Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 781–790
DOI: 10.15439/2014F337
ACSIS, Vol. 2

Supporting job-level secure access to GPGPU resources on existing grid infrastructures

John Walsh, Jonathan Dukes
School of Computer Science and Statistics,
Trinity College Dublin,
Email: John.Walsh@scss.tcd.ie, Jonathan.Dukes@scss.tcd.ie

***Abstract*—Grids provide secure, utility-like access to a wide variety of large-scale, distributed computational and storage resources. In particular, the European Grid Infrastructure (EGI) and Open Science Grid (OSG) have excelled in processing vast workloads of independent jobs for the research community.**

Researchers demand increasingly faster processing speeds to solve increasingly larger and more complex problems. To meet this need, attention has shifted over the past decade away from single-core processing models towards the use of multi-core, many-core and massively parallel computational accelerators. The increasing availability and use of General Purpose Graphic Processing Units (GPGPUs) are an example of this.

This paper addresses many of the challenges that exist in the integration of resources such as GPGPUs into Grid infrastructures. Specifically, solutions are proposed for discovering and describing GPGPU Grid resources, specifying multi-GPGPU job requirements, performing multi-GPGPU allocation to jobs, dynamically updating publicly-readable GPGPU usage information and enforcing GPGPU access control to prevent distinct jobs from inadvertently accessing the same device. The proposed solution is fully compatible with widely-used and accepted standards and middleware including the GLUE 2.0 schema and EGI Unified Middleware Distribution. A prototype implementation is also described.

I. Introduction

GRID Computing [1] developed out of the need for geographically distributed scientific communities to co-operate in order to investigate scientific problems with increasing computational complexity. The most famous example of such a community is the Particle Physics community investigating the existence of the Higgs Boson using the Large Hadron Collider (LHC). Not only do such large-scale problems require a given scientific community to share vast amounts of data among its (distributed) users, it may also be unfeasible to to process this data at a single location (known as a "Site" or "Resource Centre") – hence distributing the data processing tasks and storage to multiple locations is often required. Such grid-systems draw upon distributed computing, resource discovery and sharing, distributed data-management, authentication and authorisation, role-based access control and process accounting.

Grids developed around a "single program/single CPU" execution model. However, since 2005 the exponential growth of CPU speed and processing power has plateaued [2], and this has generated some questions about the future of computational-based scientific research using this "single program/single CPU" approach. *Parallel Processing* taking advantage of emerging multi-CPU cores (multicore) or many processing cores (many-core) on General Purpose Graphic Processing Units (GPGPUs) or Intel's Xeon Phi Computational Accelerator is regarded as a "white-knight" like solution to this problem. Indeed, the trend towards the extensive usage of GPGPUs and Intel's Xeon Phi, commonly known as "Computational Accelerators" (CAs) [3], in High Performance Computing environments can be seen in the twice-yearly "Top 500 Supercomputer" lists [4].

Support for grid-based parallel applications using Message Passing Interface (MPI) [5] has been available for a number of years [6]. No such support currently exists for the emerging CA-based parallel-processing architectures despite indications that many grid resource-centres and users were planning to incorporate CAs into their future work-plans [7] [8].

The core objective of the work presented in this paper is to address the challanges of integrating GPGPU resources into grid infrastructure in such a way that there are no changes (or minimal changes) to how the user works. In addition, there should there be no significant changes to the grid infrastructure itself. The approach taken to solve this integration problem is to first consider the more general problem of integrating any new resource. A set of *Grid Resource Integration Principles* is developed (Section III) that take these constraints into account.

In this paper: Section II reviews some of the key concepts in Grid Computing, including service-discovery (the Grid Information System) and the job submission lifecycle. Section III introduces the multi-layer abstract achitecture that seperates GPGPU job resource requests from the GPGPU allocation and access protection layers. Section IV presents a realisation of this model in the form of a prototype execution-model that extends the architecture and capabilities of a grid based on the popular Unified Middleware Distribution (UMD). This extension provides new services that address four key grid components, namely: (i) GPGPU service discovery; (ii) multi-GPGPU resource allocation through a grid job description language and batch system integration; (iii) dynamic updating of publicly-readable status information describing the GPGPU resource usage that complies with current standards; and finally, (iv) per-job access controls that prevent distinct jobs from inadvertently accessing the same GPGPUs. Section V looks at related work, and when applicable, discusses how and why the approach taken here differs. Finally, Section VI

978-83-60810-58-3/$25.00

reviews the current implementation and the scope for future work in this area.

II. Grid Computing

The terminology "Grid" and "Grid Computing" has a wide and varied interpretation [9]. With this in mind, there is a need to clarify these terms in the context of this work and thus define the scope of the work. This section introduces key concepts in Grid Computing – specifically Grids such as the European Grid Infrastructure (EGI) and Open Science Grid (OSG) that primarily provide computing and storage resources to researchers. A high-level overview of the lifecycle of a user's job, from job submission through to execution using a Grid resource, will be presented. Grids are, by nature, large-scale, distributed infrastructures and accessing Grid resources relies on maintaining a uniform, structured, global view of the Grid. An overview of the systems that maintain this information will also be presented.

Key Concepts

A Grid is a distributed collection of computational and storage resources where (i) each resource is controlled and managed solely and independently by its owner or *resource-provider* (for example a University, research centre, company or private individual) and (ii) each resource-provider has some level of control over how the resource is accessed and used. This definition is sufficiently general to include both large-scale Grids, such as EGI or OSG, and also "compute-cycle" volunteer donation systems such as BOINC [10].

The work described in this paper is concerned with large-scale Grids, such as EGI or OSG, that are composed of multiple *resource-centres*, each operated by their owner or *resource-provider*. Each resource-centre provides one or more *Compute Elements* (CEs). These are services that provide access to computational resources such as a cluster of worker nodes accessed through a batch system. A *Grid Information System* is used to publish the capability of each resource-centre, in the form of a description of the resources that it provides, the current utilisation and availability of those resources and a description of the mechanisms for accessing them. To make use of the computational resources provided by a Grid, users submit *jobs* that are described using a formal *job description language* (JDL) (or *resource-specification language* (RSL)). In simple terms, a job consists of an executable program, a specification of the software environment in which the program must run, any input parameters and data required by the program and the files that will contain the outputs from the job. Users are grouped into *Virtual Organisations* based, for example, on their research area and these groupings are used to control access to – and account for the usage of – grid resources.

Grid users can request that their jobs be executed on a specific resource, based on *a priori* knowledge of the capabilities of a resource-centre. However, ideally Grids such as EGI and OSG will allow users to submit a job and let the Grid "decide" where that job should execute. To facilitate this, as well as capturing basic information about the job (e.g. executable program, input parameters and data), a JDL description of the job can also describe the job's resource requirements (e.g. number of CPU cores, minimum CPU specification, minimum memory requirement, software environment). In this scenario, instead of submitting a job directly to a resource-centre, the job is submitted to a Grid *Workload Management System* (WMS). The WMS acts as a broker, using the information published about each resource-centre through the Grid Information System, together with the description of the job, to find all resources that match the job's requirements. Furthermore, the broker can select one of the matched resources (e.g. using pre-defined policies or heuristics) and orchestrate the execution of the job on the chosen resource.

The Grid Job Life-cycle

The usual starting point when submitting a job to the grid is the User Interface (UI). This is a service node that contains the necessary command tools to interact with other grid services. The UI is configured to interact with one or more Workload Management Systems (WMS).

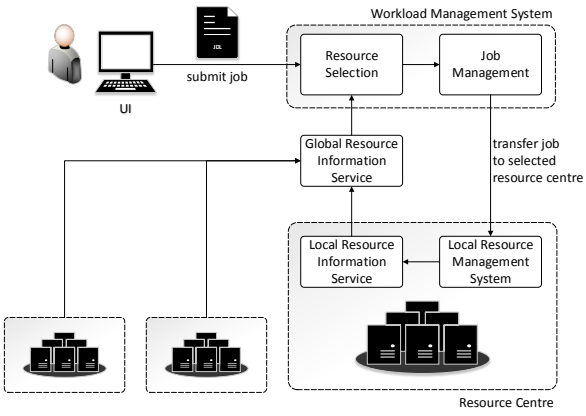

Fig. 1: The Job Submission Chain

Figure 1 illustrates the life-cycle of a single grid job, from the submission of the job in JDL on the UI, through its orchestration by a WMS, to its eventual execution at a resource-centre. A high-level outline of the flow of a grid job through the WMS includes the following stages:

1) When the job is submitted to the WMS, a copy of the JDL file and any input files specified in it are copied to the Workload Management System (WMS).
2) The WMS will determine all locations where the job can possibly run. This is the "match-making" process. The potential locations are then *ranked* in preference. The user can also influence the rank ordering by specifying a *RANK* expression in the JDL.
3) Once a target CE at a resource-centre is chosen, the WMS will engage with the CE.
4) The CE builds a *JobWrapper*. This executable is built taking into account the local batch system, also known

as a *Local Resource Management System* (LRMS), on the CE. Some of the roles of the JobWrapper are to build a job submission script for the batch system, transfer all input files from the WMS, and then submit the job to the batch system.

5) The batch system is responsible for scheduling the execution of the user's job on one or more *CPU-Cores* on a *worker node* (WN).
6) After execution, the output files specified by the JDL description are transferred back to the WMS and can be retrieved by the user.

The match-making process requires knowledge of the overall state of the distributed set of resources. This state information is managed by a "Grid Information System".

Grid Information System

The Grid Information System is a critical component required for discovering services, determining the status of resources and selecting resources for submitted jobs. This system is composed of an information model (a *schema*) for describing entities (such as computational resources and available software) and the relationship between those entities, as well as a "presentation layer" that publishes this information as a frequently-updated queryable service (a *realisation*).

The *Grid Laboratory Uniform Environment* schema (*GLUE schema*) was developed as an Open Grid Forum (OGF) "reference standard" for multi-disciplinary Grids. There are currently two major (incompatible) versions of the GLUE schema in common use – GLUE 1.3 [11] and GLUE 2.0 [12]. Grids such as EGI are currently transitioning from GLUE 1.3 to GLUE 2.0 by running services that can utilize both standards. GLUE 2.0 offers many improvements over GLUE 1.3, including a richer description of grid entities and their state and the ability to easily publish extra details about grid-entities (using "OtherInfo" attributes). Moreover, GLUE 2.0 was developed to improve inter-grid interoperability [13].

Information in a Grid Information System originates from *Information Providers*. The services that are used to access and manage grid resources (e.g. a batch system for a set of worker nodes) should provide an interface that allows the Information Provider for that resource to determine the properties and current state of the resource, transform this information into an appropriate GLUE entity and publish the information (Figure 2).

Propagating GLUE entities so that they are visible "globally" in a Grid Information System follows a natural hierarchical structure (Figure 3): GLUE entities are generated by Information Providers; the set of Information Providers (info-providers) on a particular node publish their state as a *local resource*; the set of local resources form a *domain*; and the combined set of entities from each domain yield a global view.

The GLUE information "presentation layer" is typically managed by the *BDII*([14], Sec. 3.3.5). This is an implementation of a the hierarchical Grid Information System model with three BDII "types" and a set of "information providers" that generate information about the Grid entities. As per

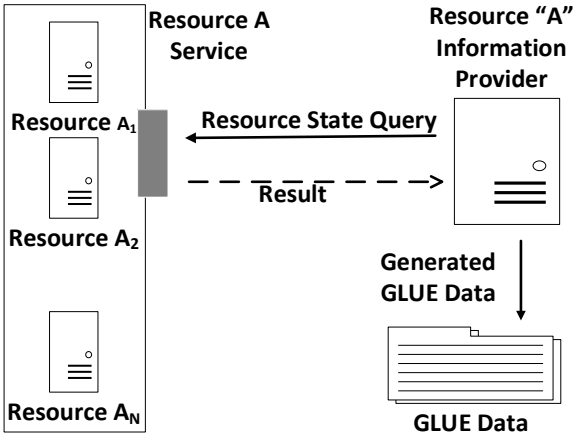

Fig. 2: Abstract Information Provider Model. An Information Provider queries a service that manages a set of Resources of type "A" at a resource centre. The Information Provider transforms the response into one or more GLUE entities.

Figure 3 the Resource-BDII (lowest BDII level) aggregates the state of a grid service node by executing a set of Generic Information Providers (GIP) plugins; the Site-BDII aggregates all the Resource-BDIIs belonging to the given site; and the Top-level BDII aggregates all the Site-BDIIs. Information is "pulled" from the lower to higher levels. In general, all queries about the state of the Grid are made through the Top-Level BDII.

III. Integrating GPGPU resources into Existing Grid Infrastructures

Many grid resource centres have already deployed GPGPU resources [7]. There is, however, no support in place for users to discover these GPGPU resources or submit jobs that specify a dependency on GPGPUs. Currently, users wishing to use these resources rely on *a priori* knowledge of the location and properties of available GPGPUs and the job submission mechanisms required to use them. Furthermore, after inspecting GLUE data published by the Top-Level BDII *lcg-bdii.cern.ch*, it was determined that from a sample of 2887 unique GlueCEUniqueID entities (i.e. interfaces to queues on the grid-connected batch systems), over 58% of these GlueCEUniqueID reported that the Torque/PBS (excluding PBSPro) batch system was used (Table I (a)). In particular, 779 GlueCEUniqueIDs (27%) used Torque/PBS 2.5.7 (Table I (b)). This is indicative of the default UMD Torque/MAUI installation that does not correctly handle generic consumable resources. From this data, it is reasonable to conclude that a significant percentage of grid systems on the EGI grid infrastructure do not support GPGPUs as consumable resources.

This paper addresses the challenge of integrating GPGPUs as first-class grid resources. From a user's perspective, these resources should be easy to discover without relying on *a*

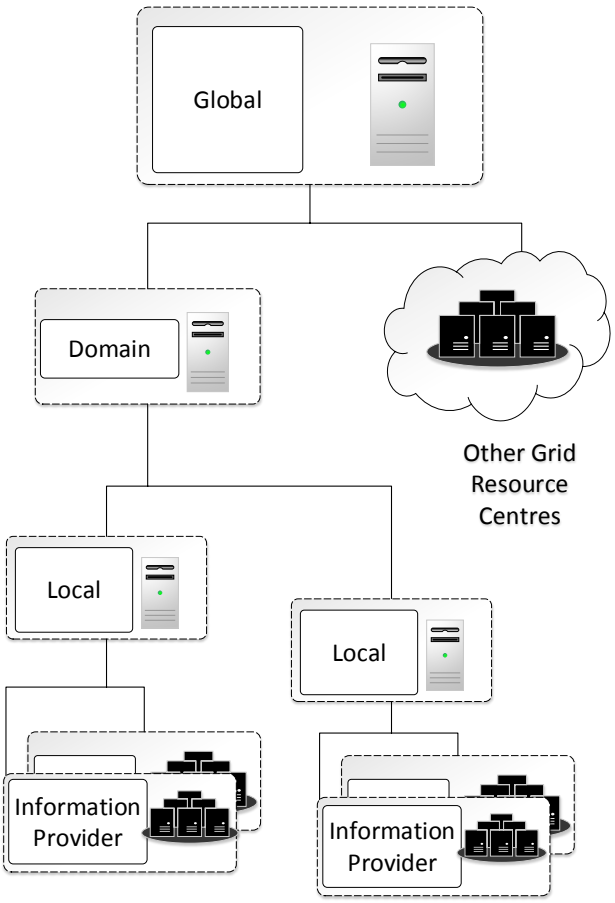

Fig. 3: The Grid Information System. GLUE data is propagated from the lowest level (generated by Information Providers) up to Global level.

TABLE I: Reported LRMS Types

(a)

LRMS Type	Count
Torque	930
PBS	740
LSF	704
Condor	296
GE/SGE	141
SLURM	63
PBSPro	12
Fork	1
Total	2887

(b)

Torque/PBS	Count
2.5.7	779
2.5.X (ex 2.5.7)	706
undefined	84
4.X	45
3.X.	24
2.4.X	8
2.3	24
Total	1670

priori knowledge (*discovery*); the specification, requirements and ranking of these resources should be independent of how the resources are locally managed (*independence*); and the user should have some assurance of exclusive access to the resource (*exclusivity*). Rather than considering GPGPUs specifically, the challenge is viewed as a generic consumable resource access problem. The approach proposed for addressing this challenge is summarised by the following *Grid Resource Integration Principles*:

- **Discovery:** The resource should be published as one or more GLUE entities. The act of publishing the resource into the Grid Information System makes the resource discoverable by users and services. An entity should, where possible, contain attributes that reflect the resources properties, and also where possible, these attributes should be *quantitative*. This allows resources of the same type to be compared and ranked against each other. (For example, the vendor, model and performance characteristics of a GPGPU may be of importance when selecting appropriate resources for a job.)
- **Independence:** There should be a method through which the required resource can be specified using a job description language. The specification should be independent of the way in which the resources are locally managed. (For example, the mechanism used to specify GPGPU requirements to the Torque/MAUI scheduling system differs from that used by SLURM.)
- **Exclusivity:** A resource allocated to a job by a batch system should be available as if it were exclusively allocated to the job. (For example, a GPGPU allocated to a job should not be available or even visible to another job running on the same worker node.)

Case Study – GPGPU integration

As an example of the application of the above *Grid Resource Integration Principles*, the use-case of GPGPUs as grid resources is considered. This use-case is interesting not only because of their parallel-processing capabilities, but because they are also representative of the class of LRMS *Generic Consumable Resources* - a class of non-CPU resources that can be managed by the LRMS.

Discovery: The salient properties that help describe a GPGPU are similar to those used to describe CPUs: vendor, model, memory, speed, benchmarked performance, number of physical GPGPUs per worker node. LRMS properties that can influence WMS orchestration and ranking include, the total number of installed GPGPUs and the number that are currently allocated through the batch system. Users may also be interested in the software required to access the resource, and basic installation details. These properties could be advertised in the Grid Information System as one or more GLUE entities and optionally used as a filter during resource selection.

Independence: The user needs an LRMS-independent way to specify the number of GPGPUs required or the number of GPGPUs that the job needs per-CPU core (this implies a minimum number of GPGPUs per worker node).

Exclusivity: The user needs assurances that two or more jobs concurrently executing on the same worker node cannot use the same GPGPU. In the absence of batch-system support for such exclusivity, an additional mechanism must be provided.

In the next section, a prototype extension of a Grid that integrates GPGPU resources is presented. This prototype introduces a GLUE 2.0 entity and GPGPU Information Providers that provides for the discovery of GPGPU resources; a means to specify GPGPU resource requirements that is independent of the batch system used; and, finally, a new mechanism to ensure that GPGPU resources are allocated exclusively to each job.

IV. Prototype Implementation

As shown in section II, the mechanism for orchestrating the execution of a grid job follows a complex chain of events. As a result, providing access to new grid resources, such as GPGPUs, is non-trivial. In particular, if these resources are to be provided by *existing* grid infrastructures that are *in-production* and in continuous use by an extensive community of users, the challenge becomes more acute. Adding support for new resources cannot be dependent, for example, on architectural changes, the replacement of core services or modifications to the GLUE schema. Instead, the approach taken must integrate with existing infrastructure, while adhering to the principles of *discovery*, *independence* and *exclusivity* (Section III).

The approach taken in this prototype is to provide a set of modular *hooks*, that in principle can be applied to many other resource types, other than GPGPUs. In this section, it will be shown that this approach requires relatively small changes to existing grid middleware.

A. Prototype Infrastructure

The focus of this prototype on the integration of Nvidia GPGPUs using the CUDA runtime and application development framework. (Nvidia GPGPUs are the most widely used GPGPU in High Performance Computing centres.) The prototype was developed using the UMD gLite grid middleware, and consists of a User Interface (UI), a Workload Management System (WMS), a Top-Level BDII (BDII), grid-security infrastructure services, and a Resource Centre using the CREAM CE. The CREAM CE uses Torque 2.5.7 as the batch system server and the MAUI batch scheduler. (MAUI was modified with a publicly available patch to enable Generic Consumable Resources.)

B. Discovery: GPGPU Schema and Information Providers

Section III considered what information about GPGPU resources should be represented through the GLUE-schema. In this prototype, a simple representation of these details is realized by using the GLUE 2.0 *ApplicationEnvironment* entity [12]. This entity is primarily used to describe the properties of software installed on worker nodes. However, a key feature of the ApplicationEnvironment is that it supports non-mandatory attributes that relate to software capacity and utilisation. This feature is used to publish installed GPGPU capacity and utilisation. Furthermore, the definition also allows for other arbitrary information to be published, such as the GPU vendor, model and speed and benchmarked performance.

(An alternative and perhaps superficially more obvious approach would have been to use the GLUE 2.0 *ExecutionEnvironment* entity, which provides for attributes such as CPU vendor, model and speed which may also used to describe GPGPUs. This approach, however, would describe an architecture in which the GPGPUs are independent of – rather than reliant on – the CPUs on the same worker nodes. In contrast, using the ApplicationEnvironment entity to describe GPUs allows the relationship between CPUs and their attached GPUs to be captured by the Grid Information System.)

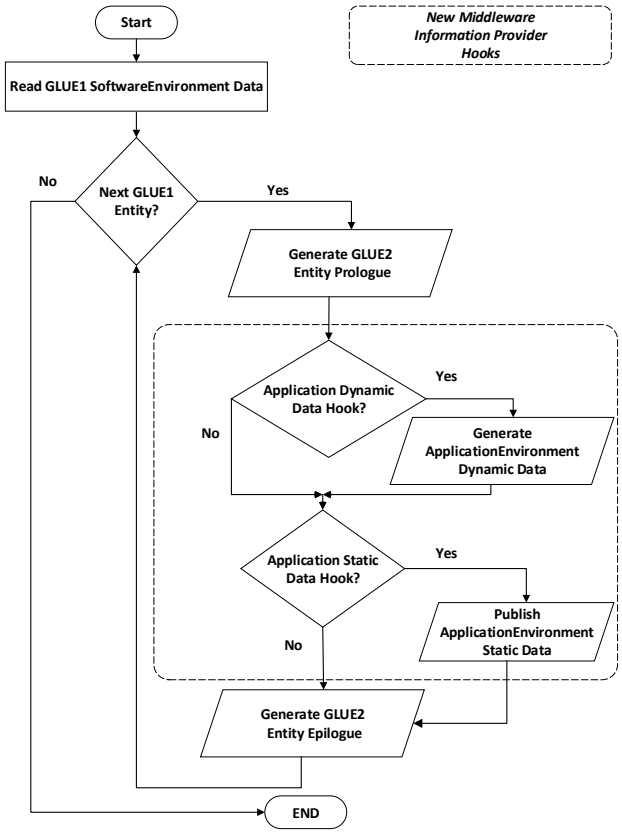

Fig. 4: Workflow of Modified ApplicationEnvironment Information Provider

The current gLite implementation of the GLUE 2.0 ApplicationEnvironment Information Provider already generates ApplicationEnvironment entities by converting GLUE 1.3 SoftwareEnvironment entities into their minimal GLUE 2.0 equivalents. In this prototype, this existing Information Provider is replaced with a new Information Provider that facilitates per-application *hooks*. For each application, separate hooks can be provided to generate static and dynamically changing GLUE data. This is illustrated in Figure 4.

In particular, the prototype uses new CUDA ApplicationEnvironment hooks that publish static data pertaining to the GPGPU hardware and CUDA software properties as well as

dynamically gathering and publishing previously unavailable GPGPU capacity and utilisation data from MAUI. Listing 1 illustrates an example of some of the output generated by the execution of a modified Information Provider where both dynamic and static hooks for the CUDA environment have been added.

Listing 1: An example of the output from the static and dynamic GLUE 2.0 CUDA ApplicationEnvironment hooks

```
...
objectClass: GLUE2ApplicationEnvironment
GLUE2ApplicationEnvironmentMaxJobs: 32
GLUE2ApplicationEnvironmentAppName: CUDA
GLUE2ApplicationEnvironmentFreeJobs: 30
...
GLUE2EntityOtherInfo: ↩
    GPUCUDAComputeCapability=2.1
GLUE2EntityOtherInfo: GPUMainMemorySize=1024
GLUE2EntityOtherInfo: GPUCoresPerMP=48
GLUE2EntityOtherInfo: GPUCores=192
GLUE2EntityOtherInfo: GPUClockSpeed=1660
GLUE2EntityOtherInfo: GPUECCSupport=false
GLUE2EntityOtherInfo: GPUVendor=Nvidia
GLUE2EntityOtherInfo: GPUPerNode=2
```

Table II indicates where the CUDA ApplicationEnvironment hooks provide additional Attribute-Value pairs that are part of the GLUE2 standard (but not generally used), and where the ApplicationEnvironment schema has been further extended (as allowed by the standard) by adding GLUE2 *OtherInfo* values. Furthermore, Table II lists the data-type of each value, the source of the data, and whether the data is generated dynamically or provided statically.

TABLE II: Extended GLUE2 CUDA ApplicationEnvironment

Standard ApplicationEnvironment		**Source**	**Creation**
MaxSlots	Integer	LRMS	Dynamic
MaxJobs	Integer	LRMS	Dynamic
FreeJobs	Integer	LRMS	Dynamic
New EntityOtherInfo Attributues		**Source**	**Creation**
ApplicationArea	String	System	Static
GPUCUDAComputeCapability	Float	GPGPU	Static
GPUMainMemorySize	Integer	GPGPU	Static
GPUMP	Integer	GPGPU	Static
GPUCoresPerMP	Integer	GPGPU	Static
GPUCores	Integer	GPGPU	Static
GPUClockSpeed	Integer	GPGPU	Static
GPUECCSupport	Boolean	GPGPU	Static
GPUVendor	String	GPGPU	Static
GPUModel	String	GPGPU	Static
GPUPerNode	Integer	LRMS	Static

C. Independence: Specifying and Handling GPGPU Job Requirements

Independence implies that the grid user should be able to specify required GPGPU resources within the existing job submission framework in a manner that is independent of any CE batch system implementation. By considering how jobs are orchestrated through a WMS, the changes required to the grid infrastructure to achieve this goal are outlined below.

JDL GPGPU requirements specification: The prototype allows a user to request GPGPUs by adding the following JDL specification:

```
GPUPerNode=X;
```

Here, *X* is the number of GPGPUs to be allocated *per node*. This specification requires no changes to the JDL Language definition syntax. As well as specifying the number of GPUs required, the JDL should also specify the ApplicationEnvironment entity that advertises the availability of the required GPGPU. An example of the complete JDL for a job requiring two GPGPUs and using the CUDA framework is shown in Listing 2.

Listing 2: Example GPGPU Job for gLite based Grids

```
[
Executable = "myScript.sh";
StdOutput = "std.out";
StdError = "std.err";
InputSandbox =  {
                 "GPGPU_acquire_prologue.sh",
                 "GPGPU_job_script.sh",
                 "GPGPU_release_epilogue.sh"
                };
OutputSandbox = { "std.out","std.err"} ;
VirtualOrganisation = "gputestvo";
Requirements = (Member("CUDA",other.↩
    GlueHostApplicationSoftwareRunTimeEnvironment)↩
    );}
GPUPerNode=2;
]
```

Once the job is submitted to the WMS, the set of potential resource-centres is filtered to include only those that advertise that they support the specified ApplicationEnvironment entity (CUDA in the above example). After applying some further job requirements, the WMS will select a matching CE, transfer the job *workload*, and then orchestrate its execution.

Resource Centre Job Execution: The orchestration of the job on the chosen resource centre CE can be classified into three stages: (i) job preparation; (ii) job submission; and (iii) job execution on the selected worker nodes(Figure 5). The *job preparation* stage is used to convert the JDL GPGPU resource specification into an LRMS specification.

During the Job Preparation phase, job input files and executable programs are transferred from the WMS to the CREAM CE. A "JobWrapper" script is created by CREAM. The JobWrapper is executed on the CE, and is responsible, among other things, for constructing a job execution environment appropriate to the LRMS. The JobWrapper script created by CREAM already contains a mechanism to pass or "Forward" job requirements to the LRMS [15]. This prototype exploits this mechanism by implementing a JobWrapper "batch requirements forwarding" script that will parse a copy of the JDL to determine if the $GPUPerNode$ value is defined and, if it is, return a suitable resource request, the format of which will be dependent on the specific batch system.

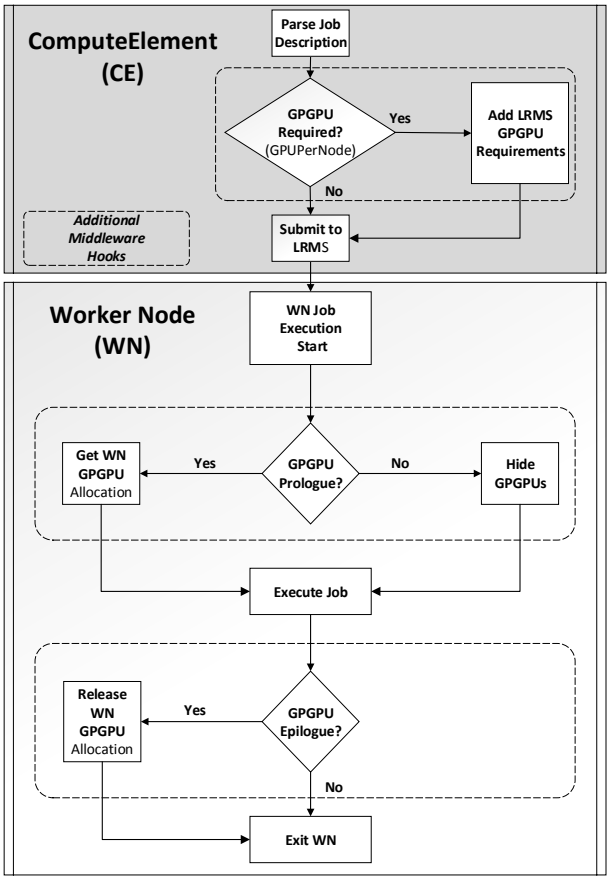

Fig. 5: Batch System GPGPU allocation process.

D. Exclusivity: Restricting Visibility of GPGPU Resources

Multi-core worker nodes allow many independent jobs to execute simultaneously. These independent jobs are protected from each other by executing in their own private disk-area, and by using file-access and process protection enforced by the operating system kernel. GPU (and consequently GPGPU) resources are designed to be accessible by all worker node users. In the case of GPGPUs exposed through a batch system, it is desirable to control access to them. Nvidia supports a number of ways in which access to its GPGPUs can be controlled:

1) the CUDA_VISIBILE_DEVICES environment variable can be used to restrict the visibility of a set of GPGPU devices in a process,
2) the Nvidia *Compute Modes* can permit/deny sharing of a GPGPU between multiple processes.

The handling of GPGPU allocation on common open-source batch systems, such as Torque/MAUI, is still problematic, in particular with mid-range GPGPUs where Nvidia-SMI tools do not report a full complement of utilisation data. A simple prototype service was developed that allows Torque/MAUI batch jobs to request a set of free GPGPUs on the worker node, and to release those GPGPUs back to the service when no longer required. Furthermore, the allocated GPGPUs are not visible to other user jobs on the worker node.

The service, which executes on each worker node, implements a lightweight web-server using the Tornado [16] framework. To improve the security of the service, it runs as an unprivileged user, GPGPU allocation states are maintained in a lightweight persistent database, and the service is internal to the worker node (i.e. it is not available to other nodes).

The server responds to two types of requests: *requestGPUS* and *releaseGPGUs*. These requests are respectively called from the within the *GPGPU_acquire_prologue.sh* and *GPGPU_release_epilogue.sh* Job Hook scripts, specified in the JDL (Listing 2).

Requests to the prototype tornado server must adhere to a strict syntax, and all malformed requests are dropped by the server. In the case the Torque implementation, a request is generated by sending a copy of the "PBS_JOBCOOKIE" and a list of *Universally Unique Identifiers* (UUIDs) - one per requested GPGPU. The PBS_JOBCOOKIE was chosen because, unlike the batch system "jobid", this value is not exposed to other users or processes.

The server manages the allocation of GPGPUs to jobs by using a single database with two tables: *UUID_JOBID* and *GPU_UUID*. The GPU_UUID table associates GPGPU devices with a job-generated UUID. The UUID_JOBID table associates UUIDs to a suitable unique value, such as the $PBS\_JOBCOOKIE$.

1) Initialisation: The GPU_UUID table is initialized at node start-up. A row is added for each physical GPGPU. Similarly, the UUID_JOBID table is created, but remains unpopulated until a GPGPU is requested by a job. Table III shows the initial state of the database tables for a node with two GPGPUs.

TABLE III: Initial State

(a)

GPUID	UUID
0	
1	

(b)

JOBID	UUID

2) Allocation Request: A GPGPU allocation request (Figure 6) should be sent to the server before the job attempts to execute any GPGPU code. When the server receives an allocation request message, a "Request Handler" will attempt to validate it. If the request is valid, then the string is converted into two component values: a JOBID (PBS_JOBCOOKIE) and a list of UUIDs. The request handler then iterates over the list of UUIDS and adds (UUID,JOBID) tuples to the UUID_JOBID table. The handler will also iterate over the list of UUIDs, selecting the first free GPGPU (i.e. rows where the UUID cell is NULL) from the GPUID_UUID table. The selected row is the updated with a new (GPUID,UUID) tuple. Changes to the database are committed. Finally, the server returns a text string to the client in the form of a comma-separated list of integers. This is the list GPGPU devices that

the the job has been allocated. This value is assigned to a *read-only* CUDA_VISIBLE_DEVICE environment variable, thereby ensuring that the user or process can no longer (inadvertently) update its value.

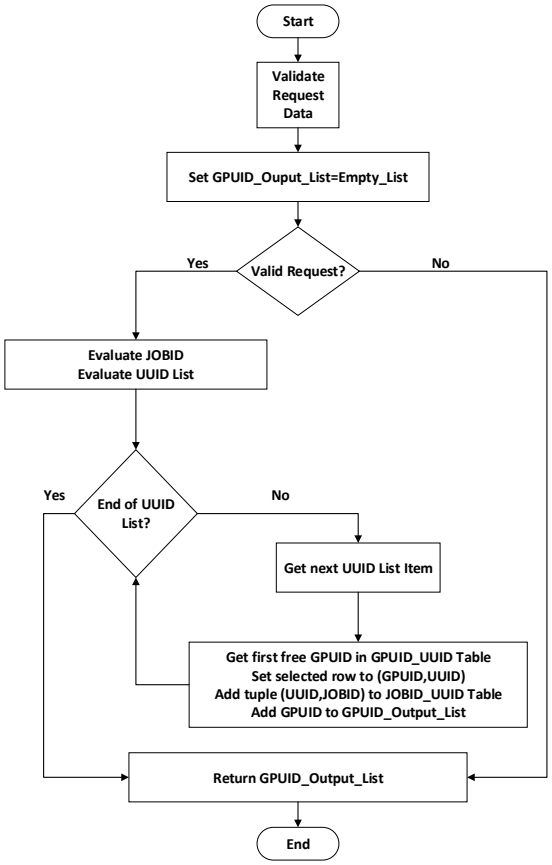

Fig. 6: Worker Node GPGPU allocation subsystem

TABLE IV: Example of a single job running on a node with two GPGPUs

(a) GPGPUs paired to UUIDs

GPUID	UUID
0	$UUID_1$
1	$UUID_2$

(b) $JOBID_1$ linked to $UUID_1$, $UUID_2$

JOBID	UUID
$JOBID_1$	$UUID_1$
$JOBID_1$	$UUID_2$

3) Release Request: A GPGPU "Release" request should be executed during the job's epilogue. This request is complementary to the "Allocation" request. As with the allocation request, the input is validated. If the request is valid, the server's "Request Handler" will remove all tuples matching the UUIDs from the UUID_JOBID table, and all the specified UUIDs from the GPUID_UUID table.

E. Results

Job submission was tested using different values for $GPUPerNode$. Sample output of a job that requested a single GPGPU but executed on a worker node with multiple GPGPUs is listed below (Listing 3):

Listing 3: Example GPGPU Job resricted to a single GPGPU

```
/usr/local/cuda/samples/1_Utilities/deviceQuery/↩
    deviceQuery Starting...
 CUDA Device Query (Runtime API) version (CUDART ↩
     static linking)
......
Detected 1 CUDA Capable device(s)
```

V. Related Work

CUDA_wrapper

CUDA_wrapper [17][18] was developed to help facilitate secure and controlled access to Nvidia-based GPGPU resources on multi-user GPGPU clusters. The wrapper acts by intercepting normal CUDA API fuction calls and overloading them with additional methods that transparently provide additional key-based access-control to the raw GPGPU device.

The approach taken in this paper avoids some drawbacks with the CUDA_wrapper method, namely: (i) Each API interception imposes a latency to handle each CUDA function call; (ii) CUDA_wrapper must be recompiled for each new version of CUDA; (iii) CUDA_wrapper did not work under testing with CUDA 5.5; and, (iv) The wrapper is vendor and LRMS (Torque) specific.

GPU resources on the Grid

There are several examples of current work where GPGPU resources have been partially integrated into Grids. These take the form of both non-virtualised GPGPU resources [19] and virtualised GPGPU resources [20]. In both cases, the grid-users job is given exclusive access to the GPGPU resource. However, both of these implementations lacks resource and service discovery and therefore require *a priori* knowledge of the existance of the resources. Moreover the virtualisation methodology imposes about 5% overhead on GPGPU access and runtimes.

BOINC and Desktop Grids

BOINC has support for multi-disciplinary computational sciences using GPGPU. For example, the *Einstein@Home* project uses GPGPUs to search for weak astrophysical signals from spinning neutron stars (also called pulsars) using data from the LIGO gravitational-wave detectors, the Arecibo radio telescope, and the Fermi gamma-ray satellite [21]. Furthermore, the EDGI Desktop Grid provides a mechanism [22] to bridge between EGI and BOINC-enabled resources. However, such combinations do not address the specific GPGPU service-discovery requirements.

HTCondor GPGPU Support

HTCondor [23] supports advanced GPGPU resource publication, service-discovery, per-job GPGPU match-making, and job-management [24]. Indeed, HTCondor is central to the the UMD WMS match-making service. However, some of the major differences between HTCondor and the work presented

in this paper include: (i) the solution is intended to be compatible with Grids using the OGF GLUE 2.0 information model; and (ii) The approach taken treats GPGPUs as an instance of generic consumable resources.

Application Hook Method

The design and implementation of the presented execution model used in this work follows a pattern similar to that used by MPI-Start [25]. In particular, this is evident in how software hooks are used to build an appropriate site-local application runtime-environment. The major differences in the implementation are: (i) MPI enabled resource centres currently publish information about the local MPI environment as a set of GLUE 1.3 SoftwareEnvironment tags. This information is inefficiently converted into a set of discrete and seemingly unrelated GLUE 2.0 ApplicationEnvironment entities; (ii) The GPGPU implementation attempts to exploit many of the newer features of the GLUE 2.0 ApplicationEnvironment definition. This includes the ability to publish GPGPU resource capacity and utilisation information.

Token based access-control

The use of UUIDs as tokens in the GPGPU allocation and release process is partially based on the use of time-limited UUID tokens in the *Puppet* [26] fabric managent system - a system used to install, configure an maintain machines on a network. In Puppet the token is used as a "shared-key" between the Puppet server and the client machine. The shared-key is used to authorise the download of the client's installation configuration file.

EGI GPGPU Working Group

The European Grid Initiative (EGI) is currently considering the GPGPU resources into grid infrastructures and one of the authors is a member of the EGI GPGPU Working Group [27]. The results presented in this paper represent independent work that may be contributed to this community effort in the future.

EGI Grid Federated Cloud Working Group

The EGI Federated Cloud Working Group uses the GLUE 2.0 ExecutionEnvironment to advertise diverse sets of (virtual) resources [28]. The authors had considered this appraoch, but deemed it to be unsuitable as a way to describe *Consumable Resources*.

VI. Conclusions and Future Work

This work shows how a GLUE 2.0 based multi-disipline scientific grid can be extended to support new models of parallel computing using GPGPUs. A methodology was developed that applied three abstract principles to this resource integration problem, namely: *Discovery*, *Independence* and *Exclusivity*.

The presented prototype is one of the first examples of where a GLUE 2.0 ApplicationEnvironment entity has been extended to include additional attributes that describe the capacity, utilisation and other properties of hardware used directly by the application itself. These attributes can be generated either statically or dynamically. The example use-case demonstrates a CUDA ApplicationEnvironment that is extended to include the total number of installed Nvidia GPGPUs, their current utilisation, and some selected hardware properties. This conforms to the *Discovery* principle.

A method was developed that allows a grid user to specify a GPGPU requirements in the Job Description Language. In particular, the prototype allows the user to specify the number of GPGPUs required per allocated Worker Node. The job requirement is converted into the native LRMS resource specification once the job enters the choosen Resource Centre. The method can also be applied to other resources made available through an LRMS. This is an application of the *Independence* principle.

Ensuring that users have guaranteed and isolated access to GPGPUs in a multi-user system can be difficult. This problem is compounded in the cases where multiple GPGPUs on the same machine can be accessed by multiple users - many batch systems do not indicate what GPGPU has been assigned to each user job. The worker node GPGPU Access Control system discusssed in Section IV-D can assuage this problem. The development of the GPGPU allocation handler for these systems ensures the *Exclusivity* principle.

Work is in progress to support enhanced job specifications, allowing the user to make job placement decisions based on a wider range of published GLUE 2.0 ApplicationEnvironment data. This allows for greater control over the resource discovery and selection process. An example based on published CUDA ApplicationEnvironment (Table II) is illustrated in Listing 4.

Listing 4: Example JDL using CUDA attributes

```
Requirements = GPUVendor=="Nvidia" && (↩
    GPUMainMemorySize >= 512);
```

Although the service demonstrated used Nvidia and CUDA, other environments such as AMD GPGPU and OpenCL can also be trivially accomodated - for example, AMD uses GPU_DEVICE_ORDINAL environment variable to restrict user visibility of AMD GPGPU devices [29]. In addition, this system can work in non-grid Torque/MAUI environments, and requires minor changes to work with other batch systems.

Finally, although the prototype implentation was tested with GPGPU resources, the model can be adapted to cater for wider range of resources, such as Intel's Xeon Phi, FPGAs and Licence controlled software. Said resources are difficult to integrate into grids based on GLUE 1.3. The prototype shows that: (i) there are cases where hardware resources can easily be treated like an ApplicationEnvironment; (ii) arbitrary information can be published about these resources - and this can be generated statically or dynamically; and (iii) job-requirements can be specified in a Job Description Langauge and transformed into batch-system directives without any major changes to the grid middleware.

ACKNOWLEDGMENT

This work was carried out on behalf of the Telecommunications Graduate Initiative (TGI) project. TGI is funded by the Higher Education Authority (HEA) of Ireland under the Programme for Research in Third-Level Institutions (PRTLI) Cycle 5 and co-funded under the European Regional Development Fund (ERDF). The authors also acknowledge the use of additional support and assitance provided by the European Grid Infrastructure. For more information, please reference the EGI - InSPIRE paper (http://go.egi.eu/pdnon).

REFERENCES

[1] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 3, pp. 200–222, Aug. 2001. doi: 10.1177/109434200101500302. [Online]. Available: http://dx.doi.org/10.1177/109434200101500302

[2] S. H. Fuller and L. I. Millett, *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011. ISBN 9780309159517. [Online]. Available: http://www.nap.edu/openbook.php?record_id=12980

[3] V. Kindratenko, "Computational Accelerator Term Revisted," https://www.ieeetcsc.org/activities/blog/Accelerated_Computing_computational_Accelerator_Term_Revisited, 02 2013.

[4] "Top 500 Supercomputers," http://www.top500.org/.

[5] W. G. et al, "MPI: The Message Passing Interface Standard, Version 2.2," http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report-book.pdf.

[6] J. Walsh, "Message Passing Interface - Current Status and Future Developments, ," EGI Technical Forum, 2010, September 2010.

[7] J. Walsh, B. Coghlan, K. Eigelis, and G. Sipos, "Results from the EGI GPGPU Virtual Team's User and Resource Centre Administrators Surveys," in *Crakow Grid Workshop 2012*, Crakow, Poland, October 2012.

[8] J. Walsh, "Presentation of Results from the EGI GPGPU Virtual Team Surveys," https://indico.egi.eu/indico/materialDisplay.py?contribId=162&sessionId=81&materialId=slides&confId=1019.

[9] H. Stockinger, "Defining the Grid: a snapshot on the current view." *The Journal of Supercomputing*, vol. 42, no. 1, pp. 3–17, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/tjs/tjs42.html#Stockinger07

[10] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004. doi: 10.1109/GRID.2004.14. ISBN 0-7695-2256-4 pp. 4–10. [Online]. Available: http://dx.doi.org/10.1109/GRID.2004.14

[11] "OGF GLUE 1.3 Specification," http://forge.gridforum.org/sf/go/doc14185.

[12] "OGF GLUE 2.0 Specification," http://glue20.web.cern.ch/glue20/.

[13] S. Burke, S. Andreozzi, F. Donno, F. Ehm, L. Field, M. Litmaath, and P. Millar, "The Impact and Adoption of GLUE 2.0 in the LCG/EGEE Production Grid," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062005, 2010. doi: 10.1088/1742-6596/219/6/062005. [Online]. Available: http://stacks.iop.org/1742-6596/219/i=6/a=062005

[14] S. B. et al., "The gLite 3.2 User Guide," https://edms.cern.ch/file/722398/1.4/gLite-3-UserGuide.pdf, July 2012.

[15] "Forward of Requirements To The Batch System," http://grid.pd.infn.it/cream/field.php?n=Main.ForwardOfRequirementsToTheBatchSystem.

[16] "Tornado HomePage," http://www.tornadoweb.org/, 2014.

[17] "CUDA Wrapper SourceForge Homepage," http://sourceforge.net/projects/cudawrapper/, 2014.

[18] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, "GPU clusters for high-performance computing," 2009. doi: 10.1109/clustr.2009.5289128 pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/clustr.2009.5289128

[19] S. Toth and M. Ruda, "Practical Experiences with Torque Meta-Scheduling in The Czech National Grid," J. Kitowski, Ed. Krakow, Poland: AGH University of Science and Technology Press, 2012, pp. 33–45.

[20] F. Vella, R. Cefala, A. Costantini, O. Gervasi, and C. Tanci, "GPU Computing in EGI Environment Using a Cloud Approach," in *International Conference on Computational Science and Its Applications (ICCSA) 2011*, 2011. doi: 10.1109/ICCSA.2011.61 pp. 150–155. [Online]. Available: http://dx.doi.org/10.1109/ICCSA.2011.61

[21] J. Breitbart and G. Khanna, "An Exploration of CUDA and CBEA for Einstein@Home," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 3-642-14389-X, 978-3-642-14389-2 pp. 486–495. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882851

[22] A. Visegrádi, J. Kovács, and P. Kacsuk, "Efficient Extension of gLite VOs with BOINC Based Desktop Grids," *Future Gener. Comput. Syst.*, vol. 32, pp. 13–23, Mar. 2014. doi: 10.1016/j.future.2013.10.012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2013.10.012

[23] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 323–356, Feb. 2005. doi: 10.1002/cpe.v17:2/4. [Online]. Available: http://dx.doi.org/10.1002/cpe.v17:2/4

[24] "How to Manage GPUs," https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToManageGpus, 2014.

[25] E. F. del Castillo, "Support to MPI Applications on the Grid." *Computing and Informatics*, vol. 31, no. 1, pp. 149–160, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/cai/cai31.html#Fernandez-del-Castillo12

[26] "Puppet IT Automation Homepage," http://puppetlabs.com/, 2014.

[27] "EGI GPGPU Working Group HomePage," https://wiki.egi.eu/wiki/GPGPU-WG, 2014.

[28] "EGI Federated Cloud Task Force Homepage," https://wiki.egi.eu/wiki/Fedcloud-tf, 2014.

[29] "AMD APP OpenCL Programming Guide," http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2014.