their execution. For example, applications may simply wish to execute their tasks on the device that will yield the result the quickest. However, there are other metrics by which applications may wish to measure their performance, such as power efficiency.

In this work, the metric used for costs is execution time. However, the concept of costs can also be applied to many metrics which are discussed here:

**Execution Time** A task cost is a measure of the expected time taken to complete. In this costing scenario the optimisation strategy is to minimise the time taken for each managed task.
3.1 High-Level Concept

**Throughput** For long running, or continuous tasks the goal may be to maximise overall throughput: in this case the task will have no determined finish point.

**Power Efficiency** Some platforms, such as large data-centres, aim to minimise power consumption where possible. Using managed tasks with power costs can enable a strategy of placing tasks based on the most power efficient devices and implementations.

**Monetisation** Many applications are now deployed to hosted environments, such as the cloud. In these environments resources are rented to the application user. Therefore, applications could attach a monetary value to executing tasks on certain devices and seek to operate within a budget or minimise expense. Cloud providers could also license device specific implementations which applications could factor in as a cost.

Using this task cost mechanism allows applications to alter behaviour by adhering to the different cost strategies.

3.1.1.4 Task Allocation

The presence of costs allows for intelligent decisions to be made for task placement. Costs also allow the current workload on a device to be estimated, and this is important when many tasks all wish to make use of devices.
Without an effective task allocation mechanism, the following pitfalls may cause problems:

**Over Provision**

If a single resource is allocated too many tasks it can become a bottleneck in the system. For a single application executing tasks in sequence, the developer may be able to derive a balanced task allocation strategy. However, when tasks execute concurrently, or multiple applications wish to use a device, static allocation can quickly lead to over provisioning of a single resource. In this scenario, the performance of tasks can degrade or the resources of a device can be exhausted, such as for GPUs which have limited memory sizes.

**Under Utilisation**

Static task allocation to a particular device, such as a GPU, can overlook opportunities to co-execute tasks on other devices. In such scenarios, all tasks of a particular type are sent to the same device, regardless of how busy that device is, and therefore other devices may idle when they could process a portion of the available workload. This issue is acknowledged in work by Serban et al. [102] which improves task execution time by allocating most work to the GPU and a portion of the work to the slower CPU which would normally be left idle.

**Device Starvation**

An additional problem that is often encountered, when using shared resources with applications that have no knowledge of other allocated tasks, is the issue of device starvation. If an application always self-
ishly sends tasks to a shared accelerator, other tasks will not have the opportunity to run, especially in the case of long running tasks.

The allocation issues presented highlight the need for concurrent tasks and applications to coordinate their execution on shared devices. However, this should not be the responsibility of the developer: rather, an intermediary is needed to orchestrate all managed tasks effectively. Using the cost approach, a management process can effectively estimate the performance of a task and how busy each device is, and place tasks accordingly.

3.1.2 Knowledge Repository

A knowledge repository, implemented as a database, is needed to store all the extra information applications need to execute managed tasks, such as device, implementation and cost data.

Having a central repository allows implementations to be shared between applications, and allows applications to determine which devices can execute the tasks they need, given the implementations available.

3.2 Implementation

This section describes the steps taken to implement the high-level concepts that have been outlined, including the major technologies used. Each component of the architectural vision is discussed, including how the various components come together to form a single managed framework for executing tasks on heterogeneous devices.
3.2 Implementation

3.2.1 OpenCL - Open Compute Language

To facilitate the creation of accelerated implementations, OpenCL (Open Computing Language) [12] is used as the foundation of the framework. OpenCL is an open standard that vendors can use to enable cross-platform, parallel programming of heterogeneous processors.

Figure 3.2 provides an overview of how OpenCL decomposes a hardware platform to provide a consistent view of all processing resources. As OpenCL is simply a standard, each device vendor must supply an implementation to allow OpenCL codes to run on their hardware. Since the architecture of various processing resources differs, the OpenCL standard defines a common programming approach to accessing and running code on these devices. The approach taken is similar to the offload approach made popular by CUDA, whereby memory is allocated on the device, data is transferred and a number of kernels are executed on a task queue to process the data. Once the kernels complete, data is read back from the device to host memory where it can be accessed by the rest of the application.
3.2 Implementation

The breakdown of the specific OpenCL hierarchy is outlined here.

**Platform** - The platform is the name given to the vendor implementation of OpenCL. Each vendor implementation provides access to its devices and therefore the state of each platform is distinct and separate.

**Device** - Each platform contains a list of its supported devices that can receive OpenCL kernels.

**Context** - A context is used to group one or more devices within a platform and allocate memory that can be managed between the devices.

**Command Queue** - A command queue is necessary to submit jobs to a device. The queue is typically processed in order, though some vendors support out of order execution on their devices.

To execute work on an OpenCL device the developer has to perform a number of steps in the application. Firstly, the platforms have to be discovered by querying the OpenCL runtime and then a device is selected from the available platforms. With a device chosen the developer then creates a context and creates and transfers the memory needed from the host to the device. Next the application needs to load and compile an OpenCL program, usually from file. Providing the program compiles successfully, the program object can be queried for the kernels that are required. Next a device queue is created and the kernels are placed on the queue to execute. Once kernels are complete, results can then be copied back to the host and the OpenCL objects destroyed.
While OpenCL is a step in the right direction and is a powerful tool for utilising heterogeneous processing technologies, it still requires much device management from the application programmer. This bloats the program, requiring a significant amount of management code that is not related to the work the program actually wants to do, and often resulting in static device allocation decisions being taken.

This thesis takes OpenCL as a good starting point for enabling access to heterogeneous technologies and builds a framework that aims to reduce these overheads to allow developers to easily write programs without the need to worry about device management. This work also enables programs to intelligently select devices to send work to based on availability of implementations, the performance of those implementations and the current demands already present on the devices available to the program.

3.2.2 SHEPARD: Scheduling on HEterogeneous Platforms using Application Resource Demands

To enable this managed task approach a framework, named SHEPARD, has been created. An overview of this framework is presented in Figure 3.3. The framework augments an application from source code compilation to run-time execution. The framework contains a number of components that facilitate the creation and management of managed tasks and this section will discuss how these components work to ease the burden on the application developer and enable applications to dynamically execute tasks on heterogeneous resources.
Figure 3.3: SHEPARD overview
3.2 Implementation

3.2.2.1 OpenCL Wrapper Library

One of the main concerns when developing for heterogeneous technologies is the overheads it imposes, particularly in writing code to target them. OpenCL provides a standardised way to execute code on supported devices; however there is a lot of verbose boiler-plate code associated with discovering the devices, creating the objects to access these devices, as well as compiling and managing the kernels that will run. This overhead results in a lot of extra code that is purely concerned with device management and not with the actual tasks that the application is designed to perform.

To reduce this overhead, a C++ library is created that wraps the native OpenCL library and aims to reduce the level of verbosity.

One convenience in particular is the ability to load OpenCL programs and kernels by name. Typically, the developer must load the OpenCL code from file, compile it and retrieve the kernels from the compiled program. In the wrapper library provided, the developer can retrieve OpenCL programs and kernels by name. The library will then load the required program from the repository, if available, or from a local file. Loaded OpenCL programs objects are then cached in memory for future use. If the program did not already exist in the repository, the library can automatically add it.

Other conveniences include the ability to automatically profile kernel executions and store the data into the repository. This data can then be used to create costs for the profiled kernels.

Listing A.1 in Appendix A presents a program which executes an OpenCL kernel using the provided OpenCL wrapper library and native OpenCL for
comparison. The listing shows a dramatic reduction in the number of lines of code required.

### 3.2.2.2 Constructing Managed Tasks

While the OpenCL wrapper library is designed to reduce the verbosity and implicitly handle many common tasks for the developer, it does not remove the need to retrieve and manage devices.

For this purpose Managed Tasks are used to allow the developer to call OpenCL functions, without the need to retrieve and manage platforms, contexts, devices or kernels. The goal is to allow the developer to simply create memory and call kernels to process these buffers with minimal overhead.

To accomplish this, any reference to explicit devices or context construction is eliminated; instead the developer uses place-holder OpenCL object references which are automatically replaced with concrete references at runtime.

Listing 3.1 shows the code a developer would write to create a managed task. In the example, the task takes an input array (`kernelData`) and performs two passes over the data, using two kernels (`kernel1`, `kernel2`).

The developer marks the function as a managed task by prefixing it with “task:”. This identifying prefix is then followed by a list of kernels that are used in the managed task separated by an underscore. The managed task also expects a number of inputs that are used to calculate the cost of the managed task at run-time. To facilitate this, the leading parameters are required to contain information on the work size to be presented to each kernel listed (`kernel1WorkSize`, `kernel2WorkSize`), in the order they are listed, and the
3.2 Implementation

Listing 3.1: Managed Task as Written by a Developer

```c
void task_<program>_<kernel1>_<kernel2> ( size_t kernel1WorkSize, size_t kernel2WorkSize, size_t readBytes, size_t writeBytes, float *kernelData, size_t kernelDataSize )
{
    oclMemory kernelBuffer = managedContext.createMemory(kernelDataSize, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, kernelData);
    setParameters(kernel1, kernelBuffer);
    setParameters(kernel2, kernelBuffer);
    managedQueue.enqueueNDRangeKernel(kernel1, 1, 0, kernel1WorkSize);
    managedQueue.enqueueNDRangeKernel(kernel2, 1, 0, kernel2WorkSize);
    managedQueue.enqueueReadBuffer(kernelBuffer, true, 0, kernelDataSize, kernelData);
}
```

total size of bytes to be written to and read from the device (\textit{readBytes}, \textit{writeBytes}).

A custom compilation step is introduced to insert the required logic to choose and retrieve the OpenCL device objects to use at runtime. The additional information provided in the managed task header allows this custom compilation step to retrieve the required kernels as well as the inputs required for the kernel and memory transfer cost functions.

After the custom compilation step is performed, the function becomes equivalent to the code in Listing 3.2. Notice that, in the transformed code, additional parameters are now passed to the managed task function and the place-holder OpenCL objects have been replaced with references to these new parameters (\textit{deviceContext}, \textit{deviceQueue}, \textit{deviceKernel1}, \textit{deviceKernel2}). The compilation pass inserts the code necessary to retrieve the OpenCL objects.
and updates all calls to the managed task to calculate the task costs, choose a device and pass the additional OpenCL parameters to the managed task.

### 3.2.2.3 Costs

Costs in this work are used to describe the expected execution time of a task. Since tasks can be manually constructed by the application developer using a number of kernels, the cost of a task needs to be calculated from the individual costs of memory transfers between the host and the device, as well as the cost of each kernel used. To facilitate this costs are modelled as linear expressions and stored in the repository.

For memory transfer costs, when SHEPARD is installed, it executes a benchmark that performs a number of reads and writes of various sized arrays...
to derive the required linear expressions. The input parameter provided for memory transfer costs is the size of the data in bytes. When constructing a managed task, the developer provides this information through the `readBytes` and `writeBytes` parameters.

For kernels, the linear expressions can be inserted into the repository directly, if known. If these expressions are not available, kernels will need to be profiled in advance and linear expressions derived from observations. It is expected that the responsibility of performing this profiling will fall on the expert developers or system administrators. However, application developers may supply example datasets that their application is likely to consume to improve any profiling. For convenience, the OpenCL wrapper library provided with this work allows such observations to be stored automatically after a kernel runs, if profiling is enabled by the developer. Linear expressions can then be derived from these recorded observations automatically.

The input parameter for a kernel cost model is the work size parameter that is provided when launching a kernel through OpenCL. This work size parameter informs the kernel of the size of data that it will execute over and is therefore an appropriate choice for this purpose. Once again, this information is provided by the developer using the `kernel1WorkSize` and `kernel2WorkSize` parameters, as shown in Listing 3.1. If a kernel is invoked multiple times the developer should multiply the work size parameters accordingly.

With costs for memory transfers and kernels, the total task cost becomes a summation of each of these individual costs.
3.2.2.4 Compiler Pass

Having provided a means to describe managed tasks, the compiler pass is responsible for inserting the additional logic to enable the application to utilise the managed tasks that have been created by the developer. The compiler pass creates a bridge between the platform and the application by inserting costs and implementations from the repository.

To achieve this, LLVM [35], a compiler project that implements its own intermediate language (LLVM IR), is used to compile the program in stages. CLANG++, an advanced compiler that is competitive with leading compilers such as gcc [36], is used to compile C++ code into LLVM IR. LLVM then allows individual compiler passes to be created which can enhance the IR code. This allows for an initial compilation to be performed, which can identify any coding errors by the programmer, before the managed tasks are enhanced by a specialised pass.

This section focuses on the steps carried out by the compiler pass to enhance the application with the objects and calls it needs to load and execute the managed implementations at run-time.

The first stage concerns initialisation of types and objects that need to be referenced within the application. These are primarily OpenCL types and arrays to hold implementation and cost references.

**Connect to Database** The first step of the pass is to connect to the repository of the target platform. This repository should contain all the devices and implementations that are available for the target system.
3.2 Implementation

Create Types and Arrays  Next the compiler pass will construct any types that are needed but not currently present in the application, such as OpenCL program objects which the developer does not have to explicitly reference. Data arrays are then constructed that will be used to hold all the information needed by managed tasks including: OpenCL platform, device, program and kernel objects, as well as cost functions for each implementation.

Identify Managed Tasks  The next stage of the pass is to identify all the managed tasks used in the application and record them so that they can be enhanced at a later point in the process. At this stage, when a managed task is discovered, information is parsed from its header relating to the kernels that the task wishes to use.

Create Hooks  The compiler pass then manually constructs library calls to the OpenCL wrapper to retrieve objects from the repository.

Update Application Entry Point  Finally, the pass adds any required initialisation logic for managed tasks to the application entry point.

The next stage of the compilation pass concerns updating the managed tasks themselves and all calls to those managed tasks.

Enhance Function  For each managed task, the compiler pass adds additional calls to retrieve the required OpenCL objects, such as devices, contexts and kernels. The pass then adds additional parameters to the managed task function to allow it to receive these objects. The
place-holder OpenCL objects used by the developer are then replaced with references to the new task parameters. This allows the managed tasks to now receive and use OpenCL objects that are retrieved at runtime. Finally, all references to the managed task within the program are updated to reflect the changes that have been made.

**Cost Function** The application also needs to be able to calculate the costs of executing each managed task on the devices available in the repository. To do this the pass uses the list of kernels that the managed task will call and retrieves the associated cost models for each implementation of that kernel, which are represented as linear expressions. Also retrieved are the memory read and write cost models. Functions are then created which evaluate these linear expressions and can be called at run-time. The input to these costs models is the work size parameter for kernels and the data size for memory transfers. This information has been provided by the developer when constructing the task. The total task cost is then derived as a summation of all the kernel and memory costs.

Creating these cost functions allows the cost of a task to be estimated using the inputs to a task at run-time.

**Update Uses** Each call to a managed task in the program must decide which device to use for that particular occurrence. To do this, a *CostEvaluator* object is created to decide which device to use. This CostEvaluator object, explained later in this chapter in section 3.2.2.6, has knowledge of the tasks already allocated to each device. It accepts the
costs of the current task which are calculated from the cost functions injected in the prior step and examines the current demand on the devices at run-time and returns a decision on which device to use. Based on this decision the appropriate OpenCL objects, i.e. kernel, context, queue and device objects for the chosen implementation, are passed to the managed task.

### 3.2.2.5 Repository

The repository is the central knowledge store for all managed devices and implementations on the target platform. Using this repository, the accelerated devices and task implementations can be decoupled from the application code and managed separately. By separating this information, devices can be added to the system and implementations can be updated or added, without requiring application developers to rewrite their code.

This section outlines what information is stored in the repository and how it is used for managed tasks.

**Device Catalogue**

This stores information on each OpenCL device available within the platform. Such information includes the device model and type as well as memory size. Memory transfer performance for each device is measured using the Scalable Heterogeneous Computing (SHOC) benchmark [103]. Read and write costs are then derived from the SHOC benchmarking and stored in the repository for each device.
Implementations

Managed tasks are composed of OpenCL kernels, and these kernels are pre-compiled and stored in the repository as binary large objects (BLOBs). This allows applications to load kernels directly without the need to compile them and avoids storing code within the repository.

Thus, kernels can be called by name within an application and different device specific kernel implementations can be associated with different devices. This allows optimised kernels to be used per device.

Cost Models

Cost models are stored as linear expressions. These describe the performance of a kernel or memory transfers for a particular device. The parameter passed to these models is the work size parameter for kernels, and the data size for memory transfers.

Storing expressions for each kernel implementation and device memory read and write allows managed tasks to be composed of many kernels and costs to be approximated for the entire task via the summation of the individual kernel costs and memory transfer costs. Additionally, for offload devices such as GPUs and Xeon PHIs, the performance of a kernel on the device should not significantly change from platform to platform, assuming the same device model. What may change is the performance of data reads and writes which is why they are stored separately in the repository.

If cost information is not available, the OpenCL library allows kernels to be profiled and the data stored automatically in the repository. This
data can then be used to derive linear expressions to represent the
kernel costs by the repository, as noted earlier.

3.2.2.6 Runtime

When an application starts it initialises the OpenCL components via the
custom wrapper library. This creates any context, device and queue objects
required to interact with the available processing resources.

When the application wishes to execute a managed task it will first per-
form the necessary cost calculation for available implementations. This in-
volves calculating the cost of each kernel used in the task and well as the
cost of reading memory from and writing memory to the device. This logic
has been inserted by a compilation step discussed earlier and is invisible to
the developer. Since managed tasks require kernel and memory buffer infor-
mation to be supplied as parameters, the information required for the cost
functions can be obtained at run-time. The kernel cost functions take the
kernel work size as input and the memory read and write costs take the asso-
ciated data size parameters as input. The summation of the memory transfer
costs and kernel costs forms the total cost of the task for a given device.

To make the ultimate decision on which device to use, a Cost Evaluator
object is created. This object accepts the task costs as inputs and will return
a decision on which device to use. The Cost Evaluator maintains a shared
memory data store which contains the total cost of tasks allocated to each
device. This data store is created if it does not already exist and is updated
each time a task is allocated to a device and allows the Cost Evaluator to
determine how much demand is currently placed on each device. Essentially,
3.3 Experiments

Each time a task is allocated to a device, its cost is added to the device’s total cost. By adding the current device cost and the cost of the current task to allocate, the cost evaluator determines which device will incur the lowest overall cost, and therefore, which device will return an answer the quickest. With the decision made, the cost of the chosen implementation will be added to the device store in shared memory.

Using this mechanism allows applications to concurrently run a number of tasks and have them be automatically load balanced among all devices with available implementations. This also allows applications to implicitly communicate and co-ordinate tasks according to the cost strategy.

With the device chosen, the application can execute the managed task. If the developer has enabled profiling on device queues, once each kernel finishes execution, kernel event data, which includes the time the kernel began and finished executing, is retrieved from OpenCL and logged to the repository automatically. This information can then be used to observe the performance of kernels and ultimately to update cost functions if necessary.

Figure 3.4 illustrates the interaction between the various SHEPARD components when executing a managed task.

3.3 Experiments

The experiments in this chapter will examine the ability of SHEPARD to successfully allocate tasks to available devices. Initial experiments will focus on SHEPARD’s ability to correctly allocate individual tasks to the fastest performing device based on the cost approach and examine if there is any
Figure 3.4: SHEPARD Interaction Diagram
3.3 Experiments

significant overhead imposed when doing so. Further experiments will examine what gains can be achieved when SHEPARD can utilise all available precessing approaches when compared to static allocation strategies.

Experimentation uses an application modelled on a real-world use case, the Delta Merge database maintenance operation that is present in the SAP HANA In-Memory Database (IMDB). The application consists of an operation that creates multiple tasks that can be executed concurrently.

Using SHEPARD to enable the application to utilise managed tasks, results demonstrate how automatic implementation selection and device allocation can improve the performance of an application without requiring large programming overheads from the developer.

3.3.1 Delta Merge

The application used to evaluate the SHEPARD approach is the Delta Merge. The delta merge is a table management operation within an In-Memory Database (IMDB) that seeks to optimise performance [83].

In-Memory Databases (IMDBs) exist today that allow the entire database to be stored in main memory, therefore avoiding the high penalties of disk access. Often, columnar storage is used in these cases which allows fast access to data of individual columns due to data being stored contiguously in memory. Using columnar data also creates the opportunity to compress the data as columns frequently have only a few unique values.

Compressing the data affords two major benefits for the database.

1. **Space Efficiency** - Compression can drastically reduce the amount of
3.3 Experiments

memory a data column uses and therefore allows larger databases to be stored in memory.

2. Efficiency of Processing - Operating on compressed data can also speed up the database as less memory needs to be read.

The method by which this column compression is achieved is dictionary encoding. Dictionary encoding works by counting the number of unique values in a column and assigning each unique value an ID. By counting the number of unique values, the minimal number of bits to describe all unique values can be calculated such that no space is wasted and values are tightly packed in memory.

Given that the values are now reduced to a unique ID, a mapping is needed to allow the real value to be retrieved for each ID. To accommodate this, the column is now fully represented using a column store of indexes, which maintains the ordering of the column for each table row, and a dictionary to map the indexes to their real values. These column and dictionary stores are referred to as the main column \( C_M \) and the main dictionary \( D_M \) respectively.

Using the minimal set of bits to describe the unique values and how they map to actual data means any changes to the table, such as addition of new values or deletion, require the dictionary encoding to be recalculated. Clearly this is not desirable for every insert or delete on a column as it would create very poor performance. To avoid this a second data structure is used to hold the differences between the main dictionary \( D_M \) and column \( C_M \) data and the current state of the column. This additional data structure is called the
3.3 Experiments

delta, and just like the main column data, contains two data structures, the \( D_D \) and the \( C_D \). The \( D_D \) is constructed as a cache sensitive B+ Tree which allows for fast insertion of new items which is vital for the transactional performance of the database. The \( C_D \) data is similarly stored as value indexes, just like the main column \( C_M \), except that these indexes are mapped using the B+ Tree, which has a slower lookup.

Given that the column now consists of two data structures, the main data store and the delta data store, every time the column is read, both of these data structures must be queried to derive the true state of the column. The structure of the table data is visualised in Figure 3.5.

Naturally, as the delta structure grows this combination of main and delta state will become more and more costly, resulting in performance degradation.

To minimise this performance degradation, the \( D_D, C_D \), is peri-
3.3 Experiments

Delta 1
Delta 2
Before Merge
Main 1
Main 2
Delta 1
Main 2
Delta 1
Main 2
Delta 2
Write Operations
Merge Operation
Write Operations
Read Operations
Read Operations
Write Operations
Main 1
Delta 1
Main 2
Main 2
Delta 2

Figure 3.6: Delta Merge Operation

Delta merge operation, depicted in Figure 3.6, can occur when the delta grows beyond a designated size, a certain time threshold has passed, or when the database has idle time. Therefore, this operation can occur intermittently throughout the execution of the database for any of the tables.

To perform the delta merge the following steps are executed:

1. **Create New Delta** - the column and table must be able to handle incoming queries while the merge operation is executing. To allow this a new delta is created at the start of the operation to record new transactions.

2. **Merge Main and Delta data structures** - This stage creates a new dictionary by merging the unique values in both the main dictionary $D_M$ and its delta $D_D$.

3. **Recode the column data** - Now that a new dictionary has been constructed, the ID values are no longer valid and must be updated to reflect the new position of the actual data items in the new dictionary.
3.3 Experiments

This phase takes each ID in the old column and replaces it with the ID of that data item in the new dictionary. The delta items are appended to the end of the column, denoting the order in which they were added to the column.

4. Replace old dictionary - Now that the new dictionary has been constructed, the old dictionary can be disposed of and replaced with the new dictionary.

For small tables, this operation is quick and imposes little overhead on the database. However, for large tables which can have hundreds of columns and in the order of millions to billions or rows, this operation can be significant. Therefore, accelerating this workload or offloading it from the main CPU altogether would be beneficial.

3.3.2 Evaluation

To evaluate the ability of the proposed solution to accelerate tasks in a managed way, the delta merge application must be modified to make use of managed tasks. Outlining the decomposition of this application into managed tasks gives an indication of the approach and effort required to utilising SHEPARD.

To begin, the delta merge must be performed for each column within a table to process. Therefore the application of a delta merge operation to a column can be described as a managed task. This results in a number of simultaneous tasks that are called when performing a delta merge on a table.

The delta merge is an operation with high data dependencies, whereby, in
the sequential algorithm, each output is dependent on exactly the previous output.

The sequential algorithm is described in Listing 3.3.

Listing 3.3: Delta Merge Algorithm

```c
void delta_merge(int* main_dict, int* main_index, int* delta_dict, int* delta_index, int* new_dict, int* new_index)
{
    int nIdx = 0;
    int mIdx = 0;
    int dIdx = 0;
    int ncIdx = 0;

    Map mMap; // map old main dictionary index to new
    Map dMap; // map old delta dictionary index to new

    // MERGE
    while (mIdx < main_dict.Size() && dIdx < delta_dict.Size())
    {
        if (main_dict[mIdx] < delta_dict[dIdx])
        {
            new_dict[nIdx] = main_dict[mIdx];
            mMap[main_idx] = nIdx++;
        } else if (main_dict[mIdx] > delta_dict[dIdx])
        {
            new_dict[nIdx] = delta_dict[dIdx++];
            dMap[delta_idx] = nIdx++;
        } else // (deltaValue == mainValue)
        {
            new_dict[nIdx] = main_dict[mIdx]
            mMap[main_idx] = nIdx;
            dMap[delta_idx] = nIdx++;
        }
    }

    // Process any remaining main_dict
    while (mainIndex < mainIndexCount)
    {
        new_dict[nIdx++] = main_dict[mIdx++];
        mMap[main_idx] = nIdx++;
    }

    // Process any remaining delta_dict
```
To decompose the delta merge into a managed task, it needs to be split into individual operations that can be implemented as kernels. From the description of the delta merge given previously, there are two clear computational stages, the dictionary merge and the column recode.

The merge kernel takes two dictionaries and produces a single unified dictionary. However, the merge needs some additional information before it can create a fully merged dictionary, specifically, the number of duplicates between the dictionaries. The sequential algorithm operates on each item in order, and thus can handle duplicates as it processes the merge. For a parallel merge, the data is partitioned and processed separately; as a result, duplicates in previous sections of the data will be unknown. Another issue is partitioning the dataset, as the partitions from the main and delta dictionaries must lie within the same range. To solve both these issues, two preprocessing steps are required, delta partitioning and duplicate discovery.

The result is five kernels that operate in sequence to create the delta merge managed task.
3.3 Experiments

**Partition Delta (PD)** The main dictionary is split into equal parts, the indexes of these parts are used to define the range of values in each data partition. Using these ranges, this kernel finds the corresponding delta dictionary indexes that contain values within each partition in the main dictionary. These indexes are stored in an output array which is then used in the merge step.

**Duplicate Discovery (DD)** Using the partitions defined in the PD step, this kernel finds and sums the number of duplicates in each data partition.

**Duplicate Scan (DS)** A generic parallel scan is performed over the duplicate array to create an output array that contains the total number of duplicates in each preceding partition.

**Merge (M)** Using the partition indexes derived in PD the merge kernel creates the new output dictionary. For each processed data partition, the output index of each new dictionary item is offset by the number of duplicates in preceding data partitions.

**Recode (R)** The final step is the recode of the old column indexes to the new indexes reflected in the new dictionary.

Each kernel used in the delta merge is generic enough to be reused or sourced externally. For instance, parallel scan, merge and map (recode) are all common operations for accelerators and many implementations are in existence.
3.3 Experiments

The delta merge operation is performed on all columns within a table. Therefore, there are two opportunities for parallelism: within each column merge, and by running more than one column merge concurrently. SHEPARD can facilitate both of these as it can allocate multiple managed tasks to different devices.

3.3.2.1 Delta Merge Costs

Since the delta merge task utilises five kernels in its operation these must all have associated costs, stored in the repository, so that they can be evaluated at runtime. The execution time of the kernels is not the only cost, so too is the time taken to transfer data from the host to the device \((THD)\) and results from the device to the host \((TDH)\).

Thus, the full cost of a delta merge, including data transfer, is described in Equation 3.3.2.1.

\[
Cost_{DM} = Cost_{THD} + Cost_{TDH} + Cost_{PD} + Cost_{DD} + Cost_{DS} + Cost_{M} + Cost_{R}
\]

(3.1)

3.3.3 Hardware Platform

The initial experimentation uses a common, consumer level, hardware configuration consisting of a single socket CPU system with a single GPU attached over PCIe 2.0. Further details are given in Table 3.1.
3.4 Results

### 3.4.1 Task Allocation

The simplest job that SHEPARD must do is to place a task on the device which can return the result the quickest. SHEPARD does this by calculating the task cost at runtime using the inputs provided to the task at run-time. Therefore the decisions SHEPARD makes are dependent on the quality of the cost models held in the repository. For these experiments, the kernels were profiled ahead of time using a number of datasets. Providing SHEPARD can accurately place individual tasks on the most appropriate device, the developer can have confidence that tasks can simply be called and not have to include their own additional scheduling logic in their applications.

A simple experiment for this is performed by scaling the input size of a delta merge task and observing where SHEPARD allocates the task. For the delta merge algorithm, there is a cross-over point where, for smaller columns, the CPU processes the result quicker, but for larger columns the GPU becomes the better device to use.

Figures 3.7a and 3.7b show the performance of column tasks for a selection of input sizes and compares the performance achieved from SHEPARD.

### Table 3.1: Hardware Platform

<table>
<thead>
<tr>
<th>Device</th>
<th>Model</th>
<th>Frequency</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i5-2500</td>
<td>3.50GHz</td>
<td>4</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Quadro 2000</td>
<td>1.25GHz</td>
<td>192 (4 x 48)</td>
</tr>
</tbody>
</table>
The results demonstrate, that by using the stored repository costs, SHEPARD allocates the tasks to the correct device for each input and causes no significant overhead.

Figure 3.8 shows the difference between execution times when OpenCL code is executed natively without SHEPARD, and then managed with SHEPARD making the allocation decisions. For the smallest columns there is minimal overhead introduced by SHEPARD. For larger columns the execution time difference is larger, mainly due to the greater variance of longer running tasks. This is clearly seen as SHEPARD managed execution is actually faster than native execution for the observed experimental runs on the largest dataset.

SHEPARD must also accommodate allocation of many tasks, not just single task instances. To observe this, multiple column merges are performed, and SHEPARD must allocate these tasks to either the CPU or GPU as they arrive. To compare how well SHEPARD’s allocation performs, static allocations of tasks are also observed:

- 100% CPU
- 100% GPU
- GPU 60%:40% CPU
- GPU 40%:60% CPU
- GPU 50%:50% CPU

Figure 3.9 compares the achieved total execution time for 10, 15 and 20 fixed size columns when allocated by SHEPARD or one of the static
3.4 Results

(a) Small columns

(b) Large columns

Figure 3.7: Column merge time per device
3.4 Results

![Execution Time Difference between Native and Managed Execution](image)

Figure 3.8: Execution time difference between native and managed execution

![Delta Merge Table Execution Time per Task Allocation](image)

Figure 3.9: Delta Merge Table Execution Time per Task Allocation

allocation strategies. The results show that SHEPARD achieves favourable performance, creating a sensible allocation that minimises execution time compared to static allocation.

3.4.2 Table Column Scaling

Delta merge tables can vary with respect to the number of columns they contain. Thus, the more columns a table has, the more delta merge tasks
3.4 Results

The results presented in Figure 3.10 show how additional speed-up can be gained from dynamically allocating workload to both the GPU and the CPU, as opposed to a single device statically. Given that SHEPARD takes care of the allocation at run-time, the developer has not had to worry about deriving an allocation schedule at design time. The application simply starts the task and SHEPARD takes care of its placement. Without the presence of SHEPARD, the application would not know the performance characteristics of the device implementations and therefore be ill equipped to make an adequate allocation.

Figure 3.10: Delta Merge Table Column Scaling
3.4 Results

3.4.3 Column Mixes

To further assess the ability of the SHEPARD framework to allocate work, a number of scenarios were created which include various mixes of columns. These column mixes, featuring dictionaries of various sizes, are intended to create differing performance characteristics between delta merge tasks. The column mixes are shown in Table 3.2 and the results from the column mix scenarios are presented in Figure 3.11.

<table>
<thead>
<tr>
<th>Column Mix 1</th>
<th>Column Mix 2</th>
<th>Column Mix 3</th>
<th>Column Mix 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 5,000,000</td>
<td>1 x 5,000,000</td>
<td>2 x 5,000,000</td>
<td>1 x 5,000,000</td>
</tr>
<tr>
<td>8 x 1,000,000</td>
<td>9 x 1,000,000</td>
<td>8 x 100,000</td>
<td>9 x 100,000</td>
</tr>
</tbody>
</table>

Table 3.2: Experiment Scenarios

From the results a clear improvement of average task performance can be seen. The rationale for this is clear: by sharing the workload dynamically, the contention on each processor is reduced.

For the total times presented in Figure 3.12, there is a slightly different outcome. Since tasks are co-located on the devices, there is the opportunity to use free cycles if each workload does not fully utilise the device processor. In the case of the delta merge, given its data dependencies and the random access required in the recode kernel in particular, there is ample scope to co-locate these tasks. Therefore, while using both devices improves the time to complete all column merges, there is one exception. For the scenario involving two columns of size 5 million and 8 of size one-hundred-thousand, the GPU is able to complete all tasks quicker. The reason behind this is
that there is the potential for greater variance on the CPU, thus when one of the large columns is scheduled to the CPU, though infrequent, it can take much longer than the GPU. The result is that while all of the other tasks finish quickly, the delta merge operation is left to wait for the final task to complete on the CPU.
3.4 Results

Figure 3.11: Delta Merge Table Column Mixes - Average Task Time

Figure 3.12: Delta Merge Table Column Mixes - Time to complete all tasks
3.5 Conclusion

The results presented in this chapter demonstrate that it is clearly advantageous for applications to leverage all available processing resources to speed-up task execution times. While it is obvious that using more processing resources will allow an application to improve its performance, the static single device allocation is indicative of the common offload paradigm when using accelerators. On the mid-range commodity hardware used in the experiments, the GPU outperforms the CPU for large columns; however this does not mean that there is no scope to use the CPU. As the experiments using the Delta Merge operation show, when multiple tasks must be performed, it is often advantageous for the application to make use of the slower CPU device when the GPU is already processing other tasks.

Additionally, while the results demonstrate the gains from using multiple processing resources, this is done without considerable overhead through SHEPARD. Without the presence of a system such as SHEPARD, the onus is on the developer to devise and code an allocation strategy for each application they write. This would also require explicit knowledge of the devices available on the target platform. Even if the developer were to devise their own allocation strategy, they would not have knowledge of tasks from other applications that may also be using processing resources, thus invalidating their allocation strategies. Using SHEPARD as a common framework for allocating tasks to devices means that there is shared knowledge of device demands between applications and individual tasks within an application. This shared knowledge, combined with SHEPARD’s decision making can al-
low concurrent tasks to better co-operate when using shared resources. This is demonstrated in the results whereby not all tasks will run on the faster device since they now have knowledge of other tasks that are already running and can, therefore, achieve faster processing response times by balancing the workload between the CPU and GPU.

3.5.1 Limitations

While the results presented in this chapter motivate the usefulness and the necessity for a system such as SHEPARD, they also point to some shortcomings in the current implementation.

The proposed system allows applications to take advantage of all available processing resources when implementations and costs are available, however this information is required at compile time. Although kernel execution performance can be logged to the repository and costs updated, the application must be recompiled to update its internal knowledge of these costs. The problem arises when kernels are updated or additional devices or implementations are made available.

While this compile time insertion of information means that there is less communication with the task repository at run-time, the overhead of re-compilation to update application information is potentially undesirable. There could also be the situation where applications are running with outdated information which could have a detrimental impact on task allocation decisions.

The work in this chapter is also limited to OpenCL devices. While
OpenCL is a powerful standard supported by many hardware vendors, it does not currently support all existing processing technologies, such as those provided by many FPGA vendors. While some FPGA vendors, such as Altera, do support OpenCL, in its current form, OpenCL is more suited to the vectorised SIMD offload approach for composing programs. Devices such as FPGAs tend to be more data flow oriented. Additionally, for many devices there may already be optimised libraries available and the ability to leverage these in the SHEPARD managed framework would be desirable.

Chapter 4 will address these issues by extending the SHEPARD framework to better accommodate more complex task scenarios and remove the compilation requirement to update task information.
This chapter expands upon the framework presented in the previous chapter and addresses many of the existing limitations in the work, such as static cost and implementation information being baked into applications.

First, this chapter will discuss the existing limitations of the approach implemented in Chapter 3. It will then introduce two main extensions to the work. The first extension is the introduction of managed plug-ins which allow more complex task implementations, which may require data pre-processing, to be fully abstracted from the view of the application developer. This new plug-in approach will allow the developer to call implementations simply by name only and impose no OpenCL management overhead. This abstraction also allows for non-OpenCL implementations to be supported through SHEPARD. The second is the addition of a management runtime that allows all static information previously required at compile time to be managed at
run-time. These two extensions serve to enhance the flexibility of the SHEPARD framework and allow task implementations to be called with greater simplicity by the application developer.

4.1 Existing Limitations

The previous chapter presented SHEPARD, a framework that allowed developers to create managed tasks composed of OpenCL kernels which are accessible from a central repository. A custom compilation step added cost information into the application directly, as well as the means to evaluate these costs and retrieve kernel implementations at run-time.

While this approach reduces the code to be written to access multiple heterogeneous devices, and removes the need to statically allocate work to devices, it introduces some hurdles in the process. Firstly, when costs, implementations or devices are updated, the application IR code needs to be recompiled. While certainly possible to accommodate, it adds an unnecessary overhead to the overall process. Additionally, developers still require some knowledge of OpenCL to create tasks. While this knowledge overhead is small, it is only acceptable for tasks that consist of just a few kernels or have a simple sequence of kernels. Some complex tasks may require data to be pre-processed in order to be executed on certain devices, for example, AOS\(^1\) to SOA\(^2\) transformations that are common for processing data in SIMD functions. Many tasks may also require a number of steps that may either not be obvious or require processing of the output of one kernel before it can

\(^1\)Array of Structures  
\(^2\)Structure of Arrays
be provided as input to another. In these cases, more overhead is placed on
the application developer to figure out how to compose their task.

Another limitation of the implementation presented in Chapter 3 is that
only OpenCL devices and implementations are supported. While composing
managed tasks through OpenCL kernels provides a great deal of flexibility,
it restricts the ability to use either legacy implementations, or non-OpenCL
devices or implementations. One prime example is in the case of FPGAs,
which are typically programmed using VHDL and application developers
are provided with a library call to allow them to use the FPGA in their
application.

In this scenario, the current SHEPARD framework is ill equipped to han-
dle FPGA tasks, meaning that the application developer has to resort to
statically assigning tasks to the FPGA.

Additionally, some tasks may not achieve good acceleration in all condi-
tions. For example, GPUs are optimised for floating point data, and have
limited support for string data types. In this case, the legacy implementa-
tion may still be required as not all scenarios are covered by the accelerated
implementations. There is a need for SHEPARD to cover such situations
by still providing access to these legacy implementations until such times as
accelerated implementations can be provided.

4.2 Managed Plug-Ins

To address the aforementioned issues from the viewpoint of the application
developer, the concept of Managed Plug-ins is introduced. Managed Plug-
ins represent fully abstracted libraries that can contain one or many tasks whose implementations are not visible to the application developer. Instead, tasks within these plug-ins are called simply by name, accept a number of parameters and return data back to the application.

An example of code which calls a task from within a managed plug-in is presented in Listing 4.1. A task object is created that takes the name of the plug-in to call as the first parameter, and the task name (i.e function) to invoke within that plug-in as the second parameter. The plug-ins are registered to the repository along with the devices they support and the tasks they contain.

In the example listing, the “Delta Merge” plug-in provides a library of delta merge tasks that accept different column data types. In the example, the developer wishes to invoke the delta merge on a double data typed column. Once the task object is created, the task parameters can be added. The task will then be invoked when “execute” is called on the task object, and this can be done either synchronously or asynchronously, with an optional callback provided to the execute function.

Listing 4.1: Plug-in Task as called by a Developer

```cpp
Shepard::ShepardTask task("DeltaMerge", "mergeDouble");

task.AddParameter(_mainDict, _mainDict->size());
task.AddParameter(_mainIndex, _mainIndex->size());
task.AddParameter(_deltaDict, _deltaDict->size());
task.AddParameter(_deltaIndex, _deltaIndex->size());
task.AddParameter(_deltaInvMap, _deltaInvMap->size());
task.AddParameter(_pValidDocIds, sizeof(_pValidDocIds));
task.execute();
```
Using this approach simplifies the code an application developer needs to write to execute tasks. Tasks within plug-ins are fixed from the viewpoint of the application developer. This is in contrast to the managed tasks in Chapter 3 which were dynamically composed using OpenCL kernels. However, using plug-ins allows the expert programmer to create more complex tasks that can be made available to applications and maintained separately.

In the example plug-in task, in Listing 4.1, the application developer simply passes the column data to a delta merge task. Upon invoking the task, the relevant task data is passed to a runtime that determines which implementation to load and when to execute, all invisible to the developer.

The plug-ins are stored as shared libraries which can be dynamically loaded at run-time. Using this approach allows for a wide range of implementations to be facilitated easily.

If tasks within a plug-in are implemented using OpenCL then all the OpenCL kernels registered to the SHEPARD repository are still accessible by these implementations, allowing plug-ins to also utilise any existing OpenCL kernel implementations. To accommodate the flexibility of OpenCL, when invoking tasks, the task function will receive the ID of the device it is using. For OpenCL implementations this ID can be used to retrieve the correct OpenCL context, device, queue and kernel objects through the OpenCL wrapper library. This means that a plug-in written using OpenCL can support a number of devices using a single code base.

Tasks within plug-ins differ in the way they are costed. The tasks composed using OpenCL kernels in Chapter 3 were costed by summation of the cost of each kernel and the cost of data transfers. Tasks within plug-ins
may not utilise OpenCL and may contain additional pre-processing or post-processing tasks, and so this cost approach cannot apply. As a consequence, individual tasks within plug-ins are costed as complete operations, requiring that they be profiled individually for each device they support.

The cost models for tasks within plug-ins are also stored as linear expressions. The input to these expressions is determined by the expert programmer. The input can simply be the size of a single parameter or it can be any mathematical expression that can be described using reverse polish notation. The repository allows these expressions to be stored for each task within a plug-in and evaluated at run-time. The derivation of these expressions is currently left as an exercise for the expert developer, however, future work may wish to implement an automated approach to creating cost models for plug-in tasks.

The main features of managed plug-ins are outlined here:

**Simplified Task Usage**

Composing tasks using OpenCL kernels requires the application developer to compose tasks manually, which includes creating device memory and copying data to and from the OpenCL context.

Managed plug-ins remove the necessity for the application developer to use OpenCL calls, as task implementations within plug-ins are fully abstracted as shared libraries that can be loaded at run-time. Instead, the application developer can simply call a task by name and provide the required parameters. In this case, the name is resolved by the SHEPARD framework, which then loads the appropriate plug-in which
contains the required task implementation.

Thus, using managed plug-ins removes almost all overhead from the application developer.

**Data Pre-Processing & Post-Processing**

For many accelerator technologies, the data must be presented to the device in a certain way to maximise performance. Typically, data should be contiguous in memory and aligned to the appropriate byte boundaries. Often, accelerators operate over individual properties of a data element; therefore, storing objects as a structure of arrays as opposed to the traditional array of structures, can improve memory performance. When composing managed tasks manually, the developer may still have to perform some additional data pre-processing steps to make their input data conform to the structure expected by the device specific implementation.

There may also be scenarios where the output of one OpenCL kernel needs to be processed before it can be used in subsequent kernels.

By wrapping tasks within a plug-in allows this step to be performed, invisible to the application developer. In this case the expert programmer can take steps to perform any additional input or output formatting, further reducing any overheads required from the application developer to access heterogeneous technologies.

**Allow access to tasks with more complex steps**

Providing implementations via plug-ins also allows for more complex
4.2 Managed Plug-Ins

tasks to be made available to the application developer as a single call. Where composing tasks using OpenCL kernels requires the application developer to generate the sequence of kernel calls required to compose a task, using a plug-in means this can be reduced to a single task call. Using plug-ins also allows complex algorithms to be hidden from the application developer, such as iteratively invoking a kernel over a dataset until a result converges or selective invocation of kernels based on input. The trade-off here is that since cost models are currently represented using linear expressions, more complex performance behaviours may not be adequately captured. The SHEPARD framework does allow custom expressions to be provided for processing task inputs, which are stored in reverse polish notation in the repository. This means that experts can define their own custom models if required, but this imposes greater administrative overhead.

Ease of maintenance

Wrapping a task’s implementation into a plug-in also means that it can be changed without affecting the applications that call it. Where adding additional steps to a managed task composed using OpenCL kernels would require the application developer to change their code, rewriting a plug-in can be done completely separately and simply updated in the repository.

Use of Non-OpenCL Implementations

Wrapping tasks in plug-ins also allows SHEPARD to allocate tasks that do not use OpenCL. Since tasks within plug-ins are costed as
complete operations, SHEPARD can maintain a consistent allocation strategy across all devices and implementations. It also means that plug-ins do not have to use OpenCL to compose their tasks. This means that devices such as FPGAs can be used alongside OpenCL implementations without requiring the application developer to distinguish between them.

Asynchronous Tasks

Another side effect of calling tasks via plug-ins is that they can easily be made asynchronous by allowing the developer to specify a callback function when they complete. For managed tasks composed using OpenCL kernels, execution is synchronous and the application developer needs to use additional threads to run these tasks asynchronously. Plug-in tasks are invoked via a library call, allowing additional convenience options, such as callbacks, which eliminate additional asynchronous coding overhead for the application developer.

4.2.1 Summary: Plug-In Tasks Vs OpenCL Composed Tasks

The trade-off between composing managed tasks from OpenCL kernels and using tasks via plug-ins comes in the form of flexibility and ease of use.

Composing managed tasks allows developers to combine kernels to execute as a single piece of work on a device to perform the tasks they need. This allows the reuse of the same kernels in a variety of tasks; however, the application developer must make sure that the data is in the format expected
for each kernel and derive the set of kernels needed to perform the required task. In addition, using more general kernels to compose a specific task may be slower than writing a specialised kernel that performs a specific task or operates on a particular set of inputs.

Therefore, depending on availability of kernels or plug-ins and the complexity of the task, the application developer may decide to use either a managed task composed from OpenCL kernels or a managed plug-in.

Additionally, while tasks composed using OpenCL kernels are costed by the kernels called and the data transfers, plug-in tasks are costed as complete tasks since they may or may not use OpenCL kernels. This means that plug-in tasks must be individually profiled, whereas tasks composed using OpenCL kernels can approximate the total task cost from the constituent kernel and memory transfer costs.

Finally, using plug-ins allows OpenCL and non-OpenCL based implementations to be used within SHEPARD and called in the same way from the application.

Having both of these approaches available to developers allows for a diverse range of accelerated implementations to be made available. Composing managed tasks from OpenCL kernels allows application developers to accelerate simple intensive sections of their code or to create more complex tasks from simple operations. The presence of managed plug-ins allows for more complex tasks, and tasks supporting non-OpenCL devices, to be provided and called simply within an application.
4.2 Managed Plug-Ins

4.2.2 Repository extensions

To facilitate the plug-in tasks, additional information is needed in the repository. The application developer needs to be able to interrogate the repository about which plug-ins are available and the tasks that they support. Additionally, the repository needs to allow the SHEPARD runtime to retrieve the plug-in implementations and the associated costs for the tasks contained within the plug-ins.

To implement this, the names of plug-ins are stored along with the tasks that these plug-ins provide. These names provide a consistent way for developers to call the tasks provided across device implementations. A separate table links these tasks to devices that support their implementation. By linking individual tasks within a plug-in to devices rather than simply the plug-ins themselves allows devices to support a subset of functionality contained within the plug-in. For instance, GPUs perform well on numerical data, but less so on strings. Therefore, the GPU plug-in is not required to provide a string based implementation but can still be used for other data types. This distinction allows SHEPARD to query the repository at run-time for devices that support a given task and prune the number of devices for which costs must be calculated.

Finally, the parameter names and types for each task are stored in the repository. Currently, this information is just for reference to allow developers to determine the inputs required for a given plug-in task. However, extending popular IDEs, such as eclipse or visual studio, could streamline the development process by providing repository information directly via
code completion prompts. This is noted as a possible extension for future work.

4.3 Enhanced Framework Summary

Figure 4.1 provides a summary view of the updated framework. A management runtime now exists as a separate entity from the applications and facilitates management of device tasks and provision of implementations. Instead of directly compiling information into the application, applications now communicate with the runtime to determine which implementations to use and when they are allowed to run.

The re-factored framework completely decouples implementations and task allocation decision logic from the applications. This allows devices, implementations and costs to be freely altered and updated at run-time, requiring no changes to the applications that call managed tasks.

The result of this new flexibility means that SHEPARD can be instructed to alter its allocation strategy should other costing mechanisms be implemented. For example, SHEPARD could be instructed to minimise execution time during periods of high demand, and minimise energy usage at other times.

Additionally, separating devices and runtime from the application also means that SHEPARD can add or remove devices from consideration when allocating tasks. This means that if a device needs to be maintained, such as being reset or having new firmware applied, SHEPARD can be instructed to stop allocating tasks to that device.
Finally, should new plug-ins be added or existing implementations be updated or enhanced, SHEPARD can be simply pick-up any changes via the repository and include the extra implementation options the next time a task is called. The same is true for updating costs.
4.3 Enhanced Framework Summary

**Figure 4.1: SHEPARD Framework**

Decoupled Costs and Decision Mechanism

- **Application Code (C++/OpenCL)**
- **Compile (Insert Logic for Managed Tasks)**
- **Receive Task & Task Parameters**
- **Implementation to Load**
- **Device Ready**
- **Update Cost Models**
- **Available Devices & Costs**
- **Feedback Actual Performance**

**Available Devices**
- CPU
- GPU
- PHI

**Execution Process**

1. Receive Task & Task Parameters
2. Implementation to Load
3. Device Ready
4. Update Cost Models
5. Available Devices & Costs
6. Feedback Actual Performance
4.4 Introducing the Management Runtime

As discussed previously, one of the main limitations of the work presented in Chapter 3 is that the compilation of managed tasks introduces static information, such as costs, that requires recompilation to update. This section discusses how the SHEPARD framework has been extended to fully decouple the application from the task repository through the introduction of a management runtime.

Additionally, this section also discusses how the framework architecture is updated to accommodate the plug-in tasks which now also support non-OpenCL devices.

4.4.1 Need for a Management Runtime

The existing approach presented in Chapter 3 required a custom compilation step to introduce cost and implementation information into the executable. This would allow the application to calculate the costs of its tasks and use these to determine which device to use and update a shared memory device cost structure which allowed all applications to query the current demand on each device.

The limitation of this approach is that, if costs are updated, due to further profiling for example, or implementations are added or modified, then the compilation pass would need to be performed again to update the relevant information in the application. This could lead to a scenario whereby applications have inconsistent or outdated cost information.

To remove this compilation-time dependency, a run-time management
process is created that is responsible for handling task requests from applications and informing the application as to which implementation it should load and when it can execute. The use of this runtime allows all device and cost information to be fully decoupled from the application and retrieved and updated at any time from the repository.

The runtime management component runs as a daemon process on the host platform. This daemon process is responsible for retrieving information from the repository for all the components in the framework and making the task allocation decisions at run-time.

When the management runtime starts it interrogates the repository for all registered devices and creates managed device instances for each device found. In turn, as each device object initialises it can perform any required start-up tasks such as retrieving device cost information and any driver loading that may be required.

The management runtime provides an interface through which devices can be interacted with, such as retrieving device task queues or instructing devices to update their cost functions from the repository.

By using a runtime to orchestrate task allocation and interact with devices, any need to recompile applications to update static implementation and cost information is removed. Since a daemon is used to create the runtime manager, it can receive administrative requests and be instructed to update its knowledge from the repository at run-time and instruct managed devices to do the same.
4.4.2 Device Abstraction

Since the plug-in mechanism supports non-OpenCL implementations, a common device abstraction is needed to effectively manage all the devices that SHEPARD has authority over.

In the implementation discussed in Chapter 3, devices were represented simply using a shared memory construct to describe and quantify the demands of tasks allocated to each device. This provided a shared view of devices to all managed tasks and allowed allocation of such tasks to be load balanced across devices.

A more robust implementation is provided in this chapter, where each device exists as an in-memory object that can be queried and updated at run-time. These device abstractions are referred to as Virtual Devices in this work.

Each virtual device object implements a fixed interface allowing tasks to be posted to, and information queried from, each device in a consistent manner, irrespective of actual implementation.

Each virtual device implements a task queue which is used to schedule the execution of managed tasks on that device. This queue based scheme allows all devices to be accommodated in the cost based allocation scheme used by SHEPARD. As each task posted to a device has an associated cost, the total cost of all tasks on the queue of a device provides an estimate for when that device can begin processing any further requests. Using this abstraction for all OpenCL and non-OpenCL devices, such as FPGAs, allows the SHEPARD framework to manage all devices in a consistent manner. Figure 4.2 provides
In addition to task queues, each virtual device selectively loads and stores, in memory, the costs associated with the implementations supported on that device. This allows for quick access to costs without having to query the repository database each time a task needs to be allocated. Having each device represented in this manner also allows individual devices to update their information either periodically or when requested at run-time. As such, when cost data is updated in the repository, the device can retrieve the new costs without having to alter any of the applications that call managed tasks. Additionally, having each device represented in this way allows them to be added or removed from task allocation consideration at run-time.

For managing devices, abstracting each device into a separate run-time object allows custom management steps to be applied to each device. For
instance, the FPGA used in experiments discussed later in this chapter re-
quires a driver to be loaded and the device reset before it can receive tasks.
Additionally, different vendors often supply their own device control and
monitoring APIs, such as the NVIDIA Management Library (NML). Ab-
stracting devices behind a common interface allows device specific interaction
and monitoring logic to be created and accessed externally in a consistent
manner between devices.

Additionally, SHEPARD also allows the definition of a legacy CPU de-
vice. While OpenCL implementations view all the CPU resources as a single
device and aim to utilise the entire CPU resource, legacy implementations
will tend to be single threaded, or use a limited number of threads. The
legacy CPU device can be defined in the SHEPARD repository with a spec-
ified number of cores. The legacy CPU virtual device will then create a
thread pool of the specified number of cores. This allows SHEPARD to al-
locate legacy tasks to a fixed pool of threads. For example, the delta merge
operation is a single threaded C++ algorithm and is usually afforded two
threads by the database, thereby allowing two column merges to be per-
formed simultaneously. SHEPARD can accommodate this by defining two
legacy CPU devices that each have a single thread. Now SHEPARD can
allocate tasks to those threads using the same queue based model it employs
for other tasks.
4.4.3 Runtime Task Allocation

Using the device abstraction and the management runtime allows task allocation measures to be changed at run-time. This can be done without requiring the runtime, devices or applications to be restarted. As tasks are called from applications the management runtime determines which implementations exist and which devices are available to service them. By using the repository to add or remove available devices or implementations, the run-time allocation of tasks can be influenced. For example, if an important application needs sole access to a given device at a certain time, the access to this device can be removed through SHEPARD. Any subsequent managed task requests will not be allocated to that device. This flexibility allows platforms to be effectively managed by an administrator without the necessity to restart applications, allowing for resource sets to be dynamically allocated to applications. Therefore, the administrator only needs knowledge of SQL and the relevant data tables in the SHEPARD repository. However, given that the runtime operates as a daemon and can receive messages, it is envisioned that an administrate interface could be provided in future to simply such operations.

Figure 4.3 provides an overview of the cost based allocation strategy used for tasks. The main steps involved are:

Post Task to runtime

When an application wishes to execute a task it first needs to communicate to the management runtime details of the task it wishes to execute and the inputs it will use.
Retrieve Implementations

The runtime manager then queries the repository to determine which implementations exist for that task, and therefore which devices can be used.

Evaluate Costs

With the set of available implementations retrieved, the management runtime evaluates the cost of each implementation using the input details passed to it by the calling application.

Factor in Device Demand

Next the management runtime queries each device that can execute the task to determine the current demand on each device. A device determines the current demand through the summation of the costs of all tasks currently waiting in its queue. Since costs currently measure execution time, the demand of a currently running task is the total expected execution time minus the time it has already been executing.
4.4 Introducing the Management Runtime

Adjusting the cost of the currently running task to reflect how long it has already been running allows for a more accurate device demand to be created and prevents long running tasks from skewing this estimate.

**Post Task to Device**

With both the task cost and device demand the management runtime will decide which device should receive the task request based on which device can complete the task soonest.

At this point the management runtime will return to the calling application task the implementation to load and execute. The application will then load the implementation.

When the device is ready to process the task, having completed any other tasks in its queue, it will signal the management runtime. The management runtime will then tell the application to execute its task using the implementation that was provided. The task will then execute and once finished, a completion signal is sent to the management runtime which can then update the associated device. If the task was called synchronously, control will then return to the main application. If called asynchronously, the callback registered, when the task was created, will be invoked.

Additionally, when a task completes, if its expected execution time is significantly different from the actual execution time, then this could be used as a trigger to perform further profiling or update cost models associated with that task. This is currently suggested as future work.

Figure 4.4 illustrates the updated interaction between the various SHEPARD components when executing a managed task.
4.4 Introducing the Management Runtime

Figure 4.4: Managed Plug-in Interaction Diagram
4.5 Experiments

To evaluate the expanded functionality of SHEPARD further experiments are performed using the delta merge use case from chapter 3.

The experiments show that SHEPARD is not only able to allocate tasks without imposing significant overhead on applications, even while the system is under significant load, but also able to make task allocations that result in overall performance similar to a static allocation based on prior knowledge of available devices and the performance of available implementations.

The experiments in this chapter expand on the those presented in chapter 3 by introducing hardware and datasets more representative of actual enterprise specification.

4.5.1 Hardware

The hardware utilised now includes enterprise grade Intel Xeon CPUs and an enterprise class NVIDIA K-Series GPU, as well as the addition of an FPGA. The specification of the hardware platform used for these experiments is outlined in Table 4.1.

The GPU and FPGA cards are both connected to the host platform via a third generation PCI express bus. However, the FPGA can only transfer data at second generation PCI express speeds and can utilise only four of the 16 available lanes.

The new device abstraction allows SHEPARD to decompose the CPU resource into multiple thread pool resources to accommodate legacy or single threaded implementations. The CPU based delta merge operation is single
4.5 Experiments

<table>
<thead>
<tr>
<th>Device</th>
<th>Model</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon CPU E5-2697 2 - Sockets</td>
<td>24 cores (48 Threads) 2.70 GHz, 256 GB</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA K20Xm</td>
<td>2688 cores 732 MHz, 6 GB</td>
</tr>
<tr>
<td>FPGA</td>
<td>Alpha Data ADM-XRC-6T1</td>
<td>Xilinx Virtex 6 FPGA, PCIe Gen2 x4</td>
</tr>
</tbody>
</table>

Table 4.1: Hardware Platform

threaded, and typically two threads are allocated by the system, allowing two operations to be performed simultaneously. Therefore, SHEPARD creates two legacy CPU devices, each supporting a single thread. This allows SHEPARD to allocate single threaded tasks and not breach the thread limit imposed by the system. These legacy CPU devices can be added or removed at run-time, allowing more or fewer threads to be allocated to delta merge tasks.

4.5.2 Dataset

The datasets used for this round of experiments have also been updated to better reflect the sizes and types of common data table columns. A study by Krueger et al. [104] analysed the customer data of 12 companies which contained 74,000 data tables per installation. Also studying the delta merge algorithm, the authors characterised the properties of the 144 most intensively used tables. The characteristics of these tables are as follows:
4.5 Experiments

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Rows</td>
<td>75.5 Million</td>
</tr>
<tr>
<td>Number of Columns</td>
<td>72</td>
</tr>
<tr>
<td>Number of Integer Columns</td>
<td>25%</td>
</tr>
<tr>
<td>Number of Long Integer Columns</td>
<td>25%</td>
</tr>
<tr>
<td>Number of Double Columns</td>
<td>25%</td>
</tr>
<tr>
<td>Number of String Columns</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table 4.2: Sample Table

1. The number of rows per table ranged from 10 million to 1.6 billion

2. The average number of rows per table was 70 million

3. The number of columns per table ranged from 2 to 399

4. The average number of columns per table was 70

5. Breakdown of unique values per column:
   (a) 1 - 32 : 60%
   (b) 32 - 1023 : 10%
   (c) 1024 - 100 million : 30%

Using this study as a guide, a data table summarising these traits is generated and outlined in Table 4.2
4.5 Experiments

4.5.3 SHEPARD versus Native

The first evaluation of the fully decoupled, run-time driven approach concerns how much overhead this process imposes on the overall execution of the tasks. To assess this, a base load is placed on the system to simulate server utilisation of varying degrees. SHEPARD must be able to receive task requests and answer them in a timely fashion, without imposing additional overhead on the system. The base load is simulated using the Stressful Application Test (stressapptest) tool [105]. This tool simulates server workload by spawning a number of threads, tasked with memory reads and writes, as well as disk I/O and compute intense operations. This allows for a mix of work to be simulated, similar to that present in a real database.

Within this environment, a delta merge operation is performed on the sample data table. The execution time of this operation is recorded during various levels of server utilisation. These levels of utilisation are outlined in Table 4.3.

These workload levels are intended to simulate existing utilisation on the server with the “high load” level fully utilising the compute resource of the server. Under these conditions, SHEPARD should be able to respond to task requests effectively and not introduce additional overhead on the application.

The delta merge is performed on the sample table using a standalone application that has the delta merge operation hard coded. The delta merge is then repeated where the application calls a SHEPARD managed task that loads the same delta merge implementation via a plug-in task. The measured execution times are presented in Figure 4.5.
### 4.5 Experiments

<table>
<thead>
<tr>
<th>Workload Level</th>
<th>#Write intensive Memory I/O Threads</th>
<th>#Compute Intensive Ops Threads</th>
<th>#Disk I/O Threads</th>
<th>RAM Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Low</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Medium</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Medium/High</td>
<td>15</td>
<td>15</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>High</td>
<td>20</td>
<td>20</td>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>

Note: Server has 48 hardware threads available

Table 4.3: Simulated Workloads

Due to variance of execution times, there is no discernible overhead from using plug-in tasks through the SHEPARD framework. From the delta merge tasks performed there are observed instances where the SHEPARD task finishes before that of the natively called delta merge. Therefore, for tasks of this type, any overhead on execution time that SHEPARD may introduce is minimal and well within the natural execution time variance of the delta merge operation. Even at high utilisation server utilisation, the SHEPARD framework is able to receive the task and retrieve and execute the implementation with negligible overhead.
4.5 Experiments

4.5.4 Available Implementations

The delta merge operation must be performed for every column within a table. As each column can be of a different data type, multiple implementations are required to accommodate this. For CPU implementations this is trivial; however for other processors additional work may be required. GPUs for instance are specialised for numerical, in particular floating point, data. String support is not as robust for GPUs and often delivers poor results. FPGAs on the other hand must have designs for each function they need to implement. There is a high overhead in not only generating these designs, but also in testing these on the actual FPGA hardware.

As a consequence of the different device characteristics, not all data types are supported by the implementations provided for each device. The CPU, being the most versatile processor supports all data types.

The GPU is specialised for numerical data and does not have a string implementation available. The FPGA is a specialised processor and due
4.5 Experiments

to the overheads of development time and testing, only a single data type implementation, integer, is available.

The data type implementations available for each device are summarised in Table 4.4.

This subset of supported implementations poses a challenge in traditional development, whereby the application developer will need to create rules to determine which subset of devices can be used for each data type. This complicates the application logic whose purpose is simply to execute a delta merge. Using SHEPARD and its repository to associate implementations with devices, this decision logic is removed from the application altogether. This means that any combination or subset of implementation support can be implemented over a range of devices, and so long as each data type is implemented for at least one device, SHEPARD can determine which device to run the delta merge operation on.

This frees the developer from the need to ensure that all implementations that could be possibly required for a device are available before the application is deployed. In the case of an FPGA in particular, waiting for all data type delta merges to be implemented and tested could impose a serious

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>Long Int</th>
<th>Double</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GPU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Available Device Implementations
delay in deployment of the application. In contrast, through SHEPARD, additional device implementations can be added at a later date, while not requiring any changes within the deployed application. For example, should a long int or double FPGA implementation be finalised at a later point in time, it can be registered to the SHEPARD repository and the next time an application requests a delta merge task of that type it can now make use of the additional implementations.

4.5.5 Scheduling Comparison

When allocating tasks at run-time, SHEPARD uses its repository to identify available device implementations and determines which device can return the result the quickest by evaluating the cost of the task combined with the current demand generated from existing tasks already allocated to each device. This allows SHEPARD to easily accommodate a range of devices and a subset of implementations across those devices without any extra overhead being placed on the application developer.

In contrast, in the absence of SHEPARD, the application will need dedicated logic to determine which devices have supported implementations for the required task, and of those, which is best to use. This will necessitate knowledge of available implementations at design time. Therefore, the application will likely need to implement a fixed allocation strategy.

Thus, to compare the ability of SHEPARD to effectively manage tasks, a number of other task allocations are also investigated.
4.5 Experiments

2 CPU Threads - This is the normal resource allocation for delta merge tasks. The intent behind this resource allocation is to prevent delta merge operations from using too many resources and negatively impacting the system’s ability to answer other queries. This scenario provides an estimate for baseline performance of the delta merge.

4 CPU Threads - When only CPU resources are available, typically only 2 threads are used by the delta merge operation. The FPGA and GPU provide two extra resources that SHEPARD can use. Therefore SHEPARD will have access to 4 resources in total, the 2 CPU threads plus the GPU and FPGA. In this scenario, 4 CPU threads are allocated for reference to compare the performance of 4 fixed CPU resources versus heterogeneous resources.

Round Robin - In this scenario SHEPARD evenly distributes tasks between the CPU, GPU and FPGA devices, without considering task costs. The greater the number of available allocation options, the more likely round-robin is to incorrectly place tasks as it cannot discern which device implementations will perform best. For these experiments the CPU is the only viable option for string columns and the FPGA can process only integers, thus limiting the potential for round robin to make poor decisions in this scenario.

Static Allocation by Data Type - This scenario fixes the allocation of delta merge column tasks based on their data type. In this scenario, the assumption is made that the developer has prior knowledge of the platform, devices and the performance of the implementations. With
this in mind, this scenario creates the type of static allocation that would be made with prior knowledge. In this case, the CPU is given all string columns and the FPGA the integer columns. Since the string columns take longer to process than the other data types, the GPU is given both the long integer and the double data type columns. This static allocation results in an even load balancing of tasks. This allocation is summarised in Table 4.5

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>Long Int</th>
<th>Double</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Static allocation of tasks

**SHEPARD Cost Based Allocation** - This scenario employs SHEPARD’s cost based allocation strategy. In this scenario SHEPARD will determine the devices available to process each column task and which device to use based on the cost of the column tasks and the overall demand present on each device.

In order for SHEPARD to be useful its cost based strategy should be able to replicate similar decisions and performance to that of the static allocation by data type.

Figure 4.6 compares the measured execution time of the entire delta merge operation for each of the allocation strategies. The execution times presented
4.5 Experiments

Figure 4.6: Delta Merge Total Execution Time per Allocation Strategy

are in seconds and represent the average execution times over three separate runs.

The results demonstrate that SHEPARD is able to effectively match the static allocation and achieve similar performance for the delta merge operation on the sample table. Firstly, SHEPARD is able to determine that only the CPU can process the string data columns and thus all string tasks are allocated to the CPU. Therefore, subsequent tasks must take account of the demand placed on the CPU resource as a consequence of the string tasks. This results in the GPU receiving most of the share of the double and long integer data typed column tasks. Additionally, since the FPGA can only process integer columns, it will have not have accrued any demand from the other data column tasks. Since the FPGA is also able to process integer columns the fastest, it is allocated all the integer column tasks that it can support. The result of using the cost approach means that the sensible allocation derived from prior knowledge of the devices and their performance
can also be replicated by SHEPARD at run-time.

When SHEPARD’s cost allocation strategy is compared with the round-robin style allocation, a clear improvement is observed. This is primarily due to the round-robin approach not having any knowledge of the cost of the tasks that is allocates. The different data types vary in the demands they place on the processors. So, despite the FPGA being allocated integers and the CPU being allocated strings, the CPU and GPU will also share double and long integer tasks. The cost strategy, on the other hand, is able to determine that strings will take a longer time to compute than the other data types and will compensate by pushing more long integer and double tasks to the GPU.

4.6 Conclusion

While heterogeneous processors can be advantageous to the overall execution time of the studied delta merge operation, the allocation of tasks among these processors is important to improve the achievable speed-up, and optimal, or near-optimal allocation is challenging. The round-robin strategy, despite its poor allocation decisions, is still able to improve performance over the fixed CPU allocations of 2 and 4 threads. However, the better allocations made through prior knowledge coupled with a fixed allocation and SHEPARD’s cost based allocation, demonstrate that further gains can be made through sensible placement of workload.

The work presented in this chapter also demonstrates the flexibility of the SHEPARD framework, allowing legacy C++ tasks, OpenCL tasks, and even FPGA tasks to be allocated in a consistent manner. Using the plug-in
based tasks, the developer is completely removed from any knowledge of the underlying implementations or devices available to the application. There is no overhead in terms of logic to execute or place tasks; developers can simply call tasks by name.

The SHEPARD repository, by allowing plug-ins to selectively implement a subset of tasks for different data types, allows various subsets of functionality to be accommodated among various devices without placing any extra burden on the application developer. The management runtime is able to parse the SHEPARD repository to derive the applicable set of devices for a given task before deciding where to place a task.

Finally, the full decoupling of the application, task implementations and run-time management simplifies the task of expert programmers to maintain their implementations and for administrators to manage their platforms. Using the SHEPARD repository, devices and implementations can be added or removed from SHEPARD’s allocation strategy. Additionally, the use of a management runtime allows costs and devices to be updated at run-time.

All this serves to create a dynamic and flexible framework that allows applications to adapt to, and maximise usage of, available resources.

4.6.1 Limitations

One important aspect not considered so far in this work is the impact on the host platform when offloading tasks to accelerators. Offloading work to accelerators can ease the burden on the host CPU and improve the overall performance of the system for certain workloads.
Additionally, until this point, investigations have centred around the delta merge operation. While an important performance maintenance operation, In-Memory Databases (IMDBs) have many other performance sensitive tasks that they must perform. For example, IMDBs are well placed to accelerate OLAP workloads due to the removal of the disk bottleneck. Heterogeneous processing technologies, such as accelerator cards, are particularly beneficial for performing data analysis over numerical datasets in particular.

Chapter 5 will greatly expand the application and analysis of SHEPARD by integrating it with an IMDB to handle analytic queries from multiple users.
Part III

Application of SHEPARD to the SAP HANA IMDB
Chapter 5

Application of SHEPARD to In-Memory Database Analytics

This chapter evaluates the use of SHEPARD to introduce and manage acceleration within an enterprise use case featuring an In-Memory Database (IMDB).

The previous chapters have focused on the creation and expansion of the management capabilities of the SHEPARD framework. These capabilities have been examined in the context of a single database maintenance operation, the delta merge.

This chapter conducts a much wider evaluation of the work by not only examining the accelerated workloads, but also investigating the actual performance improvements visible to end users in an IMDB context. In this context, the acceleration workloads will be part of user queries that must execute within an IMDB that will also have other, non-accelerated queries to carry out.
The experiments in this chapter also focus on managing a limited set of heterogeneous processing resources across many simultaneous user requests. Whereas much work to date has focused on optimising single queries on accelerators, such as GPUs, this work places emphasis on examining the actual acceleration realised when multiple users can share devices.

Analytic queries, which are compute intensive and can place significant load on the system, are used to evaluate the ability of SHEPARD to transfer the benefits of acceleration to multiple users. These queries are accelerated using OpenCL plug-ins which are managed by SHEPARD. To further evaluate the effectiveness of managed acceleration in an IMDB, a base load is simulated on the system using the SAP-H benchmark, to observe how both the accelerated analytic queries and the database performance, when performing other non-accelerated queries, is impacted.

Results demonstrate that SHEPARD can effectively manage queries from multiple simultaneous users over a limited set of shared resources. Results also show, that by using SHEPARD to manage accelerated task implementations, the load balancing of tasks across devices reduces demand on the host system, thereby indirectly improving the performance of the database as a whole.

5.1 Acceleration and IMDBs

In-Memory Databases (IMDBs) improve database performance by removing the disk bottleneck and storing all operational data in main memory. This has resulted in a shift in focus towards real-time query response, analytics
and reports, as opposed to end of day or end of month batch jobs. Therefore, for traditionally intensive tasks, such as predictive analytics, the desire is to obtain results in seconds or minutes as opposed to hours or days.

Given this drive to get results in real-time, the introduction of acceleration into IMDBs seems an obvious choice and has seen a number of studies. Unfortunately, while accelerators can achieve speed-ups for certain workloads, what is also necessary is a way to share accelerators across all users’ requests. Additionally, IMDBs can be deployed to a variety of hardware platform configurations and so the software needs to be able to adequately adapt to fully utilise all resources present. Therefore, a fixed implementation of certain functions to a fixed hardware target, such as a GPU, is not sufficient to cover the variety of hardware configurations that are now available in the market.

Existing work has motivated the use of acceleration in databases, GPUs in particular. Prior work has shown how using GPUs has achieved speed-ups for sorting [85], merge operations [88], compression [87], and many common SQL queries [86].

While such works provide clear motivation for the performance improvements that the use of acceleration technologies can provide, they do not consider the impact of other workloads on the system that may also compete for the same compute resources. In this scenario, it is unclear how much acceleration user queries will receive if other users are already making use of the devices. Additionally, no mention is made of the impact of other, non-accelerated tasks that the database may be performing concurrently to the tasks the authors study. Therefore, while the potential acceleration is
5.1 Acceleration and IMDBs

evident, the actual acceleration realised in a real, live system remains unclear.

Other work has also looked at processor selection between CPU and GPU for database queries [90, 91], but focus has been on individual and isolated query execution.

FPGAs have even been studied [89], showing not only the opportunities to introduce a variety of technologies to improve database performance, but also lend strength to the argument that offloading tasks can free up the database to process other tasks.

While the use of heterogeneous technologies within databases can achieve excellent performance gains, existing work has focused on isolated workload acceleration, fixed targets, such as GPUs, or single user scenarios.

This work broadens the investigation by applying SHEPARD to manage queries from multiple simultaneous users across a limited set of processing resources that include CPU, GPU and Intel Xeon Phi technologies. Additionally, existing work tends to focus on the speed-up of specific functions that have been accelerated while this work examines the performance also from the perspective of the end user. Therefore, this work looks at how the accelerated performance improves overall query time when simultaneous users are taken into account.

Additionally, this work places a simulated base workload on the database using the SAP-H benchmark. This is intended to examine how accelerated workloads can be affected by other work on the database, and how the other database workloads are affected when tasks are offloaded to accelerators.
5.2 Managed Analytic Queries on SAP HANA

This section provides an overview of the SAP HANA In-Memory Database and how it can be used to process custom analytic queries within the database platform.

Like most IMDBs, SAP HANA aims to provide close to real-time data analytics. To achieve this, SAP HANA utilises both high levels of parallel processing and compressed column oriented table storage.

Recall that Chapter 3 explained how SAP HANA utilised column-wise tables and column compression to improve performance. This is supported in a paper by Ababi et al. [106], showing strong performance gains in OLAP queries for column oriented tables.

5.2.1 Analytics on HANA via AFL

Having data in-memory means that it can be accessed and processed quickly. However, data analysis processing tends to be more complex than what SQL queries can describe, often requiring custom code. Commonly, the ability for developers to run custom programs on database data is provided via application servers. In this scenario, the data must be copied across network to the application server where it can then be processed externally to the database. HANA avoids this overhead by providing the Application Function Library (AFL) which allows custom code to be executed within a separate process within the database. This avoids any need to copy data across network to an external server.

AFL in turn allows developers to provide access to the implementation
via an SQL call, simplifying their use and allowing them to be invoked on demand. All extensions written using the AFL execute within the following framework of steps:

(i) **Data Copy**

Before the custom AFL implementation can execute it needs a copy of the data on which to execute. HANA makes available a compressed copy of the data columns or tables that will be consumed. Copying the data means that insertions or deletions will not affect the AFL task, and similarly, the table will not need to be locked until the AFL task completes. This is important as the AFL can accommodate any custom task, which could potentially take a long time to complete. For larger tables, making this data copy available can take a significant amount of time, especially if the database is under load. Therefore minimising the input dataset is important.

(ii) **Materialise Data**

In some cases, operating on the compressed, dictionary-encoded, data may be sufficient. However, for most algorithms, continually resolving keys using the column dictionary to retrieve the real value would be costly. To avoid this, a step called *Materialisation* is performed whereby the data is uncompressed to an array containing the actual data values. For large columns this can be a time consuming operation.

(iii) **Perform Task**

With the data now available in its native form the task can be performed. The task can then be implemented in typical C++ at this
point, no different from developing a stand-alone application.

(iv) **Output Results**

Finally, once the task completes the results need to be stored. This is often done by writing the results to an output table within HANA. These output tables can also be columnar in nature and will be compressed in HANA. If the result set of a task is large then this step can be time consuming.

When developing an AFL implementation, the developer needs to be mindful that other workloads may be operating on the database and other AFL workloads may also be present. Therefore, it is common that AFL implementations either use a limited number of CPU threads or allow the user to scale the number of threads they use when calling the AFL implementation.

### 5.2.2 Application of SHEPARD within the HANA IMDB

The AFL allows custom implementations to operate on a copy of data within the database. This is primarily targeted at complex tasks such as predictive analytics which can consume large amounts of data and take significant time to complete.

Clearly, applying acceleration technologies to these sorts of problems would be ideal. However, the database does not have a fixed hardware platform and can support a variety of hardware configurations. Therefore, technologies such as CUDA or other tools that target specific hardware vendors are not appropriate within this context.
5.2 Managed Analytic Queries on SAP HANA

Additionally, the IMDB is a multi-user environment that must support simultaneous requests from all authorised users. Therefore, static task assignment to devices such as GPUs would not be feasible as the large datasets these tasks can consume could quickly exhaust limited memory resources on accelerators, either drastically degrading performance or causing execution to terminate. Static assignment in this scenario would also not allow for vertical scaling, by adding additional accelerators, as the static allocation would not utilise any additional resources.

SHEPARD can address these issues by decoupling accelerated implementations from the database code and allowing multiple resources to be shared across simultaneous user requests.

For an AFL process, steps (i), (ii) and (iv) are concerned with retrieving data from the database and writing results back to the database. These steps are not concerned with the actual processing of the data, which is handled in step (iii) and is where most of the work is performed. Therefore, step (iii) is the ideal point in execution to utilise SHEPARD managed tasks.

Utilising SHEPARD to manage step (iii) via managed tasks provides the following benefits:

**Shared User of Accelerators**

Since multiple users may invoke AFL implemented queries, simultaneous requests for the same resource may occur. By using SHEPARD, these tasks are load balanced across the available devices according to the existing demand on each device and the estimated execution time of the task.
5.2 Managed Analytic Queries on SAP HANA

- Calculate cost per device implementation
- Examine current device task queue
- Choose Task with quickest turnaround

**AFL Process**
- Write Results
- Materialise
- Get Data

**SHEPARD**
- Call Managed Task
- Execute Task
- Receive Task
- Allocate Task
- Return Task Implementation

1. Device Parameters
2. Device Implementations
3. Costs
4. Task Queues
5. Chosen Implementation

- Get supported Devices & Implementations
- Calculate cost per device implementation
- Examine current device task queue
- Choose Task with quickest turnaround
- Return implementation to calling process
- Instruct task to execute when device is ready

Figure 5.1: Managed analytic query process using SHEPARD and AFL.

**Scaling of Resource Usage**

By using SHEPARD to allocate these tasks, the availability of resources can be changed without having to recode. For example, should a database operation have priority for a particular device, other implementations can be restricted from being allocated to this device by SHEPARD. Additionally, should extra devices be added to the database, such as adding an extra GPU, SHEPARD can utilise the new device and no code needs to be changed in the AFL implementation.

**Decoupling of Accelerated Implementations**

As steps (i), (ii) and (iv) are related to moving data from and to the database, they are unlikely to change. Step (iii) however may be optimised or enhanced at a later date. Wrapping this step as a managed task allows the developer to update the implementation in the SHEPARD repository, without having to revisit the database code.

Figure 5.1 presents an overview of an AFL query process that utilises SHEPARD managed tasks.
5.2.3 SHEPARD Run-time Task Allocation

AFL queries within the IMDB can be executed simultaneously by multiple-users with no explicit priority. Consequently, SHEPARD must handle simultaneous requests and place these requests on devices as they are received. SHEPARD uses its cost based system to estimate the execution time of each task received and uses the task queue for each device to estimate the total time any new task must wait before it can be processed. Using this information SHEPARD load-balances tasks among available processors. Balancing tasks among devices in this manner avoids any single device becoming a bottleneck.

SHEPARD does not implement task or user priority within its task allocation mechanism. Each task is treated equally and placed on a device task queue as soon as it is received. Such features, which may be desired in certain scenarios, are left as future work.

By using the task queues, multiple user requests can be accommodated on a limited number of shared resources. This is particularly useful as not all devices can support simultaneous execution from different processes, and many devices feature limited memory sizes, such as GPUs and Xeon Phis, and attempting to execute multiple workloads can exhaust these resources quickly. Therefore, management of these resources through SHEPARD is necessary.
5.3 Context of Acceleration in AFL Queries

Step (i) of an AFL query, the copy of the data, is fully managed by the database, therefore there is no opportunity to influence this phase of the query. Given this reality, any performance acceleration must be viewed within this context. Additionally, steps (ii) and (iv) interact with the database through its AFL API to materialise data and write results back to the database, therefore the opportunity to introduce acceleration into the query process is limited by these necessary steps.

To better investigate this, concrete examples of actual AFL queries are required. An example of a real library that utilises the AFL framework is the Predictive Analytics Library (PAL) [107]. PAL provides a number of commonly used analytic algorithms that cover a number of data-mining categories such as clustering, classification and regression analysis.

According to the PAL documentation, the constituent algorithms are present in the library based on their availability in other products, market surveys such as Rexer Analytics and KDNuggets, and their need in HANA applications [107]. With this motivation, three of the algorithms are chosen to investigate the acceleration of analytics within the IMDB.

5.3.1 Studied Analytic Queries

Three common analytic algorithms contained in PAL, K-Means, K-Nearest Neighbour and Multiple Linear Regression are studied.

OpenCL implementations are provided to accelerate the algorithms allowing them to be executed on CPU, GPU and Intel Xeon PHI technologies. For
comparison, multi-threaded CPU implementations are provided from PAL.

The number of threads used by the multi-threaded CPU implementations, hereafter referred to as the *native* implementations, can be configured by the user when invoked. The rationale behind this is that these algorithms can be compute intensive and run for long periods, and therefore, can impact the performance of other co-running workloads in the system. For this study, the implementations are configured to use the maximum number of threads available to maximise their performance and compare to the accelerated implementations.

### 5.3.1.1 K-Means

K-Means is a cluster analysis algorithm that seeks to put a set of data points into $K$ groups based on their spatial locality.

To do this the algorithm selects the first $K$ data points and assigns the value of the group centres to the positions of these points.

The algorithm then iteratively performs 2 steps until either no points change group, or a maximum number of iterations is reached.

1. Assign each point to the nearest class centre.

2. Update the position of each class centre according to its membership.

For comparative purposes, the algorithm implementation in this work performs a fixed number of 100 iterations.

The accelerated implementation performs the first step in an SIMD fashion using OpenCL and the second step sequentially on the CPU.
5.3.1.2 K-Nearest Neighbours (KNN)

KNN is a classification algorithm that seeks to assign a new data point to a group based on a set of labelled data.

To do this, the algorithm finds the $K$ nearest points to the new point to classify and, based on the classification of the neighbours, assigns the new point to the appropriate group.

This process is repeated for every new data point that needs to be classified.

Often K-Means is performed on the unlabelled dataset, so K-Means and KNN are commonly used together.

5.3.1.3 Multiple Linear Regression

Given a set of observations, Multiple Linear Regression seeks to describe the relationship of an output variable $Y$ (dependent variable) to a set of $N$ input variables $X$ (independent variables), as a linear equation as shown in Equation 5.1. $\alpha$ refers to the intercept and $\beta$ the coefficient value for the corresponding independent variable.

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_M X_M$$ \hspace{1cm} (5.1)

The observations are usually stored as separate rows within a database table. A convenient way to implement this algorithm is to use matrix multiplication. Matrix multiply can be multi-threaded well, and performs particularly well on accelerators.
5.3 Context of Acceleration in AFL Queries

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Large</th>
<th>Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-Means</td>
<td>100,000,000 x 2</td>
<td>10,000,000 x 2</td>
</tr>
<tr>
<td>KNN</td>
<td>100,000,000 x 2</td>
<td>10,000,000 x 2</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>8,000,000 x 26</td>
<td>1,280,000 x 26</td>
</tr>
</tbody>
</table>

Table 5.1: Algorithm Datasets (Rows x Columns)

This matrix representation of the solution is presented in Equation 5.2.

\[(A^T A)x = (A^T y)\]  \hspace{1cm} (5.2)

Given a set of \(N\) observations, \(A\) is an MxN matrix containing the values of the independent variables, and \(Y\) is a vector containing the dependent variable values. \(X\) is a vector which, when the equation is solved, will contain the coefficient for each independent variable.

5.3.2 Datasets

The experiments utilise two different sizes of input datasets, large and small. The larger tables each contain over a gigabyte of uncompressed data to be processed in the analytic queries. This is intended to place significant load on the system per analytic query.

The smaller datasets are likely to be more commonly queried by multiple users in an ad-hoc fashion.

Further details on the datasets are presented in Table 5.1.
5.3.3 Hardware Platform

The hardware platform available for experimentation consists of a dual-socket Ivy-bridge Xeon CPU with a total of 24 hyper-threaded hardware cores resulting in 48 logical cores (threads) being available. There are three accelerator cards present in the system, two Intel Xeon PHI 5110p cards and one NVIDIA K20x GPU. Further details are given in Table 5.2.

<table>
<thead>
<tr>
<th>Device</th>
<th>Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon x5650</td>
<td>24 (48 Threads)</td>
<td>256 GB</td>
</tr>
<tr>
<td>(2x) Intel 5110p</td>
<td>61 (240 Threads)</td>
<td>8 GB</td>
</tr>
<tr>
<td>NVIDIA K20x</td>
<td>2688</td>
<td>6 GB</td>
</tr>
</tbody>
</table>

Table 5.2: Experimental Hardware Platform

5.4 Analytic Query Performance

This section provides an overview of the performance of the native and accelerated queries as they execute within the HANA environment. The results present the breakdown of where time is spent in each query, providing an overview of the potential and actual speed-up within each query.

To clarify, the native queries run on the CPU, utilising as many cores as are available and the accelerated implementation runs on the device that performs best for that particular algorithm.

The execution of the query is split into 3 phases:

Data Copy

This phase of execution relates to step (i) in the AFL process where
Figure 5.2: Native Query - Proportion of Time Spent in each Query Phase

the data is copied and made available for processing.

**Algorithm Time**

This phase is concerned with only the time taken to process the algorithm portion of the query process, i.e. AFL step (iii).

**Non-Algorithm Time**

This time accounts for the other operations that execute within the AFL query process, namely materialisation of the data and writing results back to the database. Therefore, this phase includes steps (ii) and (iv) of the AFL process.

Figure 5.2 presents the breakdown of where each native query spends its time. Recall that native queries are multi-threaded implementations that run on the CPU.

For KNN, almost all of the query time is spent on processing the KNN algorithm itself. Consequently, there is a clear opportunity for acceleration to significantly improve the overall execution time of this query.

The Linear Regression query spends a significant amount of time on copying the data and also writing results, which accounts for most of the non-
5.4 Analytic Query Performance

Figure 5.3: Accelerated Query - Proportion of Time Spent in each Query Phase

algorithm time. Despite this, the algorithm itself still occupies over 55% of the total query time.

Analysing the breakdown of the K-Means queries reveals that the bottleneck shifts between the data copy and algorithm phases when looking at small and large datasets, respectively. This indicates that the time spent performing the K-Means algorithm scales at a much faster rate than the time needed to copy more data. Therefore, for larger datasets, acceleration of the algorithm can lead to greater overall query speed-up.

Understanding where each query spends its time reveals the true potential for acceleration. If the algorithm phase accounted for a small portion of the total query time then there would be little advantage gained from acceleration. The algorithms studied in this work, however, all show that the algorithm phase accounts for a large proportion of the overall query time, motivating the use of acceleration.

Figure 5.3 shows the breakdown of the same queries when an accelerated implementation is used to execute the algorithm phase. The accelerated queries were run on each device available, however, only the results from the
best performing device are presented in Figure 5.3. Therefore, the query breakdown presented in Figure 5.3 represents the best possible performance of the query with the devices and implementations available in this study. For these particular algorithms, the GPU performs best for Linear Regression and K-Nearest Neighbours, while the Xeon Phi performs best for the K-Means algorithm.

For the accelerated KNN query, the reduction of time spent in the algorithm phase is significant, falling from over 95% of the total time for the native implementations to just over 35% and 27% for small and large datasets, respectively.

Linear Regression also witnesses an equally impressive reduction in time spent on the algorithm phase. Whereas the native query spent almost 56-60% of its time on the algorithm for the small and large datasets, the accelerated query spends approximately 9% of the total query time on the algorithm.

The K-Means query’s distribution of time differed between the small and large datasets. Whereas the query using the small dataset spent only 32% of the time on the algorithm phase, when processing the large dataset over 91% of the time was spent on the algorithm phase. This trend is not observed for the accelerated query which spent 15% of the time on the algorithm for the small dataset and 10% for the large dataset. This suggests that while the native algorithm implementation scales poorly with data size, the accelerated query scales fairly well.

For each of the accelerated queries, the bottleneck has been shifted from the actual algorithm to other database tasks. For K-Means the clear bottleneck has become copying the data, while for Linear Regression writing
the results back to the database is the most significant contribution to the total execution time. KNN has a much more even distribution of time spent among all three phases, spending slightly more time on the data copy and non-algorithm phases.

While these results show a clear improvement when acceleration is applied to isolated queries, they do not show the true picture of achievable acceleration within the database. Rather, these results show the best possible acceleration outcomes, where a single user query is run on the system. Unfortunately, this is not the normal scenario for an IMDB. The reality is that a number of users will be interacting with the IMDB and a number of other workloads will compete for available resources.

In order to provide true acceleration within the database, users queries need to be able to share resources, and this is achieved via SHEPARD.

5.5 Experiments

The experiments presented in this work investigate the application of SHEPARD managed tasks to analytic workloads within an IMDB. The experiments study the actual benefit from utilising managed acceleration in an IMDB, as opposed to isolated implementations.

The experiments will examine:

Query Time Improvement

By using SHEPARD, a limited number of resources can be shared across multiple users to reduce average query time.
Distribution of Acceleration

Since SHEPARD employs a task queue system to orchestrate execution of tasks on shared devices, when large numbers of simultaneous tasks are received, tasks will need to wait on the queues. Therefore, within this constraint, how much acceleration does each user query still receive?

Reduction in System Overhead due to Offloading

By using SHEPARD to enable accelerated queries to be offloaded to accelerators, the demand on the host CPU is reduced, which can improve system performance.

5.5.1 Baseline - SAP-H

Having studied the breakdown of the analytic queries when run in isolation, it is important to also consider what happens when the database is performing other user queries that commonly occur on the system.

To do this, a baseline is first required, against which results can be compared. To this end, the SAP-H benchmark is used. SAP-H is an OLAP benchmark, based on TPC-H [108], which uses data tables more applicable to an IMDB, such as utilising more columns per table and more string typed columns.

The SAP-H benchmark simulates a number of active users on the system with each user executing a set of 18 queries. These queries are designed to be intentionally complex and when the benchmark is run with 10 users, the host processor is fully utilised.
5.5 Experiments

Using SAP-H provides a comparison for how well the database can answer queries when additional analytic workloads are placed on it. Running SAP-H in isolation results in the best possible execution time for a particular number of users. This time is compared against SAP-H when co-run alongside the analytic AFL queries to observe the impact such workloads can have on the IMDB.

Co-running SAP-H alongside the AFL analytic queries, to simulate a busy system, also allows the impact on the AFL analytic queries to be observed.

5.5.2 Multi-User Experiments

The experiments in the following sections investigate the acceleration achieved when devices are shared among multiple simultaneous users. Using SHEPARD, simultaneous user queries are allocated across available devices which use task queues to process tasks in order.

To simulate different workloads, three scenarios are created:

10 Users - Large Datasets

This scenario employs 10 users, each requesting the same query, K-Means, KNN or Linear Regression, which will consume a large dataset. The purpose of this scenario is to place extreme load on the system. Under these conditions it is expected that the introduction of SAP-H can have a profound impact on query performance, and employing accelerators can have great benefits. Therefore, this scenario represents the best case for using acceleration within the database.
5.5 Experiments

5 Users - Mixed Queries

This scenario intends to simulate a light user load on the system to investigate what benefits managed acceleration can provide outside of extreme load conditions. As databases may have periods of light and heavy traffic, it is important to investigate all aspects of performance.

100 Users - Mixed Queries

This scenario is aimed at simulating a heavier system load by employing 100 users that simultaneously issue analytic queries. This scenario investigates SHEPARD’s ability to adequately allocate tasks between a limited number of devices while also observing how much acceleration can be provided to each query in this circumstance.

For each of the experimental scenarios, the performance of the system is examined from three viewpoints.

Algorithm Time - This is the time spent computing just the analytic algorithm within the query, i.e. step (iii) in the AFL query process. This step is where SHEPARD managed tasks are called and control passes to SHEPARD to determine which device to use and when to execute.

Total Server Time - This phase of execution concerns steps (ii)-(iv) of the AFL query process. These steps are executed in a separate process within the IMDB and execute code provided by the developer. Therefore these steps include not only the time to execute the algorithm, but also to materialise the data and write any results back to the database.
**Client Time** - This time includes the *Total Server Time* as well as the data copy, step (i), which is under control of the IMDB and cannot be influenced by the developer. Therefore, this view represents the time taken from when a user submits a query until an answer is returned.

By examining the query performance from these three viewpoints, the transfer of achieved acceleration from the algorithm stage to the query as a whole can be observed. Each experiment is compared to the unmanaged CPU implementations showing how much benefit the accelerated implementations can bring to the end user when managed through SHEPARD.

For each experiment scenario, the SAP-H benchmark is also executed alongside the studied queries to observe how their performance degrades with a base system workload.

### 5.5.3 10 Users - Large Datasets

The experiments in this section explore the scenario where a number of users each execute analytic queries that consume the large datasets outlined in Table 5.1. In this scenario, the database must make a copy of each dataset available to each user and each query must materialise that data, which can be time consuming for larger datasets.

The size of the datasets means that a sustained load is placed on the system. While SHEPARD will ensure the tasks are executed in order on each device, the native CPU implementations will co-run on the host CPU. Given that the native implementations do not always fully utilise all the threads during their execution, co-running may benefit some of these queries
by better utilising the resources. However, co-running the native queries can also apply increased memory pressure on the system as well.

Three sets of experiments are performed on these datasets, one for each algorithm. Each experiment is performed with 10 users that simultaneously invoke either K-Means, KNN or Linear Regression. The experiments are then repeated with the introduction of the SAP-H benchmark which is run alongside the analytic workloads to observe the performance impact when a base load exists on the system.

Figure 5.4 presents the relative performance of each query, in terms of average execution time, versus the unmanaged native CPU implementations. In general, the accelerated implementations managed through SHEPARD all outperform the native implementations, including the case when the SHEPARD managed queries are co-run with SAP-H. As expected, all the native queries suffer performance degradation in the presence of SAP-H.

From the breakdown of where each accelerated query spends its time, presented in Section 5.4, it is clear that most acceleration will be observed in the algorithm stage, but the rest of the query, which is not accelerated, will diminish the overall observed acceleration.

This is seen most clearly in the K-Means query, where the algorithm phase of the query executes 100 times faster than the native query. However when additional phases are taken into account, the actual query speed-up is 12.65 times faster than the average native query implementation. While still a significant speed-up, in this case reducing execution time from hours to minutes, it is important to acknowledge that overall acceleration of the query is limited by database specific operations.
K-Means also suffers a significant reduction in performance when co-run with SAP-H. The primary reason for this is that a part of the K-Means algorithm must execute on the CPU. The additional SAP-H workload reduces the performance of this stage, decreasing the achieved algorithm acceleration by a quarter. The knock-on effect of this is that total query acceleration is reduced from 12.65 to 5.12 times over the native queries, which do not co-run with SAP-H.

For KNN, achieved acceleration is similarly reduced when the full query
operation is considered, falling from an acceleration of over 20 times that of the native implementation of the algorithm, to just of 6 times speed-up for the complete query. The introduction of SAP-H however has no observable impact on the accelerated KNN algorithm, due to the fact that it executes entirely on the accelerator, unlike the K-Means algorithm which contains a step performed on the CPU. Therefore, the additional reduction in acceleration observed when SAP-H is present is entirely down to performance degradation of steps (i), (ii) and (iv) of the query process. What this also shows, is that even under load, SHEPARD is able to adequately allocate and orchestrate tasks without imposing additional overhead on its managed tasks.

The linear regression query loses half of its acceleration due to the overheads from the database operations, achieving just over 3 times speed-up versus the native queries. Similarly to KNN, almost all of the algorithm step is performed on the device, bar a minor matrix transpose pre-processing step on the CPU. Therefore, most of the acceleration of the linear regression algorithm is maintained when SAP-H is present, but given that the speed-ups of the algorithm are much lower compared to KNN and K-Means, the overall speed-up when SAP-H is present is 1.24 times that of the native queries.

Observing the native queries in the presence of SAP-H, both linear regression and KNN suffer significant performance degradation due to the competition for CPU resources. The K-Means query however is only 9% worse off when SAP-H is co-run. The major reason for this is that the K-Means query runs significantly longer than any of the other queries for the large datasets, taking hours to complete, as opposed to minutes for the other queries. As a result, though the native K-Means query may take much longer to complete
Figure 5.5: 10 Users, SAP-H, Times Slower with Additional Workloads

in the presence of SAP-H, proportionally, this time is much less than in the other queries.

Figure 5.5 compares the average execution time of the SAP-H benchmark when run alongside each set of analytic queries to its baseline performance. For linear regression, the accelerated queries managed using SHEPARD impose a 2% degradation on the execution time of SAP-H, while the unmanaged native queries impose a degradation of 15%. This is due to the fact that accelerated linear regression implementation completely offloads the algorithm to the device, and by using SHEPARD, multiple users can all have their queries allocated across the shared devices. Therefore, the load on the host CPU can be reduced, minimising the impact on the other database workloads, simulated by SAP-H. The native CPU linear regression algorithm also fully utilises all available threads made available to it, therefore creating significant contention for CPU resources.

KNN places little impact on the SAP-H benchmark as it is completely offloaded for the accelerated implementations and the native implementations
mainly operate using a single thread.

K-Means imposes the largest degradation when run alongside SAP-H. For the native query this is due to the fact that K-Means uses all of the available threads for most of its execution, creating significant contention for CPU resources and resulting in SAP-H taking 29% longer to complete. This degradation is almost halved to 15% for the accelerated query due to offloading this compute intensive algorithm. However, there is still significant overhead placed on SAP-H due to the fact that part of the accelerated K-Means algorithm still employs a sequential CPU step, necessitating continual transfer of data to and from the device.

5.5.4 5 Users - Mixed Queries

The previous section focused on understanding the individual queries within a multi-user context, and in particular, using large datasets to stress the system and study the impact on the database through SAP-H.

This section focuses on a small number of users executing a mix of analytic queries on smaller datasets. The small datasets used in this section are detailed in Table 5.1. For these experiments 5 users will each execute a total of 20 queries that contain a mix of K-Means, KNN and Linear Regression queries, resulting in a total of 100 queries.

In particular, these experiments investigate the effectiveness of SHEPARD to transfer acceleration to user queries when multiple users all share a limited set of shared devices. The native CPU queries will all execute simultaneously on the CPU, while SHEPARD will enqueue multiple queries,
allowing devices to process their tasks queues in order. Therefore, while the native implementations’ performance will degrade due to resource contention, the SHEPARD managed tasks will incur waiting times on the device queues. Despite these constraints, SHEPARD is able to transfer acceleration to all queries studied, though the acceleration observed per query will not be equal for all queries.

Firstly, when observing the impact of the additional analytic queries on the database, through SAP-H, little or no overhead can be seen, as recorded in Table 5.3. This is likely due to the fact that 5 simultaneous user queries are not sufficient to fully saturate system resources on the hardware studied. Additionally, KNN does not always use the full allocation of threads, due to sequential portions of its implementation. Therefore, the system is able to accommodate 5 simultaneous analytic tasks, particularly since the datasets are much smaller in this case.

<table>
<thead>
<tr>
<th># Users</th>
<th>SHEPARD</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 5.3: SAP-H - 5 Users, Times Slower with additional workloads

Figure 5.6 shows the achieved speed-up of the SHEPARD managed accelerated queries over the unmanaged native queries. Once again, a similar falling trend in observed acceleration is seen from algorithm to client execution time. The achieved acceleration without the presence of SAP-H drops from 18 times faster for acceleration of the algorithms to 3.65 times faster for the full query, as compared to the native implementations. When SAP-H is
5.5 Experiments

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Time</th>
<th>Total Server Time</th>
<th>Client Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHEPARD</td>
<td>18.08</td>
<td>7.93</td>
<td>3.65</td>
</tr>
<tr>
<td>SHEPARD + SAP-H</td>
<td>15.27</td>
<td>5.93</td>
<td>1.61</td>
</tr>
<tr>
<td>Native + SAPH</td>
<td>10</td>
<td>100</td>
<td>0.73</td>
</tr>
</tbody>
</table>

Figure 5.6: 5 Users, Times Speed-up Versus Native Execution Time

The present the drop is from 15.27 to 1.61 times faster than native queries. While the overall acceleration is less for the smaller datasets and smaller number of users, average query time is still improved. Note that the native K-Means algorithm performs much better for smaller datasets where the data copy becomes its bottleneck, and so acceleration on smaller datasets is much less.

For the native queries, most of the performance degradation when SAP-H is present occurs when processing the analytic algorithms. For 5 users, the average native query is 1.28 times slower when SAP-H is present.

5.5.4.1 Query Performance Distribution

This section examines the distribution of query performance among all the queries performed in the experiment.

Figures 5.7a and 5.7b show the relative performance of queries run in this experiment against the best possible query time recorded for a single user scenario. In other words, the relative performance refers to how much slower the query is versus the best possible execution time for a query of that type.
Each column in these figures corresponds to an individual query instance, and instances are ordered by their performance.

Figure 5.7a shows that the K-Means queries achieve close to maximum acceleration. This is due to the fact that K-Means is the longest running query and all devices achieve a similar level or performance for this algorithm. Additionally, since the K-Means is the longest running algorithm studied, any time spent in the task queue causes a relatively minor performance degradation in proportion to the total execution time. KNN and Linear Regression, however, do see some queries with up to 5 times longer execution times for the algorithm phase. This is primarily due to the fact that these queries have short execution times and any time spent waiting on a device queue for another task to complete causes a proportionately greater overhead. However, when the total time taken processing on the server is taken into account, i.e. including data materialisation and output of results, the overheads incurred in the algorithm phase are largely diminished, owing to the relatively short amount of time spent in the algorithm phase.

Interpreting Results

Next, the breakdown of query performance over the lifetime of the experiments will be discussed. To aid the reader in interpreting the results presented in Figures 5.9 and 5.13, Figure 5.8 gives an example of the charts that will be presented in the following sections.

The top chart in Figure 5.8 shows the number of tasks queued to each device at any given time during the experiment.

In the bottom chart, each point denotes an individual query.
5.5 Experiments

(a) SHEPARD Managed Queries, Times Slower Versus Single User Time

(b) Native Queries, Times Slower Versus Single User Time

Figure 5.7: 5 Users - Per Query Execution Time Versus Single User Time
5.5 Experiments

The horizontal axis shows the time a query began its algorithm phase, i.e. step (iii), which is where control passes to SHEPARD. The vertical axis shows how long a query took to complete step (iii), which includes any queuing delay as a result of waiting for other query tasks to complete. The horizontal axis values are normalised with the earliest query starting at time 00:00. From this figure it can be seen that queries do not all start at the same time due to steps (i) and (ii) of the query process which retrieves and materialises the data.

Point “a” in Figure 5.8 shows that a task began execution at time 00:00 when the device had zero tasks queued and took just under 5 seconds to complete. Point “b” indicates a task starting at 01:28 and took 1 minute and 36 seconds to complete, while at this time the device had 20 tasks in its queue.

Figure 5.8 presents an example for accelerated queries. For the native queries which only execute on the CPU, there will be no chart given for device queues; however, interpreting the time-line for the query execution times will remain the same as for the accelerated queries.

![Figure 5.8: Example Query Time-line Chart](image_url)
5.5 Experiments

Figure 5.9 plots the breakdown of queries over time for 5 users. As can be seen from this figure, since there are only 5 simultaneous users and SHEPARD has 4 devices over which to allocate tasks, the device queues rarely exceed a single task. Therefore many of the tasks achieve close to optimal acceleration, as shown in Figure 5.9b and also when SAP-H is present in Figure 5.9d. The figures clearly show that degradation in query performance correlates to when a device queue reaches 2 or more tasks.

Figures 5.9a and 5.9c display the absolute query times per device when SAP-H is absent and present, respectively. These figures show that SAP-H does not induce a large overhead in execution time with the longest query taking no longer than 20 seconds when SAP-H is absent and 26 seconds when present.

For the native queries, Figure 5.7b shows that the degradation for each query suffered in the algorithm phase is mostly carried over to the total query time on the server. KNN however, behaves differently to the other queries, slightly outperforming a single user run for these experiments. The primary reason for this is that KNN iterates over the same dataset multiple times during its execution, using a single thread, and so, when multiple instances of this query run, they may be able to reuse some data already present in the CPU cache.

When SAP-H is introduced, K-Means and Linear Regression suffer minor additional performance degradation, whereas KNN performs over 2 times more slowly for some queries, as shown in Figures 5.10b and 5.10d. This equates to over a 4 minute increase in execution time for the native KNN query, as shown in Figures 5.10a and 5.10c, whereas the longest accelerated
queries suffer only a 6 second increase when SAP-H is present.

In summary, for 5 users, there are enough devices for SHEPARD to minimize any additional overhead from device queues and many queries can achieve close to their optimal performance. Even when queries do take longer to complete, the overhead is in the order of seconds in the worst case, as opposed to minutes for the native queries.
5.5 Experiments

(a) Query Execution Time per Device, SAP-H absent

(b) Relative Query Performance Versus Single User, SAP-H absent

(c) Query Execution Time per Device, SAP-H present

(d) Relative Query Performance Versus Single User, SAP-H present

Figure 5.9: 5 Users - Breakdown of SHEPARD Managed Query Performance
[top]Device Queue Lengths
[bottom]Query Execution Time (Algorithm Phase)
5.5 Experiments

(a) Execution Time per Query, SAP-H absent

(b) Relative Query Performance Versus Single User, SAP-H absent

(c) Execution Time per Query, SAP-H present

(d) Relative Query Performance Versus Single User, SAP-H present

Figure 5.10: 5 Users - Breakdown of Native Query Performance
5.5 Experiments

5.5.5 100 Users - Mixed Queries

This section expands the investigation to observe what happens when a large number of simultaneous users all execute analytic queries. In this scenario the same 100 queries used for the 5 user experiments are now distributed across 100 users. Therefore, each of the 100 users will execute an analytic query of either KNN, K-Means or Linear Regression.

For 5 users, SHEPARD was able to adequately load balance all queries across the available devices. However, in this scenario, the number of users far exceeds the number of devices and therefore, the device task queues will grow large. As a consequence, many tasks will incur wait times on the device queue which will be reflected in the algorithm time, i.e. the time to process the KNN, K-Means or Linear Regression tasks, which are managed by SHEPARD. With these additional overheads per query, these experiments will investigate if sharing accelerated processing devices across a large number of users can still return an average query speed-up and how much of the potential query acceleration is actually achieved by each query.

For the SAP-H benchmark, 5 users did not cause any performance degradation and the system was able to achieve the same execution time as the baseline. However, when 100 users all wish to execute analytic queries SAP-H does take much longer to execute as shown in Table 5.4. The SAP-H execution time is 8% longer when SHEPARD is used to manage the accelerated implementations, while the native queries cause SAP-H to take 14% longer. In both the accelerated and native query cases, SAP-H will incur the overhead of 100 users all requiring data to be copied and materialised
which will place pressure on the system CPU and memory resources. The additional overhead caused by the native implementations will be as a result of their extra demand for CPU resources, in direct competition with SAP-H. As discussed previously, by using SHEPARD to manage the accelerated implementations, devices process a single task at a time, so even though SHEPARD will allocate some tasks to the CPU, they do not all compete for CPU resources at the same time.

Figure 5.11 shows the achieved average query speed-up across each phase of query execution. In contrast to prior results, the drop in achieved acceleration between the algorithm phase and total server time is much less. This is due to the fact that the algorithm phase will constitute a larger proportion of the query time as SHEPARD managed tasks now have to wait in the device queues to be processed. This additional overhead will increase the overall algorithm time experienced by most of the queries. This queue overhead also negates the overheads of the SAP-H benchmark on the SHEPARD managed queries. This is due to the fact that while tasks are waiting in the queue they are not using any system resources and each device will only process a single task at a time, facilitated by the task queues. Therefore, whether SAP-H is present or not, the SHEPARD managed queries maintain an average of over 4.5 times speed-up when compared to the native queries. When examining

<table>
<thead>
<tr>
<th># Users</th>
<th>SHEPARD</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.08</td>
<td>1.14</td>
</tr>
</tbody>
</table>

Table 5.4: SAP-H - 100 Users, Times Slower with additional workloads
5.5 Experiments

the total server time, the achieved acceleration is not significantly less due to the increased algorithm times.

When the data copy included in the client time phase of execution is considered, the total achieved average query time improvement is approximately twice that of the native queries, in both cases where SAP-H is absent or co-run alongside the analytic queries. This drop in achieved acceleration will be exacerbated due to the high demand of 100 simultaneous users all requesting data copies from the database.

The unmanaged native queries suffer performance degradation of around 10% when SAP-H is present. While the time to execute the algorithm phase suffers the most degradation, this is not transferred to the client phase which observes a less severe slowdown. This is due mainly to the data copy phase which will have significant impact on the query time due to 100 simultaneous users all requesting data from the database.
5.5.5.1 Query Performance Distribution

Query performance for 100 users is analysed in the same manner as the 5 users scenario. Figures 5.12a and 5.12b show the distribution of performance for each query. As before, the values in these figures show the actual query performance recorded in the experiment relative to the best possible performance of the query, from the viewpoint of total server time and algorithm time.

Once again, for the SHEPARD managed queries, the relative slowdown observed for the algorithm time is much greater than the total query slowdown, due to the proportion of query time spent outside of this phase. The linear regression query shows the largest distribution of achieved query performance in Figure 5.12a. There are two contributing factors to this observation. The first is that the linear regression algorithm is the fastest of the three algorithms studied. Therefore, any extra time taken to process this query will show a proportionally larger slowdown than the other queries. As a consequence, any time spent waiting to process in a device queue results in a much larger relative slowdown for linear regression, as compared to K-Means and KNN. Secondly, the accelerated linear regression heavily favours the GPU, which is 4 times faster than the next best device. Therefore, any time the linear regression algorithm must wait or use a device other than the GPU, acceleration potential is lost. However, when the total server time is considered, the overall slowdown, versus the best potential acceleration, is much reduced. Since the linear regression algorithm itself accounts for only 9% of the total query time, as shown in Figure 5.3, any reduction in
achieved acceleration has a much smaller impact on the query performance as a whole. Therefore, while some of the linear regression queries take up to 50 times longer than an optimal query, the total server time slowdown is always below 10 times that of an optimal query.

For SHEPARD, the results show that K-Means suffers the least from having 100 users competing for the available resources. This is primarily because K-Means takes the longest time of the three algorithms in these experiments and the performance for each device implementation is similar. For KNN and Linear Regression the penalty for not being able to use the preferred device is much greater. Additionally, the KNN and Linear Regression algorithms complete much faster than K-Means, and as a result, any time spent waiting on the device queue causes these algorithms a proportionally greater slowdown. As is the case for 5 users, when total server time is taken into account, the perceived slowdown is much reduced due to the other overheads of accessing and preprocessing the data. As a result, despite the larger slowdowns observed from the algorithm time, the distribution of query slowdown observed in the client times is much smaller.

Figure 5.13 shows a further breakdown of query performance, including the queue lengths for each device during the experiments. There is a clear correlation between the queue length and the query performance, as expected. Figures 5.13a and 5.13c show the execution times of queries when SAP-H is respectively absent and present. In both cases, as tasks are posted to the devices, the number of tasks queued to devices becomes large. However, note that due to SHEPARD’s allocation strategy, no single device becomes a bottleneck, and even at the peak number of tasks, task times for all devices
5.5 Experiments

(a) SHEPARD Managed Queries, Times Slower Versus Single User Time

(b) Native Queries, Times Slower Versus Single User Time

Figure 5.12: 100 Users - Per Query Execution Time Versus Single User Time
are similar. The GPU is given most tasks due to its superior performance in many cases and is able to clear its task queue at a faster rate than the other devices. This results in all devices being utilised throughout the experiments. The presence of SAP-H results in much more variance of task execution time; however, the load balancing remains effective.

Figures 5.13b and 5.13d present the same experiments from the viewpoint of how many times slower each query is when co-run with 100 users. Once again, the peak in tasks to process results in the largest overheads per query. As discussed, the degree to which each query suffers relative execution time performance degradation is proportional to how long that query takes. Shorter queries, such as Linear Regression suffer proportionally more by waiting on the device queues, as opposed to longer running queries, such as K-Means. This information means that future work can make use of this to potentially prioritise workloads, while minimising the impact seen by users.

For the native implementations in Figure 5.12b, the algorithm time makes up a much greater proportion of the total execution time, and therefore, the slowdown in algorithm time is mostly reflected in total server time. For Linear Regression and KNN, each of the queries suffers a similar level of slowdown. However, the K-Means algorithm which is the longest to process, shows a large distribution of slowdown per query.

Figures 5.14b and 5.14d show the relative performance of queries versus the best case recorded by a single user in the case SAP-H is absent and present, respectively. Figures 5.14a and 5.14c show the absolute query times. KNN is the longest running algorithm and most queries take 2 - 4 times longer than a single user. However when SAP-H is present queries can take over 3
minutes longer to execute. For Linear Regression, when SAP-H is introduced most queries become at least 4 - 6 times slower, as opposed to between 2-5 times slows for many queries when SAP-H is absent. For K-Means there is a similar increase with most queries being at least 4 times slower in the absence of SAP-H and at least 6 times slower when SAP-H is present.

Comparing these results to the SHEPARD managed implementations, it can be seen that, despite that larger comparative slowdown of the SHEPARD queries, the gulf in performance between the native implementations and the accelerated implementations still allows for an overall speed-up of average query time. Though the slowdown of the SHEPARD managed queries looks comparatively larger than the native queries, they still execute much faster. Even scaling up to 100 simultaneous user queries, average query time is significantly improved using SHEPARD, and overhead on the database, as simulated by SAP-H, is much reduced.
5.5 Experiments

(a) Query Execution Time per Device, SAP-H absent

(b) Relative Query Performance Versus Single User, SAP-H absent

(c) Query Execution Time per Device, SAP-H present

(d) Relative Query Performance Versus Single User, SAP-H present

Figure 5.13: 100 Users - Breakdown of SHEPARD Query Performance
[top]Device Queue Lengths
[bottom]Query Execution Time (Algorithm Phase)
5.5 Experiments

(a) Execution Time per Query, SAP-H absent

(b) Relative Query Performance Versus Single User, SAP-H absent

(c) Execution Time per Query, SAP-H present

(d) Relative Query Performance Versus Single User, SAP-H present

Figure 5.14: 100 Users - Breakdown of Native Query Performance
5.6 Conclusions

This chapter has presented an investigation into the application of SHEPARD to manage tasks within a real-world enterprise application, namely an IMDB.

Experiments studied not only the ability for SHEPARD to handle simultaneous tasks from multiple users and manage their execution across a set of shared resources, but also analysed the impact from the view of the full query process.

By using a real-world use case, this work has demonstrated the value of introducing managed acceleration to enterprise applications and has also shown that acceleration technologies are not only applicable to HPC workloads where a single application owns all of the resources, but also where multiple user tasks may also vie for shared acceleration resources.

The investigation has shown how the queue based system employed by SHEPARD can impose overhead on individual queries, yet still maintain overall speed-up for an average user query, even when up to 100 simultaneous user queries are performed.

Results have also demonstrated that by using SHEPARD to load balance and manage allocation of tasks to accelerators, load on the host system can be reduced, freeing up resources and reducing the impact of analytic queries on the database, as simulated using the SAP-H benchmark.
Part IV

Conclusions & Future Work
Chapter 6

Conclusions

This work set out to address a number of the challenges associated with heterogeneous computing by creating the SHEPARD framework. The framework simplifies development for the application programmer through managed tasks. The SHEPARD runtime also intelligently places tasks on devices based on expected task costs and current device demand.

This work has also contributed to Hardware and Network Enhanced Software Systems for Cloud Computing (HARNESS), an EU project aimed at introducing heterogeneous resources into the cloud [109, 110]. The HARNESS project considers provision and management of network, storage and compute resources. The work in this thesis directly contributes to compute resource management, allowing the HARNESS cloud to provision hardware which SHEPARD manages. This allows applications to be deployed to a variety of different hardware configurations and take advantage of available compute resources via SHEPARD managed tasks.

Additionally, this work has also contributed to SAP by examining a number of scenarios within the HANA IMDB and demonstrating that shared heterogeneous compute resources can provide tangible benefits in multi-user
environments.

Furthermore, Intel has proposed an approach which is similar to the architecture of this work, in particular the use of a repository of implementations coupled with a runtime to manage application workloads across compute resources. This can be viewed in a technical talk presented at the Intel Developer Forum (IDF) 2015 [111].

6.1 Outcome Summary

This section outlines the main outcomes and results of the experiments in this work.

Chapter 3 showed how SHEPARD can determine a task’s execution time per device, based on dataset size, allowing it to accommodate workloads which perform best on different devices depending on data size. This chapter also demonstrated how SHEPARD can accommodate multiple tasks and allocate them effectively, outperforming a variety of static allocations.

Chapter 4 introduced plug-in tasks and described further experiments using the delta merge operation with a data table of 70 million rows based on real HANA customer data. Results showed that even under a fully utilised system, SHEPARD imposed no detectable overhead on task runtimes. SHEPARD’s scheduling mechanism was also able to outperform static and round-robin allocation strategies, while accommodating tasks that could only be performed on a subset of available devices.

Chapter 5, described the application of SHEPARD to analytic workloads within the HANA IMDB and examined the true acceleration achievable in
a large database. Despite the overheads of database copy operations and having to queue simultaneous tasks, SHEPARD could still achieve good average speed-ups for user queries. For 100 users, query performance doubled on average. For large datasets, some queries, such as K-Means, could achieve up to 12.65 times speed-up even when run with 10 simultaneous instances of that query. Additionally, by allowing SHEPARD to manage and offload work to accelerators, contention with other database workloads is reduced, resulting in better overall database performance, as evidenced by the SAP-H benchmark execution times. For 100 users, SAP-H ran 14% longer for CPU-only workloads, and only 8% longer when SHEPARD is employed to offload work. For 10 users and large datasets, the most striking result was that SAP-H ran 15% longer for Linear Regression, and only 2% longer when SHEPARD manages these queries.

6.2 Challenges Revisited

This section reviews the work presented and assesses it in the light of the challenges identified in Chapter 1.

1. Hardware Coupling and Code Dependency

Application development for heterogeneous platforms often results in targeting portions of work to specific devices, creating a hardware dependency. This work addresses this through the SHEPARD framework which separates device specific codes from high-level application code.

Initial work used a compilation step to introduce costs and management
6.2 Challenges Revisited

logic. This incurred minimal additional overhead to the development process. However, when implementations were added or cost models altered, the application needed to be recompiled: this presented an unwelcome overhead on the process that may not be suitable for all deployment scenarios.

To overcome this, the ability to use plug-ins to store implementations was created. This removed any need for recompilation when implementations or device configurations changed. Plug-ins are simply shared libraries that can be dynamically loaded at run-time, which means that creating plug-ins for the repository is a simple process.

This approach has proven to be quite flexible, allowing a broad set of devices to be managed, including OpenCL supported devices as well as non-OpenCL devices such as FPGAs. Furthermore, the management aspect of the framework allows tasks to be placed only on devices which have supported implementations, as shown for the delta merge experiments where the FPGA could only support integer data columns.

This flexibility effectively allows an application to remain portable, as long as at least one implementation of each task is available. The presence of a decoupled repository of implementations also means that device specific codes can be maintained, added to or optimized without requiring high-level application codes that use them through managed tasks to be changed.

The requirement to populate a repository constitutes the cost of taking this approach. Populating a repository requires administrative overhead to manage and maintain, and adds an additional step to installing an application onto any system. However, this installation step of registering implementa-
tions can largely be automated and therefore, should not become a significant overhead.

2. Overhead of Coding for Heterogeneous Platforms

Targeting devices such as accelerators often requires low level device specific codes, as well as additional management code to set up the device and transfer data and work to and from the device.

For managed tasks, the application developer composes a task using OpenCL kernels and supplies extra information such as the work size of the OpenCL kernels. This presents only a small overhead as this information is easily obtained. Kernel implementations, supplied by expert programmers, are managed by SHEPARD, thus hiding low level code from the application developer. Since OpenCL implementations can run on many devices, though perhaps not optimally, a single implementation can suffice until it is determined that a more optimal implementation is required.

Using OpenCL kernels allows for common operations, such as sort and merge, to be combined into custom tasks by the application developer. The initial delta merge experiments demonstrated the decomposition of a sequential C++ algorithm a managed task. However, the experiments also highlighted that this transformation may not be entirely straightforward, as some additional steps were required to convert the sequential approach into a parallel one.

Therefore, the overhead of using OpenCL calls to create managed tasks means that this approach may be limited to simple tasks, or those that contain only a few steps. Plug-in implementations avoid this by hiding the
entire task implementation from the application developer. This places more burden on the expert programmer, but allows for much more complex tasks to be made available as simple calls to the application programmer. Plug-ins also have the benefit of being able to support non-OpenCL codes. This comes at the trade-off of requiring tasks within plug-ins to be individually profiled, rather than estimating task costs from constituent kernel and memory transfer costs. This means that creating cost models for plug-ins creates more overhead for the expert programmer than managed tasks composed using OpenCL calls.

While providing managed tasks through plug-ins is more convenient for the application developer, they are not as flexible as using OpenCL kernels. Whereas multiple OpenCL kernels could be combined into a single task by the application developer, and therefore executed as a single operation on a device, tasks within plug-ins cannot be combined and must be executed separately.

From the application developer’s point of view, managed tasks are effective at simplifying executing device specific implementations as they remove any need to develop management code and any low level device codes. Device expert programmers are tasked with making their implementations available through OpenCL kernels or plug-ins. Therefore, the main overhead of utilising SHEPARD managed tasks is not placed on the application developer, but upon expert programmers and system administrators who must maintain the SHEPARD repository.
3. Diverse Nature of Performance on Devices

The third challenge considered is that different types of processing resources will have different levels of performance for tasks.

To address this the concept of costs was introduced which was used to estimate task execution times. For the tasks studied, the execution time increased linearly with the size of the input dataset, and so linear expressions proved adequate to create reliable costs. However, it is likely that many types of tasks will not scale in a linear fashion. As a result, it is likely that SHEPARD will need to support a number of more complex cost models in the future.

Initial experiments demonstrated how SHEPARD could use costs to adequately place a single task on the fastest device. Further experiments showed that SHEPARD’s allocation could handle multiple tasks as evidenced by the fact that, for the delta merge, SHEPARD’s task allocation matches or beats the performance of a static allocation which has prior knowledge of the performance of each task.

The overall allocation strategy of SHEPARD utilised two simple mechanisms; task costs and device queues to measure current device demand. In the experiments carried out, the combination of these mechanisms achieved good results. For example, when 100 HANA users executed analytic queries, SHEPARD effectively load balanced workload across all devices, without any single device becoming a bottleneck.

However, there remain some issues in the current work. To derive cost models, task implementations need to be profiled in advance. Depending on the
number of implementations available, it could take a significant time to profile all implementations adequately. Therefore, there may be the need to start with minimal or even no prior profiling observations and have SHEPARD speculatively execute and collect task measurements in a live system.

Finally, SHEPARD allocates tasks to devices as they arrive and does not prioritise any tasks. However, there may be situations where lower priority tasks can give way to high priority tasks, and longer running tasks can allow some very short tasks to execute first. Additionally, there may also be situations where a task suffers a large penalty when it is unable to execute on its preferred device, and in this case, other tasks that do not suffer large penalties when allocated to different devices could be moved to accommodate this.

4. Shared use of Accelerators

Many acceleration technologies currently available do not always support execution of work from different processes. The SHEPARD framework enables device sharing by representing devices using a task queue.

As already discussed, SHEPARD places tasks on device queues according to task costs and current device demand. This avoids any single device bottlenecking the system while other resources idle. Experiments studying analytic queries in the SAP HANA IMDB highlight this, particularly for 100 users where all devices were utilised throughout the experiments and task times per device were largely equal throughout.

Experiments utilising analytic queries executed by 100 users in HANA showed that device queues can grow large in SHEPARD. For longer running tasks
this resulted in a small proportional increase in overall execution time. However, for short tasks, time spent waiting in the device queue represented a large proportional increase in execution time. Therefore, scenarios that contain a mix of such tasks could benefit if additional processing was performed on SHEPARD’s device queues to promote short running tasks that may be more sensitive to the latency induced by queueing.

The experiments involving analytic queries showed that utilising acceleration in an IMDB can achieve large speed-ups on shared devices versus multi-threaded CPU implementations. Results also showed that by offloading work, the IMDB was better able to perform other queries on the system, due to reduced contention for the host CPU. These experiments utilised the SAP-H benchmark, which is designed to perform complex queries similar to those likely to be encountered in a real HANA system. Additionally, the analytic workloads are based on algorithms present in the PAL library which is supplied by SAP, and therefore these algorithms are likely to be performed on an actual SAP HANA system. However, there are no concrete customer case studies available for such workloads. Therefore, the experimental scenarios could be improved in future if these scenarios can be based on actual customer usage.

5. Application of Managed Acceleration to Enterprise Applications

As discussed previously in the background chapter, much work is focused on the HPC domain and less so on enterprise applications. To address this challenge this work used the SAP HANA IMDB to create experimental scenarios based on a real enterprise environment. Experiments included the mainte-
nance operation, \textit{delta merge}, as well as performing analytic queries within the database.

An IMDB is a good example of an enterprise application that can benefit from managed acceleration since it can have multiple simultaneous users executing diverse workloads, many of which may be analytic queries that can place significant demand on processing resources.

The delta merge scenario involved a standalone application that modelled the actual delta merge operation. This meant that SHEPARD could easily be applied to an isolated application code base.

The analytic queries were actually performed within the IMDB, and thus necessitated the use of SHEPARD within an existing product and all the code and dependencies that accompany it. This scenario showed the true value of SHEPARD’s approach to decoupling device specific implementations from application code. Within the IMDB code, the application developer does not need to insert any device specific code: they only need to use managed tasks. The managed tasks can be called using a few lines of code and the database can then be deployed. Typically, in this scenario, if device specific implementations need to be added or changed the database code would need to be altered and recompiled; the database would then need to be restarted or redeployed depending on which modules need to be changed.

Using SHEPARD, implementations are registered to the repository and loaded at run-time. This means that a deployed database does not need to be recompiled when device specific implementations are added. This speeds up development and deployment times for the added database functionality.
Since the SHEPARD runtime executes as an external daemon it can receive requests from different processes. For an IMDB this means different users and database processes can simultaneously call managed tasks and SHEPARD can orchestrate their execution. There is no explicit integration needed with the database code beyond calling managed tasks as the runtime exists externally to the database processes. Therefore, SHEPARD could easily accommodate both requests from the IMDB and even other applications that may also run on the same hardware platform.

However, despite the focus on the SAP HANA IMDB in this work, SHEPARD is capable of working with any application that could benefit from acceleration. Therefore, one aspect where this work can be strengthened is by broadening the range of applications that SHEPARD is applied to, beyond In-Memory Databases alone.

### 6.3 Discussion & Future Work

This work has examined two approaches to facilitating managed tasks, combined with a cost approach and allocation management runtime, and shown that applications can dynamically share, and make use of, multiple heterogeneous processing devices. However, there are a number of areas where further work can be done to advance and improve certain aspects of the approach.

The SHEPARD approach can be decomposed into a number of distinct modules and a discussion of where each part can be further investigated is presented here.

**Cost Metrics**
The idea of a cost was introduced as an abstract concept that quantifies some aspect of a task or kernel’s execution. In this work costs were used to describe execution time and thus the goal was to minimise the turnaround time of a task. Power efficiency has become a hot topic in the industry and therefore, power usage could be used as an alternative cost metric, making minimising energy usage the goal of task allocation. In the cloud space, resources are often charged based on usage or time, and so monetary cost could also be a viable cost metric, meaning applications may wish to have their tasks allocated to all devices, but within a financial budgetary constraint. Having multiple cost metrics can further enhance the flexibility of applications, allowing them to, for example, minimise for low power when turnaround time is not critical. This then introduces the issue of trade-offs, which require more complex performance models.

**Task Allocation Strategies**

Coupled with the task cost metrics, the allocation strategy is another area than can benefit from further research. This study examined workloads where tasks have no priority and were received in an ad-hoc manner, with tasks being immediately placed on the queue of the device that could yield the lowest turnaround time. In other scenarios tasks may be submitted in batches, or if not time critical, some buffering of tasks may be enabled to allow tasks to be pre-processed before allocating them to devices.

The existence of multiple cost metrics can also allow for more dynamic allocation strategies, such as sending tasks to the cheaper (monetary) or more energy efficient device in the case when execution time is similar for multiple
devices.

**Cost Representation & Derivation**

The current means of representing costs employs linear expressions. This has been sufficient for the cases studied in this work, allowing sensible decisions to be made. However, the costs are assumed to be provided or derived from observations of actual task executions on the available devices. While this fits with the ethos that implementations can be supplied with costs by third parties, it is unlikely that every type of device model can be profiled ahead of time. Therefore, a more flexible system that can adjust and re-learn costs over time could be beneficial and improve allocation decisions, provided the overhead of running such a system is not great. One possible solution is to dynamically recalculate costs when the system is not under high levels of load, or to stop recalculating costs for implementations whose performance is constant or exhibits low variation from existing costs.

**Auto-Tuning**

A popular area of research concerns auto-tuning of kernel launch parameters for OpenCL, such as in work by Spafford et al. [37], which has been shown to improve kernel performance portability for some systems. This could be a good addition to the existing solution, allowing kernels and tasks to be provided with good general case configurations, but allow SHEPARD to perform auto-tuning to speculatively improve performance on the platform on which the task runs and updating cost data accordingly.

**Co-Execution of Tasks on Devices**

This work has used a single queue to model each device, allowing demand on
the device to be estimated and preventing tasks from adversely interfering with one another. However, as device architectures evolve, they are becoming better able to accommodate execution of simultaneous workloads.

Moulding kernels to limit their resource usage to allow two or more kernels to execute simultaneously on OpenCL devices has been investigated in [79, 82]: this line of work may merit investigation in the SHEPARD context.

**Resiliency & High Availability**

In current hosted environments, providers often guarantee certain levels of availability and therefore mechanisms are in place such that if hardware fails, a backup will be available to allow applications to recover and continue executing. Since SHEPARD manages allocation of tasks and has knowledge of multiple device implementations, it can be well placed to allow tasks to recover if devices fail, by re-allocating the task to a different device. This could be a useful addition to the solution posited here and help make applications not only more dynamic and portable, but more robust and failure resistant.

**Cloud**

The cloud is another hot topic for research and introducing heterogeneous technologies into this environment is important for the industry. Toward achieving this, the output of this work is contributing to the HARNESS EU project, as highlighted previously. The addition of SHEPARD to a full cloud stack will allow applications to be dynamically deployed to various hardware configurations and take advantage of available compute resources.

**Distributed Tasks**

The current investigation has concerned only devices local to the same node.
This can be expanded to investigate allocation across shared network devices, particularly in cloud environments.

To conclude, this thesis has presented an approach to enabling applications to exploit the benefits of heterogeneous processing environments. Decoupling the device specific implementations from applications allows high level developers to be freed of the burden of low level device development and allows applications to remain portable across various platforms. Additionally, the runtime aspect of the work allows allocation decisions of tasks to devices to be made, invisible to the developer. This allows multiple applications to share devices in an effective manner.

The field of heterogeneous processing is a vast area of research and from the discussions and future work, there are clearly many more opportunities and challenges to be tackled. Heterogeneous processing offers tremendous benefits and therefore it is important that these opportunities are not ignored by enabling applications to leverage the power and flexibility that these technologies provide.
Part V

Appendix
Listing A.1: Comparison of OpenCL program creation using native library and custom wrapper library

```cpp
// contains required headers for OpenCL or OpenCL Wrapper Library
#include "MinimalExample.hpp"

/*** Begin OpenCL Wrapped Library Code ***/

using namespace oclTools;

void OpenCL_Using_Wrapper(float *hostData1, float *hostData2,
                           float *hostData3, float *hostData4, float *resultData,
                           size_t dataBytes, size_t dataElements)
{
    try {
        // Get OpenCL Objects
        oclHelper helper("Minimal Example");
        oclContext context = helper.getPlatforms()[0].createContextAll(0);
```
oclQueue queue(context, context.getDevices()[0], 0);

// get program and kernel
oclProgram program = context.createProgram("program.cl");
oclKernel kernel = pro.getKernel("computeKernel");

// Create Memory
oclMem memIn1 = context.createMemory(dataBytes, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, hostData1);
oclMem memIn2 = context.createMemory(dataBytes, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, hostData2);
oclMem memIn3 = context.createMemory(dataBytes, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, hostData3);
oclMem memIn4 = context.createMemory(dataBytes, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, hostData4);
oclMem memOut = context.createMemory(dataBytes, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, resultData);

// set kernel parameters
setParams(kernel, memIn1, memIn2, memIn3, memIn4, memOut);

// execute kernel
queue.enqueueNDRangeKernel(kernel, 1, 0, &dataElements);

// read result into host buffer
queue.enqueueReadBuffer(mem, CL_TRUE, 0, dataBytes, hostData);

} catch (oclError &err) {
    std::cout << err.getCLError() << std::endl;
}

/**
** End OpenCL Wrapped Library Code
**/

/**
** Begin Native OpenCL Code
**/

// OpenCL Objects
cl_int error;
cl_platform_id platform;
cl_device_id device;
cl_uint platforms, devices;
```c
cl_mem memIn1, memIn2, memIn3, memIn4, memOut;
cl_kernel kernel;
cl_command_queue queue;
cl_program program;

cl_int OpenCL_Native(float *hostData1, float *hostData2,
                      float *hostData3, float *hostData4, float *resultData,
                      size_t dataBytes, size_t dataElements)
{
    // Get OpenCL Objects
    error = clGetPlatformIDs(1, &platform, &platforms);
    if (error != CL_SUCCESS)
    {
        printf("Failed to get platforms: error code %d", error); return error;
    }
    error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, &devices);
    if (error != CL_SUCCESS)
    {
        printf("Failed to get devices: error code %d", error); return error;
    }
    cl_context context = clCreateContext(0, 1, &device, NULL, NULL, &error);
    if (error != CL_SUCCESS)
    {
        printf("Failed to create context: error code %d", error); return error;
    }
    queue = clCreateCommandQueue(context, device, 0, &error);
    if (error != CL_SUCCESS) {
        printf("Failed to create command queue: error code %d", error); return error;
    }

    // load program source
    std::ifstream t("program.cl");
    std::string *source = new std::string((std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>(()));
    const char *cc = source->c_str();
```
// create program from source
program = clCreateProgramWithSource(context, 1, (const char **) &cc, NULL, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create program: error code %d", error);
}

// compile the program
error = clBuildProgram(prog, 0, NULL, "", NULL, NULL);
if (error != CL_SUCCESS)
{
    printf("Failed to build program: error code %d", error);
    printf("\n Error number %d", error);
    fprintf(stdout, "\nRequestingInfo\n");
    clGetProgramBuildInfo(prog, devices, CL_PROGRAM_BUILD_LOG, 4096, build_c, NULL);
    printf("Build Log for %s_program: \n%s\n", "example", build_c);
    return error;
}

// Get kernel
kernel = clCreateKernel(prog, "computeKernel", &error);
if (error != CL_SUCCESS)
{
    printf("Failed to get kernel: error code %d", error);
    return error;
}

// create memory
memIn1 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, dataBytes, hostData1, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create memory buffer: error code %d", error);
    return error;
}
memIn2 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, dataBytes, hostData2, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create memory buffer: error code %d", error);
    return error;
}
memIn3 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, dataBytes, hostData3, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create memory buffer: error code \%d", error); return error;
}

memIn4 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, dataBytes, hostData4, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create memory buffer: error code \%d", error); return error;
}

memOut = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, dataBytes, hostResultData, &error);
if (error != CL_SUCCESS)
{
    printf("Failed to create memory buffer: error code \%d", error); return error;
}

/* Set the kernel parameters */
error = clSetKernelArg(kernel, 0, sizeof(memIn1), &memIn1);
if (error != CL_SUCCESS)
{
    printf("Failed to set kernel parameter 0: error code \%d", error); return error;
}

error = clSetKernelArg(kernel, 1, sizeof(memIn2), &memIn2);
if (error != CL_SUCCESS)
{
    printf("Failed to set kernel parameter 1: error code \%d", error); return error;
}

error = clSetKernelArg(kernel, 2, sizeof(memIn3), &memIn3);
if (error != CL_SUCCESS)
{
    printf("Failed to set kernel parameter 2: error code \%d", error); return error;
}

error = clSetKernelArg(kernel, 3, sizeof(memIn4), &memIn4);
if (error != CL_SUCCESS)
{
    printf("Failed to set kernel parameter 3: error code \%d", error); return error;
}

error = clSetKernelArg(kernel, 4, sizeof(memOut), &memOut);
if (error != CL_SUCCESS)
{
printf("Failed to set kernel parameter 4: error code %d", error); return error;
}

// execute kernel
error = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &dataElements, NULL, 0, NULL, NULL);
if (error != CL_SUCCESS)
{
    printf("Failed to execute kernel: error code %d", error); return error;
}

// read the result
error = clEnqueueReadBuffer(queue, memOut, CL_TRUE, 0, dataBytes, resultData, 0, NULL, NULL);
if (error != CL_SUCCESS)
{
    printf("Failed to read result data: error code %d ", error); return error;
}

ReleaseOpenCLObjects();

//release OpenCL objects
if (memOut) clReleaseMemObject(memOut);
if (memIn4) clReleaseMemObject(memIn4);
if (memIn3) clReleaseMemObject(memIn3);
if (memIn2) clReleaseMemObject(memIn2);
if (memIn1) clReleaseMemObject(memIn1);
if (program) clReleaseProgram(program);
if (kernel) clReleaseKernel(kernel);
if (queue) clReleaseCommandQueue(queue);
if (context) clReleaseContext(context);

/**
** Finish Native OpenCL Code
**/

int main(int argc, char **argv)
{
    float *in1 = new float[1024];
    float *in2 = new float[1024];
    float *in3 = new float[1024];
    float *in4 = new float[1024];
    float *out = new float[1024];
// create dummy data
for (int i = 0; i < 1024; i++)
{
    out[i]=0;
    in1[i] = i;
    in2[i] = i * 2;
    in3[i] = i * 3;
    in4[i] = i * 4;
}

size_t dataBytes = 1024*sizeof(float);
size_t dataElements = 1024;

// Native OpenCL C execution of kernels
OpenCL_Native(in1, in2, in3, in4, out, dataBytes, dataElements);
ReleaseOpenCLObjects(); // explicit release of objects must be handled

// Execution of kernels with OpenCL wrapper library
OpenCL_Using_Wrapper(in1, in2, in3, in4, out, dataBytes, dataElements);

return 0;
Bibliography


[21] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, “Compiling a high-level language for GPUs: (via language support for archite-


[77] Peer 1 Hosting UK, “Peer 1 Hosting UK Managed GPU Servers.” [Online]. Available: http://www.peer1hosting.co.uk/hosting/gpu-servers


[101] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund et al., “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput com-


