Communication in task-based runtimes for heterogeneous systems

Ruben Cano Diaz (rcanodia@bsc.es)

Advisor: Carlos Álvarez Martínez (calvarez@ac.upc.edu)

Co-Advisor: Daniel Jímenez (djimenez@ac.upc.edu)

June 23, 2020
Acknowledgement

I would like to thank my thesis advisor Carlos, for his guidance and infinite patience, to Dani, my Co-advisor that, although it has a tight schedule, he is always willing to make a slot if you need it, to Xavi and Vicenç, which helped me a lot of times, to my colleagues at BSC, I can’t wait to return to the office to meet you again!, to Antonio, Juanmi and Jaume, for all the support and help they provided me, and, finally, to my family and partner, you make me feel loved and are the reason I wake up in the mornings.
HPC machines in the race for exascale computing are more heterogeneous than ever. The complexity of the systems makes the programming and interoperability of the multiple accelerators a huge task that most of the times the programmer must handle directly. OmpSs-2 Programming model aims to help the programmer handle the heterogeneous systems hiding all the data communication and synchronization efforts and making offloading tasks to different devices easier. This master thesis explores some ways to handle this problems and proposes a framework to add hardware support to OmpSs-2 runtime, without relying in architecture-specific functionalities like unified-memory. The results show that the framework and implementation proposed can increase the performance of the applications when compared with previous versions of the runtime, and is able to easily support the addition of new devices to the OmpSs environment.
# Contents

1 Introduction ........................................ 1
   1.1 Objectives ........................................ 1
   1.2 Previous Publications .............................. 2

2 Background ........................................... 3
   2.1 The OmpSs-2 Programming Model ..................... 3
      2.1.1 Basics ........................................ 4
      2.1.2 Polling Services ................................ 6
      2.1.3 Dependences .................................... 6
      2.1.4 Range Dependences ................................ 7
      2.1.5 Symbol concept .................................. 8
      2.1.6 Reference Implementation ......................... 9
         2.1.6.1 Device support ............................... 9
   2.2 Heterogeneous Systems ............................... 10
      2.2.1 Accelerators .................................... 11
      2.2.2 DMA Engine and IOMMU .......................... 12
         2.2.2.1 DMA ......................................... 12
         2.2.2.2 IOMMU ........................................ 13
      2.2.3 Unified Virtual Addressing ....................... 14
   2.3 CUDA .............................................. 14
      2.3.0.1 CUDA in multi-GPU scenarios .................. 14
      2.3.0.2 Synchronous vs Asynchronous ................... 15
      2.3.0.3 CUDA Stream model ............................ 16
      2.3.0.4 CUDA Distributed Memory Management .......... 17
      2.3.0.5 What kind of job is expected on GPGPU computing? 17
   2.4 OpenCAPI .......................................... 17
   2.5 Xtasks ........................................... 18
      2.5.1 Xtasks Accelerators .............................. 18
      2.5.2 Xtasks Memory Allocations ......................... 19
      2.5.3 Xtasks Memory Management ......................... 19
      2.5.4 Xtasks Tasks ...................................... 20
   2.6 Vogel ............................................... 21
## Contents

### 3 OmpSs-2 contributions

- 3.1 Support for devices with distributed memory ........................................ 22
- 3.2 Support for CUBLAS CUDA tasks ............................................................. 22
- 3.3 Support for FPGA Tasks ................................................................. 23

### 4 Design and implementation

- 4.1 Strategy ................................................................................................... 24
- 4.2 Implementation Overview
  - 4.2.1 Accelerator ......................................................................................... 25
  - 4.2.2 Directory ............................................................................................ 26
- 4.3 Address Spaces ......................................................................................... 26
- 4.4 Entries Interval Map
  - 4.4.1 Creating new entries and partitioning new ones ................................. 28
- 4.5 Directory .................................................................................................. 29
  - 4.5.1 Directory API ..................................................................................... 29
  - 4.5.2 Registering address spaces ................................................................. 30
- 4.6 Directory Entry ......................................................................................... 30
  - 4.6.1 State of an entry .................................................................................. 30
- 4.7 The representation of a symbol ................................................................. 31
- 4.8 Streams ...................................................................................................... 32
  - 4.8.1 Reference Implementation of a Stream for any device ......................... 32
- 4.9 Registering a task
  - 4.9.1 Partitioning ......................................................................................... 34
  - 4.9.2 Allocation ............................................................................................ 35
  - 4.9.3 Reallocation ........................................................................................ 35
  - 4.9.4 Processing Symbol Regions ................................................................. 36
- 4.10 Ready-task dependences and execution workflow
  - 4.10.1 Synchronization mechanisms .............................................................. 38
  - 4.10.2 Device Workflow ................................................................................ 39
  - 4.10.2 Accelerator Loop ............................................................................... 41

### 5 Evaluation and results

- 5.1 Environment ............................................................................................... 43
  - 5.1.1 FPGA ................................................................................................. 43
  - 5.1.2 CUDA ................................................................................................. 44
- 5.2 Matrix Multiply
  - 5.2.0.1 Analysis ......................................................................................... 44
  - 5.2.1 FPGA Matrix multiply ......................................................................... 47
- 5.3 Conjugate Gradient
  - 5.3.1 GPU Performance .............................................................................. 51
  - 5.3.1.1 Execution Traces ............................................................................ 51
  - 5.3.2 Performance ......................................................................................... 53
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Conclusions and future work</td>
<td>54</td>
</tr>
<tr>
<td>7 Annex</td>
<td>55</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Release-line of features exported from OmpSs into OpenMP ........... 4
2.2 Nanos6 Polling Service API, it allows to register and unregister a service function .................................................. 6
2.3 Matrix of size N divided in blocks of size N/2, each block is represented with a different colour ........................................... 8
2.4 Non-contiguous dependences generated for the first block, in this case, the end of the first line is not followed directly by the second line, but there is an offset of N/2 elements between them ....................... 9
2.5 Diagram that shows a possible interconnection between memory, devices an the CPU .......................................................... 12
2.6 The same diagram as shown in figure 2.5 but now the CPU has two cores and a memory hierarchy ........................................... 13
2.7 CUDA API calls that serves to manage the active default context for each GPU ................................................................. 15
2.8 Execution of an Async task as seen as the GPU .......................... 15
2.9 Execution of an Async task as seen as the GPU .......................... 15
2.10 Representation of a CUDA Stream for one GPU, there can be multiple streams in parallel, as seen in the diagram, but each stream can run in parallel. ................................................................. 16
4.1 Different paradigms on partitioning the regions of the directory, at the left, we can see: In green, the existing region, in orange, the region that is being registered. At the right, we can see en blue, the new regions that have been created due to the partitioning, in orange, the new regions and in green the preexisting regions ........................................... 28
4.2 From left to right, the process of partitioning over a complex region step by step. Over the preexisting region, there is the orange region that it’s being registered, orange regions are new regions and blue regions mark a new registered region that is partitioned from a preexisting one. ... 28
4.3 Graphical Overview of the directory contents ............................ 29
4.4 Table that represents the data that it’s stored inside a symbol ....... 31
4.5 A streamable operation is an activation function which executes some asynchronous code and generates another function that checks for termination of the first one.  
4.6 Representation of the flow graph that an operation follows when adding a new operation, if there is already another operation ongoing, it’s enqueued, if not, we activate the operation and set as active the checker function.  
4.7 Flow graph of the Try Continue operation, this operation must be called multiple times while there are operations pending, it checks for the finalization of the checker function, if its finished, tries to get a new operation, if there is no operation, it ends.  
4.8 Graph that represents the states of an entry for a device, at the left, we have the actions that the device can do in order to jump between states, at the right, we can see actions that a different device can make that affect the state of the data in the device.  
4.9 Flow graph that a symbol follows when is declared as an Input  
4.10 Flow Graph that a symbol follows when it’s declared as an Input/Output  
4.11 Flow graph that a symbol follows when it’s declared as an Output  
4.12 Example of workflow steps where A is the root step, E is the last step, and the two paths (C and B,D) must finish before E can be executed.  
4.13 Directory synchronization  
4.14 Diagram that shows how the different synchronizations methods interacts when a task passes through the directory  
5.1 Performance (in GFLOPS) for matrix sizes from 2k to 16k, at different block sizes.  
5.2 As we can see, using unified memory without any kind of prefetching results in bad results for performance in all configurations (yellow line), probably due to the page-faults on the GPU. Unified Memory with prefetching (red line), seems to be competitive in the configurations (a) and (b), however, when we decrease the matrix size, it seems to decrease in performance (c) and (d), the directory approach (blue line) seems to be the one that performs better in most configurations.  
5.3 These charts show the speedup achieved using the Directory version instead of the Unified Memory with prefetching, as we can see in (a) and (b), when working with low granularity, it seems to be more noticeable the performance gain, however, using the directory version gives better performance in almost any case, noticing in the 8K matrix, a slight drop in performance when the block is the full matrix. In (c) the performance gain seems to be quite stable around 2x and in (c), when working with small matrices, the speedup goes up almost at 7x.  
5.4 Table that shows the performance in GFLOPS for Nanos5 and Nanos6 with the different configurations.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Performance plot where we can see that, while Nanos5 has greater performance, Nanos6 catches up when the problem size increases.</td>
</tr>
<tr>
<td>5.6</td>
<td>Conjugate Gradient implementation where the namespace before the name of the function specifies where that function is going to run. In this case, all dot product functions have been offloaded to GPU. In order to avoid synchronization, some helper tasks like CPU::divide and CPU::set serves to store the result of the division in the first element and set the second element in the first respectively.</td>
</tr>
<tr>
<td>5.7</td>
<td>Unified memory trace of a dot product in CUDA where the kernels take up to 1 second to execute.</td>
</tr>
<tr>
<td>5.8</td>
<td>The trace (a) corresponds to the Directory, the trace (b) corresponds to prefetching, as we can see, both share a lot of similarities in the execution of the kernels, but, while (a) describes the copies explicitly, (b) generates prefetching, which is not represented on the stream timeline.</td>
</tr>
<tr>
<td>5.9</td>
<td>Execution time for a Conjugate Gradient with 1000 iterations (less is better), as we can see, the Directory outperforms any other configuration in each scenario.</td>
</tr>
<tr>
<td>5.10</td>
<td>Execution time for a Conjugate Gradient with 1000 iterations (less is better), as we can see, the Directory outperforms any other configuration in each scenario.</td>
</tr>
<tr>
<td>7.1</td>
<td>SMP Part of the code that performs the execution of a matmul using CUDA.</td>
</tr>
<tr>
<td>7.2</td>
<td>CUDA Kernel from the University Of Colorado that performs a matrix multiplication over a block, annotated with OmpSs-2 pragmas in order to execute it with Nanos6 runtime.</td>
</tr>
<tr>
<td>7.3</td>
<td>Saxpy implementation for CUDA.</td>
</tr>
<tr>
<td>7.4</td>
<td>Dot product implementation for CUDA.</td>
</tr>
<tr>
<td>7.5</td>
<td>Scale vector implementation for CUDA.</td>
</tr>
<tr>
<td>7.6</td>
<td>Saxpy implementation.</td>
</tr>
<tr>
<td>7.7</td>
<td>Dot product implementation.</td>
</tr>
<tr>
<td>7.8</td>
<td>Scale Vector implementation.</td>
</tr>
<tr>
<td>7.9</td>
<td>Sparse matrix Vector multiplication implementation.</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

As the race towards exascale computing continues, in order to achieve the best performance possible, new heterogeneous systems with specialized accelerators are being popularized. Such systems are difficult to program and need careful data movement planning between the host SMP and the attached accelerators to deliver good performance. While the state-of-the-art relies on hardware-mechanisms with accelerator-aware memory coherence, the adoption of these in already-existing accelerators is not feasible, and a standard protocol must be enforced between all devices, which can prove difficult. In this work we propose new software methods to extend task-based runtimes for heterogeneous systems with better communication capabilities that alleviate the programmability problem while maintain or even improve the performance.

1.1 Objectives

The main objective of this master thesis is to create a framework that can be used in order to control the coherence and data management between multiple devices, even if they lack of hardware support for it.

As we create the framework, we will introduce two device implementations that serve as a demonstration of the capabilities of the system, FPGA devices and CUDA\[10\] devices. These implementations will be done on Nanos6\[12\], a runtime that implements OmpSs-2 programming model\[4\]

- **FPGA support**: inheriting FPGA compilation from OmpSs@FPGA\[5\] on Nanos5\[11\], we will for the first time run FPGA tasks on Nanos6 with range dependences.

- **GPU support**: Although there is already support for CUDA tasks, only unified memory is supported, with our implementation, the runtime will manage the memory directly, which can have a performance impact.

The final objective will be to compare the performance of applications that run on the same base runtime, but with the different possible approaches. In CUDA,
CHAPTER 1. INTRODUCTION

will benchmark unified memory, unified memory with prefetching and our approach. In FPGA, since there is no previous mature support for Nanos6, we will compare the implementation with the Nanos5 runtime one.

1.2 Previous Publications

An abstract for this thesis was presented at the BSC Doctoral Symposium 7th edition\textsuperscript{2}.
Chapter 2

Background

2.1 The OmpSs-2 Programming Model

OmpSs-2 is a task-based programming model that evolves from OmpSs. It provides parallelism to shared-memory applications in C/C++ and fortran, and is being developed at the Programming Models group in the Barcelona Supercomputing Center [3]. OmpSs evolves from the effort to integrate features from a previous programming model developed at BSC called StarSs [6], which shares the same principles and objectives. Create a programming model that has its parallelism enabled by the use of data-dependences between the different tasks of the program and is able to manage different heterogeneous accelerators integrated in the same model. OmpSs programming models aim at being a forerunner of the OpenMP [9] standard.

To achieve that, the OmpSs-2 model uses code annotations in the source which doesn’t affect the semantics of the program. This lets the compiler generate the code which will enable the parallelism. This has the advantage that, you can write the code once, and if you were to compile the code in another environment and compiler, the code will still work without any dependence to the programming model library.

OmpSs-2 model uses a thread-pool execution model, which means that there are a set number of threads that can be executing tasks at the same time, and the threads are reused instead of creating a new thread every time, thus reducing the overhead of creating and destroying a thread. At a given moment, all the threads can be idle or executing a runtime-service or executing a ready-task.

The main functionality of OmpSs-2 is the way that it performs and expresses the parallelism. The parallelism is expressed in the form of data dependences and the task creation order. This mechanism enables look-ahead instantiating, which means that the CPU can create all the tasks and continue doing useful work, or instantiating more tasks, and it eases the parallelization to the programmer, since the programmer only
has to care about which data it’s going to be needed for each task, which usually are implicit on the parameters of the function.

For this, the preferred synchronization between tasks is done via data-dependences instead of waiting for all tasks to finalize with a taskwait, only waiting for the data to be available frees the CPU from unnecessary waitings, and expresses better the intent of the code.

Apart from being a competent programming model, OmpSs-2 not only has the intention of being one of their own, but to have its features introduced to OpenMP Standard, extending OpenMP’s API in order to better support asynchronous data-flow parallelism and heterogeneity in OpenMP.

![Figure 2.1: Release-line of features exported from OmpSs into OpenMP](image)

Some of the contributions of the OmpSs programming model that are already part of the OpenMP standard include:

- Task dependences (included in OpenMP 4.0)
- OpenMP SIMD extensions (included in OpenMP 4.0)
- Task priorities (included in OpenMP 4.5)
- Task reductions (included in OpenMP 5.0)
- mutexinoutset dependence type (called commutative in OmpSs-2) (included in OpenMP 5.0)

### 2.1.1 Basics

The most important concept, and the one that all OmpSs-2 model is built around is the task. A task, in essence, is a portion of code that will be run asynchronously when the
conditions for its invocation are met. In OmpSs-2, tasks are created using a compiler directive or annotation, in C/C++, the task construct is defined as:

```c
#define oss task [clause[[], clause] ..]
function
#define oss task [clause[[], clause] ..]
structured block
```

The function or structured block that follows the construct, is the one that will be executed asynchronously.

- `private(<list>)`
- `firstprivate(<list>)`
- `shared(<list>)`
- `depend(<type>: <memory-reference-list>)`
- `<depend-type>(<memory-reference-list>)`
- `reduction(<operator>:<memory-reference-list>)`
- `priority(<expression>)`
- `cost(<expression>)`
- `if(<scalar-expression>)`
- `final(<scalar-expression>)`
- `wait`
- `for [chunksize(<expression>)]`
- `label(<string>)`

Apart from implicit synchronization using data dependences, OmpSs-2 offers a directive called `taskwait` which doesn’t have a code block associated that can be used as a synchronization point.
2.1.2 Polling Services

As cited from the master thesis where polling services where introduced \[13\] The Polling Services is the component in charge of synchronizing asynchronous tasks via polling. This piece of the system is formed by a checking mechanism called by the different SMP threads when idle or before executing a task, as well as an API to register the the data which has to be polled, which in the case of GPU tasks is CUDA events

```c
typedef int (*nanos_polling_service_t)(void *service_data);

void nanos_register_polling_service(
    char const *service_name,
    nanos_polling_service_t service_function,
    void *service_data
);

void nanos_unregister_polling_service(
    char const *service_name,
    nanos_polling_service_t service_function,
    void *service_data
);
```

Figure 2.2: Nanos6 Polling Service API, it allows to register and unregister a service function

The interface (which definition is on figure 2.2) consists of the definition of a nanos_polling_service_t function which describes how a polling service is represented. A polling service is a single function that receives some service specific data and returns an integer. This integer represents whether the service has to be removed from the service list to be run after executing its execution. To compliment this definition a couple of functions to register and unregister services are added.

2.1.3 Dependences

Dependences is the mechanism which OmpSs-2 uses to determine the data flow and parallelism of a program.

When an OmpSs-2 task is created, the runtime environment uses the data dependence associated to the task in order to perform a dependence analysis, which decides which tasks must be executed before and which ones must be executed after.

This data dependence analysis is done by matching the dependences of the entrant task with the preexisting ones, looking for Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read (WAR) dependences. If one of these is found, the tasks
becomes a successor of the corresponding tasks, and will only be ran after all prede-
cessors finish their execution. If no clash was detected between the new task and the
preexisting ones, the task is prepared to be run at creation.

Defining a data dependence over a task it’s the same as notifying to the runtime
which is the intent (read, write...) for each region of data that the task will potentially
use.

Another important factor of the dependency-system, is the dependence-scope. Two
tasks can have a dependency between them, if and only if, they share the same father
(Sibling Task), if it’s not the case, for example, a father that has a dependency over
a big region of memory, creates minor tasks that depend on that region, the runtime
will not mark the children tasks as successors of the father task.

The basic types of dependency are:

- **in**: Specifies that the data must be available for reading, this enforces a depen-
dency if there is a previous out/inout task over the same memory reference.

- **out**: Specifies that the data will be overwritten, this enforces a dependency if
there is a previous task over the same memory reference.

- **inout**: Specifies that the data needs to be readable and writable, enforces a
dependency if there is a previous task over the same memory reference.

While there are other three clauses: commutative, concurrent and reduction,
they act similar to an inout dependence, which from our project perspective it’s suf-
ficient.

### 2.1.4 Range Dependences

In OmpSs-2, dependence-clauses act over a region of memory instead of over a single-
position in memory as a default behaviour. The runtime is responsible for computing
the intersection between dependences in order to figure out the dependences between
tasks. Annotating the dependences precisely is one of the fundamental parts of OmpSs-
2 and has the advantage of improving readability and give the runtime sufficient infor-
mation to manage copies of prefetching of data.

There are two ways to enforce a dependence over a memory region:

- **Array Sections** allows to refer to multiple elements inside an array with one
single expression
  
  - A[N]: The dependence will be enforced for the range A[0] to A[N]
  - A[lower:upper]: The dependence will be enforced for A[lower] to A[upper]
  - A[lower; size]: The dependence will be enforced from A[lower] to A[lower + size]
• **Shaping Expressions** allows to cast a pointer into an array type to recover the potential dimensionality lost when passing pointers without the inner array type. These shaping expressions are put before the symbol name \([\text{dim2}...][\text{dim1}]A\)

  – Example: \([n][n]A\) will recast the pointer A into a matrix of nxn in order to compute dependences.

### 2.1.5 Symbol concept

We call a Symbol the data that is going to be used by a task for performing the operations. In that sense, the information needed by the symbols is similar to the information needed by the dependency system, a set of regions that are going to be used.

However, a Symbol is more than a simple range, a Symbol can be a superset of range dependences, where the offset between dependencies matter.

Imagine a matrix of size \(N^*N\), which is blocked in four blocks, each of \(N/2^*N/2\) elements, and that we have an accelerator which performs a matrix multiplication over each block.

![Matrix of size N divided in blocks of size N/2, each block is represented with a different colour](image)

**Figure 2.3:** Matrix of size \(N\) divided in blocks of size \(N/2\), each block is represented with a different colour

If we create a task, which operates over the first block, the dependences would be generated as non-continuous data-accesses:
When working with unified-memory or in SMP, ensuring the read-write order using dependences, is sufficient to ensure the correct execution of this part, however, when we work with devices with discrete memory, it’s not sufficient to ensure that each dependence has an address in the device, but that the offsets between them are maintained.

This is due to the fact that it’s not feasible nor performant to modify the code of an accelerator at runtime to adapt to the new memory layout for each region.

For this reason, a symbol is a set of non-contiguous data-accesses that must be allocated on a discrete memory respecting the offsets between the data-accesses.

### 2.1.6 Reference Implementation

The reference implementation of OmpSs-2 is based on the Mercurium Source-to-source compiler and the Nanos6 Runtime Library.

Mercurium source-to-source compiler provides the necessary support for transforming the high-level directives or annotations into a parallelized version of the application which interfaces with Nanos6. The final executable is compiled with a native compiler of the machine it will run on.

The Nanos6 runtime system library provides services to manage all the parallelism in the user-application, including, the creation of the task, the synchronization and the data movement, and provides support for heterogeneous resources.

#### 2.1.6.1 Device support

Device support doesn’t appear on OmpSs-2 Specification, however, Nanos6 and Mercurium reference implementation has support for CUDA tasks on Unified-memory capable Devices.
To mark a task as a device task, there is an undisclosed pragma:

```
#pragma oss target(<device>)
```

To be able to generate cuda tasks, the cuda portion of code must be an outline-task defined in a header, with the implementation in a separate file, which needs to be compiled with NVIDIA’s CUDA compiler.

When programming using unified memory, there is an architecture restriction, in IBM Power9 machines, tasks can accept any host-addressable memory as a parameter, however, in x86-64 and ARM machines, all the memory regions that will be used in CUDA are required to be allocated using a CUDA API reserved for this use, which registers memory that can be used in both, GPU and CPU.

### 2.2 Heterogeneous Systems

The target heterogeneous architecture for this project is: a CPU fully-capable of running Linux, coupled to an accelerator or set of accelerators which can be used to offload tasks from the main CPU.

The main CPU consists in multiple cores, following a cache-based hierarchy, with each process having their own virtual-address space mapped in physical DRAM. The accelerators may or may not have a private DRAM and memory hierarchy, and share the same DRAM as the CPU via a memory controller.

The cache-based hierarchy is used to keep the most frequent used data and code in near and fast storage, which is key to achieve good performance, and the virtual-addressing in its core is needed to avoid fragmentation of memory (physically-noncontiguous data is seen as contiguous in the virtual address-space), process isolation and other perks.

This two mechanisms of performance and convenience work well in each individual device, since internally there are cache-coherent interconnects and an MMU (Memory Management Unit) which handles the virtual address space. However, when a separate device accesses the DRAM of the system, all accesses are usually done via a DMA (Direct Memory Access) controller, which only handles physical memory.

This means that devices don’t share the same virtual address space as the CPU-Process, may not have coherent memory-accesses to memory that it’s currently cached by the CPU and simple virtual-to-physical address translation won’t work for data that fits in more than one single page per-default, since the pages could be in different physical addresses, and algorithms and data-managing expects them to be contiguous.

To simplify this problem, an IOMMU (Input Output Memory Management Unit) can be placed between the memory and the accelerators, the IOMMU is a mechanism to allow the same CPU-MMU system to devices outside the CPU fabric, providing a virtual-memory system to devices. It can be programmed or share the same page table as a selected process.
While this can solve the data-fragmentation and allows the device to access the same regions as the CPU when referring to the same pointer in a virtual address space, it doesn’t solve the memory-hierarchy problem, in which data in DRAM is not updated. This makes it so that the usual system to fix this problem becomes in registering large contiguous un-paged and un-cached memory regions which ensure that the memory is coherent in DRAM.

This has its drawbacks, since the un-cachable memory is not performant in an SMP system, since reads always have maximum latency. To avoid that, it’s usual to use normally-allocated memory, and copy this data to the un-cachable memory when is going to be used, costing up to double the needed space and making additional work in some systems.

2.2.1 Accelerators

Hardware accelerators is computer hardware made to perform some functions more efficiently (usually exploiting parallelism) that the work that a Central Processing Unit (CPU) can.

When accelerating software, we can perform the operations in a CPU (no accelerator), in a mix between CPU and custom accelerators (Mixed approach) and in accelerators only (fully accelerated). Each approach has its advantages and in order to decide which approach is the best for our application, we must understand the limitations.

To understand this limitations we must take into account:

- The connection between the main processing element and the accelerator
- Shared memory or discrete memory
- Throughput and latency
- Working Size of the accelerator

For example, processors these days often come with a cryptographic accelerator embedded in the sock (on-chip accelerators) that often allows to perform parallel executions, or, instruction extensions that behave like normal instructions.

In the first example, is usual to not have a coherent access between the accelerator and the processor to the memory, due to the existence of a cache, and the processor must ensure that the data is flushed into memory before sending a task to the accelerator, however, if it’s an instruction and it’s tied to the processor, the access to the memory is coherent and following the same mechanism of the running processor.

Here, we can see the firsts design decisions, while in the first case, we can execute in parallel, the setup, flush, latency to get the data, latency to release the data, and check for finalization can create sufficient overhead to make the accelerator slower than if done in software. However, in the instruction approach, we are limiting the processing of the
CPU until these instructions end, but the finalization check and memory movements are faster and implicit.

There is no right answer to when run in an accelerator or not, as in the previous example, if the task that is sent to run in the accelerator is big enough (in time), we can perform useful work in the CPU, overlapping the execution of the hardware-algorithm with the software. However, if the tasks we are going to run are rather small, the CPU Instruction approach may be the best one.

2.2.2 DMA Engine and IOMMU

2.2.2.1 DMA

DMA stands for Direct Memory Access, it refers to memory accesses from devices to physical addresses of the main memory. Each device has its own DMA Controller to perform memory petitions to the Main Memory, this special hardware frees the device from doing busywaiting to get the data since it retrieves it asynchronously.

![Diagram that shows a possible interconnection between memory, devices and the CPU](image)

Figure 2.5: Diagram that shows a possible interconnection between memory, devices and the CPU

In the single case, where there are no memory hierarchy in any device, all DMA Accesses will access the memory coherently. This means that at any time the CPU and any other device will have the same information available. However, in a real scenario, devices and CPU have their own memory hierarchy that serves as a cache or as a working memory, meaning that the data is not always up-to-date in memory, so further control mechanisms are needed to ensure coherence.

In Cache-coherent system, DMA-write-petitions signals the cache controller for invalidation, and DMA-read-petitions cache the controller for flush. This ensures that the memory read from DMA is coherent.

However, in Non-coherent systems, software must ensure that cache-lines are flushed before an outgoing DMA transfer is started, and invalidated before a device makes
a read, another approach could be to mark the CPU-pages as non-cacheable. This introduce overheads over DMA operations, but is needed to maintain coherence.

\subsection{IOMMU}

The DMA works on physical-memory without any abstraction, however, CPUs adopted the concept of virtual memory, which makes possible having a contiguous address space that in physical memory is not contiguous. This means that, the addresses you work with in a CPU program are not physical address, but virtual, and a hardware called MMU translates this addresses to physical addresses.

While this works well in a CPU process-environment, this generates some problem when talking to accelerators. The first one is that, due to the dissonance between the virtual-addressing and physical-addressing, accelerators no longer know how to generate DMA-Accesses to the correct address, because for each virtual address, a different, mutable physical address needs to be accessed. To solve this problems, the IOMMU hardware translates memory-petitions from a virtual address space to the physical address space at the time of performing a DMA-Access, this ensures that, no matter how fragmented are pages in memory, we will get the correct address to access.

Also, the main use of IOMMU currently, is to restrict the set of addresses a device can access to, since a DMA Request can potientially read and write any part of the memory, is a huge security threat to let devices access all the memory, so the IOMMU is programmed to access only known addresses.

This can be challenging to implement device-coherent memory accesses, since it may be necessary to develop additional drivers to handle the hardware through OS-specific IOMMU API, and extend the capabilities to allow selecting process pages in order to share the virtual address space.

Also, a hardware IOMMU may be not available in all platforms, so when an IOMMU
is missing, additional hardware to perform the IOMMU job must be included in the accelerator, or configured inside an fpga.

2.2.3 Unified Virtual Addressing

The Unified Virtual Addressing (UVA) is the capability that allows accelerators to share the virtual addresses with the running process. This doesn't necessarily mean that the accelerator can access all the virtual address space, or that the copies doesn't need to be explicitly invoked, there is not a winning standard and there are a lot of approaches, hardware and software, that, while sharing similar concepts, the execution and transparency to the user vary.

We will discuss the following approaches:

- CUDA
- OpenCAPI
- Xtasks
- Vogel

2.3 CUDA

CUDA (Compute Unified Device Architecture) It’s a parallel programming model developed and maintained by NVIDIA that allows general purpose computing on CUDA-Capable devices.

CUDA extends the C/C++ programming language in order to include new language symbols that dictates whether or not a function must be compiled to run on a GPU (also called Kernel) and the invocation methods to call this GPU-Device with the hardware assignation. While code that uses these c++ extensions must be compiled with the propietary NVIDIA CUDA compiler, the generated objects are generated by the compiler installed on the system which can be (cLANG or GCC). This means that the output object can be linked against non-nvidia generated objects.

CUDA also provides a c/c++ library which can be used to interact between the GPU and the CPU, with operations such as allocating memory, freeing memory and interacting with the CUDA ASYNC functionality.

2.3.0.1 CUDA in multi-GPU scenarios

All cuda API calls are binded to a per-thread active GPU. This means that, the allocations, and CUDA objects that we obtain through the API are only valid to the GPU that’s marked as active.
cudaError_t cudaGetDeviceCount(int* num);
cudaError_t cudaGetDevice(int* dev);
cudaError_t cudaSetDevice(int dev);

Figure 2.7: CUDA API calls that serves to manage the active default context for each GPU

On the code listing 2.7 cudaGetDeviceCount retrieves the number of CUDA-Capable devices that can accept work, cudaGetDevice gets the thread-active device and cudaSetDevice activates the given device for the thread.

Managing correctly the active device and ensuring the operations are done in the correct context it’s crucial in order to run CUDA correctly.

2.3.0.2 Synchronous vs Asynchronous

CUDA exposes two different API functions, ones that are asynchronous and will not block the execution of the CPU and ones that will only return from the function when the operation has been satisfied.

While asynchronously-execution CUDA API calls can prove more challenging that synchronously calling and waiting for results, the later method of execution it’s incompatible with HPC and the runtime work.

However, ignoring the fact that the CPU could be doing useful work, having an asynchronous behaviour doesn’t necessarily mean better performance.

Figure 2.8: Execution of an Async task as seen as the GPU

As we can see on the figure 2.8 the async behaviour of the call is not improving the execution time of the kernel itself. However, with an async behaviour, we can enqueue new kernels that are not dependent on the first one, thus overlapping the copies and the execution of different kernels, as seen on the figure 2.9

Figure 2.9: Execution of an Async task as seen as the GPU
With this method, the more kernels we can enqueue, the more we overlap kernel
execution with copies, while using the GPU concurrency capability to run more than
one kernel at the same time, if possible.

### 2.3.0.3 CUDA Stream model

Streams is the CUDA response to the asynchronous execution model. A stream models
a queue of executions that will happen as if they were to run synchronously, so, while all
the operations on a stream run synchronously, multiple streams can run concurrently
as seen in figure [2.10].

Creating a new stream is as easy as performing the following call in the correct context:

```c
cudaError_t cudaStreamCreate ( cudaStream_t* pStream );
```

After we have a stream, all the operations (copies, executions...) that are enqueued in
that stream will be executed sequentially. For synchronizing in a stream-environment,
there are multiple options, however, the only one we will work with in this project, it’s
the cudaEvent synchronization, which has the following API functions associated:

```c
cudaError_t cudaStreamCreate ( cudaStream_t* pStream );
cudaError_t cudaEventCreate ( cudaEvent_t* event );
cudaError_t cudaEventRecord ( event, streamid );
cudaError_t cudaEventQuery ( cudaEvent_t event );
```

An event is an item that can be enqueued inside a stream, and, when the stream
reaches the event point, the execution status of the previous operation is stored in the
stream. To record and already-sent operation, we only need to call cudaEventRecord
with the information about the event we will use to record and the stream we are going
to check. After that, to get the status of the previous operation, we only have to call
to cudaEventQuery and check the return value, which will tell us if it’s pending, it
finished with errors or it finished successfully.

![Figure 2.10: Representation of a CUDA Stream for one GPU, there can be multiple streams in parallel, as seen in the diagram, but each stream can run in parallel.](image-url)
2.3.0.4 CUDA Distributed Memory Management

When working with CUDA Distributed memory model, the CUDA API exports some functions to interact with the memory, which in essence allows to get an allocation of memory in the device RAM, free that memory and copy data in and out it. Any address allocated by these functions that can be used in function calls to GPU kernels.

\begin{verbatim}
cudaError_t cudaFree ( void* devPtr );
cudaError_t cudaMalloc ( void** devPtr, size_t size );
cudaError_t cudaMemcpyAsync(void * dst, const void * src, size_t count,
                                      cudaMemcpyKind kind, cudaStream_t stream);
\end{verbatim}

2.3.0.5 What kind of job is expected on GPGPU computing?

NVIDIA unified memory contains a hardware-enabled MMU which translates host virtual memory addresses into gpuMemory physical addresses and performs the needed hidden copies when needed.

It’s implementation is based in the same principles that the CPU Memory Subsystem, there are mapped pages inside the GPU MMU, with a TLB (Translation Lookaside Buffer) that stores recent virtual to physical address translations. When in PASCAL and VOLTA GPUs, an address is not in the GPU memory, it generates a fault and the two SM (Streaming Multiprocessors) that share the TLB are stalled until the access is satisfied. This is necessary because on a fault, the GPU may evict some pages in order to fit the new ones. The unified memory driver processes the faults, removes duplicates and updates mappings and transfers the data. The fault handling adds significant overhead over the performance of the GPU.

This makes that, while data-transfer and data-coherence it’s easier to maintain with CUDA Unified memory, the performance of the application can vary drastically depending on the data pattern of the application.

2.4 OpenCAPI

OpenCAPI\[8\] by IBM is a protocol that tries to ease the programmability of devices making the devices able to access coherently main memory. Itjungle has an article which expands in the progression of OpenCAPI in time, citing they page we find that Coherent Accelerator Processor Interface (CAPI) is an IBM protocol that has evolved among time. CAPI 1.0 was the first generation and implemented coherent memory protocol over PCIe 3.0 controllers on Power8, but avoiding unnecessary overhead and just using the necessary to allow coherent memory addressing between devices. CAPI 2.0 was similar to 1.0 but over PCIe 4.0 controllers, which had more bandwidth. After this point, IBM saw more benefit in creating a new 25Gb/sec port on their Power9 chips in order to implement and improved CAPI protocol over that port. So, with
CAPI 3.0 and CAPI 4.0 (now called OpenCAPI), the Power9 Chip support up to 48 lanes of OpenCAPI ports, which offer better latency numbers than PCIe.

With OpenCAPI, virtual-to-physical address translations occurs in the host processor and enables a shared virtual address space. With this, the complexity and resources needed in the accelerator decreases, specially helpful in the FPGA world, and allows pointer-chasing and any kind of structure without any driver involvement.

2.5 Xtasks

Xtasks is a support library that allows an user to create tasks and execute them in a supported FPGA with a previously generated accelerators created using the ompss stack.

2.5.1 Xtasks Accelerators

Xtasks has a unique structure per accelerator implemented inside the FPGA, which contains the following information:

- Unique Identifier: Serves only as identifier
- Frequency (Hz): The actual speed the accelerator inside the FPGA is running at in Hertz
- Type: An integer value which contains the implemented function signature.
- Description: A description of the implemented type
- Maximum Number of Tasks: The number of tasks the accelerator can handle (-1 is undefined).

In order to get this information, Xtasks exposes an API that allows to get the number of available accelerators:

```c
xtasks_stat xtasksGetNumAccs(size_t *count);
```

A list of handles to the different accelerators:

```c
xtasks_stat xtasksGetAccs(
    size_t const maxCount,
    xtasks_acc_handle *array,
    size_t *count
);
```

and finally, from an accelerator handle, it allows us to get the information struct specified before:
2.5.2 Xtasks Memory Allocations

Xtasks offers a Malloc and Free implementation, however, because of the host is not allowed to access the allocated data directly, there is no pointer to data directly returned from a malloc call, but an opaque pointer to a memory handle, which serves as an allocation handle.

When trying to get the base address of the allocation, in order to send some data to the device, an API call that returns the physical-address associated to the memory handle is needed, the case of xtaskGetAccAddress.

xtasks_stat xtasksMalloc(size_t len, xtasks_mem_handle *handle);
xtasks_stat xtasksFree(xtasks_mem_handle handle);
xtasks_stat xtasksGetAccAddress(
    xtasks_mem_handle const handle,
    uint64_t *addr
);

2.5.3 Xtasks Memory Management

Xtasks offers its own set of memcpy implementations that are able to communicate with the memory handles allocated by the xtasksMalloc function.

The function expects as parameters the handle to the allocated memory, the offset since the beginning of that memory, the length, an user pointer to copy the data from or to, and the kind of memcpy that is needed (Copy to device or copy from device).

xtasks_stat xtasksMemcpy(xtasks_mem_handle const handle,
    size_t offset,
    size_t len,
    void *usr,
    xtasks_memcpy_kind const kind
);

There is an alternative way of performing a memcpy, making use of the xtasks asynchronous API, which as an extra parameter accepts a copy handle, which can be tested to know if the copy has finished or not:
2.5.4 Xtasks Tasks

The main functionality of Xtasks is that it provides the necessary interface to offload functionality to an accelerator. When creating a task, we must take into account the following parameters:

```c
xtask_stat xtasksCreateTask(
    xtasks_task_id const id,
    xtasks_acc_handle const accId,
    xtasks_task_id const parent,
    xtasks_comp_flags const compute,
    xtasks_task_handle *handle
);
```

- **Id**: A custom arbitrary identifier that we set
- **AcceleratorId**: The accelerator we are sending this task to
- **Parent**: The task-id of the parent, 0 if non-existent.
- **Compute**: Defines if the accelerator must execute the task or not
- **Handle**: Where a handle to the task will be set in order to control it.

So, when creating a task, a handle is generated, which will serve as an entry point for adding the parameters the task expects to have.

```c
xtask_stat xtasksAddArg(
    xtasks_arg_id const id,
    xtasks_arg_flags const flags,
```
\texttt{xtasks_arg_val const value,  
\texttt{xtasks_task_handle const handle
\)};

- Id: Which argument we are passing, goes from 0 to \( N \) being \( N \) the number of
  parameters the accelerator expects
- flags: Defines some properties of the argument related to placing and caching
- Value: The value that the accelerator will receive
- Handle: The task handle of the task we are inserting this argument on

Once the task is created, we can submit it with simply calling the Xtasks with the

task handle:
\texttt{xtasks_stat xtasksSubmitTask(xtasks_task_handle const handle);};

To check for task finalization we can use the function
\texttt{xtasks_stat xtasksTryGetFinishedTask(xtasks_task_handle *handle, xtasks_task_id *id);};

which, if returns a success, handle and \( id \) will be the task handle and \( id \) from a task
that has finished.

When we finish with the cleanup of the task, the interface requires us to delete the

task to free resources, it can be done with the following API call:
\texttt{xtasks_stat xtasksDeleteTask(xtasks_task_handle *handle);};

\section{Vogel}

This method from the paper \textit{Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU[14]} explores an implementation of SVM (Shared Virtual Memory) using a hardware IOMMU in high-end FPGA Boards that support it, or a soft-IOMMU implemented in an FPGA in other cases.

In essence, the problem that Vogel tries to solve, and the way it solves it, makes it

similar to NVIDIA CUDA unified memory, which, at the same time, matches with our
end goal although we take a software approach.
Chapter 3

OmpSs-2 contributions

3.1 Support for devices with distributed memory

Nanos6 support for devices relied before this work in the transparency that unified memory accesses for CUDA offered, since no extra data synchronization was needed from the point of view of the runtime, as it was managed by CUDA, CUDA tasks were more of an adaptation of an SMP task with asynchronous synchronization for checking when a task finished.

The use of this transparency has hidden implications, for example, in x86-64 machines, all the data that was going to be shared with the GPU needed to be allocated using an specific CUDA API call, while data allocated this way, can be read from both, GPU and CPU, this already forces the user to change the code in order to support this feature.

With the device support developed in this project, we make the data sharing totally transparent to the user, so it can use any kind of address that it’s addressable from the CPU side.

While we talk about supporting CUDA, the same mechanism that we developed serves as an entry point for other programming models and devices, for example, interoperability with OpenCL, or adding new devices like FPGA or ASICS to OmpSs-2 runtime is possible with this new framework. In fact this project offers a software mechanism that allows any kind of device or abstraction to be held coherent regardless of any kind of hardware support for sharing memory between devices.

3.2 Support for CUBLAS CUDA tasks

As stated by the official nvidia docs, The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA® CUDA™ runtime. It allows the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU).

When we want to execute cuBLAS kernels, we must have the context of the GPU
we are going to use active, and a valid stream that will be used internally by cuBLAS, 
this, unfortunately, can’t be done with usual SMP tasks, because an SMP task doesn’t 
have a GPU or a Stream associated, however, the usual CUDA tasks couldn’t be used 
either, because they contained a direct kernel call, and did not supported function 
calls. To being able to solve this limitations, a new type of task was created, which, in 
the eyes of the runtime, it’s a CUDA task, but when is launched, instead of running 
the code that launches a kernel, it runs an user-defined function as if an SMP task 
was run. This means that, now, the task has an active GPU and a stream associated, 
which can be retrieved by the new API call we created for this situation:

```c
cudaStream_t nanos6_get_current_cuda_stream(void);
```

and the execution of the cuBLAS calls will have the data coherent at the moment of 
executing the task. While this is explored for cuBLAS task, an user can be creative 
in the uses of this, and can, for example, call it’s own kernels manually or call other 
external libraries.

### 3.3 Support for FPGA Tasks

OmpSs runtime Nanos5 supports FPGA tasks with discrete dependencies, this means 
that, when a task that has dependences arrives, having with it the size and the region 
of the dependences, only the address of the first element is used as a coherence and 
identification method for that region. Dependencies that are subregions of a big region, 
are tracked like if they were foreing to the big one. In OmpSs-2 runtime, this is not 
the case, so, for this reason, until now there was no infrastructure in OmpSs-2 runtime 
that supported sharing host memory with devices.

The FPGA implementation uses the support for device discrete memory mentioned 
in the previous paragraph as base for the implementation. Serving, jointly with CUDA, 
as an example of implementation for the new device infrastructure developed in this 
project.

Since there isn’t support for FPGA tasks yet in OmpSs-2 runtime, mercurium 
does not generate the code for this kind of tasks, so manually editing the mercurium- 
generated code is needed in order to run FPGA tasks, also, the bitstream (the “code” 
that programs the FPGA) is generated using OmpSs@FPGA infrastructure, which 
makes the generation of accelerators for the FPGA and the interface with them easier 
to implement.

In addition to that, in order to forward the arguments of a task to the FPGA task, 
a new Nanos6 API call was needed:

```c
void nanos6_fpga_addArg(int index, int symbolData, 
                        void *taskHandle, void *address);
```

which serves as a method to bind the data to the task before sending it to thhe 
accelerator.
Chapter 4

Design and implementation

4.1 Strategy

As we discussed in the background, there are a lot of strategies that can be applied to this problem. The most common approach are the page-level strategies, and, if possible, page-level strategies with hardware support. These strategies are interesting due to two principal factors:

- We know that, at a page level, all the memory of the belongs to the process
- The access pattern is not known, and a random access must be assumed.

So, in a page-level strategy, the system doesn’t know which page the user is going to use, and has a page-fault system which moves the page between the different devices or processor. This strategy makes sharing the data between devices and the CPU transparent, hiding all the complexity.

However, these methods usually don’t handle well the data-sharing of read regions between multiple devices, which can make a page move between devices, which implies performance degradation, also, at a page level, there is potential to have false sharing, which can hurt the performance even further.

Also, the hardware capabilities to make these kind of system work, are not widely available in all platforms or in all vendors, which makes it less portable to any kind of device.

However, the strategy we will follow is different, for each piece of functionality that will be offloaded to an accelerator, we will mark the symbols passed as parameters as device-symbols, which, with a logic layer that handles the different accelerators, we will perform the copies and translations needed before executing the task.

While this method is by no means transparent to the user, all changes when we link this method to a task-based runtime that supports range dependences, like Nanos6.

The runtime, to be able to generate the dependence graph, needs the information about which parts of the data are going to be used, and which purpose do they serve.
This information, however, overlaps with the information needed by the logic we propose in order to handle the different devices. This makes it “transparent” to the user, since, when creating a task, the dependence data they are already supplying, is the same that the data our system will use in order to perform the data copies and maintain coherence.

This approach, comes with hidden benefits, which can’t be underrated.

- Eliminates false-sharing completely, range-based coherence
- Allows any kind of device to have software-based coherence
- Allows more control over the memory

4.2 Implementation Overview

Due to the fact that all the components that intervene in the implementation are tied together, an overview of the different components can be beneficial for understanding the architecture.

4.2.1 Accelerator

A new abstraction called accelerator is created, an accelerator is the representation of a physical device. This means that, for each GPU, there will be an instance of an accelerator registered in the runtime. In more complex devices, like FPGA, where inside an FPGA multiple accelerators can be instantiated, there will be only one accelerator instance, the FPGA, and all the processing for the virtual accelerators inside this FPGA will be done inside this abstraction.

- **Accelerator Events**: In some architectures, like in CUDA, some actions require that an event completes in order to be able to proceed with the computation or to check if a task has finished. A queue is created that will contain all the events that are being tracked, followed with the functionality that must be executed when they are completed.

- **Async Steps**: When some actions don’t have native event or streamable support, they end up in a synchronization abstraction called steps, while more insight will be given later, they work similar to the events.

- **Streams**: A stream can be seen as a queue of operations that are ensured to be executed in order. This abstraction is taken from the CUDA API, and must be implemented for other devices like FPGAs.

- **Accelerator Polling**: A function that is called time-to-time that proceed with obtaining new tasks, processing them, and making advance the execution of all the tasks that are waiting for events or synchronization.
• **Taskwaits:** When a taskwait arrives, the directory will add into a queue all the operations that must be executed, the polling service will proceed with them.

While the task is not a new component of the runtime, there are some changes made to support discrete memory, copies and coherence in the runtime. Now, when a task registers its data accesses, a new structure inside the task is created which contain the different symbols, and, for each symbol, its regions with the type of dependency it is.

### 4.2.2 Directory

When we initiate the runtime, each accelerator that is going to be handled by the directory is registered, this is important because the size of the entries is dynamic, taking into account the number of the accelerators that are going to be used.

When a task is pre-ready, so we need to manage the copies of the task, it is sent to the directory, which, for each symbol, will check the state of the regions that define the symbol in the directory entry map, and if they are not found, will create an entry for them.

After checking where the data is, it will generate the necessary copies and update the necessary directory entries in order to ensure that, before executing the data, the data is valid.

- **Directory Entry Map:** This is a range-based list of accesses, it contains directory entries. While it’s an implementation detail, it allows the partition of entries in case a new access boundary fall inside an already existing entry. This functionality is key in order to support all the cases.

- **Directory Entry:** A directory entry contains the status of a host-region of memory, and its translation into a device memory space, if existent.

- **Device Allocation:** A device allocation is the allocation that translates a host-region of memory into a device-region of memory. A Device Allocation can be shared between multiple Directory Entries.

### 4.3 Address Spaces

An address space is an abstraction of a region of memory, let it be the virtual-memory address space of the host, the memory of a GPU or a sub-region of another address space which will be treated like it’s own.

The Address Space, for the directory, it’s highly tied to the hardware accelerator abstraction, meaning that each hardware accelerator must have an Address Space.

For accelerators like FPGA, where multiple accelerators reside inside the same chip with the same address-space, the Accelerator abstraction must support the handling
of the multiple virtual accelerators that reside in the single FPGA Accelerator and, consequently, it’s address space.

An address space is defined by:

- It’s own virtual page-size
- It’s own index inside the directory
- The device type of the address space
- The device handler for it’s own API calls

The principal functions that an address space must support are:

- accel_allocate: A function that synchronously allocates a chunk of memory in the address-space.
- accel_free: A function that synchronously deallocates a chunk of memory previously allocated with accel_allocate.
- copy_in: A function that asynchronously copies a host-addressable region of memory to the address space, appending the operation into a stream.
- copy_out: A function that asynchronously copies an address-space addressable region of memory to a host-addressable memory region, appending the operation into a stream.
- copy_between: A function that asynchronously copies an address-space addressable region of memory to the address-space of a device of the same type, appending the operation into a stream.

4.4 Entries Interval Map

In the core of the directory resides the interval map. This structure is the most critical and contains the information about the state of all regions that are being tracked by the directory. Its interface has two main functionalities.

- Create and partition the map for each access
- Obtain all the entries for an access
Figure 4.1: Different paradigms on partitioning the regions of the directory, at the left, we can see: In green, the existing region, in orange, the region that is being registered. At the right, we can see en blue, the new regions that have been created due to the partitioning, in orange, the new regions and in green the preexisting regions.

Figure 4.2: From left to right, the process of partitioning over a complex region step by step. Over the preexisting region, there is the orange region that it’s being registered, orange regions are new regions and blue regions mark a new registered region that is partitioned from a preexisting one.

4.4.1 Creating new entries and partitioning new ones

When the Directory is empty and a new region is going to be registered, this region does not need any more work than to be registered inside the directory, however, when there is more than one entry, we must check for overlapping regions inside the directory and act in consequence, any kind of overlapping is possible and we must ensure that the new regions or partitions contain the correct information.

In figure 4.1 we can see the different data-registration patterns that the directory can handle in the view of an specific already-existing access. In orange, we can see the regions that are going to be added to the map, in green, the current existing entries of the map and in blue, the new partitions that needed to be done in the directory, but that contain at this point the same information as the region partitioned.

In the left part of the diagram, it can be seen the superposition that will cause the access, and at the right part, we can see how it affected to the final structure.

Since mapping all the cases is quite difficult, since the layout of the registered accesses can be as chaotic as the user programs it to be, the regions are handled step by step from left to right following the example figure 4.2.
4.5 Directory

A directory is a data structure that centralizes the information of the state and location of the data through an heterogeneous system with different address spaces, let it be processor caches, main memory, data in devices or another machines. The granularity of a Directory is a discussion driven by the usual restrictions set by the hardware implementations, a highly-associative cache with the monetary, latency and area cost associated to the storage. However, in this thesis, a software directory implementation is proposed, which handles this problem at a software-level, making viable a byte-granularity approach, and capable of handling any kind of device.

In the figure 4.3, we can see all the parts that conform or are needed by the directory semantically divided by colours.

![Figure 4.3: Graphical Overview of the directory contents.](image)

4.5.1 Directory API

The Device-Directory interface is simple and concise:

- registerAddressSpace: This function registers an accelerator, returning an identifier to be used inside the directory
• initialize: Once all Accelerators are registered, it initializes the internal structures.

• registerRegions: This function tries to register all the symbols of a task in the directory, it creates a synchronization barrier that will prevent the task from being executed until all the symbols are valid in the accelerator the task will run.

• taskwait: This function ensures that the data is valid on the host.

4.5.2 Registering address spaces

The Device-Directory is a structure that must be generated at runtime. This is due to the fact that at compile-time we don’t know the number of devices that are going to interact with the directory, and, since the number of devices directly impact the size of the inner structures, all address spaces must be registered before the first use of the directory.

The registerAddressSpace function handles the Accelerator-registration, returning a handler that will identify the accelerator inside the directory.

Once all accelerators have been registered, a call to initialize will setup the directory for its use.

4.6 Directory Entry

When we talk about a directory entry, we are referring to the piece of information that is absolutely needed in order to track a region of memory. Since it keeps track of the multiple devices that are instantiated in Nanos6 (including one entry for "SMP"), the size of the Directory Entry is dynamic. The minimum information that needs this entry type is:

• The host memory region that is mapping
• The per-device base allocation of this region
• The state of this region per device
• The dirty device (The device where the data is modified, can be none)

4.6.1 State of an entry

When talking about the state of an entry, we must assume that, on creation, all the entries are VALID on the SMP. The two principal states are VALID and INVALID, which are the normal state of the entry. But, since we want to make the copies asynchronous, we are forced to have a transitory state: PENDING_TO_VALID in order to avoid, having two tasks: A, B which share a symbol, from having A performing a
copy and B advancing the execution of A without the copy having finished. Also, there is a "hidden state" called modified. If a device have a modified copy of the data, the device will be in a VALID and a unique index that tracks the dirty device will be marked with the index of the device. This means, only one device can have the data modified, and while the dirty device is set, only that device is valid.

There are 3 possible states of an entry per device:

- **VALID**: the data is ready to be used and nothing has to be done to run tasks on this device.
- **INVALID**: the data must be retrieved from a device that has a VALID state, or the HOST if there is no VALID state.
- **PENDING_TO_VALID**: the data is not yet VALID on the device, but the respective asynchronous copies have been created and, when completed, will be set to valid.

### 4.7 The representation of a symbol

A symbol is represented by a set of accesses that may, or may not, be contiguous in memory, but the offset between them must remain valid.

To support this feature, when Mercurium compiler generates the task data accesses using the pragma information given by the user, it adds a symbol identifier that goes from 0 to the number of symbols a specific task has. This identifier is applied to all accesses of the same symbol. This way, when each separated access for a task is processed, we can create an structure inside the task that:

- Contains the (currently unknown) address in the target device
- The full host-region the symbol is referring to
- A list of accesses separated between its access type

This is because for a symbol, the full region must be contiguous in memory, but not necessarily all the regions of that symbol must share the same attributes.

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>HostAddressRegion</td>
</tr>
<tr>
<td>DeviceAddressRegion</td>
</tr>
<tr>
<td>set:IN_REGIONS</td>
</tr>
<tr>
<td>set:OUT_REGIONS</td>
</tr>
<tr>
<td>set:INOUT_REGIONS</td>
</tr>
</tbody>
</table>

**Figure 4.4**: Table that represents the data that it’s stored inside a symbol.
4.8 Streams

A stream, as defined by CUDA, is "a sequence of operations that execute on the device in the order in which they are issued by the host code". So, basically it’s a queue of operations that are ensured to be executed in order. Also, there is no limit on how many queues of this style we can create.

This is an useful methodology that hides the synchronization away from the generator code, which means that we can process multiple asynchronous calls, with the benefits of ensuring the order and being able to process more data instead of hanging.

However, not all devices offer an asynchronous API, nor have an implementation for all the platforms, this is the case of the Xtasks API, which provides an asynchronous API but it behaves like if it were synchronous in some platforms, affecting drastically the performance.

Due to the device-specific methodology to run, handle copies and check for termination, each accelerator must create it’s own abstraction, the directory handles the Stream as an unknown data structure and facilitates it to address-space functions when needed.

4.8.1 Reference Implementation of a Stream for any device

As we discussed, a stream is a representation of a queue of operations that must be performed in sequence. This means that: to continue with the execution of the next task in the queue, the latest one must have finished.

Due to that fact, in reality, a queued operations must perform two works: First, when an operation is firstly run, it must send to execute the operation in another thread. After that, it must check for the ending of the executed task.

This model is similar to what in c++, std::async supports, however, as the runtime generates it’s own thread pool to handle tasks, adding more threads would restrain the performance. For this, this implementation makes use of dependency-free runtime-generated tasks, and asks the implementor to setup a method to detect the task finalization.

When adding an operation to a Stream, the methodology is to have an "Activator" and a "Finalizer". The "Activator" is an anonymous function that executes the asynchronous code and generates a Finalizer. The Finalizer is an anonymous function that returns true if the operation has finished, we call the operations that can be added into a stream, streamable operations ass seen in figure 4.5.
Generates Activation Function
Check For Termination Function
Streamable Operation

Figure 4.5: A streamable operation is an activation function which executes some asynchronous code and generates another function that checks for termination of the first one.

The Custom Stream implementation offers the following methods:

- addOperation: Adds an operation to the stream, It accepts as a parameter a function (Activator) that generates another function (Finalizer) that returns true or false whether or not the task generated by the activator has ended. In c++ notation would be: std::function<std::function<bool>>>, in figure 4.6 we can see the flow graph for this function.

- trycontinue: It’s an asynchronous function that must be called while there is an active operation in the stream, it generates runtime-created tasks that will check for a finalized operation, and launch a new one if needed. In figure 4.7 we can see the flow graph for this function.
4.9 Registering a task

When a task has selected an accelerator it will run on, it has to pass through the directory, registering and ensuring that the data is valid before the execution. There are multiple paths the task will have to follow through:

- Partitioning
- Allocation
- Reallocation
- Process input regions
• Process output regions
• Process input/output regions

4.9.1 Partitioning

When a task arrives, we don’t know if the regions specified by the symbols of the task are already present on the directory or not, to ensure that the region is being tracked, we check the map, and add and create the partitions on the inner map, if needed. This ensures that the following calls to the interval map will succeed and return the correct data.

4.9.2 Allocation

We must ensure that, for each host region represented by the task’s symbols, there is a contiguous memory allocation sufficient to contain that region. To do so, we get the first directory entry that matches the beginning of our symbol, and obtain it’s device allocation. After that, we loop through all the entries which are inside the symbol range, and check them against the device allocation previously obtained.

If at any point, any allocation was null, or different that the first allocation retrieved, a new allocation is performed on the device.

To perform an allocation over a region, there are things to consider:

• There must be space in the accelerator
• Which size should be allocated and locality optimizations

If the allocation can’t be performed, the task can’t run, so we will return an error when creating the task, that must be handled to free memory.

We could, also, decide to allocate the exact memory we need to satisfy the symbol allocation, however, too many allocations of little size could hurt the performance. To sort these problems, the accelerator must define a page-size, which will be used as a hint for the directory at the time of allocating memory, and, if the directory entries inside the page-allocation-region doesn’t have a valid allocation, this new one will be applied.

4.9.3 Reallocation

When a symbol wasn’t continuously allocated, and a new allocation is created, before assigning this new allocation to the entries that perform the symbol, we must ensure that, for each entry, the latest version of the data is not exclusively owned there. If this happens, a copy between the previous allocation and the new allocation must be performed in order to ensure that the data we are working with still valid.
4.9.4 Processing Symbol Regions

When a new symbol is going to be processed, the directory logic behaves differently depending on which is the actual state of the directory entry (VALID, INVALID or PENDING) and which kind of access we are registering. The protocol we implement (figure 4.8) consists in 4 states, where one of them is a transition between states.

- **Input Regions:** This state is for read-only regions of data, that, in consequence, can be shared between devices without invalidation. An overview of the processing protocol can be found on the figure 4.9
  
  - VALID: No need to do anything
  - INVALID: We will set the entry as PENDING_TO_VALID, generate the asynchronous copies from a device where this region is valid, and will generate a callback which will set this entry as VALID once the copy is performed.
  - PENDING_TO_VALID: Occurs when another task is already copying this region. We generate an awaiter, which blocks the execution of the task until this entry it’s VALID.

- **Output Regions:** Since when a region is set as output, the data that it’s already present is not important, we invalidate the entry for all the devices and set it VALID and dirty directly, as we can see on the figure 4.11

![Diagram](image)

**Figure 4.8:** Graph that represents the states of an entry for a device, at the left, we have the actions that the device can do in order to jump between states, at the right, we can see actions that a different device can make that affect the state of the data in the device.
• **Input/Output regions:** When a region is going to be read and write at the same time, we need to have the valid data present on the device. This means that inout it’s more restrictive than out region, since it forces us to do a copy, and can’t be shared as an input region. The behaviour of this logic can be seen on the figure 4.10

  - **VALID:** We process the entry as if it were an output region
  - **INVALID:** We will set the entry as PENDING_TO_VALID, generate the asynchronous copies from a device where this region is valid, and will generate a callback which will set this entry as VALID and dirty once the copy is performed.
  - **PENDING_TO_VALID:** Occurs when another task is already copying this region. We generate an awaite, which blocks the execution of the task until this entry it’s VALID.

![Flow graph](image)

**Figure 4.9:** Flow graph that a symbol follows when is declared as an Input

![Flow graph](image)

**Figure 4.10:** Flow Graph that a symbol follows when it’s declared as an Input/Output
4.10 Ready-task dependences and execution workflow

When the dependences on a task have been cleared, in an SMP context, it means that the task is ready to be executed. This is, however, not valid when there is an heterogeneous system with non-unified memory between accelerators and memory. While the task can be executed, we must ensure that the data is valid in the accelerator before the task can run. This creates another level of dependency, the data-dependency system.

When a task passes through the first level of dependencies, it becomes a "pre-ready" task, and will perform the preparations to be executed. Unlike the traditional dependency system, these dependencies can be both, active and passive dependencies, being an active dependency a work that is invoked by the dependency in order to get cleared, and a passive dependency being an awaiting to look for finalization of an already-invoked dependency by other task that has not finished yet.

We call the Execution workflow the series of steps that must be run in order to execute a task.

4.10.0.1 Synchronization mechanisms

At this point, we have discussed widely about Streams, a synchronization mechanism that allows the execution of some functionality in a certain order. While we made an abstraction, which can be used for any kind of functionality, devices like CUDA, while supports running CPU code in its native streams, a stream code can’t be executed multiple times in order to check for finalization of some action without advancing, this means that, for example, the finalization of a copy invoked by another task can be checked inside a CUDA stream, but we would have to do busywaiting and hang the CPU until the condition has been satisfied, if not, the next function on the stream
would run. This is important because we want to maintain the compatibility with native streams for performance reasons, at the same time that we want to ensure that some alternative synchronization is done.

In order to have more flexible synchronization, we divide each task into two synchronization realms, the **Stream synchronization** and the **Workflow Step** synchronization.

A **Workflow Step** is a synchronization mechanism with data and functionality associated, which ensures, via the creation of a synchronization graph of multiple steps, that all the prerequisites to proceed the functionality are completed. To put it simple, given five steps: A, B, C, D, E, if we create the synchronization graph following this dependence order of figure [4.12] in this example, firstly A will be executed, which bifurcates into two paths, which can be run in parallel, but before the step E can be executed, as there are two paths that points to that step, both paths must have finished before executing E, making E a synchronization point.

![Figure 4.12: Example of workflow steps where A is the root step, E is the last step, and the two paths (C and B,D) must finish before E can be executed.](image)

### 4.10.1 Device Workflow

At some point, we have to build up the infrastructure to run the task, this is called the Device Workflow. We divide the functionality in the two synchronization mechanisms:

- **Steps**: In the steps we put the functionality that it’s not directly supported by the streams, the flow of the steps begins with the awaiters, after the awaiters, the task execution and finalizes with the freed of dependences.

- **Streams**: In the streams, we put all the asynchronous copies and, the events to check for finalization of these copies, and, after all these copies are already in the stream, and the Step subsystem finalizes all the awaiters, it puts at the end of the current stream the task execution and finalization event.

As we discussed earlier, the Directory is the one that will be generating the necessary data copies or awaiters for a task. It does that by requesting two step synchronization
points and a stream. The Stream is received by the accelerator that pertains to the task that it’s going to be executed. The directory will enqueue all the copies in the stream, and add between the two synchronization points all the awaiters that must be completed before continue with the task execution. As we can see in figure 4.13, intermediate steps can be created by the directory, if needed, in order to await for data. Notice that, since a stream is already sequential, we don’t need to check for copy-finalization of the new copies performed in this workflow before enqueuing the task execution, since this is done automatically by the stream system.

This division between steps and streams ensure the compatibility with CUDA, where it’s a good idea to enqueue the maximum number of operations as fast as possible, while, other operations like the awaiter one, although possible to implement in CUDA streams, there is no good method to implement it without actually stalling the CPU.

When the task has finished executing (Detected asynchronously), the next step executed is the one that frees dependencies in order to let the dependency system know that the task has finished, at this point, the task has finished completely, in the figure 4.14 we can see a diagram of how both synchronization methods operate.

Figure 4.13: Directory synchronization

Figure 4.14: Diagram that shows how the different synchronizations methods interacts when a task passes through the directory
4.10.2 Accelerator Loop

One of the design decisions when working with accelerators is, in absence of an intelligent load balancer and data-aware-scheduling, is to have, for each accelerator, a polling service which tries to obtain tasks of it’s own type from the scheduler, create the device workflow making use of the directory, and run the task. This way, multiple accelerators of the same type can concurrently access the scheduler and load balance on a task-number basis. Also, in order to incentive the balancing and put a limit in the number of concurrent streams executing, we fix the number of tasks that an accelerator can be run at the same time.

Also, when talking about device operations that must be performed periodically, we must not forget about the asynchronous finalization of operations. Since our proposal generates fully-asynchronous code, a decision was made to: merge inside the polling service the additional workflow-dependant functionality that asynchronously continues the execution of the workflow, checks for tasks finalization and, in case of devices which doesn’t support streams, executes the stream logic.

We can then, divide the functionality in this processes:

- **Process New Task:** We try to get, as many tasks as we can, from the scheduler. Each task is processed by the device-workflow, which uses the directory in order to generate copies and awaiters.

- **Process Events:** We check for the finalization of events, which can be copies finished, where the directory must be notified, or tasks that have finished and are waiting to be freed.

- **Process Taskwaits:** Taskwaits are an special case where all the copies that the directory generates, can be run in parallel. Also, since there is no stream associated to a taskwait, and the data can be in multiple kinds of devices, in the meanwhile, the taskwait is a semi-synchronous operation. Each taskwait copy is done in a Nanos6 task synchronously, but all the needed copies are run concurrently. When processing taskwaits on the accelerator loop, we check if the taskwait operation has finished, and proceed to continue with the execution of the taskwait when it does.

- **Process Interrupted Steps:** When there is asynchronous functionality that must be kept track of, but there is no native support for events in the device, for example, awaiters (Wait until a directory Entry is valid), an interrupted step tries try perform the check of validity, and if it fails, adds itself to the interrupted step queue again.

- **Platform Dependent Actions:** This action is reserved to device-specific implementations, for example, in platforms like FPGA with Xtasks, where there is no native stream implementation, there is a list of streams in the accelerator
that must be processed in order to advance towards the eventual execution of the task. This is done here.
Chapter 5

Evaluation and results

5.1 Environment

To test our proposal, we need different machines with the necessary compatible hardware accelerators, while our proposal works for any kind of accelerators, given that an implementation of the device-specific functions are provided, we will test it in two of our available clusters:

- **Ikergune ZU102**: A development board that features a Zynq UltraScale+ 9EG System-on-Chip which runs a 4-core Arm Cortex-A53, with 4GB of RAM, and includes an FPGA with 600K System Logic Cells, 32Mb of memory, 2520 DSPs and 328 I/O Pins.

- **CTE-Power9**: CTE-POWER is an IBM-Power9 processor cluster which runs under Red Hat Enterprise Linux Server 7.4 with infiniband as network. It’s composed by 2 login nodes and 52 compute nodes, where each of them has two IBM Power9 8335-GTH processors, with 20 cores per socket and 4 threads per core, which means that each node has 160 threads available, and a 512 GB main memory divided in 16 dimms of 32 GB each. As accelerators, each of them have 4 GPU NVIDIA V100 (Volta) with 16 GB HBM2.

5.1.1 FPGA

Since there is no other implementation of similar features in Nanos6 for devices, the way of validating the results and performance will be checking against the Nanos5 implementation, which, runs under other constraints and runtime. While the obtained results may differ due to the differences in the runtime architecture, it will result in an overview of how well the Directory behaves in this context.
5.1.2 CUDA

In order to performance-check the Directory, we will try three implementations of Nanos6:

- CUDA with Directory: Our proposal, which is discussed in this project.
- CUDA Unified Memory: Nanos6 with support for CUDA.
- CUDA Prefetching: Nanos6, using unified memory and prefetching the data that each task is going to use.

5.2 Matrix Multiply

One of the most elementary algorithms that applications often use is matrix multiplication, which can be performed in multiple ways. It can seem simple, but to make a really performance-tuned implementation there needs to be a huge understanding of the memory subsystem, so we can use algorithms which data fits inside the lower level memory hierarchy so the data don’t become a bottleneck.

For this reason, Blocked Matrix Multiplication is a way to increase performance due to the fact that it decreases the memory bottleneck.

There can be two approaches:

- Contiguous Block Matrix Multiplication: The matrix is already divided by blocks that are contiguous in memory, or, a copy is made in order to make them contiguous before computation.
- Non-contiguous Block Matrix Multiplication: Each block is not contiguous in memory and no modifications are done to the origin matrix in order to apply the algorithm.

In this case, Contiguous Block Matrix Multiplication is used, allocating directly the matrix in a blocked way.

5.2.0.1 Analysis

The code implementation for our matrix multiply can be seen on the figures 7.1 and 7.2, with the help of a script, we created for multiple configurations of blocksize and runtimes the performance obtained as seen in the table 5.1 and represented on the plot 5.2.

It’s worth noticing how, for low granularity, in most cases unified memory seems to be working better, however, as soon as the task work increases, the directory and prefetcher approaches tends to increase the performance drastically, but unified memory never ends up catching up. In the figure 5.3 we can see an speedup plot which we
compare the Directory performance against the prefetcher, where, we can see, that most of the cases the Directory outperforms drastically the prefetcher in this application.

In general terms, we can see that our Directory approach outperforms the simple usage of Unified Memory, getting from 10x up to 35x performance increase, and outperforms Unified Memory with Prefetching up to 6.5x, without changing the code.

<table>
<thead>
<tr>
<th>BS</th>
<th>DIRECTORY</th>
<th>PREFETCHER</th>
<th>UNIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>62.6933</td>
<td>30.0028</td>
<td>148.773</td>
</tr>
<tr>
<td>256</td>
<td>569.3204</td>
<td>201.02</td>
<td>205.99</td>
</tr>
<tr>
<td>512</td>
<td>1910.8154</td>
<td>833.82</td>
<td>207.46</td>
</tr>
<tr>
<td>1024</td>
<td>2115.2687</td>
<td>2037.96</td>
<td>217.26</td>
</tr>
<tr>
<td>2048</td>
<td>2143.2134</td>
<td>1971.13</td>
<td>155.2159</td>
</tr>
<tr>
<td>4096</td>
<td>2150.7068</td>
<td>2132.56</td>
<td>264.064</td>
</tr>
<tr>
<td>8192</td>
<td>2102.2524</td>
<td>2012.51</td>
<td>261.0056</td>
</tr>
<tr>
<td>16384</td>
<td>1934.0378</td>
<td>911.61</td>
<td>55.69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BS</th>
<th>DIRECTORY</th>
<th>PREFETCHER</th>
<th>UNIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>63.4714</td>
<td>27.66</td>
<td>140.3312</td>
</tr>
<tr>
<td>256</td>
<td>484.2637</td>
<td>207.0474</td>
<td>157.7808</td>
</tr>
<tr>
<td>512</td>
<td>1602.3242</td>
<td>831.2172</td>
<td>205.2358</td>
</tr>
<tr>
<td>1024</td>
<td>1988.6877</td>
<td>1806.6096</td>
<td>219.8572</td>
</tr>
<tr>
<td>2048</td>
<td>2059.0183</td>
<td>1831.9637</td>
<td>229.1546</td>
</tr>
<tr>
<td>4096</td>
<td>1986.758</td>
<td>1799.5722</td>
<td>243.7039</td>
</tr>
<tr>
<td>8192</td>
<td>1487.1127</td>
<td>1665.6996</td>
<td>78.1371</td>
</tr>
</tbody>
</table>

Figure 5.1: Performance (in GFLOPS) for matrix sizes from 2k to 16k, at different block sizes.
Figure 5.2: As we can see, using unified memory without any kind of prefetching results in bad results for performance in all configurations (yellow line), probably due to the page-faults on the GPU. Unified Memory with prefetching (red line), seems to be competitive in the configurations (a) and (b), however, when we decrease the matrix size, it seems to decrease in performance (c) and (d), the directory approach (blue line) seems to be the one that performs better in most configurations.
### Figure 5.3: These charts show the speedup achieved using the Directory version instead of the Unified Memory with prefetching, as we can see in (a) and (b), when working with low granularity, it seems to be more noticeable the performance gain, however, using the directory version gives better performance in almost any case, noticing in the 8K matrix, a slight drop in performance when the block is the full matrix. In (c) the performance gain seems to be quite stable around 2x and in (c), when working with small matrices, the speedup goes up almost at 7x.

#### 5.2.1 FPGA Matrix multiply

As the objective in this project was to create a framework that can be used with multiple devices, we wanted to offer and try more than one device implementation, we decided that FPGAs would be the logical inclusion to the runtime, as OmpSs already have support for FPGAs using OmpSs@FPGA system.

With OmpSs@FPGA, we generated 3 matmul accelerators of blocks of 256x256 at 300Mhz in a single bitstream, adapted the source code of the matmul in order to run under OmpSs-2 directives and modified the intermediate code in order to add the additional support which the mercurium compiler doesn’t provide yet for Nanos6. After that, we tried to run a Matrix Multiply with different configurations, which produced the results shown on the figures 5.4 5.5

However, this hit in performance doesn’t necessarily mean that the Directory is the
Matrix Size | 1024 | 2048 | 3072 | 4096  
--- | --- | --- | --- | ---  
Nanos5 | 146.83 | 153.87 | 155.81 | 156.943  
Nanos6 | 57.76 | 86.17 | 93.1 | 107.98

**Figure 5.4:** Table that shows the performance in GFLOPS for Nanos5 and Nanos6 with the different configurations.

**Figure 5.5:** Performance plot where we can see that, while Nanos5 has greater performance, Nanos6 catches up when the problem size increases.

cause of the problem, as we saw, when increasing the size problem, the performance tends to get better too. This can be for multiple reasons like, the change from discrete dependences (Nanos5) to ranges (Nanos6) or the thread management, we need to take into account that the whole runtime is different, since we couldn’t compare with another Nanos6 version, more insight is needed and will be investigated in the future. However, right now, there is a working implementation for a device other than GPUs, where GPUs and FPGAs can coexists in the runtime.

### 5.3 Conjugate Gradient

A conjugate gradient is an algorithm that gives the numerical solution of a system of linear equations that is symmetric and positive-definite.

While there are algorithms that probably perform better than a CG on a dense matrix, for example, backsubstitution, but for really big sparse matrices, iterative algorithms like CG are the most efficient ones.

It’s defined as an iterative algorithm, suitable for computation on sparse matrices, (although it can be used in dense matrices too, other algorithms, like backsubstitution should be explored first).

The implementation we are going to use consist on the following kernels, which we implemented for CPU and GPU:
• Dot Product: Calculates the Dot product between two equally-sized vectors.

• Sparse Matrix Vector Multiplication: or SpMV, Performs a matrix multiplication, like the one benchmarked in the previous section, but over a sparse matrix (Most of the elements are 0). The operation performed is the same, but instead of allocating all the matrix, the only elements stored are the non-zero elements, and a data structure that helps retrieve that elements taking into account its position.

• Single Precision A per X plus Y: or Saxpy, for a pair of vectors, multiplies all the elements of the second vector by a constant, and adds it to the same position of the first one.

• Scale Vector: Multiplies all the vector by a constant.
float cpuConjugateGrad(int *I, int *J, float *val, float *x,
    float *Ax, float *p, float *r,
    int nnz, int N, float tol) {

    float alpha = 1.0, alpham1 =-alpha;
    int k = 1;

    const float tol2 = tol*tol;
    const int max_iter = 1000;

    float r0 = 0.0, r1 = -1, dot = -1, b, a;
    float *r0_p = &r0, *r1_p = &r1, *dot_p = &dot, *b_p=&b, *a_p=&a;

    CPU::cpuSpMV(I, J, val, nnz, N, alpha, x, Ax);
    CPU::saxpy(Ax, r, &alpham1, N, POSITIVE);
    GPU::dotProduct(r, r, r1_p, N);

    while (k <= max_iter)
    {
        if (k > 1)
        {
            CPU::divide(b_p, r1_p, r0_p);
            CPU::scaleVector(p, b_p, N);
            CPU::saxpy(r, p, &alpha, N, POSITIVE);
        }
        else CPU::setRange(p,r,N);

        CPU::cpuSpMV (I, J, val, nnz, N, alpha, p, Ax);
        GPU::dotProduct(p, Ax, dot_p, N);
        CPU::divide (a_p,r1_p, dot_p);
        CPU::saxpy (p, x, a_p, N, POSITIVE);
        CPU::saxpy (Ax, r, a_p, N, NEGATIVE);
        CPU::set (r0_p, r1_p);
        GPU::dotProduct(r, r, r1_p, N);
        k++;
    }

    #pragma oss taskwait
    return r1;
}

Figure 5.6: Conjugate Gradient implementation where the namespace before the name of the function specifies where that function is going to run. In this case, all dot product functions have been offloaded to GPU. In order to avoid synchronization, some helper tasks like CPU::divide and CPU::set serves to store the result of the division in the first element and set the second element in the first respectively.
CHAPTER 5. EVALUATION AND RESULTS

5.3.1 GPU Performance

In order to compare the GPU performance, we will offload different functions and see how it performs. In the first case, we will compare the performance of offloading the dot product function, this function was selected because it shares data with the saxpy function, which means that synchronization must work in order to get the correct result. Since we are more interested in seeing which is the performance hit of copies, instead of real performance, this scenario setups a pretty good way to check the directory performance versus the other counterparts.

In this experiment, we run 1000 iterations of CG in the following configurations:

- SMP Only tasks
- Dot products on GPU using Directory
- Dot products on GPU using Unified Memory with prefetching
- Dot products on GPU using Unified Memory as is

After trying to execute 1000 iterations of Unified Memory, we gave up, the execution time was abnormally high, taking up to 43 seconds for only 30 iterations. In order to understand better why this was happening, we decided to trace the program using the nvidia profiler[7] which returned the trace form figure 5.7.

![Figure 5.7: Unified memory trace of a dot product in CUDA where the kernels take up to 1 second to execute](image)

5.3.1.1 Execution Traces

Since the performance reported is not generated by the runtime, but per the CUDA runtime, the execution time for these traces is valid, and a problem in data-sharing when using unified memory is occurring, probably, the pages where the data that the dot product is handling is being shared with the data other tasks are using, and there is a false-sharing problem which generates this penalty. However, with unified memory with semi-explicit copies (prefetching), this problem didn’t occur, so the cuda driver probably handles things differently in that scenarios, since there is no page faults involved after a prefetch (if the data that is being prefetched is all the data that is going to be used).
So, this arises the question, how different should behave the CUDA with prefetching than CUDA with the directory? Since both explicitly state the copies that must be done, we expect to see similar traces in this end,

\[\text{Figure 5.8: The trace (a) corresponds to the Directory, the trace (b) corresponds to prefetching, as we can see, both share a lot of similarities in the execution of the kernels, but, while (a) describes the copies explicitly, (b) generates prefetching, which is not represented on the stream timeline.}\]
5.3.2 Performance

We executed the different configurations and extracted the table referenced on the figure [5.9]. While there was no performance reason to run this CUDA conjugate gradient with more than one GPU, in order to validate the functionality of the Directory it is convenient to check multiple accelerators scenarios, also, since the only thing that changes between tests is the runtime, the user code remains the same, for the same scenario we can detect how well or worse our system performs compared to the others.

<table>
<thead>
<tr>
<th>#GPU</th>
<th>DIRECTORY</th>
<th>SMP</th>
<th>PREFETCHER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.91</td>
<td>22.5</td>
<td>21.9</td>
</tr>
<tr>
<td>2</td>
<td>21.07</td>
<td>22.5</td>
<td>22.01</td>
</tr>
<tr>
<td>4</td>
<td>21.52</td>
<td>22.5</td>
<td>22.49</td>
</tr>
</tbody>
</table>

Figure 5.9: Execution time for a Conjugate Gradient with 1000 iterations (less is better), as we can see, the Directory outperforms any other configuration in each scenario.

Figure 5.10: Execution time for a Conjugate Gradient with 1000 iterations (less is better), as we can see, the Directory outperforms any other configuration in each scenario.
Chapter 6
Conclusions and future work

In this master thesis we have designed and implemented new copy and coherency methods that allows task-based runtimes to support accelerators. Our proposal includes a layer abstraction for devices, a directory which manages the copies and coherence between multiple accelerators, and a device execution workflow that allows device tasks to be run. Current implementation has been integrated into the OmpSs-2 runtime and supports CUDA and FPGA devices. Therefore, the programmer will not need to worry about manually handling the data and synchronizations when using different devices using OmpSs-2 programming model.

We have evaluated the functionality and performance with extensively used applications, like matrix multiplication and conjugate gradient for both FPGAs and CUDA, running under OmpSs-2 programming model, in a mature manner. For the case of CUDA, in addition, results show better performance than using the currently-available coherency methods: Unified Memory and Unified Memory with prefetching. This demonstrates that it is possible to obtain better performance using software copies and coherence mechanisms than using unified memory hardware acceleration without hurting the programmability of the end user.

After this first design space exploration and proof-of-concept design, we can see several opportunities to work on. For example, the system could be used to create a cluster where nodes are handled as devices, with an abstraction that shares the data via network, or even create a data-tracing capability which logs portions of data with the use of tasks, etc. In the near future, our work will be to look ways to improve the current implementation, and do more benchmarking in order to ensure that this is a path worth to be visited, which all the indicators say yes.
void matmul( int m, int l, int n,
        int mDIM, int lDIM, int nDIM,
        REAL **tileA, REAL **tileB, REAL **tileC )
{
    for(int i = 0; i < mDIM; i++)
        for (int j = 0; j < nDIM; j++)
            for (int k = 0; k < lDIM; k++)
                Muld(
                    tileA[i*lDIM+k],
                    tileB[k*nDIM+j],NB,NB,
                    tileC[i*nDIM+j],NB
                );
}

Figure 7.1: SMP Part of the code that performs the execution of a matmul using CUDA.
#pragma oss task in(([NB][NB]A)[0;NB][0;NB])
→ in(([NB][NB]B)[0;NB][0;NB]) out(([NB][NB]C)[0;NB][0;NB])
→ device(cuda) ndrange(2, NB, NB)

__global__ void Muld(REAL* A, REAL* B, int wA, int wB, REAL* C, int NB)
{
  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;

  // Index of the first/last sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd = aBegin + wA - 1;

  int aStep = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep = BLOCK_SIZE * wB;

  REAL Csub = 0;
  for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
  {
    __shared__ REAL As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ REAL Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB*ty + tx] += Csub;
}

Figure 7.2: CUDA Kernel from the University Of Colorado[1] that performs a matrix multiplication over a block, annotated with OmpSs-2 pragmas in order to execute it with Nanos6 runtime.
__global__ void saxpy(float *x, float *y, float* a, int size, bool s)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    float at = s?a:-a;
    if(i<size) y[i] = at * x[i] + y[i];
}

Figure 7.3: Saxpy implementation for CUDA

__global__ void dotProduct(float *vecA, float *vecB, float *dot, unsigned int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid==0) *dot = 0;
    while (tid < n){
        double t=vecA[tid] * vecB[tid];
        atomicAdd(dot,t);
        tid+=blockDim.x * blockDim.x;
    }
}

Figure 7.4: Dot product implementation for CUDA

__global__ void scaleVector(float *vec, float* alpha, int size)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<size) vec[i] = *alpha * vec[i];
}

Figure 7.5: Scale vector implementation for CUDA

void saxpy(float *x, float *y, float *a, int size, bool s) {
    if (s)
        for (int i = 0; i < size; i++)
            y[i] = *a * x[i] + y[i];
    else
        for (int i = 0; i < size; i++)
            y[i] = -*a * x[i] + y[i];
}

Figure 7.6: Saxpy implementation
void dotProduct(float *vecA, float *vecB, float *dot, int size) {
    float result = 0.0;
    for (int i = 0; i < size; i++) {
        result = result + (vecA[i] * vecB[i]);
    }
    *dot = result;
}

Figure 7.7: Dot product implementation

void scaleVector(float *vec, float *alpha, int size) {
    for (int i = 0; i < size; i++) {
        vec[i] = *alpha * vec[i];
    }
}

Figure 7.8: Scale Vector implementation

void cpuSpMV(int *I, int *J, float *val, int nnz, int num_rows, float alpha, float *inputVecX, float *outputVecY) {
    for (int i = 0; i < num_rows; i++) {
        int num elems this_row = I[i + 1] - I[i];
        float output = 0.0;
        for (int j = 0; j < num elems this_row; j++)
            output += alpha * val[I[i] + j] * inputVecX[J[I[i] + j]];
        outputVecY[i] = output;
    }
}

Figure 7.9: Sparse matrix Vector multiplication implementation
Bibliography


[3] Programming models @ bsc, . URL https://pm.bsc.es/


[5] Programming models @ bsc, . URL https://pm.bsc.es/ompss-at-fpga


