View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositori Institucional de la Universitat Jaume I

DEPARTAMENT D'ENGINYERIA I CIÈNCIA DELS COMPUTADORS
UNIVERSITAT JAUME I

SIU043
Trabajo Final de Máster

MÁSTER EN SISTEMAS INTELIGENTES
Curso 2013 / 2014

Memoria Técnica del Proyecto

Un gestor de GPUs remotas para clusters HPC

Proyecto presentado por el alumno:

Sergio Iserte Agut

Dirigido por ***Rafael Mayo Gual***

Castellón, a 29 de Julio de 2014

Resumen

SLURM es un gestor de recursos para *clusters* que permite compartir una serie de recursos heterogéneos entre los trabajos en ejecución. Sin embargo, SLURM no está diseñado para compartir recursos como los procesadores gráficos (GPUs). De hecho, aunque SLURM admita *plug-ins* de recursos genéricos para poder manejar GPUs, éstas sólo pueden ser accedidas de forma exclusiva por un trabajo en ejecución del nodo que las hospeda. Esto es un serio inconveniente para las tecnologías de virtualización de GPUs remotas, cuya misión es proporcionar al usuario un acceso completamente transparente a todas las GPUs del *cluster*, independientemente de la ubicación concreta, tanto del trabajo como de la GPU.

En este trabajo presentamos un nuevo tipo de dispositivo en SLURM, "rgpu", para conseguir que una aplicación desde su nodo acceda a cualquier GPU del *cluster* haciendo uso de la tecnología de virtualización de GPUs remotas, rCUDA. Además, con este nuevo mecanismo de planificación, un trabajo puede utilizar tantas GPUs como existan en el *cluster*, siempre y cuando estén disponibles. Finalmente, presentamos los resultados de varias simulaciones que muestran los beneficios de este nuevo enfoque, en términos del incremento de la flexibilidad de planificación de trabajos.

Palabras clave

SLURM, gestor de recursos, virtualización GPU, rCUDA

Agradecimientos

Por un lado quiero dar las gracias a todos los componentes del grupo de investigación HPC&A de la UJI, en especial a los miembros del Crazy Hector's Lab, por la armonía y el buen ambiente de trabajo que generan. También, agradecer a los miembros del grupo GAP de la UPV por la ayuda prestada para el desarrollo del proyecto y por haberme dejado utilizar su infraestructura para realizar los experimentos.

Por otro lado, no podría dejar de agradecer el apoyo de mi familia, haciendo una mención especial a mi abuelo Joaquín, mis padres Javier y Dina, mi hermano Jorge y por supuesto, mi querida Ana.

Índice general

I Memoria Técnica del Proyecto 9

1. Introducción 11

1.1. Motivación . . . 11

1.2. Objetivos . . . 13

1.3. Metodología . . . 13

1.4. Organización de la memoria . . . 13

2. Descripción del proyecto 15

2.1. Introducción teórica . . . 15

2.1.1. RPC . . . 15

2.1.2. Gestor de recursos . . . 15

2.1.3. SLURM . . . 16

2.1.4. GPU . . . 16

2.1.5. CUDA . . . 16

2.1.6. rCUDA . . . 16

2.2. Estimación de recursos . . . 17

2.3. Planificación temporal . . . 17

2.3.1. Identificación de tareas . . . 17

2.3.2. Estimación de la duración de las tareas . . . 18

2.4. Requisitos del proyecto . . . 18

3. Descripción de SLURM **21**
3.1. Introducción 21
3.2. Arquitectura 22
3.2.1. Demonio central *slurmctld* 24
3.2.2. Demonios locales *slurmd* 24
3.2.3. Demonio de la base de datos *slurmdbd* 25
3.3. Servicios y Operaciones 25
3.3.1. Comandos de usuario 25
3.3.2. Plug-ins 26
3.3.3. Capa de Comunicación 27
3.3.4. Seguridad 27
3.3.5. Inicio de los trabajos 28
3.3.5.1. Modo Interactivo 28
3.3.5.2. Modo Lote 30
3.3.5.3. Modo Reserva 30

4. Descripción de rCUDA **33**
4.1. Introducción 33
4.2. Arquitectura 33

5. Integración de la virtualización de GPUs con rCUDA en SLURM **37**
5.1. Introducción 37
5.2. Cambios en SLURM 38
5.2.1. Configuración 38
5.2.2. Estructura de los paquetes RPC 38
5.2.3. Ficheros comunes 39
5.2.3.1. Paquetes RPC 39
5.2.3.2. Módulo GRes (*Generic Resources*) 39
5.2.3.3. Variables de entorno 40
5.2.4. Demonio controlador *slurmctld* 40
5.2.4.1. Configuración 40
5.2.4.2. Comunicación 40
5.2.4.3. Trabajo 41
5.2.4.4. Etapa de trabajo 41
5.2.5. Plug-ins 41
5.3. Cómo utilizar la nueva funcionalidad de SLURM 42
5.4. Utilizando SLURM con GPUs virtualizadas 43

6. Evaluación de prestaciones **45**
6.1. Aplicaciones 45
6.1.1. GPU-BLAST 45
6.1.2. LAMMPS 45
6.1.3. MCUDA-MEME 46
6.1.4. GROMACS 46
6.2. Cargas de trabajos 46
6.3. Experimentación 47
6.4. Resultados 49

7. Conclusiones **53**

Bibliografía **55**

II Documentos Anexos **59**

A. Ficheros de configuración de SLURM **61**
A.1. slurm.conf 61
A.2. gres.conf 62
A.3. slurmdbd.conf 62

B. *Scripts* auxiliares **65**
B.1. GPU-Blast 65
B.2. LAMMPS 65
B.3. MCUDA-MEME CUDA 66
B.4. MCUDA-MEME rCUDA 66
B.5. GROMACS 66
B.6. genera_carga.py 67

C. SLURM for rCUDA User's Guide **69**
C.1. Introduction 69
C.2. Configuration 69
C.2.1. Applying the patch and installation 69
C.2.2. Controller node environment variables 70
C.2.3. Configuration files 70
C.2.3.1. slurm.conf 70
C.2.3.2. gres.conf 71
C.3. How to use modified SLURM 71
C.3.1. Submision options 72
C.3.1.1. Using `sbatch` 72
C.3.2. Submision examples 73
C.4. Further Information 73

D. Cambios realizados en los ficheros de SLURM **75**

Parte I

Memoria Técnica del Proyecto

Capítulo 1

Introducción

En este primer capítulo de presentación de la memoria, se introducirá al lector en la temática del proyecto, empezando con el apartado de motivación, en el que se describirá el interés del tema. A continuación se presentarán los objetivos específicos del proyecto y una breve descripción del entorno de trabajo en el que se ha desarrollado. Finalmente, se muestra la planificación temporal del proyecto y se evalúan los recursos necesarios para llevarlo a cabo.

1.1. Motivación

En los últimos años, la utilización de los aceleradores de cálculo en las instalaciones dedicadas a computación de alto rendimiento se ha consolidado como una clara opción para incrementar la productividad de estas instalaciones. En particular, la utilización de los procesadores gráficos (GPUs) para el cálculo de propósito general ha sido la opción más adoptada, llegando incluso a incorporarse en los grandes supercomputadores que aparecen en los primeros lugares de la lista Top500[10]. Esta incorporación se ha debido en parte a la aparición de entornos de programación, como CUDA y OpenCL, que permiten, de una forma más o menos sencilla, la ejecución de parte del código de las aplicaciones en las GPUs. Las GPUs son procesadores masivamente paralelos que ofrecen un gran rendimiento para aplicaciones con un alto grado de paralelismo de datos, mostrando un bajo rendimiento en aquellas partes de la aplicación que no presentan esta característica. Otro aspecto a tener en cuenta a la hora de utilizar las GPUs para cálculo de propósito general es que tienen un espacio de direccionamiento separado del de la CPU. Por ello la forma más común de utilizar las GPUS es la siguiente. Cuando se desea que una parte de la aplicación se ejecute en la GPU, se debe:

1. Copiar desde el espacio de memoria de la CPU al espacio de memoria de la GPU los datos que se van a tratar.
2. Cargar, también desde la memoria de CPU a la memoria de GPU, el código que debe ejecutar la GPU para tratar esos datos (que se suele denominar kernel).
3. Ordenar a la GPU que ejecute dicho código.

4. Copiar desde el espacio de memoria de la GPU al de la CPU los resultados generados por la ejecución de dicho código

Así pues, con esta forma de funcionamiento, las aplicaciones factibles de ser aceleradas utilizando GPUs son aquellas que presentan partes de código con un alto grado de paralelismo, y en las que el tiempo necesario para las distintas transferencias entre el espacio de direccionamiento de la CPU y el espacio de direccionamiento de la GPU, suponen un pequeño coste respecto a la reducción de tiempo que se obtiene al realizar la ejecución en la GPU. Por otra parte, la práctica totalidad de los grandes supercomputadores presenta una arquitectura de cluster, en donde un conjunto de cientos o miles de nodos colaboran en la ejecución de un código paralelo. Esta colaboración se realiza utilizando redes de altas prestaciones, entre las que destacan en la actualidad aquellas que se basan en el estándar de InfiniBand. En estos grandes supercomputadores, si se desea utilizar la potencia de cálculo de las GPUs, se debe incorporar al menos una GPU por nodo. Esto tiene diversos inconvenientes, que no pueden ser obviados:

- Económicos: incorporar el GPU a cada uno de los nodos incrementa TCO (Total Cost of Ownership), que incluye tanto los costes de adquisición como los de funcionamiento y mantenimiento de las instalaciones.
- Ecológicos: incorporar una GPU a cada uno de los nodos incrementa la energía necesaria para el funcionamiento del sistema, lo que no sólo repercute a nivel económico, sino también a nivel de la huella de carbono para la instalación.
- De rendimiento: no todas las aplicaciones son susceptibles de beneficiarse de su ejecución en una GPU. Ni siquiera aquellas que pueden obtener beneficio, lo pueden hacer durante todas las fases de su ejecución, sino que la mayor parte de veces sólo es posible en determinadas partes de la misma. Esto hace que se disponga de un hardware (GPU) que sólo se utiliza durante un cierto tiempo, pudiendo estar inactivo durante largos periodos.

Todo esto ha hecho que en los últimos años se hayan desarrollado tecnologías que permiten la virtualización de GPUs para el cálculo de propósito general basándose en el acceso a GPUs remotas que pueden ser compartidas entre distintos nodos. Así, con un menor número de GPUs, equipando sólo a algunos nodos con GPUs, se pueden obtener los mismos beneficios que dotando a todos los nodos de GPUs. Entre las diversas tecnologías de virtualización de GPUs podemos nombrar rCUDA[3][8] y vCUDA [9]. De estas tecnologías rCUDA es la que tiene un mayor grado de madurez, permitiendo el acceso a GPUs remotas utilizando las capacidades de las redes InfiniBand con un sobrecoste muchas veces despreciable respecto a la utilización de GPUs locales. Para que estas herramientas sean útiles en grandes instalaciones es necesario que se integren dentro de los entornos de planificación de trabajos que en ellas se utilizan. Esto es, los sistemas de colas de trabajos deben ser conscientes de la existencia de un nuevo recurso, las GPUs remotas, que puede ser compartido entre varios trabajos, y también deben poder controlar el uso que se hace de ellos. En este trabajo se presenta el desarrollo que se ha realizado para integrar rCUDA dentro de SLURM, de forma que los trabajos, además de solicitar los recursos estándar de cómputo y almacenamiento, también pueden solicitar la utilización de GPUs remotas, siendo todo ello gestionado por la herramienta de planificación de forma automática.

1.2. Objetivos

El objetivo de este trabajo es integrar la tecnología de virtualización de GPUs rCUDA en el gestor de recursos cluster SLURM, siguiendo estos pasos:

1. Estudiar el funcionamiento de rCUDA
2. Estudiar el funcionamiento de SLURM
3. Integrar la tecnología rCUDA en SLURM
4. Comparar tiempos de ejecución de cargas de trabajos con y sin rCUDA en SLURM.

1.3. Metodología

Tras haberse familiarizado con la estructura y el funcionamiento de rCUDA y SLURM, se procederá a insertar en el código de SLURM la lógica necesaria para que gestione las GPUs remotas como un recurso diferente. Este nuevo recurso deberá poder ser compartido (cuando se solicite) por los diferentes trabajos de las colas.

1.4. Organización de la memoria

La primera parte de esta memoria explica la parte técnica del proyecto, que se distribuye del siguiente modo:

En el capítulo 2 se describe a fondo la arquitectura y el funcionamiento del gestor de recursos SLURM.

A continuación, en el capítulo 3, se explica en detalle el proceso de instalación y configuración de SLURM en el cluster.

Para comprobar el funcionamiento de SLURM, se han generado unas cargas de trabajos. Estas cargas se describen en el capítulo 4 y sirven para ver el comportamiento de SLURM dependiendo del tipo de trabajo que se ejecuta. También en este capítulo, se analizan los resultados obtenidos tras las ejecuciones.

Finalmente, se dedica un capítulo a las conclusiones, las cuales recopilan las ideas y aspectos más interesantes, además del trabajo futuro.

En la segunda parte de la memoria, se encuentran los documentos anexos que amplían la información sobre el proyecto llevado a cabo.

Capítulo 2

Descripción del proyecto

En este capítulo se va a explicar el desarrollo completo del proyecto. Se parte de una serie de definiciones de los conceptos básicos relacionados. Se sigue con la estimación temporal y de costes de las tareas que lo componen. Se termina con la explicación de la ejecución completa del proyecto junto con los resultados. También se realiza una comparativa de los resultados generados en el proyecto con los requisitos iniciales, para comprobar su grado de cumplimiento.

2.1. Introducción teórica

En este apartado se explican los términos básicos y fundamentales para familiarizarse con las herramientas y sistemas con los que se trabaja en el proyecto y cuyo uso va a ser continuo a lo largo de la memoria. De este modo, se facilitará la compresión de las explicaciones que se encuentran a lo largo de este documento.

2.1.1. RPC

El Remote Procedure Call (RPC) (del inglés, Llamada a Procedimiento Remoto) es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos. El protocolo es un gran avance sobre los sockets usados hasta el momento. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC. Las RPC son muy utilizadas dentro del paradigma cliente-servidor. Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando éste de vuelta el resultado de dicha operación al cliente [?].

2.1.2. Gestor de recursos

Un gestor de recursos es una herramienta capaz de asignar los recursos disponibles dentro de un sistema, siguiendo unas directrices, a las aplicaciones que requieren el uso de esos recursos.

2.1.3. SLURM

SLURM es un sistema de gestión de recursos y de planificación de trabajos open-source, tolerante a fallos y escalable para todo tipo de clusters. SLURM incluye gestión de trabajos, gestión de particiones y control sobre el estado de las máquinas que forman el cluster. Además proporciona tres servicios clave:

- Reserva recursos de forma exclusiva o no para los usuarios durante un tiempo determinado, permitiendo a éstos utilizar dichos recursos.
- Ofrece un entorno para el lanzamiento y monitorización de los trabajos que se encuentran en el sistema.
- Dispone de un arbitraje para el manejo de la cola de tareas dependiendo de los recursos requeridos por los trabajos y aquellos disponibles.

2.1.4. GPU

La GPU es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo de la CPU en ciertas aplicaciones. Para el propósito de este proyecto se utilizan las GPUs para aprovechar la gran potencia de cálculo que ofrecen. El uso de GPUs para hacer cálculos de aplicaciones no relacionadas con los gráficos, se denomina GPUGPU, GPU de propósito general.

2.1.5. CUDA

CUDA son las siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por nVidia que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de nVidia.

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes (que es lo que hacen las GPU al procesar gráficos, su tarea natural), una GPU podrá ofrecer un gran rendimiento en campos que podrían ir desde la biología computacional a la criptografía por ejemplo [?].

2.1.6. rCUDA

rCUDA es una herramienta de virtualización de GPUs remotas que ofrece pleno soporte con CUDA, lo que significa que las aplicaciones originales no deben ser modificadas. rCUDA implementa todas las funciones del *CUDA Runtime API*, excepto las relacionadas con gráficos. Incluye comunicaciones altamente optimizadas sobre TCP e InfiniBand, además de tener capacidad para trabajar como multi-hilo y multi-nodo. rCUDA se puede utilizar en las mismas plataformas Linux que CUDA, proporcionando también soporte para arquitecturas x86 y ARM.

2.2. Estimación de recursos

- Un cluster de 9 nodos de los cuales 8 se configuraron como nodos de cómputo, mientras que el otro fue el *front-end*. Cada nodo venía equipado con una placa base Supermicro 1027GF-TRF, dos procesadores Intel Xeon E5-2620 (Ivy Bridge) hexa-core a 2.1 GHz y 32 Gbytes de memoria SRAM DDR3 a 1.6 GHz. Además cada uno de ellos contó con un acelerador NVIDIA Tesla K20 GPU conectado a la placa base a través de un PCIe 2.0 x16. La comunicación entre los nodos se hizo con un conmutador Mellanox SX6025 (InfiniBand FDR-compatible), ya que cada nodo tenía instalada una tarjeta de red Mellanox ConnectX-3 VPI single-port (InfiniBand FDR-compatible). El ancho de banda teórico de la configuración de la red era de 56 Gbytes/s.

  El sistema operativo instalado en cada nodo fue un CentOS 6.4; la red de comunicaciones ejecutó el controlador Mellanox OFED 2.1-1.0.0; y las GPUs utilizaron CUDA 5.5 y el controlador NVIDIA 331.62; la versión del virtualización de GPUs remotas rCUDA utilizada fue la 4.1.

- Una estación de trabajo con conexión a Internet para poder conectarse remotamente al *frontend* del cluster.

2.3. Planificación temporal

En esta sección del capítulo se identifican las tareas a llevar a cabo en el desarrollo del proyecto y su duración.

2.3.1. Identificación de tareas

A continuación se describe cada una de las tareas identificadas:

- **1: Estudiar la estructura de rCUDA**

  Análisis a fondo del funcionamiento de rCUDA.

- **2: Codificación de la nueva lógica de SLURM**

  Durante esta tarea se desarrollará todo el código necesario para la integración de rCUDA en SLURM. El desarrollo implica tanto modificaciones en el código fuente de SLURM, como creación de plug-ins y nuevas funciones.

- **3: Realización de pruebas**

  Esta actividad consiste en llevar a cabo todo tipo de pruebas y buscar posibles errores de la aplicación. En caso de encontrar errores se deberán solucionar y volver a repetir el proceso hasta que la aplicación funcione perfectamente.

1	Estudiar la estructura de rCUDA	15
2	Codificación de la nueva lógica de SLURM	205
3	Realización de pruebas	45
4	Redacción del manual de uso	5
5	Redacción de la memoria	20
6	Preparación de la presentación	9
7	Presentación del proyecto	1

Cuadro 2.1: Duración estimada de las tareas

- **4: Redacción del manual de uso**

  El manual de usuario es el documento de ayuda destinado al usuario.
  En esta tarea se redactará el documento que contendrá los pasos a seguir para utilizar la nueva versión de SLURM.

- **5: Redacción de la memoria**

  Durante esta tarea se redactará la memoria del proyecto. En ella se detallarán: los objetivos, los detalles de implementación, la documentación, etc.

- **6: Preparación de la presentación**

  En esta tarea se preparará el índice y las transparencias para la presentación del proyecto. También se preparará un ejemplo del funcionamiento y llevarán a cabo los ensayos de la exposición oral.

- **7: Presentación del proyecto**

  En esta última actividad se realizará la presentación del proyecto ante el tribunal. En la presentación se dará a conocer los principales resultados del proyecto desarrollado.

2.3.2. Estimación de la duración de las tareas

La estimación de la duración de las actividades se ha realizado en base a experiencias previas en otros proyectos. En la tabla 2.1 se pueden ver las duraciones previstas para cada tarea identificada.

2.4. Requisitos del proyecto

Esta sección define los requisitos necesarios para que el proyecto se considere satisfactoriamente finalizado.

- SLURM contará con un nuevo recurso para denominar las GPUs remotas (rGPUs).
- el funcionamiento de SLURM no se verá alterado, tan sólo ampliado.

- no se permitirá el uso simultáneo de los recursos genéricos GPU (local) y rGPU (remota).
- SLURM deberá poder asignar GPUs remotas a los trabajos que necesiten GPU para su ejecución.
- rCUDA deberá ejecutar la aplicación en las rGPUs que SLURM le ha asignado.
- SLURM deberá tener una política de selección de rGPUs (plug-in) que:
  - se base en la política de selección de recursos consumibles `select/cons_res`.
  - tenga en cuenta el número de rGPUs y cantidad de memoria solicitada por el trabajo.
  - seleccione rGPUs de una lista ordenada que contenga todas las rGPUs de la partición.
  - priorice la selección de rGPUs locales, es decir, de nodos asignados al trabajo.
- SLURM deberá escribir el resultado de la planificación de rGPUs en variables de entorno para la configuración de rCUDA.

Capítulo 3

Descripción de SLURM

Este capítulo describe la herramienta de gestión de recursos y planificación de trabajos para clusters Linux. Tras una introducción, se analiza a fondo la arquitectura y los servicios que ofrece SLURM.

3.1. Introducción

SLURM es una herramienta para Linux de código abierto, tolerante a fallos y altamente escalable para clusters Linux de todos los tamaños. El diseño resultante SLURM ofrece estas características:

- **Simplicidad**: es suficientemente simple de usar para un usuario que tenga interés.
- **Código abierto**: está disponible para todo el público de forma gratuita y se distribuye con licencia GNU.
- **Portabilidad**: se ha escrito en lenguaje C y utiliza el motor de configuración GNU *a*utoconf. Aunque inicialmente estaba dirigido a sistemas Linux, en la actualidad puede ser ejecutado en otros sistemas operativos basados en UNIX. Además el mecanismo de plug-ins que tiene SLURM permite configurarlo y ejecutarlo en diferente tipo de arquitecturas.
- **Independencia de interconexión**: admite varios protocolos de comunicación y tiene en cuenta diferentes topologías de red.
- **Escalabilidad**: el diseño permite ejecutar SLURM en cluster con miles de nodos, demostrando su alto rendimiento.
- **Tolerancia a fallos**: puede manejar diversos tipos de fallo sin necesidad de cancelar la carga de trabajos. Los recursos reservados para un trabajo en un nodo que haya sufrido algún problema, serán gestionados de modo que el trabajo disfrutará de los recursos en otro lugar.
- **Seguridad**: cuenta con diferentes tipos de encriptado para la autenticación de los usuarios en el sistema.

- **Amigable con el administrador de sistemas**: utiliza una configuración simple y mayormente centralizada, que se puede modificar en tiempo de ejecución sin que la carga de trabajos se vea afectada.

Como cualquier gestor de cargas de trabajos en clusters, SLURM tiene tres funcionalidades principales:

- Permite a los usuarios la reserva de recursos, en modo exclusivo y/o compartido, durante un tiempo determinado.
- Ofrece un entorno para el inicio, ejecución y monitorización de trabajos en un conjunto de nodos.
- Arbitra el uso de los recursos mediante la gestión de una cola de trabajos pendientes.

Opcionalmente, SLURM dispone de plug-ins con los que ampliar su funcionalidad y/o modificar su comportamiento.

3.2. Arquitectura

Por un lado, SLURM ejecuta en el nodo de gestión un demonio central *slurmctld* para monitorizar recursos y trabajos. También pueden haber demonios de respaldo que hagan el trabajo del controlador principal, en el caso de que este falle.

Por el otro, cada servidor de cómputo (nodo) ejecuta un demonio *slurmd*. Se puede decir que el demonio está a la espera de trabajos, cuando un trabajo llega, lo ejecuta y devuelve el estado. Acto seguido, se queda de nuevo esperando más trabajo. Estos demonios en los nodos de cómputo proveen al sistema un modo de tolerancia a fallos basado en las comunicaciones jerárquicas.

Finalmente, existe un demonio (aunque su uso es opcional) *slurmdbd* que se utiliza para guardar la información de registro de los eventos que suceden en el sistema, en una base de datos.

La comunicación entre los demonios (ver secciones 3.2.1, 3.2.2 y 3.2.3) y la interacción con el usuario (ver sección 3.3.1), se produce siguiendo el esquema de la figura 3.1.

Así pues, las entidades controladas por los demonios de SLURM son: recursos computacionales, particiones (agrupaciones lógicas de nodos), trabajos y etapas (conjuntos de tareas dentro de un trabajo). Por lo tanto, a un trabajo se le asigna un conjunto de recursos de un partición concreta. Una vez reservado los recursos, el usuario puede lanzar trabajos paralelos en forma de etapas de trabajo. Por lo que una sola etapa de trabajo puede utilizar todos los nodos, pero también varias etapas de un trabajo pueden utilizar parcialmente los recursos reservados para el trabajo. Por ejemplo: en la parte izquierda de la Figura 3.2 vemos como los 5 nodos asignados a un trabajo son utilizado por la misma etapa de trabajo. Al contrario, en la derecha el trabajo crea una tarea que utilizará dos nodos de una reserva de 4 nodos. Este comportamiento vendrá dado por la propia naturaleza del trabajo y la aplicación que ejecute éste. Es decir, el primer ejemplo corresponde a un trabajo en el que se crean procesos en todos los nodos reservados, mientras que en el segundo caso un trabajo a reservado nodos que la tarea que se encarga de la ejecución no va a utilizar.

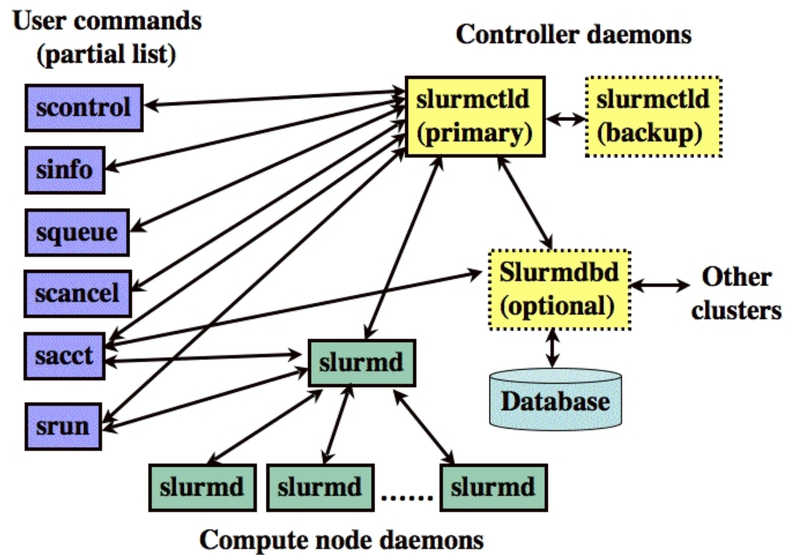

Figura 3.1: Componentes de SLURM

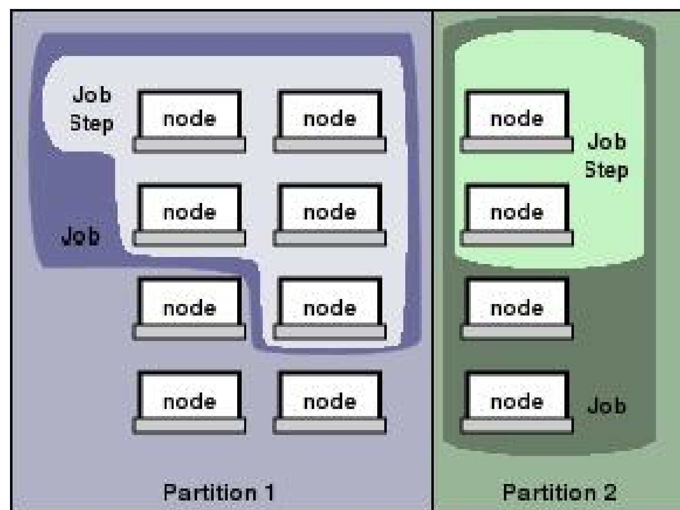

Figura 3.2: Entidades de SLURM

3.2.1. Demonio central *slurmctld*

Este demonio multi-hilo se ejecuta sólo en un nodo, llamado el nodo de gestión. Es el encargado de leer el fichero de configuración y mantiene información de los estados de las diferentes entidades. Este demonio está compuesto por tres subsistemas que se describen a continuación:

- **Gestor de nodos**: monitoriza el estado y la configuración de cada nodo del cluster. Él recibe los mensajes, de forma asíncrona, de cambio de estado de los demonios *slurmd*. También se encarga de pedir a los demonios de los nodos de cómputo informes periódicos del estado del nodo.
- **Gestor de particiones**: agrupa nodos en conjuntos disjuntos, asigna límites de trabajos y controla el acceso a cada partición. Este gestor se encarga de reservar los nodos para los trabajos (ante la petición del Gestor de trabajos) dependiendo de sus requisitos.
- **Gestor de trabajos**: procesa las peticiones de los trabajos y los prioriza en la cola. Periódicamente se encarga de revisar la cola para asegurarse de los trabajos que pidan recursos disponibles entren en ejecución. Cuando todos los nodos asignados a un trabajo informan a este gestor de que la tarea ha finalizado, el gestor de trabajos revisa de nuevo la cola de trabajos pendientes.

3.2.2. Demonios locales *slurmd*

En cada uno de los nodos de cómputo existirá una instancia de demonio local *slurmd*. Este demonio multi-hilo se ejecuta en modo superusuario, lo que otorga privilegios suficientes para ejecutar trabajos en nombre de otros usuarios. Este demonio lleva a cabo 5 tareas, correspondientes a estos subsistemas:

- **Estado de la máquina**: responde las peticiones de *slurmctld*, de forma asíncrona, referentes al estado del nodo.
- **Estado del trabajo**: responde las peticiones de *slurmctld*, de forma asíncrona, referentes al estado de los trabajos.
- **Ejecución remota**: inicia, monitoriza y limpia procesos pertenecientes a un trabajo.
- **Servicio de copia de flujo**: maneja los STDERR, STDOUT y STDIN para tareas remotas. Esto puede implicar redirección y siempre implica almacenamiento local de la salida de los trabajos para evitar bloqueo local de tareas.
- **Control del trabajo**: propaga señales y peticiones de terminación de trabajos a los procesos relacionados con SLURM.

Todo lo descrito en estos dos demonios se puede ver de manera esquemática en la figura 3.3.

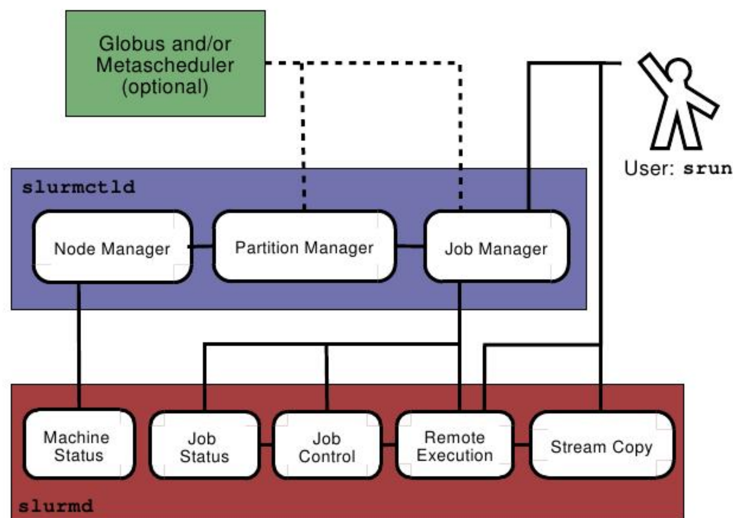

Figura 3.3: Arquitectura de los subsistemas de SLURM

3.2.3. Demonio de la base de datos *slurmdbd*

Para registrar todos los eventos y la información que estos generan, el demonio multihilo *slurmdbd* escribe los datos directamente en una base de datos. El controlador se comunica directamente con este demonio y también lo puede hacer el usuario a través del comando `sacct`. Este demonio puede registrar datos de diferentes clusters desde una única ubicación.

3.3. Servicios y Operaciones

Gracias a los servicios que ofrece SLURM el usuario puede interactuar con los demonios. A continuación, se describen estos servicios y se detalla como funcionan.

3.3.1. Comandos de usuario

El usuario puede interactuar con SLURM utilizando una serie de comandos disponibles. Estos comandos se pueden ejecutar desde cualquier lugar del cluster y son los siguientes:

- **srun**: envía trabajos a la cola o inicia etapas de trabajo. Cuenta con un gran número de opciones.
- **sbatch**: envía el script de un trabajo a la cola para su posterior ejecución. Este script suele contener una o varias invocaciones del comando srun para la ejecución de tareas en paralelo.

- **salloc**: reserva recursos para un trabajo en tiempo real. Este comando devuelve una shell con los recursos asignados al trabajo y en la cual se ejecutan comandos srun para lanzar trabajos en paralelo.
- **scancel**: cancela trabajos.
- **sinfo**: informa del estado del sistema.
- **squeue**: informa del estado de los trabajos.
- **sacct**: obtiene información de los trabajos y de las etapas que se están ejecutando o han terminado.
- **smap**: ofrece información de los trabajos, nodos y particiones. También muestra gráficamente la topología de la red.
- **sview**: es una interfaz de usuario que da información gráfica de los trabajos, nodos y particiones.
- **strigger**: determina disparadores, para los diferentes eventos que pueden suceder en el sistema.
- **sbcast**: retransmite un fichero desde el disco duro local al resto de discos de los otros nodos.
- **sattach**: añade una entrada o salida estándar (o de error) a un trabajo o a una etapa. También se puede quitar.
- **scontrol**: herramienta administrativa para monitorizar y modificar la configuración y estado del cluster.
- **sacctmgr**: herramienta administrativa para gestionar la base de datos.

3.3.2. Plug-ins

El mecanismo de plug-ins de propósito general que usa SLURM permite utilizarlo en diferente tipo de infraestructuras. Estos plug-ins son bibliotecas dinámicas que se cargan explícitamente en tiempo de ejecución. Los plug-ins proveen una implementación personalizada de una API bien conocida conectada a una funcionalidad determinada. Las funciones que se pueden ampliar y modificar son las relativas a estas tareas:

- Almacenamiento de datos históricos (*Accounting Storage*)
- Almacenamiento y recolección de datos energéticos (*Account Gather Energy*)
- Autenticación de las comunicaciones (*Authentication of communications*)
- Puntos de control (*Checkpoint*)
- Criptografía (*Cryptography (Digital Signature Generation)*)
- Recursos genéricos (*Generic Resources*)

- Envío de trabajos (*Job Submit*)
- Recolección de datos históricos de los trabajos (*Job Accounting Gather*)
- Registro de terminación de los trabajos (*Job Completion Logging*)
- Lanzadores de tareas (*Launchers*)
- Implementación de MPI (*MPI*)
- Adelantamiento de trabajos encolados (*Preempt*)
- Priorización de los trabajos (*Priority*)
- Seguimiento de procesos pertenencientes a trabajos (*Process tracking (for signaling)*)
- Selección de nodos (*Node selection*)
- Conmutación o interconexión (*Switch or interconnect*)
- Afinidad de tareas (*Task Affinity*)
- Topología de la red (*Network Topology*)

3.3.3. Capa de Comunicación

Para la comunicación entre sus procesos, SLURM utiliza sockets *Berkeley* y el protocolo de transporte de paquetes RPC.

3.3.4. Seguridad

El modelo de seguridad de SLURM se basa en las premisas:

- Cualquier usuario puede enviar y ejecutar trabajos paralelos al sistema, además de poder cancelar los suyos propios.
- Cualquier usuario puede ver la configuración y el estado del sistema.
- Sólo los usuarios con privilegios podrán cancelar cualquier trabajo y modificar la configuración de SLURM.

SLURM cuenta con varios mecanismos de autenticación vía plug-ins. Mientras que uno de ellos `none` no emplea ninguna credencial, `authd` y `Munge` utilizan criptografía para generar las credenciales. El más usado de estos mecanismos es `Munge` y funciona del siguiente modo:

Los demonios *munged* ejecutándose en cada nodo confirman la identidad del usuario y generan la credencial. Esta credencial contiene: el identificador del usuario, el identificador del grupo, la marca de tiempo, tiempo de vida y otros datos menos relevantes. El *munged* utiliza una clave privada para generar un mensaje de código de autenticación

(MAC) para la credencial. A su vez, utiliza una llave publica para encriptar la credencial que incluye el MAC. Los demonios SLURM transmiten esta credencial encriptada y cuando la reciben la envían al demonio *munged* del nodo. Una vez allí, la credencial se desencripta utilizando la llave privada y devuelve el identificador del usuario y del grupo del usuario que generó la credencial. En el caso de SLURM, la información de la credencial incluye el identificador del nodo, para asegurar que la información se use solamente en los nodos a los que va destinados.

Cuando los recursos son reservados por el demonio controlador de SLURM, se crea un *job step credential* combinando el identificador del usuario, del trabajo, de la etapa del trabajo, la lista de recursos reservados (nodos) y la credencial del tiempo de vida. Este *job step credential* es encriptado por *slurmctld* haciendo uso de su clave privada. Esta credencial se devuelve al solicitante (por ejemplo el proceso *srun*) junto con la respuesta de la reserva hecha. El controlador también se encarga de enviar la credencial creada a los demonios locales *slurmd* para el inicio de la etapa del trabajo. Estos demonios locales desencriptan la credencial con la llave publica del controlador para verificar que el usuario realmente tiene permisos para acceder a los recursos del nodo local. Además, también utiliza la credencial para la autenticación de los flujos de entrada, salida y error.

3.3.5. Inicio de los trabajos

Existen tres modos para que el usuario puede ejecutar trabajos en SLURM, en los siguiente apartados se describe como se lleva a cabo este proceso para cada uno de ellos.

3.3.5.1. Modo Interactivo

En el modo interactivo *stdout* y *stderr* se muestran en tiempo real a través del terminal del usuario. También *stdin* y las señales son enviadas a las tareas a través del terminal. En la figura 3.4 se puede ver el esquema general de conexiones que se producen durante el inicio de un trabajo interactivo. A continuación se describe:

1. *srun* solicita a *slurmctld* que se inicie un *job step* (etapa de un trabajo) que requiere que se reserven ciertos recursos.
2. Si la operación va bien, *slurmctld* le responde con el identificador del trabajo, la lista de nodos reservados y la credencial del trabajo.
3. *srun* inicializa puertos de escucha para cada tarea y envía un mensaje a cada *slurmd*, de los nodos reservados, para pedir que se inicie el proceso remoto.
4. Cada *slurmd* empieza a ejecutar su tarea y se conecta con *srun* para los flujos *stdout* *stderr*.

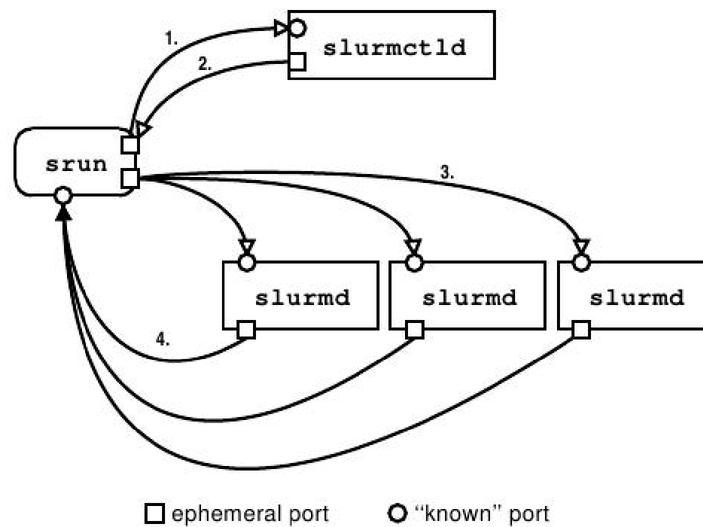

Figura 3.4: Esquema de las conexiones que se producen durante el inicio de un trabajo

EL proceso de inicio de un trabajo, desde que el usuario lo envía al sistema está esquematizado en la figura 3.5. Como se puede ver en ella, el usuario quiere ejecutar un trabajo desde su terminal. Para ello utiliza el comando *srun* que una vez procesado, pide en un mensaje que *slurmctld* le reserve los recursos y que se inicie un *job step*. En este punto, *srun* se mantiene a la espera de la respuesta de *slurmctld*, la cual puede tardar debido a que los recursos que se han solicitado no están disponibles. Cuando llega la respuesta, esta contiene el *job step credential*, la lista de nodos reservados, cpus por nodo, etc. Con esta información *srun* envía a cada *slurmd* de los nodos seleccionados una petición para que se inicie un *job step*. *slurmd* verifica que el trabajo sea válido revisando el *job step credential* para responder la petición de *srun*.

Cada *slurmd* crea un hilo para manejar el trabajo, también se invoca un hilo para cada tarea requerida. Estos hilos se conectan a un puerto abierto por *srun* para redirigir los datos que aparezcan en *stdout* y *stderr*. Hecho esto, es momento de que el hilo empiece a ejecutar el programa que el usuario ha enviado al sistema.

Cuando finaliza el programa del usuario, los hilos encargados de las tareas registran la salida y envían un mensaje avisando a *srun* de que su ejecución a terminado. Cuando todos estos procesos terminan, el hilo que manejaba el trabajo del usuario deja de existir. *srun* esperará a recibir el mensaje desde todos los nodos implicados para poder avisar al controlador de que el trabajo a terminado y ya no necesita los recursos reservados. Acto seguido, el proceso *srun* termina.

El controlador *slurmctld*, al recibir el mensaje de *srun* libera los recursos reservados y envía una petición a demonios locales involucrados para que ejecuten un epílogo, que no es más que un fragmento de código con instrucciones sobre que hacer en el momento de que el controlador libere los recursos.

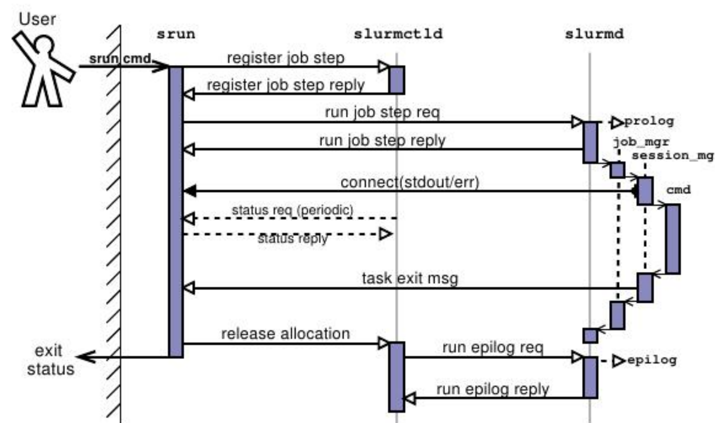

Figura 3.5: Inicio de un trabajo interactivo

3.3.5.2. Modo Lote

La figura 3.6 muestra el proceso que se sigue para iniciar un trabajo en modo lote, o lo que es lo mismo, un trabajo encolado. Una vez un trabajo en modo lote (*batch)* es enviado a la cola, *srun* envía a *slurmctld* una petición para ejecutar un trabajo en modo lote, que contiene la localización de la entrada y salida del trabajo a procesar, el directorio actual, el entorno y los recursos necesarios. *slurmctld* encola la petición en su cola de prioridad ordenada.

Una vez los recursos estén disponibles y la prioridad es la debida, *slurmctld* reserva los recursos para el trabajo y contacta con el primer nodo de la lista de reservados, para pedirle que inicie el trabajo del usuario. En este caso, el trabajo puede contener una llamada a *srun* o un *script* con múltiples invocaciones a *srun*. El demonio *slurmd* del nodo remoto responde al controlador iniciando: el trabajo (*job*), la tarea (*step*) y el *script* del usuario. Los procesos *srun* ejecutados desde el *script* tendrán acceso a los recursos reservados y podrán iniciar los *job step* en los nodos que dispongan.

Terminada la ejecución *job step*, el proceso *srun*, invocado desde el *script*, notifica a *slurmctld* y termina. El *script* continua su ejecución que puede contener más *job step*. Cuando la ejecución del *script* se completa, el hilo de la tarea que ejecutaba el *script* recoge el estado de la salida del programa y envía a *slurmctld*, un mensaje informando de esta salida. *slurmctld* se percata de que el trabajo a sido completado y envía a todos los demonios locales involucrados una petición para ejecutar el epílogo. Hecho esto, el controlador libera los recursos reservados.

3.3.5.3. Modo Reserva

En este modo, los usuarios esperan reservar recursos para un trabajo y de forma interactiva ejecutar *job step* en esa reserva. El proceso de inicio de trabajos en este modo (*alloc*) está representado en la figura 3.7.

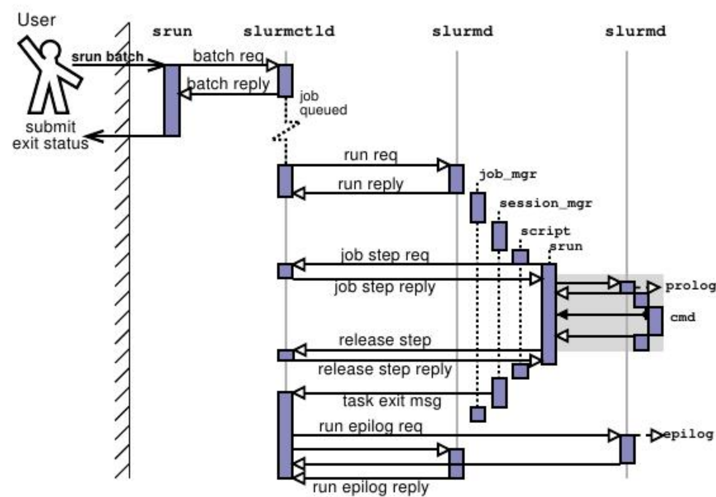

Figura 3.6: Inicio de un trabajo encolado

El proceso *srun* envía una petición de recursos, a la cual el controlador *slurmctld* responderá, si es posible, con una lista de nodos seleccionados, un identificador de trabajo, etc. El proceso *srun* abre una nueva *shell* en el terminal del usuario, que permitirá a éste acceder a los recursos reservados. *srun* esperará a que el usuario salga de la *shell* antes de considerar el trabajo completado.

Desde la nueva *shell* se puede lanzar *job step*, ya que ésta es el propio trabajo. El usuario mediante estos *job step* podrá solicitar recursos, siempre y cuando el trabajo los tenga asignados.

Cuando se ejecuta un *srun* desde la nueva *shell*, éste lee el entorno y las opciones que el usuario ha dado al trabajo, entonces notifica al controlador que se ha iniciado un nuevo *job step* dentro del trabajo. *slurmctld* registra el *job step* y responde a *srun* con la credencial del trabajo. A partir de aquí, *srun* inicia los trabajos con el mismo método que se ha visto en modo interactivo (sección: 3.3.5.1).

Finalmente, cuando el usuario sale del *shell*, el proceso *srun* original recibe el estado de salida, notifica a *slurmctld* y termina. Como en los otros casos, el controlador ejecutará el epílogo en los nodos reservados y los liberará.

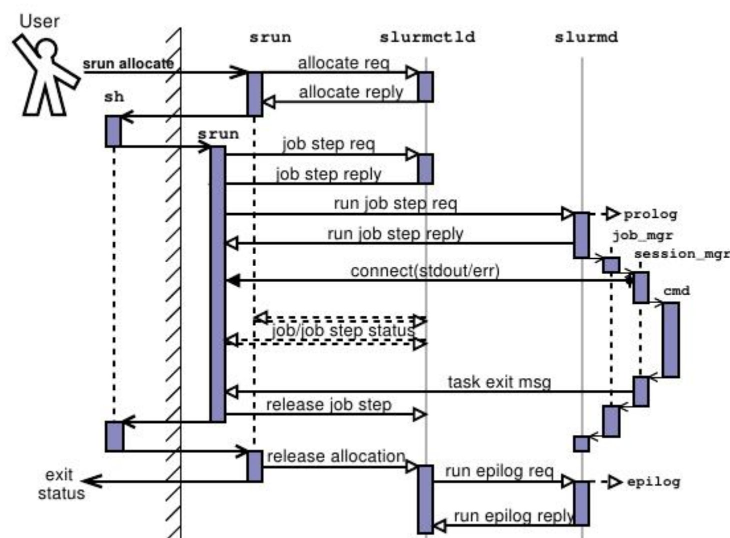

Figura 3.7: Inicio de un trabajo en modo reserva

Capítulo 4

Descripción de rCUDA

Este capítulo describe la herramienta de virtualización de GPUs rCUDA. Tras una introducción, se analiza a fondo la arquitectura y los servicios que ofrece rCUDA.

4.1. Introducción

rCUDA ofrece acceso transparente a cualquier GPU instalada en un *cluster*, independientemente del nodo donde se esté ejecutando la aplicación que solicita servicios GPGPU. Por tanto, rCUDA es realmente útil en los siguientes escenarios: *i)* en un *cluster* equipado con rCUDA se puede reducir el número total de GPUs del sistema, aumentando el ratio de utilización de los aceleradores hardware; *ii)* rCUDA también puede ser aprovechado para acelerar significativamente la computación de datos paralelos en un *cluster* convencional, añadiendo al sistema un número reducido de aceleradores, menor que el número total de nodos; *iii)* rCUDA aumenta el número de GPUs accesibles por una aplicación, que pasa de tener tan sólo las GPUs locales, a disponer de todas las disponibles en el *cluster*. En resumen, son muchos los casos en los que sacrificando un poco de tiempo de ejecución podemos obtener ahorros considerables en energía, mantenimiento, espacio y refrigeración.

4.2. Arquitectura

El entorno rCUDA está divido en dos grandes módulos cliente y servidor, como se describe en la Figura 4.1:

- La parte del cliente consiste en una colección de funciones envoltorio, las cuales reemplazan a NVIDIA CUDA Runtime (proporcionado por NVIDIA como una biblioteca compartida) en el nodo cliente (nodo sin GPU). La biblioteca cliente se encarga de enviar las funciones CUDA al servidor y recuperar el resultado. Este mecanismo hace creer a la aplicación que tiene acceso directo a la GPU física.

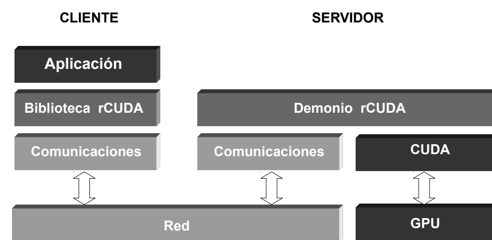

Figura 4.1: Esquema general de la arquitectura de rCUDA.

- La parte del servidor se ejecuta en cada nodo con al menos una GPU. El servidor recibe, interpreta y ejecuta la llamada a la API reenviada por el cliente. Se utiliza un proceso diferente para procesar cada ejecución remota sobre un contexto de GPU independiente, lo que permite la multiplexación de GPUs.

Como puede verse en la Figura 4.1, rCUDA responde a una arquitectura distribuida cliente-servidor. Los clientes hacen uso de la biblioteca de funciones envoltorio que realizan las llamadas a CUDA *runtime* API. Así pues, los clientes obtienen acceso a los dispositivos virtualizados, mientras que los nodos que hospedan los dispositivos físicos ejecutan los demonios que procesan las peticiones a CUDA. Ya que los clientes y servidores se comunican a través de la red, se hace uso de un protocolo personalizado de comunicaciones de alto rendimiento, para optimizar la transferencia de datos en la red.

Lado del Cliente: la biblioteca envoltorio CUDA *runtime* API se instala en un nodo sin GPU que necesite la capacidad de procesar código para GPU. Este cliente intercepta, procesa y reenvía las llamadas al servidor. Una vez la biblioteca de funciones envoltorio se carga dinámicamente, automáticamente se establece la conexión con el servidor (o servidores) especificado en la, correspondiente variable, de entorno. Cuando la biblioteca se descarga, la conexión se cierra automáticamente y los recursos son liberados. ésto evita la necesidad de extender el CUDA *runtime* API original, con funciones explicitas de inicialización y destrucción. Para cada llamada CUDA se produce esta secuencia de tareas: (1) comprobaciones locales (depende de cada función); (2) mapeados opcionales, p.e., para asignar identificadores a los punteros o localmente almacenar información que se recuperará más tarde; (3) empaquetar los argumentos junto con un identificador de función; (4) enviar la petición de ejecución al servidor; (5) en funciones síncronas, esperar a la respuesta del servidor.

Lado del servidor: el demonio servidor, localizado en los nodos que ofrecen servicios de aceleración, se encarga de recibir, interpretar y ejecutar las llamadas a CUDA. Por cada ejecución remota se crea un nuevo proceso (usando *pre-fork* para un mayor rendimiento) que ejecuta todas las peticiones de una sola aplicación remota, en un contexto de GPU independiente. Por tanto, la multiplexación de la GPU se consigue generando un proceso diferente en el servidor por cada ejecución remota sobre un nuevo contexto de GPU. Esto también asegura que el servidor sobreviva a una excepción en la que el proceso sea abortado (p.e., ante una llamada CUDA inapropiada).

Este diseño permite que en un *cluster* HPC donde los trabajos son planificados y asignados a diferentes procesadores de propósito general (CPUs), todas las GPUs pueden ser

compartidas, de forma segura, por distintos trabajos. Las aplicaciones pueden ser ejecutadas concurrentemente siempre y cuando haya suficiente memoria en el dispositivo y es el driver del dispositivo quien controlará la ejecución concurrente de aplicaciones en diferentes contextos activos, usando su propio planificador.

Capítulo 5

Integración de la virtualización de GPUs con rCUDA en SLURM

Este capítulo describe el trabajo realizado en cuanto a desarrollo y modificaciones de la versión original de SLURM, para soportar la virtualización de GPUs mediante rCUDA. El código fuente de SLURM está organizado de forma de árbol de directorios. Cada directorio contiene los ficheros relativos a una parte diferenciada del funcionamiento de SLURM. Para cada uno de estos ficheros se describen los cambios aplicados. Aparte de las modificaciones, también se han añadido nuevos plug-ins, que no dejan de ser nuevos directorios añadidos al árbol.

5.1. Introducción

Las modificaciones llevadas a cabo para conseguir la funcionalidad esperada, fueron las siguientes:

1. Se han añadido nuevas estructuras de datos a SLURM para poder trabajar con la información de las GPUS, requerida por los trabajos, particiones y nodos.
2. Se ha modificado el módulo GRes (*Generic Resources*), que es el encargado de gestionar los recursos genéricos de los nodos, en este caso las GPUs. Se ha añadido a este módulo la lógica necesaria para que todas las GPUs del *cluster* sean accesibles desde cualquier nodo, lo que implica que se puedan compartir entre los nodos.
3. Se han implementado dos nuevos plug-ins. El primero de ellos "`gres/rgpu`", declara un nuevo recurso genérico en el sistema, la *GPU remota*. El segundo "`select/cons_rgpu`" es responsable de seleccionar los recursos, en especial las *rGPU*. El código de este plug-in de selección de recursos está basado en el ya existente "`select/cons_res`", por lo que se puede esperar un comportamiento similar entre ambos.
4. En los paquetes RPC, usados en las comunicaciones entre los demonios, se han añadido nuevos campos con información de *rGPU*.
5. Finalmente, se ha añadido el código necesario para escribir las variables de entorno que leerá rCUDA:

- RCUDA_DEVICE_COUNT, da a conocer el número de dispositivos que necesita este trabajo.
- RCUDA_DEVICE_*X*, indica la IP del nodo donde la *rGPU* está instalada.

Tras estos cambios, el usuario puede enviar trabajos a la cola de tres modos diferentes:

- **Original (SLURM)**: El comportamiento de SLURM es el correspondiente a la versión 2.6.2.
- **Exclusivo (rCUDAex)**: SLURM desacopla las GPUs de los nodos, aunque éstas sólo pueden ser usadas por un trabajo a la vez.
- **Compartido (rCUDAco)**: Igual que el modo exclusivo, pero los trabajos pueden compartir las GPUs.

5.2. Cambios en SLURM

A continuación se explican los cambios en el código que se llevaron a cabo. Además, en el Anexo D se puede ver el resultado de la ejecución del comando `diff` aplicado a la versión original y la versión modificada.

5.2.1. Configuración

Para la configuración de la instalación se añadieron todos los plug-ins creados al fichero `./configure.ac`. Añadir estos plug-ins a la configuración global, permite que se generen automáticamente sus `Makefile`.

```
src/plugins/select/cons_rgpu/Makefile
src/plugins/gres/rgpu/Makefile
```

5.2.2. Estructura de los paquetes RPC

La comunicación entre procesos SLURM se lleva a cabo intercambiando paquetes RPC. Para poder transmitir la información referente a las rGPUS, se modificaron las estructuras de datos para que albergaran esta información. Esta información se debe transmitir entre: procesos de envío de trabajos a la cola (`srun`, `sbatch` y `salloc`), demonio controlador, demonios locales y procesos encargados de las etapas de los trabajos. Por tanto, se han añadido nuevos campos de estructuras de datos intercambiadas entre los procedimientos de envíos de trabajos, reservas de recursos o creación de etapas de trabajos.

A las estructuras afectadas se les ha añadido estos campos:

```
uint32_t nrgpu;        /* cantidad de rgpus asignadas */
char *rgpulist;        /* lista de rgpus asignadas */
```

5.2.3. Ficheros comunes

En el directorio `./src/common` se encuentra el código común que ejecutan todos los procesos SLURM.

5.2.3.1. Paquetes RPC

Los paquetes RPC enviados entre procesos pasan por una fase de empaquetado, al ser enviados, y una fase de desempaquetado, al ser recibidos. Como se han añadido nuevos campos a las estructuras de datos de los paquetes, también se tienen que adaptar las funciones encargadas de estos procedimientos para que empaqueten y desempaqueten los nuevos campos referentes a las rGPUs.

La información de las rGPUs viaja de unos procesos a otros, concretamente, entre el proceso que envía el trabajo a la cola y el demonio controlador; y a la hora de distribuir tareas sobre los nodos.

Ejemplos de empaquetado y desempaquetado:

```
packstr(msg->rgpu_list, buffer);
safe_unpackstr_xmalloc(&tmp_ptr->rgpu_list, &uint32_tmp, buffer);
```

5.2.3.2. Módulo GRes (*Generic Resources*)

El módulo GRes de SLURM se encarga de gestionar los recursos genéricos como las GPUs. Como se ha explicado anteriormente, este módulo es incapaz de compartir recursos entre trabajos, ni entre nodos. Por lo tanto, cualquier operación que involucre a algún recurso genérico, pasará por este código.

La idea general para conseguir que este módulo se comportara como se esperaba fue:

1. Llamar a la función que reserva rGPUs, desde la política de selección (se explica más adelante).
2. Recorrer las rGPUs del nodo afectado.
3. Comprobar que la rGPU tenga suficiente memoria libre.
4. Actualizar los registros del trabajo con el identificador de la rGPU, actualizar los registros de la rGPU con el identificador del trabajo.

5.2.3.3. Variables de entorno

Para hacer llegar a rCUDA la información sobre la planificación de rGPUs, se escriben variables de entorno una vez se han asignado recursos al trabajo. En concreto se escribe:

- `RCUDAPROTO`: determina el protocolo de comunicación que utilizará el cliente de rCUDA. Antes de escribir el protocolo "TCP", utilizado por defecto, se lee del entorno esta misma variable, por si hay establecido otro protocolo de comunicación.
- `LD_LIBRARY_PATH`: se añade la ruta de las bibliotecas de rCUDA o CUDA, rGPU o GPU respectivamente, dependiendo del tipo de recurso que ha solicitado el trabajo.
- `RCUDA_DEVICE_COUNT`: indica el número de dispositivos reservados por el trabajo. Estas rGPUs pueden ser compartidas o no.
- `RCUDA_DEVICE_X`: para cada dispositivo reservado se indica el nombre del nodo donde se ubica y el identificador del dispositivo en ese nodo. Por tanto, existirá una variable de entorno para cada rGPU asignada al trabajo llamada `RCUDA_DEVICE_X`, donde la `X` es identifica a la rGPU para ese trabajo.

Las variables de entorno, arriba descritas, se escriben en cualquier modo de envío de trabajo: `srun`, `sbatch` y `salloc`.

5.2.4. Demonio controlador *slurmctld*

En el directorio `./src/slurmctld` se encuentran los ficheros referentes al demonio controlador. Las modificaciones han afectado las siguientes áreas:

5.2.4.1. Configuración

Durante el arranque del demonio y tras haber leído la configuración desde el fichero, el controlador hace un recuento global de rGPUs por partición y no por nodo, como suele hacer con todos los recursos genéricos. Por tanto, para cada partición se analizan sus nodos en busca de rGPUs y así calcular el total de la partición.

5.2.4.2. Comunicación

El demonio controlador procesa los RPC para reservar recursos para un trabajo, además se encarga de responder a los procesos que piden recursos para trabajos, con la información de las rGPUs asignadas al trabajo.

5.2.4.3. Trabajo

Una vez procesado el mensaje que informa que un nuevo trabajo ha sido enviado a la cola, es momento de analizar semánticamente las especificaciones del trabajo. Para ello, el módulo gestor de trabajos analiza de forma habitual todos los recursos requeridos por el trabajo, poniendo especial atención a los recursos del tipo rGPU. En caso de que el trabajo solicite recursos de este tipo, las funciones modificadas y añadidas a SLURM se encargan de separar las rGPUs de los recursos genéricos habituales. De este modo, se lleva a cabo el análisis semántico de las rGPUs aparte, sin interferir en el funcionamiento original. Después de este proceso, se obtiene una instancia del trabajo, que es la que almacena toda la información referente a él. La selección de los recursos exactos que se van a asignar a un trabajo se realiza a continuación y es ejecutada por el plug-in `select/cons_rgpu`.

5.2.4.4. Etapa de trabajo

Del mismo modo que se crea un nuevo trabajo, se crean las etapas. El procedimiento es análogo: se crea la instancia de la etapa y se procesan sus parámetros. Aunque, en este caso no se llama al selector de recursos, porque cada etapa pertenece a un sólo nodo asignado previamente al trabajo. La reserva de recursos para la etapa se hace a través del módulo GRes que utiliza las nuevas funciones que operan con rGPUs.

5.2.5. Plug-ins

Todos los plug-ins que admite SLURM se encuentran en el directorio `./src/plugins`, aunque cada tipo de plug-in se encuentra en un subdirectorio:

- El recurso rGPU se declara en el sistema añadiendo un nuevo plug-in `./src/plugins/gres/rgpu`.
- La selección de rGPUs se lleva a cabo mediante una política implementada en el plug-in `./src/plugins/select/cons_rgpu`. Antes de empezar el proceso de reserva de recursos, se comprueba que el trabajo disponga de los recursos requeridos. Para ello, se evalúa cada rGPU y su capacidad de albergar el trabajo. En caso de no disponer de recursos, el trabajo se encolará (si los recursos están temporalmente ocupados) o bien se descartará (si no existen suficientes recursos en el sistema). El algoritmo prioriza la reserva de las rGPUs locales. Lo que significa que primero se buscan rGPUs disponibles en los nodos que conforman la lista de nodos asignados al trabajo. Si éstas no son suficientes, se seguirán buscando rGPUs por la partición hasta satisfacer la demanda.

  Tras la ejecución del trabajo, este plug-in se encarga de llamar a las rutinas que liberan recursos. Como cualquier operación sobre rGPUs, ésta se ejecuta en el módulo GRes a través de las nuevas funciones añadidas.

5.3. Cómo utilizar la nueva funcionalidad de SLURM

Para empezar, el fichero de configuración ‘‘slurm.conf’’ debe incluir las siguientes líneas:

```
RcudaModeDefault=(exclusive | shared)
RcudaDistDefault=(global | node)
RgpuMinMemory=256

SelectTypeParameters = CR_CORE
SelectType = select/cons_rgpu
GresTypes = gpu,rgpu
```

Estas líneas indican a SLURM las configuraciones por defecto que se utilizarán en caso de que no se especifique el modo, la distribución y la memoria mínima que se reservará de cada rGPU. El modo indica si la rGPU podrá ser compartida por varios trabajos simultáneos. La distribución determina si la cantidad de rGPUs requeridas es global, o bien, la cantidad por nodo. La cantidad mínima de memoria se asignará cada trabajo que requiera rGPUs en modo compartido y no haya sido configurado con una cantidad de memoria durante el envío.

Tras configurar los parámetros por defecto, deberemos establecer como plug-in de selección ‘‘select/cons_rgpu’’. Debido a que para otorgar rGPUs a un trabajo es necesario asignar como mínimo un core, es necesario configurar el core como unidad mínima de planificación. Además, se indica que existen los recursos genéricos gpu (local GPU) y rgpu (GPU remota) en el sistema. En este mismo fichero también se tienen que configurar los nodos, por ejemplo:

```
NodeName=mlxc2 CPUs=12 Sockets=1 CoresPerSocket=12 ThreadsPerCore=1 Gres=gpu:2,rgpu
    :2
```

Aparte de la configuración habitual del nodo, este fragmento indica a SLURM que el nodo mlxc2 tiene dos GPUs que pueden ser usadas de forma local o remota.

El fichero de configuración global se puede ver en A.1.

Finalmente, el proceso de configuración concluirá indicando a SLURM las propiedades de sus recursos genéricos, para el caso anterior sería:

```
Name=gpu File=/dev/nvidia0
Name=gpu File=/dev/nvidia1
Name=rgpu File=/dev/nvidia0 Cuda=3.5 Mem=5G
Name=rgpu File=/dev/nvidia1 Cuda=3.5 Mem=5G
```

Notar que en estos casos la GPU local sólo necesita su ruta, mientras que a la GPU remota hay que añadirle también la versión de *compute capability* y la memoria.

El fichero de configuración de recursos genéricos utilizado se puede ver en A.2.

5.4. Utilizando SLURM con GPUs virtualizadas

Una vez el controlador `slurmctld` se inicia, hace un conteo de rGPUs en cada partición del *cluster*. Tras esto, SLURM se mantiene a la espera de trabajos. Nosotros mostraremos el modo de operación utilizando `srun`, aunque también se podría utilizar `salloc` y `sbatch`. En concreto:

```
srun -N1 --gres=rgpu:4:1G job.sh
```

Con esta línea se pide a SLURM que reserve 1 nodo y 4 GPUs remotas con 1 GB de memoria cada una, para así poder ejecutar el *script* `job.sh`. También se puede enviar trabajos al planificador indicándole específicamente en qué nodos se deben ejecutar:

```
srun -w‘‘mlxc2’’ --gres=rgpu:4:1G job.sh
```

En este caso se fuerza a que la reserva sea del nodo `mlxc2`, no importa que el nodo tenga una GPU o no, aunque en este último caso el nodo deberá acceder a 4 GPUs remotas.

Después de enviar el trabajo entra en juego el plug-in `‘‘select/cons_rgpu’’`. Éste verificará si el trabajo no necesita rGPUs para hacer la planificación tal y como lo haría el plug-in original `‘‘select/cons_res’’`. En caso de que sí necesite hacer uso de rGPUs, se activará el `rgpu_mode`.

Aparte de pedir el número de rGPUs y la memoria para un trabajo, también se ha implementado la posibilidad de limitar la mínima *compute capability* de las rGPUs. Además, se han añadido parámetros para configurar el modo de utilizar las rGPUs, modo exclusivo o compartido. En el Anexo **??** se puede encontrar una explicación detallada de como utilizar cada opción implementada.

Dependiendo de los recursos requeridos y disponibles, el trabajo se ejecutará o se encolará. Cuando se ejecute, los recursos que ha pedido el trabajo permanecerán reservados. La reserva se lleva a cabo en el módulo GRes, donde las estructuras de datos se actualizarán debidamente. La política de selección utilizada prioriza el uso de rGPUs localizadas en el nodo de ejecución. Si la demanda no ha sido totalmente satisfecha, el algoritmo seguirá buscando recursos en una lista ordenada hasta satisfacerla.

Cuando termine la ejecución del trabajo los recursos anteriormente reservados serán liberados para poder ser usados por otros trabajos.

Capítulo 6

Evaluación de prestaciones

En este capítulo hemos llevado a cabo una evaluación del nuevo módulo SLURM. Para ello hemos configurado un cluster con la versión de SLURM que trabaja con rCUDA; hemos estudiado la escalabilidad de varias aplicaciones científicas; hemos centrado nuestro estudio en dos tipos de configuración de lanzamiento de aplicaciones, rendimiento y productividad; hemos probado el sistema reduciendo el número de recursos disponibles para comparar los resultados entre GPUs físicas y remotas.

6.1. Aplicaciones

Aunque se podría haber utilizado trabajos sintéticos basados en el comando `sleep` para generar carga y evaluar a SLURM, se decidió utilizar aplicaciones funcionales que se utilizan en entornos reales. Esto da mayor solidez a nuestra fase de experimentación ya que estará exenta de recibir comentarios que ataquen a su falta de realismo. Por tanto, para generar una carga de trabajos heterogénea y real se seleccionaron estas aplicaciones debido a sus características:

6.1.1. GPU-BLAST

Versión acelerada con GPUs de la herramienta bioinformática BLAST, la cual se usa para la búsqueda y alineación de secuencias de proteínas biomoleculares. Esta versión `1.1_ncbi-blast-2.2.26` de GPU-BLAST se ejecuta en un único procesador, aunque puede crear varios hilos de ejecución paralela. [1]

6.1.2. LAMMPS

Simulador de dinámica molecular con un alto grado de paralelismo. Desde la perspectiva de su implementación, la aplicación es multi-hilo, multi-proceso y necesita al menos 1 GPU, pero puede aprovechar utilizar varias. Esta versión (`1Feb14 - mvapich2`) está destinada a utilizar MPI sobre InfiniBand. [2]

[1] http://archimedes.cheme.cmu.edu/?q=gpublast

[2] http://lammps.sandia.gov

6.1.3. MCUDA-MEME

Versión del algoritmo de descubrimiento de motivos MEME (versión `4.4.0`), que combina CUDA, MPI y OpenMP para el procesamiento en paralelo. Con características similares a LAMMPS, MCUDA-MEME es multi-hilo y multi-proceso, pero debido a su implementación necesitará una GPU por cada proceso. La versión de MPI utilizada es MVAPICH2 que nos permitirá utilizar la red InfiniBand. [3]

6.1.4. GROMACS

Simulador de dinámica molecular que procesa ecuaciones Newtonianas de movimiento de cientos de millones de partículas. Su principal uso es en moléculas bioquímicas, como las proteínas, lípidos y ácidos nucleicos los cuales tienen muchas interacciones complicadas. La versión que hemos utilizado (`4.6.5`) utiliza procesos MPI y hilos OpenMP para paralelizar sus cálculos. La aplicación está preparada para obtener unos resultados similares para un mismo número de cores sin importar la distribución procesos/hilos. GROMACS es la única aplicación utilizada que no precisa de GPUs para su ejecución. [4]

Todas las aplicaciones que realizan cálculo en GPUs se encuentran en el catálogo de aplicaciones de NVIDIA. [5].

6.2. Cargas de trabajos

La combinación de rCUDA y SLURM permite que las GPUs sean compartidas por varios trabajos. Para ello, es necesario indicar la máxima cantidad de memoria requerida por cada trabajo. Debido a que el controlador de la GPU está cargado en memoria de la GPU, no toda la memoria de ésta estará disponible para nuestros trabajos. Lo mismo pasa con el demonio de rCUDA que también se encuentra cargado. Es más, por cada proceso que sea gestionado por rCUDA, se necesitará un demonio independiente. A lo que hay que sumar siempre un demonio rCUDA ocioso (cargado en memoria de GPU) a la espera de trabajo. Por lo tanto, el valor a calcular se corresponde a la memoria del propio trabajo, a la memoria del controlador de la GPU y a la memoria del demonio de rCUDA. Así pues, la cantidad máxima de memoria vendrá dada por la expresión: $App_{MaxMem} + NVIDIA\ Controlador + (rCUDAd \cdot (hilos + 1))$.

Hemos generado tres cargas de diferente duración teórica. Estas cargas se han generado aleatoriamente, pero son reproducibles si se mantiene la semilla de las funciones que calculan números aleatorios. El generador de cargas de trabajo recibe el tiempo teórico que debería durar la ejecución, por lo que va añadiendo nuevos trabajos aleatorios que se encargan de una de las cuatro aplicaciones. Elegir una aplicación para añadir a la carga, es un proceso aleatorio donde la probabilidad de cada una de ser elegida es de un 25 %. El código encargado de generar las cargas de trabajos se puede ver en B.6. Destacar que el

[3] https://sites.google.com/site/yongchaosoftware/mcuda-meme
[4] http://www.gromacs.org
[5] http://www.nvidia.com/object/gpu-applications.html

tiempo teórico debería ser similar a la duración de la ejecución secuencial de los trabajos. En nuestro caso, los trabajos se solapan por lo que el tiempo de ejecución es mucho menor (como se verá en la siguiente sección). La tabla 6.1 muestra los tiempos de ejecución de cada aplicación y este es el tiempo que utiliza el *script* para añadir trabajos a la carga, mientras la suma del tiempo de los trabajos no supere el tiempo teórico dado.

En los Anexos B.1, B.2, B.3, B.4 y B.5 se encuentran los *scripts* que se ejecutan con el comando de SLURM `salloc`.

Cuadro 6.1: Máximo rendimiento de las aplicaciones

Aplicación	Configuración	Tiempo de ejecución (s)
GPU-Blast	1 proceso con 6 hilos	21
LAMMPS	5 procesos mono-hilo en 5 nodos diferentes	15
MCUDA-MEME	4 procesos mono-hilo en 4 nodos diferentes	165
GROMACS	2 procesos, con 12 hilos cada uno, en 2 nodos	167

La tabla 6.2 contiene la descripción detallada, para cada duración, de la cantidad de instancias de cada aplicación. El orden de los trabajos es independiente del tipo de carga, por lo que las modificaciones en la configuración de los trabajos no supone una llegada de trabajos a la cola distinta.

En resumen, tenemos 4 escenarios con 6 cargas cada uno. Además, esas cargas se dividen en 2 grupos dependiendo si requieren GPUs o rGPUs. Ambos grupos cuentan con las 3 duraciones teóricas de carga.

Cuadro 6.2: Descripción de las cargas de trabajos

Aplicación	Carga		
	2 horas	4 horas	8 horas
GPU-Blast	12	43	81
LAMMPS	18	47	90
MCUDA-MEME	18	36	77
GROMACS	23	42	79
Total	71	168	327

6.3. Experimentación

Para esta fase del proyecto hay que aclarar cómo fue configurado SLURM. Para poder permitir adelantamientos entre trabajos se utilizó la política de planificación `backfill`, que permite el adelantamiento de trabajos. Además, la selección de recursos consumibles fue realizada por la política `cons_rgpu` que es la única que permite el uso de GPUs remotas.

Utiliza la implementación MVAPICH2 de MPI, especialmente preparada para la tecnología InfiniBand. Y para enviar trabajos a la cola escogimos el comando `salloc` para

trabajos multi-proceso, ya que `srun` necesita que la aplicación esté enlazada a la implementación de la biblioteca PMI de SLURM.

En los experimentos llevados a cabo se pueden distinguir fácilmente tres propósitos:

1. Conseguir el mayor rendimiento posible. A partir de los datos extraídos de un análisis de escalabilidad, hemos obtenido para cada aplicación su óptima configuración de ejecución en nuestro cluster. Esta análisis medía tiempos de ejecución de las aplicaciones con diferente configuración de número de procesos e hilos, siempre cuando fuera posible. Aunque nuestro cluster estuviera equipado con 8 GPUs, restringimos a 4 el uso de GPUs para MCUDA-MEME, ya que nuestro objetivo era reducir el número de GPUs en el cluster (hasta 4 unidades) manteniendo el número de nodos de cómputo. Por el contrario, LAMMPS se configura con 5 GPUs debido a su implementación. La tabla 6.3 resume la mejor configuración para ambos modos de ejecución. Es importante matizar que la búsqueda del máximo rendimiento es perfectamente entendible cuando hayan pocos trabajos a ejecutar. Sin embargo, nuestras cargas contienen un considerable número de trabajos, por lo que esta configuración será propensa a desperdiciar recursos, ya que muchos trabajos esperarán a que se liberen el resto de recursos.

2. Aumentar la productividad global. Para este propósito se han añadido varios grados de libertad que evitan que cada proceso se ejecute en un nodo diferente. Aunque sabemos que esta configuración no va a mapear de manera óptima los recursos a los trabajos, pero si que se reducirá el tiempo de inactividad de muchos recursos. La tabla 6.4 refleja como aplicaciones tales como GPU-Blast y LAMMPS, no fuerzan a sus procesos a ejecutarse en diferentes nodos (se ha eliminado el argumento `-N`). Sin embargo, también se aprecia que las configuraciones de MCUDA-MEME y GROMACS permanecen sin cambios. La explicación para MCUDA-MEME es que la aplicación necesita 4 GPUs (uno por proceso); LAMMPS es capaz de usar la misma GPU para diferentes procesos, mientras que MCUDA-MEME no es capaz. GROMACS experimenta un aumento dramático en el tiempo de ejecución si dos o más instancias comparten un nodo, debido a que los hilos se mapean a los mismos cores en todas las instancias. Por este motivo, decidimos dar a cada instancia acceso exclusivo a dos nodos. Tras estos cambios se observó un sustancial aumento en la productividad de trabajos por minuto.

3. Nuestro plan era reducir progresivamente el número de GPUs en el nodo, desde 8 hasta 6 y 4. Así pues, se modificó la orden de lanzamiento de LAMMPS para que casara con la nueva "plataforma". Esto supuso cambiar a `-n5 -c1 --gres=rgpu:4:3275M` la orden de LAMMPS, porque ya no habían 5 GPUs en el cluster.

Finalmente, a la hora de enviar tan gran ráfaga de trabajos en un tiempo tan reducido, experimentamos que el controlador de SLURM se saturaba. Para paliar este imprevisto se añadió un retraso entre cada envío. Este retraso aumenta a medida que los trabajos son encolados partiendo desde un valor cercano al cero.

Cuadro 6.3: Parámetros de lanzamiento para obtener el máximo rendimiento individual

Aplicación	Lanzamiento con CUDA	Lanzamiento con rCUDA
GPU-Blast	-N1 -n1 -c6 –gres=gpu:1	-N1 -n1 -c6 –gres=rgpu:1:1686M
LAMMPS	-N5 -n5 -c1 –gres=gpu:1	-N5 -n5 -c1 –gres=rgpu:5:3275M
MCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:163M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12

Cuadro 6.4: Parámetros de lanzamiento para obtener la máxima productividad global

Aplicación	Lanzamiento con CUDA	Lanzamiento con rCUDA
GPU-Blast	-n1 -c6 –gres=gpu:1	-n1 -c6 –gres=rgpu:1:1686M
LAMMPS	-n5 -c1 –gres=gpu:1	-n5 -c1 –gres=rgpu:5:3275M
MCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:163M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12

6.4. Resultados

Esta sección contiene una serie de gráficas que muestran el comportamiento de los resultados obtenidos en cuanto a tiempo de ejecución y productividad del sistema (trabajos por minuto).

Las Figuras 6.1 y 6.2 sirven para aclarar lo explicado anteriormente respecto a las configuraciones de lanzamiento de trabajos. La primera de ellas muestra que buscar el máximo rendimiento por aplicación (ver Tabla 6.3) genera tiempos similares entre los modos de GPUs y rGPUs. Aunque trabajando en el modo rCUDA se comparten GPUs, las restricciones de utilizar cierto número de nodos provoca una sobrecarga que retrasa el avance global. Por ejemplo, la ejecución en paralelo de 12 instancias (12 son los cores de cada nodo) en los mismos 4 nodos implica un tiempo de 1.884 segundos. Por el contrario, la ejecución en serie necesita 165 segundos $\times$ 12 instancias = 1.980, tiempo bastante similar al de la ejecución en paralelo. En la siguiente Figura se aprecia que evitando que cada proceso se ejecute en un nodo diferente (ver Tabla 6.4) se consiga reducir el tiempo de ejecución y aumentar la productividad. Comparándolas, se ve que no sólo se obtienen mejores resultados utilizando GPUs locales con CUDA, si no que la ganancia utilizando rCUDA es muy destacable.

A simple vista, en las Figuras 6.2, 6.3 y 6.4 podemos apreciar que rCUDA reduce el tiempo de ejecución, en varios casos en un factor mayor de 2$\times$, esto está bien reflejado en los resultados en el cluster de 4 GPUs. Aun más destacable es el hecho de que rCUDA mantenga el índice de productividad independientemente del número de GPUs, al contrario de lo que sucede al utilizar CUDA. Eliminar GPUs de varios nodos supone un efecto muy negativo para CUDA, tal y como se muestra en las gráficas.

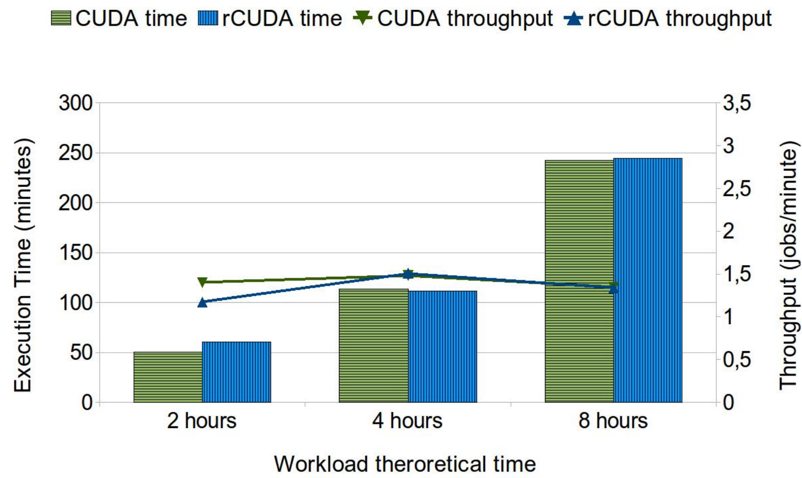

Figura 6.1: Ejecución en plataforma con 8 GPUs. Lanzamiento configurado para obtener el máximo rendimiento.

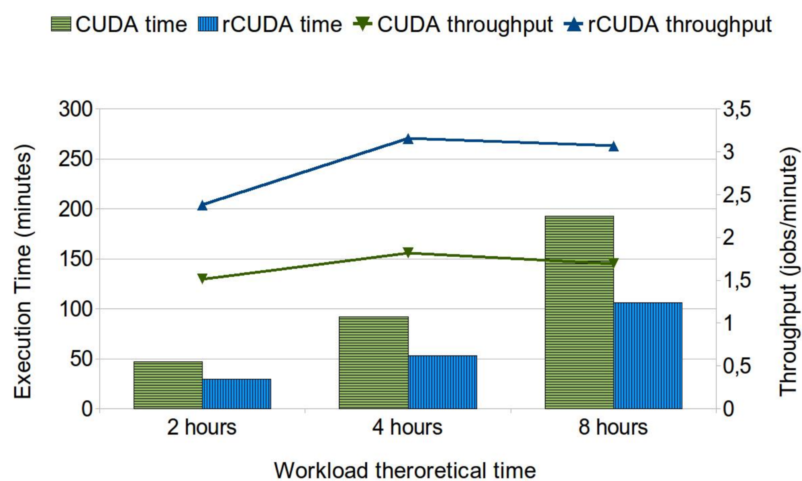

Figura 6.2: Ejecución en plataforma con 8 GPUs

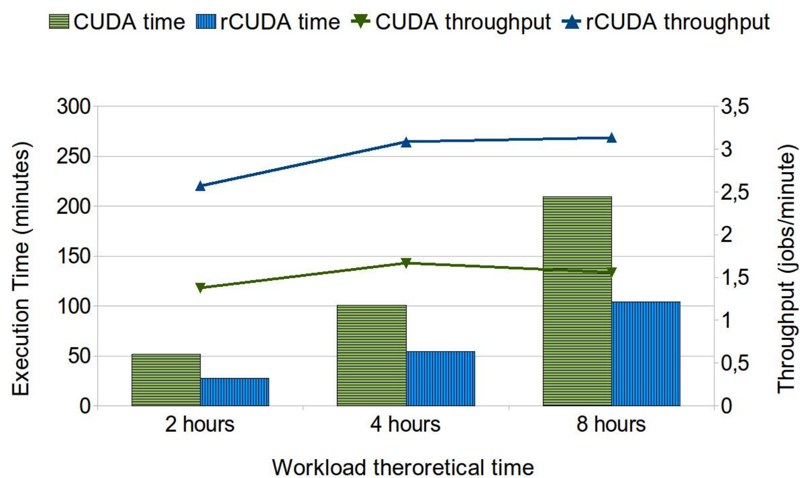

Figura 6.3: Ejecución en plataforma con 6 GPUs

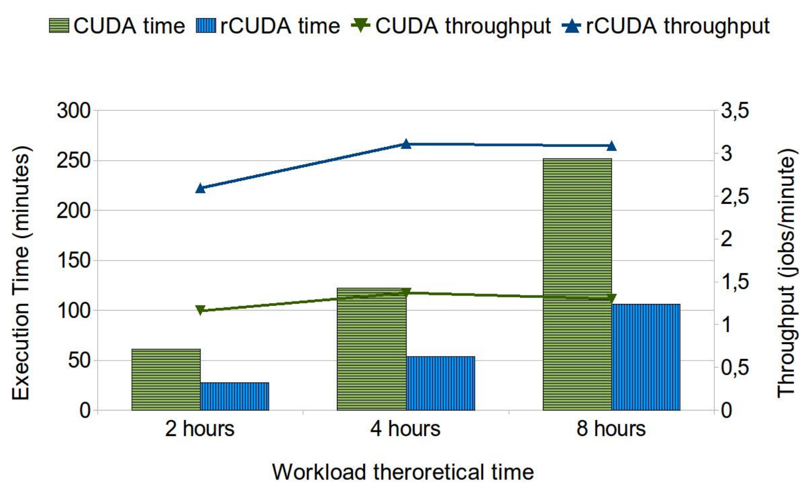

Figura 6.4: Ejecución en plataforma con 4 GPUs

Capítulo 7

Conclusiones

En este proyecto hemos integrado la gestión de GPUs remotas en el planificador de trabajos SLURM. Con estas modificaciones y utilizando un virtualizador de GPUs remotas como rCUDA, las GPUs en un *cluster* son virtualmente separadas del nodo donde están físicamente instaladas. Así pues, los trabajos ejecutándose en un nodo pueden utilizar GPUs de otros nodos. Hemos añadido a SLURM la definición de un nuevo recursos genérico SLURM y parámetros para que el planificador pueda administrar las RGPUs. Además, hemos realizado diversas modificaciones para aumentar la funcionalidad en varios comandos de SLURM.

También hemos llevado a cabo una extensa evaluación de prestaciones de la nueva funcionalidad en un cluster real. Con este propósito hemos definido una colección de cargas de trabajos sintéticas para demostrar tanto la funcionalidad como el aumento de rendimiento que se obtiene. Pudiendo así reducir el número de recursos físicos del *cluster* sin afectar al resultado.

Actualmente, la versión de SLURM-rCUDA adopta decisiones de planificación que involucran a las GPUs dependiendo de su memoria. En un futuro, tenemos planeado implementar otros algoritmos de planificación que aparte de tener en cuenta la memoria de la GPU, tenga también en cuenta: la carga computacional, la energía consumida o la distancia de red entre nodos.

Bibliografía

[1] SLURM website, http://www.schedmd.com

[2] Andy B. Yoo, Morris A. Jette, Mark Grondona, *"SLURM: Simple Linux Utility for Resource Management"*, in Job Scheduling Strategies for Parallel Processing, L. Rudolph and U. Schwiegelshohn, Editors. 2003, SpringerVerlag. p. 44-60.

[3] Quadrics Resource Management System http://www.quadrics.com/website/pdf/rms.pdf

[4] Distributed Production Control System http://www.llnl.gov/icc/lc/dpcs_overview.html

[5] Beowulf Distributed Process Space http://brpoc.sourceforge.net

[6] Y. Georgiou, "Resource and Job Management in High Performance Computing", PhD Thesis, Joseph Fourier University, France, 2010.

[7] Seren Soner, Can Özturan. Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager.

[8] Seren Soner, Can Ozturan, Itir Karaca. Extending SLURM with Support for GPU Ranges.

[9] Gerald Sabin P. Sadayappan. On Enhancing the Reliability of Job Schedulers.

[10] Susanne M. Balle and Dan Palermo. Enhancing an Open Source Resource Manager with Multi-Core/Multi-threaded Support.

[11] Jiadong Wu, Weiming Shi, and Bo Hong. Dynamic Kernel/Device Mapping Strategies for GPU-assisted HPC Systems.

[12] C. Reano, R. Mayo, E.S. Quintana-Orti F. Silla, J. Duato A.J. Pena. Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization, IEEE Cluster, 2013.

[13] Antonio J. Peña et al. An efficient implementation of GPU virtualization in high performance clusters, in Euro-Par Workshops, 2009.

[14] Antonio J. Peña et al. Performance of CUDA virtualized remote GPUs in high performance *clusters*, in ICPP, 2011.

[15] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Orti and G. Quintana-Orti. Exploiting the capabilities of modern GPUs for dense matrix computations, Concurr. Comput. : Pract. Exper., vol. 21, no. 18, pp. 24572477, 2009.

[16] D. P. Playne and K. A. Hawick. Data parallel three-dimensional CahnHilliard field equation simulation on GPUs with CUDA, in International Conference on Parallel and Distributed Processing Techniques and Applications, H. R. Arabnia, Ed., 2009, pp. 104-110.

[17] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Hwu, Z.P . Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs, in Proceedings of the 2008 conference on Computing Frontiers (CF'08), pp. 261-272. ACM, New York, 2008.

[18] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid Aerodynamic performance prediction on a *cluster* of graphics processing units, in Proceedings of the 47th AIAA Aerospace Sciences Meeting, no. AIAA 2009-565, Jan. 2009.

[19] Y. C. Luo and R. Duraiswami. Canny edge detection on NVIDIA CUDA, in Computer Vision on GPU, 2008.

[20] A. Gaikwad and I. M. Toke. GPU based sparse grid technique for solving multidimensional options pricing PDEs, Proceedings of the 2nd Workshop on High Performance Computational Finance 2009, pp. 6:1–6:9.

[21] NVIDIA. *The NVIDIA CUDA API Reference Manual Version 5*, NVIDIA 2012.

[22] Khronos OpenCL Working Group. OpenCL 1.2 Specification, 3 June 2013 http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[23] Giunta, Giulio et al. A GPGPU transparent virtualization component for high performance computing clouds, Euro-Par, 2010.

[24] Oikawa, Minoru et al. DS-CUDA: a middleware to use many GPUs in the cloud environment, in SC, 2012.

[25] Lin Shi et al. vCUDA: GPU accelerated high performance computing in virtual machines, IPDPS, 2009.

[26] Gupta, Vishakha et al. GViM: GPU-accelerated virtual machines, in HPCVirt, 2009.

[27] Tyng-Yeu Liang et al. GridCuda: a grid-enabled CUDA programming toolkit, WAINA, 2011.

[28] Zillians, Inc., V-GPU: GPU virtualization, 2013, http://www.zillians.com/vgpu

[29] Barak, A. et al. A package for OpenCL based heterogeneous computing on *clusters* with many GPU devices, *Cluster*, 2010.

[30] Kegel, P. et al. dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-many-core systems, 2012

[31] Kim, Jungwon et al. SnuCL: an OpenCL framework for heterogeneous CPU/GPU *clusters*, ICS, 2012.

[32] Shucai Xiao et al. VOCL: an optimized environment for transparent virtualization of graphics processing units. InPar, 2012.

[33] B. Nitzberg , J. M. Schopf , J. P. Jones. PBS Pro: Grid computing and scheduling attributes, Grid resource management: state of the art and future trends, Kluwer Academic Publishers, Norwell, MA, 2004.

[34] Moab Workload Manager Documentation,
http://docs.adaptivecomputing.com/mwm/7-2-8/help.htm

[35] Torque Resource Manager Documentation,
http://docs.adaptivecomputing.com/torque/4-2-8/help.htm

[36] LSF (Load Sharing Facility) Features and Documentation,
http://www.platform.com/workload-management/high-performance-computing

[37] N. Capit, G.D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components, in 5th Int. Symposium on *Cluster* Computing and the Grid, Cardiff, UK, 2005, pp. 776-783, IEEE.

[38] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux *clusters*, in Proceedings of the 4th Annual Showcase and Conference (LINUX-00), Berkeley, CA, 2000, pp. 217-224, The USENIX Association.

[39] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira. Workload Management with LoadLeveler. IBM, rst ed., Nov 2001. ibm.com/redbooks

[40] T. Tannenbaum, D. Wright , K. Miller , M. Livny. Condor: a distributed job scheduler, Beowulf *cluster* computing with Linux, MIT Press, Cambridge, MA, 2001.

[41] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid, in Proc. First IEEE International Symposium on *Cluster* Computing and the Grid (1st CC-GRID'01), Brisbane, Australia, May 2001, pp. 35-39, IEEE Computer Society (Los Alamitos, CA).

[42] F. Silla. rCUDA: towards energy-efficiency in GPU computing by leveraging low-power processors and InfiniBand interconnects. In HPC Advisory Council Spain Conference 2013, Barcelona, Spain, 2013.

[43] Lawrence Livermore National Laboratory. SLURM Generic Resource (GRES) Scheduling. https://computing.llnl.gov/linux/slurm/gres.html, 2012.

[44] Infiniband. http://es.wikipedia.org/wiki/InfiniBand

[45] MVAPICH2. http://mvapich.cse.ohio-state.edu

[46] GPU Applications. http://www.nvidia.com/object/gpu-applications.html

[47] The Top500 list. http://www.top500.org

[48] Jette, M., Auble, D. SLURM: Resource Management from the Simple to the Sophisticated. In Lawrence Livermore National Laboratory, SLURM User Group Meeting, October 2010.

Parte II

Documentos Anexos

Apéndice A

Ficheros de configuración de SLURM

En este documento se encuentra el contenido de los ficheros de configuración de SLURM para el cluster VirtualGap.

A.1. slurm.conf

```
ClusterName=VirtualGap
ControlMachine=virtualgap
SlurmUser=slurm
SlurmdUser=slurm
SlurmctldPort=6817
SlurmdPort=6818
AuthType=auth/munge
StateSaveLocation=/nfs/gap/slurm/var/slurm_state
SlurmdSpoolDir=/nfs/gap/slurm/var/slurmd.%n
SlurmctldPidFile=/nfs/gap/slurm/var/slurmctld.pid
SlurmdPidFile=/nfs/gap/slurm/var/slurmd.%n.pid
ProctrackType=proctrack/pgid
CacheGroups=0
ReturnToService=1
SlurmctldTimeout=300
SlurmdTimeout=300
InactiveLimit=0
MinJobAge=300
KillWait=30
Waittime=0

SchedulerType=sched/backfill
#SchedulerParameters=max_job_bf=15,interval=60

#MessageTimeout=300

SchedulerPort=7321

#MpiDefault=none
#MpiDefault=pmi2

RcudaModeDefault=shared
RcudaDistDefault=global
RgpuMinMemory=255

SelectType=select/cons_rgpu
SelectTypeParameters=CR_CORE
GresTypes=rgpu,gpu

AccountingStorageType=accounting_storage/slurmdbd
```

```
#AccountingStorageType=accounting_storage/none
AccountingStoreJobComment=YES
AccountingStorageHost=virtualgap
AccountingStoragePort=6819
AccountingStorageUser=slurm
JobCompHost=localhost
#JobCompPass=qwerty
JobCompUser=slurm
JobCompPort=3306
JobCompType=jobcomp/mysql
#JobCompType=jobcomp/none

FastSchedule=1
CryptoType=crypto/munge
SlurmctldDebug=1
SlurmctldLogFile=/nfs/gap/slurm/var/slurmctld.log
SlurmdDebug=1
SlurmdLogFile=/nfs/gap/slurm/var/slurmd.%n.log

NodeName=mlxc2i1 NodeHostname=mlxc2 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxf2i1 NodeHostname=mlxf2 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc3i1 NodeHostname=mlxc3 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc6i1 NodeHostname=mlxc6 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc7i1 NodeHostname=mlxc7 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc8i1 NodeHostname=mlxc8 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc10i1 NodeHostname=mlxc10 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1
NodeName=mlxc11i1 NodeHostname=mlxc11 CPUs=12 Sockets=2 CoresPerSocket=6
    ThreadsPerCore=1 RealMemory=32072 TmpDisk=29528 State=UNKNOWN Gres=gpu:1,rgpu:1

PartitionName=main Nodes=mlxf2i1,mlxc2i1,mlxc3i1,mlxc6i1,mlxc7i1,mlxc8i1,mlxc10i1,
    mlxc11i1 Default=YES MaxTime=INFINITE State=UP
```

A.2. gres.conf

```
Name=gpu File=/dev/nvidia0
Name=rgpu File=/dev/nvidia0 Cuda=3.5 Mem=4726M
```

A.3. slurmdbd.conf

```
ArchiveEvents=yes
ArchiveJobs=yes
ArchiveSteps=no
ArchiveSuspend=no

AuthType=auth/munge

SlurmUser=slurm
DbdHost=virtualgap
DbdPort=6819

DebugLevel=1
PurgeEventAfter=1months
```

```
PurgeJobAfter=1months
PurgeStepAfter=1months
PurgeSuspendAfter=1months

StorageHost=virtualgap
StoragePort=3306
StoragePass=qwerty
StorageType=accounting_storage/mysql
StorageUser=slurm
```

Apéndice B

Scripts auxiliares

En este documento se encuentran todos los *scripts* de apoyo que se han implementado durante el desarrollo del proyecto.

B.1. GPU-Blast

```
#!/bin/bash
set -e
if [ $# -ne 2 ]; then
        echo "error USAGE: ./script.sh 'job id' 'mem rgpu'"
        exit 1
fi

file="$1_$2.job"
touch $file
chmod 774 $file

if [ $2 -eq -1 ]; then
        printenv | egrep -i 'CUDA_VISIBLE_DEVICES|SLURM_NODELIST|JOB_ID' > $file
else
        printenv | egrep -i 'RCUDA_|JOB_ID' > $file
fi

path_home=/nfs/gap/siserte/gpu-blast_executions
path_app=/nfs/APPS/APPS/GPU-BLAST/1.1_ncbi-blast-2.2.26
$path_app/bin/blastp -db $path_home/sorted_env_nr -query $path_app/TESTS/queries/
    SequenceLength_00003000.txt -num_threads 6 -gpu t
```

B.2. LAMMPS

```
#!/bin/bash
set -e
if [ $# -ne 2 ]; then
        echo "error USAGE: ./script.sh 'job id' 'mem rgpu'"
        exit 1
fi
file="$1_$2.job"
touch $file
chmod 774 $file

if [ $2 -eq -1 ]; then
```

```
        printenv | egrep -i 'CUDA_VISIBLE_DEVICES|SLURM_NODELIST|JOB_ID' > $file
else
        printenv | egrep -i 'RCUDA_|JOB_ID' > $file
fi

path_home=/nfs/gap/siserte/lammps_executions
path_app=/nfs/APPS/APPS/LAMMPS/lammps-1Feb14/mvapich2+CUDA55/bin
path_mpirun=/nfs/LIBS/LIBS/MVAPICH2/2.0b/bin/mpirun
path_test=/nfs/gap/siserte/lammps55_executions/input_lammps
$path_mpirun -np $SLURM_NPROCS -hosts $SLURM_JOB_NODELIST $path_app/lmp_g++ -var x
    4 -var y 4 -var z 8 -sf cuda < $path_test/my_in.lj
```

B.3. MCUDA-MEME CUDA

```
#!/bin/bash
set -e
if [ $# -ne 2 ]; then
        echo "error USAGE: ./script.sh 'job id' 'mem rgpu'"
        exit 1
fi
file="$1_$2.job"
touch $file
chmod 774 $file
printenv | egrep -i 'CUDA_VISIBLE_DEVICES|SLURM_NODELIST|JOB_ID' > $file


./create_nodelist_file_no_repeating $SLURM_JOB_NODELIST $SLURM_JOBID
/nfs/LIBS/LIBS/MVAPICH2/2.0b/bin/mpirun_rsh -ssh -export -np $SLURM_NNODES -
    hostfile machines$SLURM_JOBID MV2_SMP_USE_LIMIC2=1 MV2_IBA_HCA=mlx4_0
    MV2_NUM_PORTS=1 MV2_SHOW_ENV_INFO=2 ./mcuda-meme nrsf_testcases/nrsf_500.fasta
    -dna -mod oops -maxsize 500000 -num_threads 1
```

B.4. MCUDA-MEME rCUDA

```
#!/bin/bash
set -e
if [ $# -ne 2 ]; then
        echo "error USAGE: ./script.sh 'job id' 'mem rgpu'"
        exit 1
fi
file="$1_$2.job"
touch $file
chmod 774 $file
printenv | egrep -i 'RCUDA_|JOB_ID' > $file

./create_nodelist_file $SLURM_JOB_NODELIST $SLURM_JOBID $SLURM_NPROCS
/nfs/LIBS/LIBS/MVAPICH2/2.0b/bin/mpirun_rsh -ssh -export -np $SLURM_NPROCS -
    hostfile machines$SLURM_JOBID MV2_SMP_USE_LIMIC2=1 MV2_IBA_HCA=mlx4_0
    MV2_NUM_PORTS=1 MV2_SHOW_ENV_INFO=2 ./mcuda-meme nrsf_testcases/nrsf_500.fasta
    -dna -mod oops -maxsize 500000 -num_threads 1
```

B.5. GROMACS

```
#!/bin/bash
set -e
if [ $# -ne 2 ]; then
        echo "error USAGE: ./script.sh 'job id' 'mem'"
        exit 1
fi
file="$1_$2.job"
touch $file
chmod 774 $file

APPS_PATH=/nfs/APPS/APPS
LIBS_PATH=/nfs/LIBS/LIBS
GROMACS_MPI_PATH=$APPS_PATH/GROMACS/4.6.5/MPI/mvapich2
GROMACS_MPI_BIN=$GROMACS_MPI_PATH/bin
GROMACS_MPI_LIB=$GROMACS_MPI_PATH/lib
GROMACS_MPI_INC=$GROMACS_MPI_PATH/include
MVAPICH2_PATH=$LIBS_PATH/MVAPICH2/2.0b
MVAPICH2_BIN=$MVAPICH2_PATH/bin
MVAPICH2_LIB=$MVAPICH2_PATH/lib
MVAPICH2_INC=$MVAPICH2_PATH/include
FFTW_SINGLE_PATH=$LIBS_PATH/FFTW/3.3.3/SINGLE
FFTW_SINGLE_LIB=$FFTW_SINGLE_PATH/lib
FFTW_DOUBLE_PATH=$LIBS_PATH/FFTW/3.3.3/DOUBLE
FFTW_DOUBLE_LIB=$FFTW_DOUBLE_PATH/lib

export LD_LIBRARY_PATH=$GROMACS_MPI_LIB:$MVAPICH2_LIB:$FFTW_SINGLE_LIB:
    $FFTW_DOUBLE_LIB:$LD_LIBRARY_PATH
export PATH=$GROMACS_MPI_BIN:$MVAPICH2_BIN:$PATH
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./create_nodelist_file $SLURM_JOB_NODELIST $SLURM_JOBID $SLURM_NPROCS
cat machines$SLURM_JOBID
mpirun_rsh -ssh -export -np $SLURM_NPROCS -hostfile machines$SLURM_JOBID
    MV2_SMP_USE_LIMIC2=1 MV2_IBA_HCA=mlx4_0 MV2_NUM_PORTS=1 MV2_SHOW_ENV_INFO=2
    $GROMACS_MPI_PATH/bin/mdrun_mpi -pin on -pinoffset 0 -nsteps 100000 -s /nfs/gap
    /siserte/gromacs_executions/bpti.tpr
```

B.6. genera_carga.py

```
#coding: latin1
import random, sys, os
random.seed(2)

class App:
        def __init__(self, id, orden, ruta, memoria, duracion):
                self.nameID = id                #nombre identificador del tipo de
                    app
                self.orden = orden              #comando slurm de envÃo de trabajo
                self.ruta = ruta                #ruta del script a ejecutar
                self.memoria = memoria          #memoria de GPU necesaria (en MB)
                self.duracion = duracion        #duracion de la ejecucion (en
                    segundos), calculada a priori
        def __str__(self):
                return "%s - %d" % (self.nameID, self.duracion)

def inicializa_apps_shared():
        apps = []
        cont_apps = dict()

        app = App("GPUBlast", "srun -n1 -c6 --rcuda-mode=shar --rcuda-distribution=
            global --gres=rgpu:1:1686M", "gpublast.sh", 1686, 21)
        apps.append(app)
        cont_apps[app.nameID] = 0
```

```
        app = App("mCUDA-MEME", "salloc -n4 -c1 --rcuda-mode=shar --rcuda-
            distribution=global --gres=rgpu:4:163M", "./mcudameme_rcuda.sh", 163,
            165)
        apps.append(app)
        cont_apps[app.nameID] = 0

        app = App("LAMMPS", "salloc -n4 -c1 --rcuda-mode=shar --rcuda-distribution=
            global --gres=rgpu:4:3275M", "./lammps.sh", 3275, 15)
        apps.append(app)
        cont_apps[app.nameID] = 0

        app = App("GROMACS", "salloc -N2 -n2 -c12", "./gromacs.sh", 0, 167)
        apps.append(app)
        cont_apps[app.nameID] = 0

        return (apps, cont_apps)

def inicializa_apps_original():
        apps = []
        cont_apps = dict()

        app = App("GPUBlast", "srun -N1 -n1 -c6 --gres=gpu:1", "gpublast.sh", -1,
            21)
        apps.append(app)
        cont_apps[app.nameID] = 0

        app = App("mCUDA-MEME", "salloc -N4 -n4 -c1 --gres=gpu:1", "./
            mcudameme_cuda.sh", -1, 165)
        apps.append(app)
        cont_apps[app.nameID] = 0

        app = App("LAMMPS", "salloc -N4 -n4 -c1 --gres=gpu:1", "./lammps.sh", -1,
            15)
        apps.append(app)
        cont_apps[app.nameID] = 0

        app = App("GROMACS", "salloc -N2 -n2 -c12", "./gromacs.sh", 0, 167)
        apps.append(app)
        cont_apps[app.nameID] = 0

        return (apps, cont_apps)

if __name__ == "__main__":
        if len(sys.argv) == 2:
                tiempo = int(sys.argv[1]) * 60
        else:
                print "Error - USAGE: python carga_heter.py 'minutes'"
                sys.exit(1)
        cadena = "_%i_rcuda_v2" % (tiempo /60)
        (apps, cont_apps) = inicializa_apps_shared()
        #(apps, cont_apps) = inicializa_apps_original()
        cargash1 = open("carga%s.sh" % cadena, "w")
        timesleep=100000
        cronometro=0
        njob=0
        jobList = []
        while cronometro < tiempo:
                idjob = random.randint(0,len(apps)-1)
                jobList.append(idjob)
                app = apps[idjob]
                idapp = app.nameID
                comando = "%s %s %i %i &\nusleep %i\n" % (app.orden, app.ruta, njob
                    , app.memoria, timesleep)
                cargash1.write(comando)
                cronometro += apps[idjob].duracion
                njob += 1
                cont_apps[idapp] += 1
                if (njob % 10) == 0:
                        timesleep += 250000
        cargash1.close()
```

Apéndice C

SLURM for rCUDA User's Guide

C.1. Introduction

On the one hand, SLURM is a resource manager tool that dispatches jobs to resources according to the specified policies and constrained by a specific criteria. It is an open-source resource manager designed for Linux clusters of all sizes. However, SLURM does not allow to share generic resources, such as GPUs, among nodes as it does with CPUs.

On the other hand, the rCUDA framework enables the concurrent usage of CUDA-compatible devices remotely. To enable a remote GPU-based acceleration, this framework creates virtual CUDA-compatible devices on those machines without a local GPU. These virtual devices represent physical GPUs located in remote hosts offering GPGPU services.

In order to use rCUDA framework on clusters where SLURM is installed, a code modification over the original SLURM sources has to be done. In this document, the steps to install and run the modified SLURM are explained.

C.2. Configuration

This software is a patch to modify automatically the original SLURM (2.6.2 version) code. Once modified, a new installation of SLURM is required. Now, SLURM is going to be able to share GPU (referred to as `rgpu`) among nodes.

C.2.1. Applying the patch and installation

The steps to change the original code are:

1. Download SLURM-2.6.2 from SLURM website
2. Move the patch into SLURM folder
3. Execute patch command: `patch -p1 -i rCUDA.patch`
4. Install SLURM

C.2.2. Controller node environment variables

Set the "RCUDAPROTO" environment variable to IB if an InfiniBand network connection is used (TCP by default).

Set the "RCUDAPATH" environment variable according to the rCUDA client side middleware location ("$HOME/rCUDA/framework/rCUDAl" by default).

Set the "CUDAPATH" environment variable according to the CUDA location ("/usr/local/cuda/lib64" by default).

C.2.3. Configuration files

C.2.3.1. slurm.conf

This file has to be the same (or with the same content) in all nodes where SLURM is running (controller node included).

There is a set of parameters that the user has to change manually. This parameters are used to notice the controller that a remote GPU (`rgpu`) can be shared by the nodes.

```
SelectType=select/cons_rgpu
SelectTypeParameters=(CR_CORE | CR_CPU | CR_SOCKET | ...)
GresTypes=rgpu,gpu
```

Moreover, the user is entitled to alter the default behaviour by adding in the configuration the next fields:

```
RcudaModeDefault=(exclusive | shared)
RcudaDistDefault=(global | node)
RgpuMinMemory=256
```

Being the mode by default `exclusive` (what means that only one job at once can be run in a RGPU), the user can change it, and set the mode `shared` (the RGPU can host jobs while it has enough memory).

The distribution by default is `global`, so the amount of requested RGPUs is the total amount. If the distribution used is `node`, the total amount of RGPUs will be given by the amount of requested RGPUs and the amount of requested nodes (both numbers are involved in a multiplication in order to obtain the total amount of RGPUs).

The field `RgpuMinMemory` indicates the minimun quantity of MB used by a job in the GPUs, if its quantity has not been declared in the submision (by default: 512). In this excerpt, the quantity is set as 256 MB. It is important to be aware that this field only allows integers and the units can not be included.

Finally, in the nodes description the user has to add this resource (`rgpu`) in each node where one or more GPUs are located.

```
NodeName=n15 NodeAddr=n15 CPUs=2 Sockets=1 CoresPerSocket=2
ThreadsPerCore=1 RealMemory=2005  Gres=rgpu:1,gpu:1
NodeName=n16 NodeAddr=n16 CPUs=8 Sockets=1 CoresPerSocket=4
ThreadsPerCore=2 RealMemory=7682 Gres=rgpu:4,gpu:4
NodeName=n17 NodeAddr=n17 CPUs=8 Sockets=1 CoresPerSocket=4
ThreadsPerCore=2 RealMemory=7682
```

In this sample nodes n15 and n16 have 1 and four GPUs and n17 does not have any GPU.

C.2.3.2. gres.conf

This file must exist in each node where a generic resource is installed. Using the previous sample, this is the gres.conf file content used in n16:

```
Name=rgpu File=/dev/nvidia0 Cuda=2.1 Mem=1535m
Name=rgpu File=/dev/nvidia1 Cuda=1.3 Mem=131072K
Name=rgpu File=/dev/nvidia2 Cuda=3.0 Mem=2048M
Name=rgpu File=/dev/nvidia3 Cuda=3.5 Mem=5g
Name=gpu File=/dev/nvidia0
Name=gpu File=/dev/nvidia1
Name=gpu File=/dev/nvidia2
Name=gpu File=/dev/nvidia3
```

As can be seen in this example, apart from the path of the device, for each `rgpu`, the user has to add the GPU Compute Capability (referred to as `Cuda`) and the Memory (referred to as `Mem`) parameters. Using one line per device.

The `Cuda` field expects two integers separated by a dot, which refer to the major and minor version. While the `Mem` field expects and integer and the units (see the example above).

C.3. How to use modified SLURM

Once the patch is applied and SLURM is installed and configured, the user is going to work in a standard SLURM version with new features.

C.3.1. Submision options

The request of rgpu resources will be done by the --gres option during the submision. Futhermore, the user can also change the mode and the distribution of RCUDA with the command line options.

- --rcuda-mode=(excl|shared): whether the job needs the whole rgpu or not.
- --rcuda-distribution=(global|node): if the number of requested rgpus is the total or is the number of RGPUs per node.
- --gres=rgpu:X:Y:Z: where
  - X = [1-9]+[0-9]* is the number of requested RGPUs.
  - Y = [1-9]+[0-9]*[kKmMgG] is the minimun memory required in each requested rgpu.
  - Z = [1-9]\.[0-9](cc|CC) is the minimun "compute capability" in each requested rgpu.

C.3.1.1. Using sbatch

The options in sbatch mode must be given in the script file with:

```
#SBATCH --gres=rgpu

#SBATCH --rcuda-distribution

#SBATCH --rcuda-mode
```

Moreover, we are supposed to define, manually in the script, the rCUDA libraries path in the environment variable $LD_LIBRARY_PATH.

If we have set the environment variable $RCUDAPATH before, we will include in the script:

```
export LD_LIBRARY_PATH=$RCUDAPATH:$LD_LIBRARY_PATH
```

Otherwise, the rCUDA path should be appended to $LD_LIBRARY_PATH.

C.3.2. Submision examples

So far, the submision options have been explained, they will be better understood by these examples, though.

- A job requiring 4 exclusive `rgpus` and 2 nodes:

```
$ srun -N2 --rcuda-mode=excl --rcuda-distribution=global
       --gres=rgpu:4 script.sh
```

- A job requiring 8 exclusive `rgpus` and 2 nodes (4 `rgpus` per node):

```
$ srun -N2 --rcuda-mode=excl --rcuda-distribution=node --gres=rgpu:4
       script.sh
```

- A job requiring 3 `rgpus` and 2 nodes, but the `rgpus` must have, at least, 1500 MB of free memory:

```
$ srun -N2 --rcuda-mode=shared --rcuda-distribution=global
       --gres=rgpu:3:1500M script.sh
```

- A job requiring 6 exclusive `rgpus` and 2 nodes (3 `rgpus` per node), but the `rgpus` must have both: at least 8GB of memory and a compute capability greater than or equal to 2.1:

```
$ srun -N2 --rcuda-mode=exlc --rcuda-distribution=node
       --gres=rgpu:3:8G:2.1cc script.sh
```

- Submitting a job with `salloc` which requires 4 `rgpus` and will create 4 processes:

```
$ salloc -n4 -c1 --rcuda-mode=shar --rcuda-distribution=global
         --gres=rgpu:4:3275M ./script.sh
```

- Submitting a job with `sbatch`:

```
$ sbatch ./script.sh
```

  The file `script.sh` will contain the `rgpu` requeriments:

```
#SBATCH --gres=rgpu:2:2048M
#SBATCH --rcuda-distribution=global
#SBATCH --rcuda-mode=excl
```

C.4. Further Information

For further information about SLURM for rCUDA, please refer to rCUDA Support Group.

For information related to rCUDA, please refer to www.rCUDA.net.

Apéndice D

Cambios realizados en los ficheros de SLURM

En este documento se encuentra la salida del comando `diff` aplicado entre los ficheros modificados respecto a la versión de SLURM 2.6.2.

```
--- slurm-2.6.2/configure.ac    2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/configure.ac    2014-06-25 09:59:56.029663932 +0200
@@ -606,6 +606,10 @@
                 testsuite/slurm_unit/api/Makefile
                 testsuite/slurm_unit/api/manual/Makefile
                 testsuite/slurm_unit/common/Makefile
+                src/plugins/select/cons_rgpu/Makefile
+                  src/plugins/select/cons_rgpu_2/Makefile
+                  src/plugins/select/cons_rgpu_3/Makefile
+                src/plugins/gres/rgpu/Makefile
                 ]
 )

--- slurm-2.6.2/slurm/slurm.h.in        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/slurm/slurm.h.in        2014-07-17 16:47:00.660862465 +0200
@@ -1112,6 +1112,10 @@
        char *std_out;          /* pathname of stdout */
        uint32_t wait4switch;   /* Maximum time to wait for minimum switches */
        char *wckey;            /* wckey for job */
+
+        /* RCUDA SPECIFIC */
+        uint16_t rcuda_mode;          /* 0 for default, 1 for "exclusive", 2 for "
    shared" mode */
+        uint16_t rcuda_dist;          /* 0 for default, 1 for "global", 2 for "node
    " distribution */
 } job_desc_msg_t;

 typedef struct job_info {
@@ -1204,6 +1208,8 @@
        uint32_t wait4switch;   /* Maximum time to wait for minimum switches */
        char *wckey;            /* wckey for job */
        char *work_dir;         /* pathname of working directory */
+
+        char *rgpu_list;        /* string with the assigned rgpus */
 } slurm_job_info_t;

 #ifndef __PERMAPI__H__
@@ -1252,6 +1258,8 @@
         * of each subarray is designated by the corresponding value in
         * the tasks array. */
        uint32_t **tids;        /* host id => task id mapping */
+       uint32_t nrgpu;
+       char *rgpu_list;        /* assigned list of rgpus */
 } slurm_step_layout_t;
```

```
 typedef struct slurm_step_io_fds {
@@ -1414,6 +1422,8 @@
        char **spank_job_env;    /* environment variables for job prolog/epilog
                                  * scripts as set by SPANK plugins */
        uint32_t spank_job_env_size;     /* element count in spank_env */
+    uint32_t nrgpu;
+    char *rgpulist;              /* assigned list of rgpus */
 } slurm_step_launch_params_t;

 typedef struct {
@@ -1664,6 +1674,8 @@
        uint32_t pn_min_memory;  /* minimum real memory per node OR
                                  * real memory per CPU | MEM_PER_CPU,
                                  * default=0 (no limit) */
+       uint32_t nrgpu;
+       char *rgpu_list;         /* assigned list of rgpus */
 } resource_allocation_response_msg_t;

 typedef struct job_alloc_info_response_msg {
@@ -2134,6 +2146,12 @@
        uint16_t z_16;           /* reserved for future use */
        uint32_t z_32;           /* reserved for future use */
        char *z_char;            /* reserved for future use */
+
+        /* RCUDA SPECIFIC */
+        uint16_t rcuda_mode;          /* 0 for default, 1 for "exclusive", 2 for "
    shared" mode */
+        uint16_t rcuda_dist;          /* 0 for default, 1 for "global", 2 for "node"
     distribution */
+        uint32_t rgpu_min_mem;        /* MB */
+
 } slurm_ctl_conf_t;

 typedef struct slurmd_status_msg {
--- slurm-2.6.2/src/common/env.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/env.c        2014-07-17 17:13:53.764858443 +0200
@@ -929,9 +929,57 @@
        int rc = SLURM_SUCCESS;
        uint32_t node_cnt = alloc->node_cnt;
        uint32_t cluster_flags = slurmdb_setup_cluster_flags();
-
+
+       char *envvar_library, *envvar_home;
+       char *str = xstrdup("");
+       envvar_home = getenv("HOME");
+       envvar_library = getenv("LD_LIBRARY_PATH");
+
        _setup_particulars(cluster_flags, dest, alloc->select_jobinfo);
-
+
+       if(alloc->rgpu_list != NULL){
+               /* WRITING RCUDA ENVIRONMENT VARIABLES */
+               int rgpuCount=0;
+               char *tok, *envvar_proto, *tmp_str, *envvar_rcuda;
+               char aux[2048];
+
+               envvar_proto = getenv("RCUDAPROTO");
+               envvar_rcuda = getenv("RCUDAPATH");
+
+               if(envvar_rcuda == NULL)
+                       xstrcat(envvar_rcuda , "rCUDA/framework/rCUDAl");
+               if(envvar_library == NULL)
+                       xstrfmtcat(str, "%s/%s:", envvar_home, envvar_rcuda);
+               else
+                       xstrfmtcat(str, "%s/%s:%s:", envvar_home, envvar_rcuda,
    envvar_library);
+               if(envvar_proto == NULL)
+                               env_array_overwrite_fmt(dest, "RCUDAPROTO", "%s", "
    TCP");
```

```
+
+                tmp_str = xstrdup(alloc->rgpu_list);
+                tok = strtok(tmp_str,",");
+                while (tok != NULL) {
+                        sprintf(aux, "RCUDA_DEVICE_%i", rgpuCount);
+                        env_array_overwrite_fmt(dest, aux, "%s", tok);
+                        rgpuCount++;
+                        tok = strtok(NULL,",");
+                }
+                env_array_overwrite_fmt(dest, "RCUDA_DEVICE_COUNT", "%i", rgpuCount
    );
+        } else {
+                /* WRITING CUDA PATH IN LD_LIBRARY_PATH */
+                char *envvar_cuda;
+                envvar_cuda = getenv("CUDAPATH");
+
+                if(envvar_cuda == NULL)
+                        xstrcat(envvar_cuda , "/usr/local/cuda/lib64:/usr/local/
    cuda/lib");
+
+                if(envvar_library == NULL)
+                        xstrfmtcat(str, "%s", envvar_cuda);
+                else
+                        xstrfmtcat(str, "%s:%s", envvar_cuda, envvar_library);
+        }
+        env_array_overwrite_fmt(dest, "LD_LIBRARY_PATH", "%s", str);
+        xfree(str);
+
         if (cluster_flags & CLUSTER_FLAG_BG) {
                 select_g_select_jobinfo_get(alloc->select_jobinfo,
                                             SELECT_JOBDATA_NODE_CNT,
@@ -1038,7 +1086,7 @@
                                                     desc->task_dist,
                                                     desc->plane_size)))
                 return SLURM_ERROR;
-
+
        tmp = _uint16_array_to_str(step_layout->node_cnt,
                                    step_layout->tasks);
        slurm_step_layout_destroy(step_layout);
@@ -1084,9 +1132,65 @@
        uint16_t cpus_per_task;
        uint16_t task_dist;
        uint32_t cluster_flags = slurmdb_setup_cluster_flags();
-
+
+        char *envvar_library, *envvar_home;
+        char *str = xstrdup("");
+        envvar_home = getenv("HOME");
+        envvar_library = getenv("LD_LIBRARY_PATH");
+
        _setup_particulars(cluster_flags, dest, batch->select_jobinfo);

+        /* FIX IT
+         * rCUDA sbatch command. Due to in shared directories sbatch must be run
    with slurm user
+         * the HOME does not contain the rCUDA directory.
+         * Moreover, from this function it is not possible to get environment
    variables,
+         * as has been done in other similar functions.
+         * The env vars LD_LIBRARY_PATH and RCUDAPROTO cannot be set, hence the env
    vars
+         * are not available from here.
+         */
+        if(batch->rgpu_list != NULL){
+                /* WRITING RCUDA ENVIRONMENT VARIABLES */
+                int rgpuCount=0;
+                char *tok, *envvar_proto, *tmp_str, *envvar_rcuda;
+                char aux[2048];
+
```

```
+                envvar_proto = getenv("RCUDAPROTO");
+                envvar_rcuda = getenv("RCUDAPATH");
+
+                if(envvar_rcuda == NULL)
+                        xstrcat(envvar_rcuda , "rCUDA/framework/rCUDAl");
+                if(envvar_library == NULL)
+                        xstrfmtcat(str, "%s/%s:", envvar_home, envvar_rcuda);
+                else
+                        xstrfmtcat(str, "%s/%s:%s:", envvar_home, envvar_rcuda,
    envvar_library);
+                //if(envvar_proto == NULL)
+                //              env_array_overwrite_fmt(dest, "RCUDAPROTO", "%s", "
    TCP");
+
+                tmp_str = xstrdup(batch->rgpu_list);
+                tok = strtok(tmp_str,",");
+                while (tok != NULL) {
+                        sprintf(aux, "RCUDA_DEVICE_%i", rgpuCount);
+                        env_array_overwrite_fmt(dest, aux, "%s", tok);
+                        rgpuCount++;
+                        tok = strtok(NULL,",");
+                }
+                env_array_overwrite_fmt(dest, "RCUDA_DEVICE_COUNT", "%i", rgpuCount
    );
+        } else {
+                /* WRITING CUDA PATH IN LD_LIBRARY_PATH */
+                char *envvar_cuda;
+                envvar_cuda = getenv("CUDAPATH");
+
+                if(envvar_cuda == NULL)
+                        xstrcat(envvar_cuda , "/usr/local/cuda/lib64:/usr/local/
    cuda/lib");
+
+                if(envvar_library == NULL)
+                        xstrfmtcat(str, "%s", envvar_cuda);
+                else
+                        xstrfmtcat(str, "%s:%s", envvar_cuda, envvar_library);
+        }
+        env_array_overwrite_fmt(dest, "LD_LIBRARY_PATH", "%s", str);
+        xfree(str);
+
         /* There is no explicit node count in the batch structure,
          * so we need to calculate the node count. */
         for (i = 0; i < batch->num_cpu_groups; i++) {
@@ -1235,11 +1339,61 @@
         uint32_t node_cnt = step->step_layout->node_cnt;
         uint32_t cluster_flags = slurmdb_setup_cluster_flags();

+        char *envvar_library, *envvar_home;
+        char *str = xstrdup("");
+        envvar_home = getenv("HOME");
+
         tpn = _uint16_array_to_str(step->step_layout->node_cnt,
                                    step->step_layout->tasks);
         env_array_overwrite_fmt(dest, "SLURM_STEP_ID", "%u", step->job_step_id);
         env_array_overwrite_fmt(dest, "SLURM_STEP_NODELIST",
                                 "%s", step->step_layout->node_list);
+
+        envvar_library = getenv("LD_LIBRARY_PATH");
+
+        if(step->step_layout->rgpu_list != NULL){
+                /* WRITING RCUDA ENVIRONMENT VARIABLES */
+                int rgpuCount=0;
+                char *tok, *envvar_proto, *tmp_str, *envvar_rcuda;
+                char aux[2048];
+
+                envvar_proto = getenv("RCUDAPROTO");
+                envvar_rcuda = getenv("RCUDAPATH");
+
+                if(envvar_rcuda == NULL)
```

```
+                       xstrcat(envvar_rcuda , "rCUDA/framework/rCUDAl");
+               if(envvar_library == NULL)
+                       xstrfmtcat(str, "%s/%s:", envvar_home, envvar_rcuda);
+               else
+                       xstrfmtcat(str, "%s/%s:%s:", envvar_home, envvar_rcuda,
    envvar_library);
+               if(envvar_proto == NULL)
+                               env_array_overwrite_fmt(dest, "RCUDAPROTO", "%s", "
    TCP");
+
+               tmp_str = xstrdup(step->step_layout->rgpu_list);
+               tok = strtok(tmp_str,",");
+               while (tok != NULL) {
+                       sprintf(aux, "RCUDA_DEVICE_%i", rgpuCount);
+                       env_array_overwrite_fmt(dest, aux, "%s", tok);
+                       rgpuCount++;
+                       tok = strtok(NULL,",");
+               }
+               env_array_overwrite_fmt(dest, "RCUDA_DEVICE_COUNT", "%i", rgpuCount
    );
+       } else {
+               /* WRITING CUDA PATH IN LD_LIBRARY_PATH */
+               char *envvar_cuda;
+               envvar_cuda = getenv("CUDAPATH");
+
+               if(envvar_cuda == NULL)
+                       xstrcat(envvar_cuda , "/usr/local/cuda/lib64:/usr/local/
    cuda/lib");
+
+               if(envvar_library == NULL)
+                       xstrfmtcat(str, "%s", envvar_cuda);
+               else
+                       xstrfmtcat(str, "%s:%s", envvar_cuda, envvar_library);
+       }
+       env_array_overwrite_fmt(dest, "LD_LIBRARY_PATH", "%s", str);
+       xfree(str);
+
        if (cluster_flags & CLUSTER_FLAG_BG) {
                char geo_char[HIGHEST_DIMENSIONS+1];

--- slurm-2.6.2/src/common/gres.c       2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/gres.c       2014-07-17 17:12:40.908858623 +0200
@@ -83,6 +83,7 @@
 #include "src/common/xmalloc.h"
 #include "src/common/xstring.h"
 #include "src/common/read_config.h"
+#include "job_resources.h"

 #define GRES_MAGIC 0x438a34d4

@@ -635,6 +636,8 @@
                {"Count", S_P_STRING},  /* Number of Gres available */
                {"CPUs" , S_P_STRING},  /* CPUs to bind to Gres resource */
                {"File",  S_P_STRING},  /* Path to Gres device */
+               {"Cuda", S_P_STRING},
+               {"Mem", S_P_STRING},
                {NULL}
        };
        int i;
@@ -695,8 +698,6 @@
        } else if (p->count == 0)
                p->count = 1;

-       s_p_hashtbl_destroy(tbl);
-
        for (i=0; i<gres_context_cnt; i++) {
                if (strcasecmp(value, gres_context[i].gres_name) == 0)
                        break;
@@ -706,6 +707,49 @@
                _destroy_gres_slurmd_conf(p);
```

```
                return 0;
        }
+
+       if (gres_context[i].plugin_id==1970300786) {
+               /* Parsering Cuda Capability Version and GPU Memory */
+               if (s_p_get_string(&tmp_str, "Cuda", tbl)) {
+                       char *tok = NULL;
+                       int ver1, ver2;
+                       tok = strtok(tmp_str, ".");
+                       ver1 = atoi(tok);
+                       if (ver1<1)
+                               fatal("Invalid gres data for %s, Cuda Major not
    allowed, Cuda Capabily=%s", p->name, tmp_str);
+                       tok = strtok(NULL, "");
+                       ver2 = atoi(tok);
+                       if (ver2<0)
+                               fatal("Invalid gres data for %s, Cuda Minor not
    allowed, Cuda Capabily=%s", p->name, tmp_str);
+                       xfree(tmp_str);
+                       p->cc_version = ver1 * 1000 + ver2 * 10;
+               } else
+                       fatal("Invalid gres data for %s, Cuda Capability does not
    specified in gres.conf", p->name);
+
+               if (s_p_get_string(&tmp_str, "Mem", tbl)) {
+                       tmp_long = strtol(tmp_str, &last, 10);
+                       if ((tmp_long==LONG_MIN)||(tmp_long==LONG_MAX)) {
+                               fatal("Invalid gres data for %s, Count=%s", p->name
    ,
+                                       tmp_str);
+                       }
+                       if ((last[0]=='k')||(last[0]=='K'))
+                               tmp_long *= 1024;
+                       else if ((last[0]=='m')||(last[0]=='M'))
+                               tmp_long *= (1024*1024);
+                       else if ((last[0]=='g')||(last[0]=='G'))
+                               tmp_long *= (1024*1024*1024);
+                       else {//if (last[0] != '\0') {
+                               fatal("Invalid gres data for %s, Mem=%s", p->name,
+                                       tmp_str);
+                       }
+                       p->mem = tmp_long;
+                       xfree(tmp_str);
+               } else
+                       fatal("Invalid gres data for %s, RGPU Memory does not
    specified in gres.conf", p->name);
+       }
+
+       s_p_hashtbl_destroy(tbl);
+
        p->plugin_id = gres_context[i].plugin_id;
        *dest = (void *)p;
        return 1;
@@ -903,6 +947,10 @@
                        pack32(gres_slurmd_conf->cpu_cnt, buffer);
                        pack8(gres_slurmd_conf->has_file, buffer);
                        pack32(gres_slurmd_conf->plugin_id, buffer);
+                       if (gres_slurmd_conf->plugin_id==1970300786) {
+                               pack32(gres_slurmd_conf->cc_version, buffer);
+                               pack64(gres_slurmd_conf->mem, buffer);
+                       }
                        packstr(gres_slurmd_conf->cpus, buffer);
                        packstr(gres_slurmd_conf->name, buffer);
                }
@@ -921,7 +969,8 @@
 extern int gres_plugin_node_config_unpack(Buf buffer, char* node_name)
 {
        int i, j, rc;
-       uint32_t count, cpu_cnt, magic, plugin_id, utmp32;
+       uint64_t mem;
```

```
+	uint32_t count, cpu_cnt, magic, plugin_id, utmp32, cc_ver;
	uint16_t rec_cnt, version;
	uint8_t has_file;
	char *tmp_cpus, *tmp_name;
@@ -948,6 +997,10 @@
		safe_unpack32(&cpu_cnt, buffer);
		safe_unpack8(&has_file, buffer);
		safe_unpack32(&plugin_id, buffer);
+		if (plugin_id==1970300786) {
+			safe_unpack32(&cc_ver, buffer);
+			safe_unpack64(&mem, buffer);
+		}
		safe_unpackstr_xmalloc(&tmp_cpus, &utmp32, buffer);
		safe_unpackstr_xmalloc(&tmp_name, &utmp32, buffer);

@@ -998,6 +1051,10 @@
		tmp_cpus = NULL;	/* Nothing left to xfree */
		p->name = tmp_name;	/* We need to preserve for accounting! */
		p->plugin_id = plugin_id;
+		if (plugin_id==1970300786) {
+			p->cc_version = cc_ver;
+			p->mem = mem;
+		}
		list_append(gres_conf_list, p);
	}
	slurm_mutex_unlock(&gres_context_lock);
@@ -1033,6 +1090,13 @@
	xfree(gres_node_ptr->topo_gres_bitmap);
	xfree(gres_node_ptr->topo_gres_cnt_alloc);
	xfree(gres_node_ptr->topo_gres_cnt_avail);
+	//debug("RCUDA freeing cc_version, mem_rgpu_avail, mem_rgpu_alloc...");
+	if (gres_node_ptr->cc_version)
+		xfree(gres_node_ptr->cc_version);
+	if (gres_node_ptr->mem_rgpu_avail)
+		xfree(gres_node_ptr->mem_rgpu_avail);
+	if (gres_node_ptr->mem_rgpu_alloc)
+		xfree(gres_node_ptr->mem_rgpu_alloc);
	xfree(gres_node_ptr);
	xfree(gres_ptr);
 }
@@ -1448,6 +1512,58 @@
	return rc;
 }

+/* Built the rgpu list of the node */
+extern int _rgpu_node_config_validate(gres_state_t *gres_ptr)
+{
+	int i, rc = SLURM_SUCCESS;
+	bool cc=true, alloc=true, avail=true;
+	gres_node_state_t *gres_data;
+	ListIterator iter;
+	gres_slurmd_conf_t *gres_slurmd_conf;
+
+	gres_data = (gres_node_state_t *) gres_ptr->gres_data;
+
+	//debug("\n");
+	//debug2("RCUDA %s(%s,%d) - - gres_data->gres_cnt_found: %i", __FILE__,
    __func__, __LINE__, gres_data->gres_cnt_found);
+	if (!gres_data->cc_version) {
+		cc = false;
+		gres_data->cc_version = xrealloc(gres_data->cc_version, gres_data->
    gres_cnt_found * sizeof(uint32_t));
+		if (gres_data->cc_version == NULL)
+			fatal("xrealloc: malloc failure - gres_data->cc_version");
+	}
+	if (!gres_data->mem_rgpu_alloc) {
+		alloc = false;
+		gres_data->mem_rgpu_alloc = xrealloc(gres_data->mem_rgpu_alloc,
    gres_data->gres_cnt_found * sizeof(uint64_t));
+		if (gres_data->mem_rgpu_alloc == NULL)
```

```
+                       fatal("xrealloc: malloc failure - gres_data->mem_rgpu_alloc
    ");
+       }
+       if (!gres_data->mem_rgpu_avail) {
+               avail = false;
+               gres_data->mem_rgpu_avail = xrealloc(gres_data->mem_rgpu_avail,
    gres_data->gres_cnt_found * sizeof(uint64_t));
+               if (gres_data->mem_rgpu_avail == NULL)
+                       fatal("xrealloc: malloc failure - gres_data->mem_rgpu_avail
    ");
+       }
+       iter = list_iterator_create(gres_conf_list);
+       if (iter==NULL)
+               fatal("list_iterator_create: malloc failure");
+       i = 0;
+       while ((gres_slurmd_conf = (gres_slurmd_conf_t *) list_next(iter))) {
+               if (gres_slurmd_conf->plugin_id!=1970300786)
+                       continue;
+               if (! cc)
+                       gres_data->cc_version[i] = gres_slurmd_conf->cc_version;
+               if (! alloc)
+                       gres_data->mem_rgpu_alloc[i] = 0LLU;
+               if (! avail)
+                       gres_data->mem_rgpu_avail[i] = gres_slurmd_conf->mem;
+               //debug("\tRCUDA %s(%s,%d) - - gres_data->mem_rgpu_alloc[%i]: %lu",
    __FILE__, __func__, __LINE__, i, gres_data->mem_rgpu_alloc[i]);
+               i++;
+       }
+       //debug("\n");
+       list_iterator_destroy(iter);
+       return rc;
+}
+
 /*
  * Validate a node's configuration and put a gres record onto a list
  * Called immediately after gres_plugin_node_config_unpack().
@@ -1467,7 +1583,7 @@
                                         uint16_t fast_schedule,
                                         char **reason_down)
 {
-       int i, rc, rc2;
+       int i, rc, rc2, rc3;
        ListIterator gres_iter;
        gres_state_t *gres_ptr;

@@ -1493,6 +1609,10 @@
                rc2 = _node_config_validate(node_name, orig_config, new_config,
                                         gres_ptr, fast_schedule,
                                         reason_down, &gres_context[i]);
+               if (gres_context[i].plugin_id==1970300786) {
+                       rc3 = _rgpu_node_config_validate(gres_ptr);
+                       rc2 = MAX(rc2, rc3);
+               }
                rc = MAX(rc, rc2);
        }
        slurm_mutex_unlock(&gres_context_lock);
@@ -1751,6 +1871,72 @@
        new_gres->gres_cnt_alloc  = gres_ptr->gres_cnt_alloc;
        if (gres_ptr->gres_bit_alloc)
                new_gres->gres_bit_alloc = bit_copy(gres_ptr->gres_bit_alloc);
+
+       if (gres_ptr->topo_cnt == 0)
+               return new_gres;
+
+       new_gres->topo_cnt          = gres_ptr->topo_cnt;
+       new_gres->topo_cpus_bitmap = xmalloc(gres_ptr->topo_cnt *
+                                        sizeof(bitstr_t *));
+       new_gres->topo_gres_bitmap = xmalloc(gres_ptr->topo_cnt *
+                                        sizeof(bitstr_t *));
+       new_gres->topo_gres_cnt_alloc = xmalloc(gres_ptr->topo_cnt *
```

```
+                                                   sizeof(uint32_t));
+       new_gres->topo_gres_cnt_avail = xmalloc(gres_ptr->topo_cnt *
+                                                   sizeof(uint32_t));
+       for (i=0; i<gres_ptr->topo_cnt; i++) {
+               new_gres->topo_cpus_bitmap[i] =
+                       bit_copy(gres_ptr->topo_cpus_bitmap[i]);
+               new_gres->topo_gres_bitmap[i] =
+                       bit_copy(gres_ptr->topo_gres_bitmap[i]);
+               new_gres->topo_gres_cnt_alloc[i] =
+                       gres_ptr->topo_gres_cnt_alloc[i];
+               new_gres->topo_gres_cnt_avail[i] =
+                       gres_ptr->topo_gres_cnt_avail[i];
+       }
+       return new_gres;
+}
+
+static void *_rgpu_node_state_dup(void *gres_data)
+{
+       int i;
+       gres_node_state_t *gres_ptr = (gres_node_state_t *) gres_data;
+       gres_node_state_t *new_gres;
+
+       if (gres_ptr == NULL)
+               return NULL;
+
+       if (gres_ptr->mem_rgpu_avail == NULL)
+               return NULL;
+
+       new_gres = xmalloc(sizeof(gres_node_state_t));
+       new_gres->gres_cnt_found  = gres_ptr->gres_cnt_found;
+       new_gres->gres_cnt_config = gres_ptr->gres_cnt_config;
+       new_gres->gres_cnt_avail  = gres_ptr->gres_cnt_avail;
+       new_gres->gres_cnt_alloc  = gres_ptr->gres_cnt_alloc;
+       if (gres_ptr->gres_bit_alloc)
+               new_gres->gres_bit_alloc = bit_copy(gres_ptr->gres_bit_alloc);
+       //debug("\n");
+       //debug2("RCUDA %s(%s,%d) - - gres_ptr->gres_cnt_avail:%i", __FILE__,
    __func__, __LINE__, gres_ptr->gres_cnt_avail);
+       if (gres_ptr->gres_cnt_avail > 0) {
+               xassert(gres_ptr->mem_rgpu_avail);
+               xassert(gres_ptr->mem_rgpu_alloc);
+               xassert(gres_ptr->cc_version);
+
+               new_gres->cc_version = xmalloc(gres_ptr->gres_cnt_avail*sizeof(
    uint32_t));
+               new_gres->mem_rgpu_alloc = xmalloc(gres_ptr->gres_cnt_avail*sizeof(
    uint64_t));
+               new_gres->mem_rgpu_avail = xmalloc(gres_ptr->gres_cnt_avail*sizeof(
    uint64_t));
+               for (i = 0; i<gres_ptr->gres_cnt_avail; i++) {
+                       //debug("\tRCUDA %s(%s,%d) - - dup before: i=%i gres_ptr->
    mem_rgpu_alloc[i]=%lu new_gres->mem_rgpu_alloc[i]=%lu ", __FILE__, __func__,
    __LINE__, i, gres_ptr->mem_rgpu_alloc[i], new_gres->mem_rgpu_alloc[i]);
+                       //debug("\tRCUDA %s(%s,%d) - - dup before: i=%i gres_ptr->
    mem_rgpu_alloc[i]=%lu gres_ptr->mem_rgpu_avail[i]=%lu gres_ptr->cc_version[i]=%
    lu \n", __FILE__, __func__, __LINE__, i, gres_ptr->mem_rgpu_alloc[i], gres_ptr
    ->mem_rgpu_avail[i], gres_ptr->cc_version[i]);
+                       new_gres->cc_version[i] = gres_ptr->cc_version[i];
+                       new_gres->mem_rgpu_alloc[i] = gres_ptr->mem_rgpu_alloc[i];
+                       new_gres->mem_rgpu_avail[i] = gres_ptr->mem_rgpu_avail[i];
+                       //debug("\tRCUDA %s(%s,%d) - - dub after: i=%i gres_ptr->
    mem_rgpu_alloc[i]=%lu new_gres->mem_rgpu_alloc[i]=%lu ", __FILE__, __func__,
    __LINE__, i, gres_ptr->mem_rgpu_alloc[i], new_gres->mem_rgpu_alloc[i]);
+
+               }
+       }
+       //debug("\n");
        if (gres_ptr->topo_cnt == 0)
                return new_gres;
```

```
@@ -1803,7 +1989,10 @@
                for (i=0; i<gres_context_cnt; i++) {
                        if (gres_ptr->plugin_id != gres_context[i].plugin_id)
                                continue;
-                       gres_data = _node_state_dup(gres_ptr->gres_data);
+                       if (gres_ptr->plugin_id==1970300786)
+                               gres_data = _rgpu_node_state_dup(gres_ptr->
    gres_data);
+                       else
+                               gres_data = _node_state_dup(gres_ptr->gres_data);
                        if (gres_data) {
                                new_gres = xmalloc(sizeof(gres_state_t));
                                new_gres->plugin_id = gres_ptr->plugin_id;
@@ -1819,7 +2008,7 @@
        }
        list_iterator_destroy(gres_iter);
        slurm_mutex_unlock(&gres_context_lock);
-
+
        return new_list;
 }

@@ -3755,6 +3944,11 @@
                                rc2 = ESLURM_INVALID_GRES;
                                continue;
                        }
+                       if (list_count(job_gres_list)==0) {
+                               info("step %u.%u has gres spec, job has not null
    but none",
+                                       job_id, step_id);
+                               continue;
+                       }
                        /* Now make sure the step's request isn't too big for
                         * the job's gres allocation */
                        job_gres_iter = list_iterator_create(job_gres_list);
@@ -4820,3 +5014,1196 @@
        }
        slurm_mutex_unlock(&gres_context_lock);
 }
+
+/**************************** RCUDA *****************************************/
+
+static int _validate_rgpu_data(char *tok, uint64_t *res){
+       char *last_num = NULL;
+       uint64_t cnt = strtol(tok, &last_num, 10);
+       if (strlen(last_num)>1)
+               return SLURM_ERROR;
+       else if (last_num[0]=='\0')
+               ;
+       else if ((last_num[0]=='k')||(last_num[0]=='K'))
+               cnt *= 1024;
+       else if ((last_num[0]=='m')||(last_num[0]=='M'))
+               cnt *= (1024*1024);
+       else if ((last_num[0]=='g')||(last_num[0]=='G'))
+               cnt *= (1024*1024*1024);
+       else
+               return SLURM_ERROR;
+       if (cnt<0)
+               return SLURM_ERROR;
+       *res = (uint64_t) cnt;
+       return SLURM_SUCCESS;
+}
+
+static int _validate_compute_capability(char *tok, int tok_size, uint32_t *res){
+
+       char last, next_last;
+       char *minor_v, *aux;
+       int major, minor;
+       uint32_t version = 0;
+
```

```
+        if (tok_size < 5){
+                error("_validate_compute_capability: some character missing (
    detected only %i in %s)", tok_size, tok);
+                return SLURM_ERROR;
+        }
+        last = tok[tok_size-1];
+        next_last = tok[tok_size-2];
+        if ((last == 'c' || last == 'C') && (next_last == 'c' || next_last == 'C'))
    {
+                //cutting the string after testing that it is a correct request
+                tok[tok_size-2] = '\0';
+                //major = strtok_r(tok, ".", &minor);
+                major = strtol(tok, &minor_v, 10);
+                minor = strtol(minor_v+sizeof(char), &aux, 10);
+                version = major*1000 + minor*10;
+        } else
+                return SLURM_ERROR;
+
+        *res = version;
+        return SLURM_SUCCESS;
+}
+
+/*
+ * The same function as _job_config_validate(), but with the capability
+ * of processing rgpu jobs. This function is supossed to parser the
+ * quantity of rgpus and the memory requested, besides the usual
+ * gres parsering carried out by _job_config_validate()
+ */
+static int _job_config_validate_2(char *config, uint32_t *gres_cnt,
+                         slurm_gres_context_t *context_ptr,
+                         uint64_t *mem_cnt, uint32_t *com_cap)
+{
+        char *last_num = NULL, *last = NULL, *tmp_str, *tok;
+        int n_toks, rc, size;
+        uint32_t cc = 0;
+        uint64_t cnt, cnt_mem = 0;
+        char *toks[3];
+        char end;
+
+        if (!strcmp(config, context_ptr->gres_name)) {
+                cnt = 1;
+        } else if (!strncmp(config, context_ptr->gres_name_colon,
+                context_ptr->gres_name_colon_len)) {
+                if (context_ptr->plugin_id==1970300786) {
+                        tmp_str = xstrdup(config);
+                        tok = strtok_r(tmp_str, ":", &last);
+                        //tok == rgpu
+                        tok = strtok_r(NULL, ":", &last);
+                        n_toks=0;
+                        while(tok) {
+                                toks[n_toks++] = xstrdup(tok);
+                                tok = strtok_r(NULL, ":", &last);
+                        }
+
+                        if (n_toks > 3)
+                                return SLURM_ERROR;
+
+                        if (n_toks > 1){
+                                //strtol(toks[n_toks-1], &last_num, 10);
+                                size  = strlen(toks[n_toks-1]);
+                                end = toks[n_toks-1][size-1];
+                                //testing the last character of the last tok
+                                if ((end =='c')||(end =='C')){
+                                        //the tok is likely to be the compute
    capability
+                                        rc = _validate_compute_capability(toks[
    n_toks-1], size, &cc);
+                                        if (rc == SLURM_ERROR)
+                                                return SLURM_ERROR;
+                                        if (n_toks == 3) {
```

```
+                                               rc = _validate_rgpu_data(toks[1], &
    cnt_mem);
+                                               if (rc == SLURM_ERROR)
+                                                       return SLURM_ERROR;
+
+                                       }
+                               } else {
+                                       if (n_toks == 2){
+                                       // if the ending is not "c", it should be
    the memory tok
+                                               rc = _validate_rgpu_data(toks[1], &
    cnt_mem);
+                                               if (rc == SLURM_ERROR)
+                                                       return SLURM_ERROR;
+
+                                       } else
+                                               return SLURM_ERROR;
+                               }
+                       }
+                       rc = _validate_rgpu_data(toks[0], &cnt);
+                       if (rc == SLURM_ERROR)
+                               return SLURM_ERROR;
+               }//rgpu plugin
+               else {//normal plugin
+                       config += context_ptr->gres_name_colon_len;
+                       cnt = strtol(config, &last_num, 10);
+                       if (last_num[0]=='\0')
+                               ;
+                       else if ((last_num[0]=='k')||(last_num[0]=='K'))
+                               cnt *= 1024;
+                       else if ((last_num[0]=='m')||(last_num[0]=='M'))
+                               cnt *= (1024*1024);
+                       else if ((last_num[0]=='g')||(last_num[0]=='G'))
+                               cnt *= (1024*1024*1024);
+                       else
+                               return SLURM_ERROR;
+                       if (cnt<0)
+                               return SLURM_ERROR;
+               }
+       } else
+               return SLURM_ERROR;
+
+       *gres_cnt = (uint32_t) cnt;
+       *mem_cnt = (uint64_t) cnt_mem;
+       *com_cap = (uint32_t) cc;
+       return SLURM_SUCCESS;
+}
+
+/*
+ * The same function as _job_state_validate(), but with the capability
+ * of parsering the memory required.
+ */
+static int _job_state_validate_2(char *config, void **gres_data,
+                       slurm_gres_context_t * context_ptr)
+{
+       int rc;
+       uint32_t gres_cnt, com_cap;
+       uint64_t mem_cnt;
+
+
+       rc = _job_config_validate_2(config, &gres_cnt, context_ptr, &mem_cnt, &
    com_cap);
+       if ((rc==SLURM_SUCCESS) && (gres_cnt>0)) {
+               gres_job_state_t *gres_ptr;
+               gres_ptr = xmalloc(sizeof(gres_job_state_t));
+               gres_ptr->gres_cnt_alloc = gres_cnt;
+
+               gres_ptr->rgpu_cnt_mem = mem_cnt;
+               *gres_data = gres_ptr;
+       } else
```

```
+               *gres_data = NULL;
+       return rc;
+}
+
+/*
+ * The same function as gres_plugin_job_state_validate(), but isolating
+ * the rgpu processing, which will be carried out in
+ * gres_rgpu_job_state_validate().
+ * Given a job's requested gres configuration, validate it and build
+ * a gres list and special strings for the rgpu management.
+ * IN req_config - job request's gres input string
+ * OUT gres_list - List of Gres records for this job to track usage
+ * OUT rgpu_req - job request of rgpus, string
+ * OUT new_req - job request of gres (no rgpu), string
+ * RET SLURM_SUCCESS or ESLURM_INVALID_GRES
+ */
+extern int gres_plugin_job_state_validate_2(char *req_config, List *gres_list,
+                               char **rgpu_req, char **new_req)
+{
+       char *tmp_str, *tok, *last = NULL;
+       int i, rc, rc2;
+       gres_state_t *gres_ptr;
+       void *job_gres_data;
+
+       if ((req_config==NULL)||(req_config[0]=='\0')) {
+               *gres_list = NULL;
+               return SLURM_SUCCESS;
+       }
+
+       if ((rc = gres_plugin_init())!=SLURM_SUCCESS)
+               return rc;
+
+       slurm_mutex_lock(&gres_context_lock);
+       tmp_str = xstrdup(req_config);
+       tok = strtok_r(tmp_str, ",", &last);
+       while (tok&&(rc==SLURM_SUCCESS)) {
+               rc2 = SLURM_ERROR;
+               for (i = 0; i<gres_context_cnt; i++) {
+                       rc2 = _job_state_validate_2(tok, &job_gres_data,
+                               &gres_context[i]);
+                       if ((rc2!=SLURM_SUCCESS)||(job_gres_data==NULL))
+                               continue;
+                       if (gres_context[i].plugin_id==1970300786) {
+                               *rgpu_req = xstrdup(tok);
+                               debug3("sched: job_create: gres %s found for this
    job", *rgpu_req);
+                               break;
+                       }
+                       if (*gres_list==NULL) {
+                               *gres_list = list_create(_gres_job_list_delete);
+                               if (*gres_list==NULL)
+                                       fatal("list_create malloc failure");
+                       }
+                       //rewrite string req_config
+                       if (!*new_req)
+                               *new_req = xmalloc(2048);
+                       else {
+                               xstrcat(*new_req, ",");
+                       }
+                       xstrfmtcat(*new_req, "%s", tok);
+                       //
+                       gres_ptr = xmalloc(sizeof(gres_state_t));
+                       gres_ptr->plugin_id = gres_context[i].plugin_id;
+                       gres_ptr->gres_data = job_gres_data;
+                       list_append(*gres_list, gres_ptr);
+                       break; /* processed it */
+               }
+               if (rc2!=SLURM_SUCCESS) {
+                       info("Invalid gres job specification %s", tok);
+                       rc = ESLURM_INVALID_GRES;
```

```
+                       break;
+               }
+               tok = strtok_r(NULL, ",", &last);
+       }
+       slurm_mutex_unlock(&gres_context_lock);
+
+       xfree(tmp_str);
+
+       return rc;
+}
+
+/*
+ * Given a job's requested rgpus configuration, validate it and build
+ * a gres record. Quite similar to the original function
+ * gres_plugin_job_state_validate(),
+ * but only for the gres plugin gres/rgpu.
+ * IN req_config - job request's gres input string
+ * OUT gres_rgpu - gres_state_t record for the rgpus of this job
+ * RET SLURM_SUCCESS or ESLURM_INVALID_GRES
+ */
+extern int gres_rgpu_job_state_validate(char *req_config, void **gres_rgpu)
+{
+       int i, rc, rc2;
+       gres_state_t *gres_ptr;
+       gres_job_state_t *aux;
+       void *job_gres_data;
+
+       if ((req_config==NULL)||(req_config[0]=='\0'))
+               return ESLURM_INVALID_GRES;
+
+       if ((rc = gres_plugin_init())!=SLURM_SUCCESS)
+               return rc;
+
+       slurm_mutex_lock(&gres_context_lock);
+       for (i = 0; i<gres_context_cnt; i++) {
+               rc2 = _job_state_validate_2(req_config, &job_gres_data, &
    gres_context[i]);
+               if ((rc2!=SLURM_SUCCESS)||(job_gres_data==NULL))
+                       continue;
+               gres_ptr = xmalloc(sizeof(gres_state_t));
+               gres_ptr->plugin_id = gres_context[i].plugin_id;
+               gres_ptr->gres_data = job_gres_data;
+               aux = (gres_job_state_t *) gres_ptr->gres_data;
+               *gres_rgpu = gres_ptr;
+               break; /* processed it */
+
+               if (rc2!=SLURM_SUCCESS) {
+                       info("Invalid gres job specification %s", req_config);
+                       rc = ESLURM_INVALID_GRES;
+                       break;
+               }
+       }
+       slurm_mutex_unlock(&gres_context_lock);
+
+       return rc;
+}
+
+/* This function calculates the percentage of free memory of the GPUs of the node
+ * IN node_id - id of the node to process
+ * IN gres_list - gres_state_t record of the rgpus in the node
+ * IN job_mem - amount of rgpu memory required by the job
+ * RET node_gpu_list - list of gpu_data with the calculated weigth
+ */
+extern List weight_my_gpus(int node_id, List gres_list, uint64_t job_mem)
+{
+       int i;
+       List node_gpu_list = NULL;
+       ListIterator iter;
+       node_gpu_list = list_create(NULL);
+       gres_state_t * node_gres_ptr;
```

```
+        gres_node_state_t *gres_ptr;
+        struct gpu_data *gpu;
+
+        iter = list_iterator_create(gres_list);
+        while ((node_gres_ptr = (gres_state_t *) list_next(iter)))
+                if (node_gres_ptr->plugin_id == 1970300786)
+                        break;
+        list_iterator_destroy(iter);
+        gres_ptr = (gres_node_state_t *) node_gres_ptr->gres_data;
+
+        for (i=0; i<gres_ptr->gres_cnt_avail; i++) {
+                if ((gres_ptr->mem_rgpu_alloc[i] + job_mem) > gres_ptr->
    mem_rgpu_avail[i])
+                        continue;
+                gpu = xmalloc(sizeof(struct gpu_data));
+                gpu->node_id = node_id;
+                gpu->gpu_id = i;
+                gpu->weight = (gres_ptr->mem_rgpu_alloc[i] + job_mem) * 100 /
    gres_ptr->mem_rgpu_avail[i];
+                list_append(node_gpu_list, gpu);
+        }
+        return node_gpu_list;
+}
+
+extern int _rgpu_job_alloc_id(struct job_resources *job, void *job_gres_data,
+                void *node_gres_data, int node_cnt, int node_offset,
+                uint32_t job_id, char *node_name, uint32_t * rgpus_remaining,
    uint32_t gpu_id)
+{
+        gres_job_state_t *job_gres_ptr = (gres_job_state_t *) job_gres_data;
+        gres_node_state_t *node_gres_ptr = (gres_node_state_t *) node_gres_data;
+        uint64_t job_mem_req, rgpu_free_mem;
+        char * gres_name = "rgpu";
+
+        /*
+         * Validate data structures. Either job_gres_data->node_cnt and
+         * job_gres_data->gres_bit_alloc are both set or both zero/NULL.
+         */
+        xassert(node_cnt);
+        xassert(node_offset>=0);
+        xassert(job_gres_ptr);
+        xassert(node_gres_ptr);
+        if (job_gres_ptr->node_cnt==0) {
+                job_gres_ptr->node_cnt = node_cnt;
+                if (job_gres_ptr->gres_bit_alloc) {
+                        error("gres/%s: job %u node_cnt==0 and bit_alloc is "
+                              "set", gres_name, job_id);
+                        xfree(job_gres_ptr->gres_bit_alloc);
+                }
+                job_gres_ptr->gres_bit_alloc = xmalloc(sizeof(bitstr_t *)*
+                        node_cnt);
+                if (job_gres_ptr->rgpu_mem_alloc) {
+                        error("gres/%s: job %u node_cnt==0 and rgpu_mem_alloc is "
+                              "set", gres_name, job_id);
+                        xfree(job_gres_ptr->rgpu_mem_alloc);
+                }
+                job_gres_ptr->rgpu_mem_alloc = xmalloc(sizeof(uint64_t *)*
+                        node_cnt);
+        } else if (job_gres_ptr->node_cnt<node_cnt) {
+                error("gres/%s: job %u node_cnt increase from %u to %d",
+                        gres_name, job_id, job_gres_ptr->node_cnt, node_cnt);
+                if (node_offset>=job_gres_ptr->node_cnt)
+                        return SLURM_ERROR;
+        } else if (job_gres_ptr->node_cnt>node_cnt) {
+                error("gres/%s: job %u node_cnt decrease from %u to %d",
+                        gres_name, job_id, job_gres_ptr->node_cnt, node_cnt);
+        }
+
+        /*
+         * Select the specific resources to use for this job.
```

```
+        */
+        if (job_gres_ptr->gres_bit_alloc[node_offset]) {
+        } else if (node_gres_ptr->gres_bit_alloc) {
+                job_gres_ptr->gres_bit_alloc[node_offset] =
+                        bit_alloc(node_gres_ptr->gres_cnt_avail);
+                if (job_gres_ptr->gres_bit_alloc[node_offset]==NULL)
+                        fatal("bit_copy: malloc failure");
+                job_gres_ptr->rgpu_mem_alloc[node_offset] =
+                        xmalloc(sizeof(uint64_t)*node_gres_ptr->gres_cnt_avail);
+                if (job_gres_ptr->rgpu_mem_alloc[node_offset]==NULL)
+                        fatal("bit_copy: malloc failure");
+        }
+
+        /* only one gpu */
+        /* requesting min memory for the job */
+        job_mem_req = job->rgpumem_min;
+        /* rgpu full */
+        if (bit_test(node_gres_ptr->gres_bit_alloc, gpu_id))
+                return SLURM_SUCCESS;
+        /* compute capability smaller than required */
+        if (job->rgpucc > node_gres_ptr->cc_version[gpu_id])
+                return SLURM_SUCCESS;
+        /* exclusive mode */
+        if (job->rgpumem_max == 0) {
+                /* rgpu is being used by others */
+                if (node_gres_ptr->mem_rgpu_alloc[gpu_id] > 0)
+                        return SLURM_SUCCESS;
+                /* if it is free, request all the memory */
+                else
+                        job_mem_req = node_gres_ptr->mem_rgpu_avail[gpu_id];
+        }
+        /* both modes (excl and shar) */
+        /* calculating how much memory is free */
+        rgpu_free_mem = node_gres_ptr->mem_rgpu_avail[gpu_id] - node_gres_ptr->
    mem_rgpu_alloc[gpu_id];
+        /* rgpu has enough memory */
+        if (job_mem_req > rgpu_free_mem)
+                return SLURM_SUCCESS;
+        /* job fits in rgpu memory */
+        if ((job_mem_req + node_gres_ptr->mem_rgpu_alloc[gpu_id]) > node_gres_ptr->
    mem_rgpu_avail[gpu_id])
+                return SLURM_SUCCESS;
+
+        debug("RCUDA_ALLOC %i - Node(%i):%s GPU:%i Mem_Job:%lu "
+                "Mem_Alloc:%lu Mem_total:%lu",
+                job_id, node_offset, node_name, gpu_id,
+                job_gres_ptr->rgpu_mem_alloc[node_offset][gpu_id],
+                node_gres_ptr->mem_rgpu_alloc[gpu_id],
+                node_gres_ptr->mem_rgpu_avail[gpu_id]);
+
+        node_gres_ptr->mem_rgpu_alloc[gpu_id] += job_mem_req;
+        job_gres_ptr->rgpu_mem_alloc[node_offset][gpu_id] = job_mem_req;
+
+        /* if rgpu is full, set the bit */
+        if (node_gres_ptr->mem_rgpu_alloc[gpu_id] == node_gres_ptr->mem_rgpu_avail[
    gpu_id]){
+                node_gres_ptr->gres_cnt_alloc++;
+                bit_set(node_gres_ptr->gres_bit_alloc, gpu_id);
+        }
+        /* set the bit in the job's bitmap */
+        bit_set(job_gres_ptr->gres_bit_alloc[node_offset], gpu_id);
+        *rgpus_remaining -= 1;
+
+        return SLURM_SUCCESS;
+}
+
+/*
+ * Allocate rgpus to a job and update node and job rgpus information
+ * IN job_gres_status          - information about the rgpus of the job
+ * IN node_gres_list           - node's rgpu list built by
```

```
+ *                                       gres_plugin_node_config_validate() ->
+ *
    _rgpu_node_config_validate()
+ * IN node_offset                   - zero-origin index to the node of interest
+ * IN job_id                        - job's ID (for logging)
+ * IN node_name                     - name of the node (for logging)
+ * IN/OUT rgpus_remaining           - number of RGPUs that still have not
+ *                                                                          been
    allocated
+ * IN node_cnt                      - total number of nodes originally
+ *                                                                          allocated
    to the job
+ * RET SLURM_SUCCESS or error code
+ */
+extern int gres_rgpu_job_alloc_id(struct job_resources *job, List node_gres_list,
+                    int node_offset, uint32_t job_id, char *node_name,
+                    uint32_t * rgpus_remaining, uint32_t gpu_id)
+{
+       int i, rc, rc2;
+       ListIterator node_gres_iter;
+       gres_state_t *job_gres_ptr, *node_gres_ptr;
+       job_gres_ptr = (gres_state_t *) job->rgpu_job_state;
+       uint32_t node_cnt = bit_set_count(job->rgpu_node_bitmap);
+
+       bool plugin_found = false;
+
+       if (job_gres_ptr==NULL)
+               return SLURM_SUCCESS;
+       if (node_gres_list==NULL) {
+               error("gres_job_alloc: job %u has gres specification while "
+                       "node %s has none", job_id, node_name);
+               return SLURM_ERROR;
+       }
+
+       rc = gres_plugin_init();
+
+       slurm_mutex_lock(&gres_context_lock);
+
+       if (job_gres_ptr->plugin_id==1970300786) {
+               plugin_found = true;
+       } else
+               error("gres_rgpu_job_alloc: no rgpu plugin configured "
+               "for data type %u for job %u and node %s",
+               job_gres_ptr->plugin_id, job_id, node_name);
+
+       for (i = 0; i<gres_context_cnt; i++) {
+               if (job_gres_ptr->plugin_id==
+                       gres_context[i].plugin_id)
+                       break;
+       }
+       if (i>=gres_context_cnt) {
+               error("gres_rgpu_job_alloc: no plugin configured "
+                       "for data type %u for job %u and node %s",
+                       job_gres_ptr->plugin_id, job_id, node_name);
+               /* A likely sign that GresPlugins has changed */
+       }
+
+       node_gres_iter = list_iterator_create(node_gres_list);
+       while ((node_gres_ptr = (gres_state_t *)
+               list_next(node_gres_iter))) {
+               if (job_gres_ptr->plugin_id==node_gres_ptr->plugin_id)
+                       break;
+       }
+       list_iterator_destroy(node_gres_iter);
+       if (node_gres_ptr==NULL) {
+               error("gres_rgpu_job_alloc: job %u allocated gres/%s "
+                       "on node %s lacking that gres",
+                       job_id, gres_context[i].gres_name, node_name);
+       }
+       rc2 = _rgpu_job_alloc_id(job, job_gres_ptr->gres_data,
```

```
+                       node_gres_ptr->gres_data, node_cnt,
+                       node_offset, job_id, node_name, rgpus_remaining, gpu_id);
+       if (rc2!=SLURM_SUCCESS)
+               rc = rc2;
+
+       slurm_mutex_unlock(&gres_context_lock);
+
+       return rc;
+}
+
+/*
+ *  Allocate rgpus while the quantity of remaining rgpus for the job is
+ * greater than 0. If the job needs more rgpus than the node hosts
+ * in the next call to this function by other node, the allocation
+ * of rgpus in this new node will carry on.
+ */
+extern int _rgpu_job_alloc(struct job_resources *job, void *job_gres_data,
+               void *node_gres_data, int node_cnt, int node_offset,
+               uint32_t job_id, char *node_name, uint32_t * rgpus_remaining)
+{
+       //debug2("RCUDA_DEALLOC job:%i (pid:%d - thread:%u): %s(%s,%d)", job_id,
    getpid(), (unsigned int)pthread_self(), __FILE__, __func__, __LINE__);
+       int i;
+       gres_job_state_t *job_gres_ptr = (gres_job_state_t *) job_gres_data;
+       gres_node_state_t *node_gres_ptr = (gres_node_state_t *) node_gres_data;
+       uint64_t job_mem_req, rgpu_free_mem;
+       char * gres_name = "rgpu";
+
+       /*
+       struct tm *tm;
+       time_t t;
+       char str_time[100];
+       */
+
+       /*
+        * Validate data structures. Either job_gres_data->node_cnt and
+        * job_gres_data->gres_bit_alloc are both set or both zero/NULL.
+        */
+       xassert(node_cnt);
+       xassert(node_offset>=0);
+       xassert(job_gres_ptr);
+       xassert(node_gres_ptr);
+       if (job_gres_ptr->node_cnt==0) {
+               job_gres_ptr->node_cnt = node_cnt;
+               if (job_gres_ptr->gres_bit_alloc) {
+                       error("gres/%s: job %u node_cnt==0 and bit_alloc is "
+                               "set", gres_name, job_id);
+                       xfree(job_gres_ptr->gres_bit_alloc);
+               }
+               job_gres_ptr->gres_bit_alloc = xmalloc(sizeof(bitstr_t *)*
+                       node_cnt);
+               if (job_gres_ptr->rgpu_mem_alloc) {
+                       error("gres/%s: job %u node_cnt==0 and rgpu_mem_alloc is "
+                               "set", gres_name, job_id);
+                       xfree(job_gres_ptr->rgpu_mem_alloc);
+               }
+               job_gres_ptr->rgpu_mem_alloc = xmalloc(sizeof(uint64_t *)*
+                       node_cnt);
+       } else if (job_gres_ptr->node_cnt<node_cnt) {
+               error("gres/%s: job %u node_cnt increase from %u to %d",
+                       gres_name, job_id, job_gres_ptr->node_cnt, node_cnt);
+               if (node_offset>=job_gres_ptr->node_cnt)
+                       return SLURM_ERROR;
+       } else if (job_gres_ptr->node_cnt>node_cnt) {
+               error("gres/%s: job %u node_cnt decrease from %u to %d",
+                       gres_name, job_id, job_gres_ptr->node_cnt, node_cnt);
+       }
+
+       /*
+        * Select the specific resources to use for this job.
```

```
+          */
+         if (job_gres_ptr->gres_bit_alloc[node_offset]) {
+         } else if (node_gres_ptr->gres_bit_alloc) {
+                 job_gres_ptr->gres_bit_alloc[node_offset] =
+                         bit_alloc(node_gres_ptr->gres_cnt_avail);
+                 if (job_gres_ptr->gres_bit_alloc[node_offset]==NULL)
+                         fatal("bit_copy: malloc failure");
+                 job_gres_ptr->rgpu_mem_alloc[node_offset] =
+                         xmalloc(sizeof(uint64_t)*node_gres_ptr->gres_cnt_avail);
+                 if (job_gres_ptr->rgpu_mem_alloc[node_offset]==NULL)
+                         fatal("bit_copy: malloc failure");
+                 for (i = 0; ((*rgpus_remaining>0) && (i<node_gres_ptr->
    gres_cnt_avail)); i++) {
+                         /* requesting min memory for the job */
+                         job_mem_req = job->rgpumem_min;
+                         /* rgpu full */
+                         if (bit_test(node_gres_ptr->gres_bit_alloc, i))
+                                 continue;
+                         /* compute capability smaller than required */
+                         if (job->rgpucc > node_gres_ptr->cc_version[i])
+                                 continue;
+                         /* exclusive mode */
+                         if (job->rgpumem_max == 0) {
+                                 /* rgpu is being used by others */
+                                 if (node_gres_ptr->mem_rgpu_alloc[i] > 0)
+                                         continue;
+                                 /* if it is free, request all the memory */
+                                 else
+                                         job_mem_req = node_gres_ptr->mem_rgpu_avail
    [i];
+                         }
+                         /* both modes (excl and shar) */
+                         /* calculating how much memory is free */
+                         rgpu_free_mem = node_gres_ptr->mem_rgpu_avail[i] -
    node_gres_ptr->mem_rgpu_alloc[i];
+                         /* rgpu has enough memory */
+                         if (job_mem_req > rgpu_free_mem)
+                                 continue;
+                         /* job fits in rgpu memory */
+                         if ((job_mem_req + node_gres_ptr->mem_rgpu_alloc[i]) >
    node_gres_ptr->mem_rgpu_avail[i])
+                                 continue;
+
+                         /* updating memory state in rgpu
+                         t = time(NULL);
+                         tm = localtime(&t);
+                         strftime(str_time, sizeof(str_time), "%H:%M:%S", tm);
+
+                            debug("RCUDA_ALLOC(1/2) %i (%s) Node:%s GPU:%i Mem_Job:%
    lu "
+                                 "Mem_Free:%lu Mem_Alloc:%lu Mem_total:%lu
    Position_array:%p Position_array[i]:%p",
+                                 job_id, str_time, node_name, i, job_mem_req,
+                                 node_gres_ptr->mem_rgpu_avail[i] - node_gres_ptr->
    mem_rgpu_alloc[i],
+                                 node_gres_ptr->mem_rgpu_alloc[i], node_gres_ptr->
    mem_rgpu_avail[i],
+                                 (void *) (node_gres_ptr->mem_rgpu_alloc),
+                                 (void *) (&node_gres_ptr->mem_rgpu_alloc[i]));
+                           */
+
+                         debug("RCUDA_ALLOC %i - Node(%i):%s GPU:%i Mem_Job:%lu "
+                                 "Mem_Alloc:%lu Mem_total:%lu",
+                                 job_id, node_offset, node_name, i,
+                                 job_gres_ptr->rgpu_mem_alloc[node_offset][i],
+                                 node_gres_ptr->mem_rgpu_alloc[i],
+                                 node_gres_ptr->mem_rgpu_avail[i]);
+
+                         //xassert(job_mem_req <= node_gres_ptr->mem_rgpu_avail[i]);
+                         node_gres_ptr->mem_rgpu_alloc[i] += job_mem_req;
```

```
+                               job_gres_ptr->rgpu_mem_alloc[node_offset][i] = job_mem_req;
+                               //xassert(node_gres_ptr->mem_rgpu_alloc[i] > 0);
+                               /*
+                                       debug("RCUDA_ALLOC(2/2) %i (%s) Node:%s GPU:%i Mem_Job
    :%lu "
+                                               "Mem_Free:%lu Mem_Alloc:%lu Mem_total:%lu
    Position_array:%p Position_array[i]:%p",
+                                               job_id, str_time, node_name, i, job_mem_req,
+                                               node_gres_ptr->mem_rgpu_avail[i] - node_gres_ptr->
    mem_rgpu_alloc[i],
+                                               node_gres_ptr->mem_rgpu_alloc[i], node_gres_ptr->
    mem_rgpu_avail[i],
+                                               (void *) (node_gres_ptr->mem_rgpu_alloc),
+                                               (void *) (&node_gres_ptr->mem_rgpu_alloc[i]));
+                               */
+                               /* if rgpu is full, set the bit */
+                               if (node_gres_ptr->mem_rgpu_alloc[i] == node_gres_ptr->
    mem_rgpu_avail[i]){
+                                       node_gres_ptr->gres_cnt_alloc++;
+                                       bit_set(node_gres_ptr->gres_bit_alloc, i);
+                               }
+                               /* set the bit in the job's bitmap */
+                               bit_set(job_gres_ptr->gres_bit_alloc[node_offset], i);
+                               *rgpus_remaining -= 1;
+                       }
+               }
+               return SLURM_SUCCESS;
+}
+
+/*
+ * Allocate rgpus to a job and update node and job rgpus information
+ * IN job_gres_status           - information about the rgpus of the job
+ * IN node_gres_list            - node's rgpu list built by
+ *                                      gres_plugin_node_config_validate() ->
+ *
    _rgpu_node_config_validate()
+ * IN node_offset               - zero-origin index to the node of interest
+ * IN job_id                    - job's ID (for logging)
+ * IN node_name                 - name of the node (for logging)
+ * IN/OUT rgpus_remaining       - number of RGPUs that still have not
+ *                                                                      been
    allocated
+ * IN node_cnt                  - total number of nodes originally
+ *                                                                      allocated
    to the job
+ * RET SLURM_SUCCESS or error code
+ */
+extern int gres_rgpu_job_alloc(struct job_resources *job, List node_gres_list,
+                       int node_offset, uint32_t job_id, char *node_name,
+                       uint32_t * rgpus_remaining)
+{
+       int i, rc, rc2;
+       ListIterator node_gres_iter;
+       gres_state_t *job_gres_ptr, *node_gres_ptr;
+       job_gres_ptr = (gres_state_t *) job->rgpu_job_state;
+       uint32_t node_cnt = bit_set_count(job->rgpu_node_bitmap);
+
+       bool plugin_found = false;
+
+       if (job_gres_ptr==NULL)
+               return SLURM_SUCCESS;
+       if (node_gres_list==NULL) {
+               error("gres_job_alloc: job %u has gres specification while "
+                       "node %s has none", job_id, node_name);
+               return SLURM_ERROR;
+       }
+
+       rc = gres_plugin_init();
+
+       slurm_mutex_lock(&gres_context_lock);
```

```
+
+       if (job_gres_ptr->plugin_id==1970300786) {
+               plugin_found = true;
+       } else
+               error("gres_rgpu_job_alloc: no rgpu plugin configured "
+               "for data type %u for job %u and node %s",
+               job_gres_ptr->plugin_id, job_id, node_name);
+
+       for (i = 0; i<gres_context_cnt; i++) {
+               if (job_gres_ptr->plugin_id==
+                       gres_context[i].plugin_id)
+                       break;
+       }
+       if (i>=gres_context_cnt) {
+               error("gres_rgpu_job_alloc: no plugin configured "
+                       "for data type %u for job %u and node %s",
+                       job_gres_ptr->plugin_id, job_id, node_name);
+               /* A likely sign that GresPlugins has changed */
+       }
+
+       node_gres_iter = list_iterator_create(node_gres_list);
+       while ((node_gres_ptr = (gres_state_t *)
+               list_next(node_gres_iter))) {
+               if (job_gres_ptr->plugin_id==node_gres_ptr->plugin_id)
+                       break;
+       }
+       list_iterator_destroy(node_gres_iter);
+       if (node_gres_ptr==NULL) {
+               error("gres_rgpu_job_alloc: job %u allocated gres/%s "
+                       "on node %s lacking that gres",
+                       job_id, gres_context[i].gres_name, node_name);
+       }
+       rc2 = _rgpu_job_alloc(job, job_gres_ptr->gres_data,
+               node_gres_ptr->gres_data, node_cnt,
+               node_offset, job_id, node_name, rgpus_remaining);
+       if (rc2!=SLURM_SUCCESS)
+               rc = rc2;
+
+       slurm_mutex_unlock(&gres_context_lock);
+
+       return rc;
+}
+
+/* Deallocate every single rgpu allocated in this node by this job */
+static int _rgpu_job_dealloc(void *job_gres_data, void *node_gres_data,
+                int node_offset, uint32_t job_id, char *node_name)
+{
+       //debug2("RCUDA_DEALLOC job:%i (pid:%d - thread:%u): %s(%s,%d)", job_id,
    getpid(), (unsigned int)pthread_self(), __FILE__, __func__, __LINE__);
+       int i, len, len2, gres_cnt;
+       gres_job_state_t *job_gres_ptr = (gres_job_state_t *) job_gres_data;
+       gres_node_state_t *node_gres_ptr = (gres_node_state_t *) node_gres_data;
+
+       char * gres_name = "rgpu";
+       /* calcular fecha actual
+       struct tm *tm;
+       time_t t;
+       char str_time[100];
+       */
+
+       /*
+        * Validate data structures. Either job_gres_data->node_cnt and
+        * job_gres_data->gres_bit_alloc are both set or both zero/NULL.
+        */
+       xassert(node_offset>=0);
+       xassert(job_gres_ptr);
+       xassert(node_gres_ptr);
+       if (job_gres_ptr->node_cnt<=node_offset) {
+               error("gres/%s: job %u dealloc of node %s bad node_offset %d "
+                       "count is %u", gres_name, job_id, node_name, node_offset,
```

```
+                               job_gres_ptr->node_cnt);
+                       return SLURM_ERROR;
+               }
+
+               if (node_gres_ptr->gres_bit_alloc && job_gres_ptr->gres_bit_alloc &&
+                       job_gres_ptr->gres_bit_alloc[node_offset]) {
+                       len = bit_size(job_gres_ptr->gres_bit_alloc[node_offset]);
+                       len2 = bit_size(node_gres_ptr->gres_bit_alloc);
+                       if (len2!=len) {
+                               error("gres/%s: job %u and node %s bitmap sizes differ "
+                                       "(%d != %d)", gres_name, job_id, node_name, len,
    len2);
+                               len = MIN(len, len2);
+                       }
+
+                       for (i = 0; i<len; i++) {
+                               if (!bit_test(job_gres_ptr->gres_bit_alloc[node_offset], i)
    )
+                                       continue;
+                               /*
+                               t = time(NULL);
+                               tm = localtime(&t);
+                               strftime(str_time, sizeof(str_time), "%H:%M:%S", tm);
+                               debug("len2=%i, len=%i, found=%i, i=%i, node_offset=%i",
    len2, len, node_gres_ptr->gres_cnt_found, i, node_offset);
+                               */
+                               debug("RCUDA_DEALLOC %i - Node(%i):%s GPU:%i Mem_Job:%lu "
+                                       "Mem_Alloc:%lu Mem_total:%lu",
+                                       job_id, node_offset, node_name, i,
+                                       job_gres_ptr->rgpu_mem_alloc[node_offset][i],
+                                       node_gres_ptr->mem_rgpu_alloc[i],
+                                       node_gres_ptr->mem_rgpu_avail[i]);
+                               /*debug("mem_rgpu_free:%lu, (avail:%lu - alloc:%lu)",
    node_gres_ptr->mem_rgpu_avail[i] - node_gres_ptr->mem_rgpu_alloc[i],
    node_gres_ptr->mem_rgpu_avail[i], node_gres_ptr->mem_rgpu_alloc[i]);
+                               */
+
+                               //xassert(node_gres_ptr->mem_rgpu_alloc);
+                               //xassert(node_gres_ptr->mem_rgpu_alloc[i] > 0);
+                               node_gres_ptr->mem_rgpu_alloc[i] -= job_gres_ptr->
    rgpu_mem_alloc[node_offset][i];
+                               /*
+                               debug("RCUDA_DEALLOC(2/2) %i (%s) Node:%s GPU:%i Mem_Job:%
    lu "
+                                       "Mem_Free:%lu Mem_Alloc:%lu Mem_total:%lu
    Position_array:%p Position_array[i]:%p",
+                                       job_id, str_time, node_name, i,
+                                       job_gres_ptr->rgpu_mem_alloc[node_offset][i],
+                                       node_gres_ptr->mem_rgpu_avail[i] - node_gres_ptr->
    mem_rgpu_alloc[i],
+                                       node_gres_ptr->mem_rgpu_alloc[i], node_gres_ptr->
    mem_rgpu_avail[i],
+                                       (void *) (node_gres_ptr->mem_rgpu_alloc),
+                                       (void *) (&node_gres_ptr->mem_rgpu_alloc[i]));
+                               */
+                               if (node_gres_ptr->mem_rgpu_alloc[i]==0) {
+                                       bit_clear(node_gres_ptr->gres_bit_alloc, i);
+                                       node_gres_ptr->gres_cnt_alloc--;
+                               } else if (node_gres_ptr->mem_rgpu_alloc[i]<0)
+                                       error("gres/%s: job %u dealloc node %s gres "
+                                       "count underflow", gres_name, job_id, node_name);
+                       }
+               } else {
+                       node_gres_ptr->gres_cnt_alloc = 0;
+                       error("gres/%s: job %u node %s gres count underflow",
+                               gres_name, job_id, node_name);
+               }
+
+               if (job_gres_ptr->gres_bit_alloc&&
+                       job_gres_ptr->gres_bit_alloc[node_offset]&&
```

```
+               node_gres_ptr->topo_gres_bitmap&&
+               node_gres_ptr->topo_gres_cnt_alloc) {
+               for (i = 0; i<node_gres_ptr->topo_cnt; i++) {
+
+                       gres_cnt = bit_overlap(job_gres_ptr->
+                               gres_bit_alloc[node_offset],
+                               node_gres_ptr->
+                               topo_gres_bitmap[i]);
+                       node_gres_ptr->topo_gres_cnt_alloc[i] -= gres_cnt;
+               }
+       }
+       return SLURM_SUCCESS;
+}
+
+/*
+ * Deallocate rgpus from a job and update node and job rgpus information
+ * IN job_gres_status   - information about the rgpus of the job
+ * IN node_gres_list    - node's rgpu list built by
+ *                        gres_plugin_node_config_validate() ->
+ *                                                _rgpu_node_config_validate
    ()
+ * IN node_offset       - zero-origin index to the node of interest
+ * IN job_id            - job's ID (for logging)
+ * IN node_name         - name of the node (for logging)
+ * RET SLURM_SUCCESS or error code
+ */
+extern int gres_rgpu_job_dealloc(void * job_gres_status, List node_gres_list,
+                       int node_offset, uint32_t job_id, char *node_name)
+{
+       int i, rc, rc2;
+       ListIterator node_gres_iter;
+       gres_state_t *job_gres_ptr, *node_gres_ptr;
+       char *gres_name = "rgpu";
+       job_gres_ptr = (gres_state_t *) job_gres_status;
+
+       if (job_gres_status==NULL) {
+               return SLURM_SUCCESS;
+       }
+       if (node_gres_list==NULL) {
+               error("gres_job_dealloc: job %u has gres specification while "
+                       "node %s has none", job_id, node_name);
+               return SLURM_ERROR;
+       }
+
+       rc = gres_plugin_init();
+
+       slurm_mutex_lock(&gres_context_lock);
+       if (job_gres_ptr->plugin_id!=1970300786) {
+               error("gres_rgpu_job_dealloc: no plugin configured "
+                       "for data type %u for job %u and node %s",
+                       job_gres_ptr->plugin_id, job_id, node_name);
+       }
+
+       for (i = 0; i<gres_context_cnt; i++) {
+               if (job_gres_ptr->plugin_id==
+                       gres_context[i].plugin_id)
+                       break;
+       }
+       if (i>=gres_context_cnt) {
+               error("gres_plugin_job_dealloc: no plugin configured "
+                       "for data type %u for job %u and node %s",
+                       job_gres_ptr->plugin_id, job_id, node_name);
+               /* A likely sign that GresPlugins has changed */
+               gres_name = "UNKNOWN";
+       }
+       node_gres_iter = list_iterator_create(node_gres_list);
+       while ((node_gres_ptr = (gres_state_t *)
+               list_next(node_gres_iter))) {
+               if (job_gres_ptr->plugin_id==node_gres_ptr->plugin_id)
+                       break;
```

```
+	}
+	list_iterator_destroy(node_gres_iter);
+	if (node_gres_ptr==NULL) {
+		error("gres_rgpu_job_dealloc: node %s lacks gres/%s "
+			"for job %u", node_name, gres_name, job_id);
+	}
+
+	rc2 = _rgpu_job_dealloc(job_gres_ptr->gres_data,
+		node_gres_ptr->gres_data, node_offset, job_id, node_name);
+
+	if (rc2!=SLURM_SUCCESS)
+		rc = rc2;
+
+	slurm_mutex_unlock(&gres_context_lock);
+	return rc;
+}
+
+/*
+ * The same function as _step_state_validate(), but with the capability
+ * of parsering the memory required. Similar to the process of the
+ * job_state_validate().
+ */
+static int _step_state_validate_2(char *config, void **gres_data,
+			slurm_gres_context_t * context_ptr)
+{
+	int rc;
+	uint32_t gres_cnt, com_cap;
+	uint64_t mem_cnt;
+
+	rc = _job_config_validate_2(config, &gres_cnt, context_ptr, &mem_cnt, &
    com_cap);
+	if ((rc==SLURM_SUCCESS) && (gres_cnt>0)) {
+		gres_step_state_t *gres_ptr;
+		gres_ptr = xmalloc(sizeof(gres_step_state_t));
+		gres_ptr->gres_cnt_alloc = gres_cnt;
+		gres_ptr->rgpu_cnt_mem = mem_cnt;
+		gres_ptr->com_cap = com_cap;
+		*gres_data = gres_ptr;
+	} else
+		*gres_data = NULL;
+
+	return rc;
+}
+
+/*
+ * The same function as gres_plugin_step_state_validate(), but isolating
+ * the rgpu processing, which will be carried out in
+ * gres_rgpu_step_state_validate().
+ * Given a step's requested gres configuration, validate it and build
+ * a gres list and special strings for the rgpu management.
+ * This is a very similar method to the job_state_validate(), and the
+ * global idea is the same, but this is used for the steps of the job.
+ * IN req_config - step request's gres input string
+ * OUT step_gres_list - List of Gres records for this step to track usage
+ * IN job_gres_list - List of Gres records for this job
+ * IN job_id, step_id - ID of the step being allocated.
+ * RET SLURM_SUCCESS or ESLURM_INVALID_GRES
+ */
+extern int
+gres_plugin_step_state_validate_2(char *req_config,
+				List *step_gres_list, List job_gres_list,
    uint32_t job_id,
+				uint32_t step_id, char **rgpu_req, char **new_req
    )
+{
+	char *tmp_str, *tok, *last = NULL;
+	int i, rc, rc2, rc3;
+	gres_state_t *step_gres_ptr, *job_gres_ptr;
+	void *step_gres_data, *job_gres_data;
+	ListIterator job_gres_iter;
```

```
+
+	*step_gres_list = NULL;
+	if ((req_config==NULL)||(req_config[0]=='\0'))
+		return SLURM_SUCCESS;
+
+	if ((rc = gres_plugin_init())!=SLURM_SUCCESS)
+		return rc;
+
+	slurm_mutex_lock(&gres_context_lock);
+	tmp_str = xstrdup(req_config);
+	tok = strtok_r(tmp_str, ",", &last);
+	while (tok&&(rc==SLURM_SUCCESS)) {
+		rc2 = SLURM_ERROR;
+		for (i = 0; i<gres_context_cnt; i++) {
+			rc2 = _step_state_validate_2(tok, &step_gres_data,
+				&gres_context[i]);
+			if ((rc2!=SLURM_SUCCESS)||(step_gres_data==NULL))
+				continue;
+			if (gres_context[i].plugin_id==1970300786) {
+				*rgpu_req = xstrdup(tok);
+				break;
+			}
+			if (job_gres_list==NULL) {
+				info("step %u.%u has gres spec, job has none",
+					job_id, step_id);
+				rc2 = ESLURM_INVALID_GRES;
+				continue;
+			}
+
+			/* Now make sure the step's request isn't too big for
+			 * the job's gres allocation */
+			job_gres_iter = list_iterator_create(job_gres_list);
+			if (job_gres_iter==NULL)
+				fatal("list_iterator_create: malloc failure");
+			while ((job_gres_ptr = (gres_state_t *)
+				list_next(job_gres_iter))) {
+				if (job_gres_ptr->plugin_id==
+					gres_context[i].plugin_id)
+					break;
+			}
+			list_iterator_destroy(job_gres_iter);
+			if (job_gres_ptr==NULL) {
+				info("Step %u.%u gres request not in job "
+					"alloc %s", job_id, step_id, tok);
+				rc = ESLURM_INVALID_GRES;
+				_step_state_delete(step_gres_data);
+				break;
+			}
+
+			job_gres_data = job_gres_ptr->gres_data;
+			rc3 = _step_test(step_gres_data, job_gres_data, NO_VAL,
+				true, gres_context[i].gres_name,
+				job_id, step_id);
+			if (rc3==0) {
+				info("Step %u.%u gres higher than in job "
+					"allocation %s", job_id, step_id, tok);
+				rc = ESLURM_INVALID_GRES;
+				_step_state_delete(step_gres_data);
+				break;
+			}
+
+			if (*step_gres_list==NULL) {
+				*step_gres_list = list_create(
+					_gres_step_list_delete);
+				if (*step_gres_list==NULL)
+					fatal("list_create malloc failure");
+			}
+			//rewrite string req_config
+			if (!*new_req)
+				*new_req = xmalloc(2048);
```

```
+                               else {
+                                       xstrcat(*new_req, ",");
+                               }
+                               xstrfmtcat(*new_req, "%s", tok);
+                               //
+                               step_gres_ptr = xmalloc(sizeof(gres_state_t));
+                               step_gres_ptr->plugin_id = gres_context[i].plugin_id;
+                               step_gres_ptr->gres_data = step_gres_data;
+                               list_append(*step_gres_list, step_gres_ptr);
+                               break; /* processed it */
+                       }
+                       if (rc2!=SLURM_SUCCESS) {
+                               info("Invalid gres step %u.%u specification %s",
+                                       job_id, step_id, tok);
+                               rc = ESLURM_INVALID_GRES;
+                               break;
+                       }
+                       tok = strtok_r(NULL, ",", &last);
+               }
+               slurm_mutex_unlock(&gres_context_lock);
+
+               xfree(tmp_str);
+
+               return rc;
+}
+
+/*
+ * Given a step's requested rgpus configuration, validate it and build
+ * a gres record. Quite similar to the original function
+ * gres_plugin_step_state_validate(), but only for the
+ * gres plugin gres/rgpu.
+ * OUT rgpu_gres - gres_state_t record for the rgpus of this step
+ * IN req_config - job request's gres input string
+ * IN job_rgpu - number of rgpu required by the step
+ * IN job_memrgpu - quantity of memory required by the step
+ * RET SLURM_SUCCESS or ESLURM_INVALID_GRES
+ */
+extern int
+gres_rgpu_step_state_validate(char *req_config, void ** rgpu_gres,
+                               uint32_t job_rgpu, uint64_t job_memrgpu)
+{
+       char *tmp_str;
+       int i, rc, rc2 = SLURM_SUCCESS;
+       gres_state_t *step_gres_ptr;
+       void *step_gres_data;
+       gres_step_state_t *aux;
+
+       if ((req_config==NULL)||(req_config[0]=='\0'))
+               return SLURM_SUCCESS;
+
+       if ((rc = gres_plugin_init())!=SLURM_SUCCESS)
+               return rc;
+
+       slurm_mutex_lock(&gres_context_lock);
+       tmp_str = xstrdup(req_config);
+       for (i = 0; i<gres_context_cnt; i++) {
+               rc2 = _step_state_validate_2(tmp_str, &step_gres_data,
+                       &gres_context[i]);
+               if ((rc2!=SLURM_SUCCESS)||(step_gres_data==NULL))
+                       continue;
+               if (gres_context[i].plugin_id==1970300786) {
+                       aux = (gres_step_state_t *) step_gres_data;
+                       if (aux->gres_cnt_alloc>job_rgpu) {
+                               info("Invalid number of rgpus step specification");
+                               _step_state_delete(step_gres_data);
+                               rc2 = ESLURM_INVALID_GRES;
+                               break;
+                       } else if (aux->rgpu_cnt_mem>job_memrgpu) {
+                               info("Invalid quantity of memory of rgpu step
        specification");
```

```
+                               _step_state_delete(step_gres_data);
+                               rc2 = ESLURM_INVALID_GRES;
+                               break;
+                       }
+                       // new in version issue016, which allow to request x rgpus
    in y nodes (x*y rgpus)
+                       aux->gres_cnt_alloc = job_rgpu;
+                       //
+                       step_gres_ptr = xmalloc(sizeof(gres_state_t));
+                       step_gres_ptr->plugin_id = gres_context[i].plugin_id;
+                       step_gres_ptr->gres_data = step_gres_data;
+                       *rgpu_gres = step_gres_ptr;
+                       break; /* processed it */
+               }
+       }
+       if (rc2!=SLURM_SUCCESS) {
+               //info("Invalid gres step %u.%u specification %s",
+               //        job_id, step_id, tok);
+               rc = ESLURM_INVALID_GRES;
+       }
+       slurm_mutex_unlock(&gres_context_lock);
+       xfree(tmp_str);
+
+       return rc;
+}
+
+/*
+ * Allocate resource rgpu to a step and update job and step
+ * rgpu information.
+ * IN step_rgpu_bitstr - step's rgpu record built by gres_rgpu_step_state_validate
    ()
+ * IN job_rgpu_bitstr - job's rgpu record built by gres_rgpu_job_state_validate()
+ * IN job_id, step_id - ID of the step being allocated.
+ * OUT output - string with the rgpus allocated.
+ * RET SLURM_SUCCESS or error code
+ */
+extern int gres_rgpu_step_alloc(void *step_rgpu, bitstr_t *step_rgpu_bitstr,
+                       bitstr_t *job_rgpu_bitstr, char *rgpu_list, uint32_t job_id
    ,
+                       uint32_t step_id, char **output)
+{
+       char *tmp_str, *tok, *dev_list;
+       gres_state_t *step_gres_ptr;
+       gres_step_state_t *step_gres_data;
+       int remaining;
+       int i_bit, i_first, i_last;
+
+       step_gres_ptr = (gres_state_t *) step_rgpu;
+       step_gres_data = step_gres_ptr->gres_data;
+
+       /* Such as SALLOC works, the bits will be allocated if they are free.
    Anyway,
+        * if the number of requested bits is higher than the free bits, they will
+        * be selected again, no matter if they are already allocated. */
+       i_first = 0;
+       i_last = bit_size(job_rgpu_bitstr);
+       remaining = step_gres_data->gres_cnt_alloc;
+
+       for (i_bit = i_first; (i_bit<i_last) && (remaining>0); i_bit++) {
+               if (bit_test(job_rgpu_bitstr, i_bit))
+                       continue;
+               bit_set(job_rgpu_bitstr, i_bit);
+               bit_set(step_rgpu_bitstr, i_bit);
+               remaining -= 1;
+       }
+       if (remaining>0) {
+               for (i_bit = i_first; (i_bit<i_last) && (remaining>0); i_bit++) {
+                       if (bit_test(step_rgpu_bitstr, i_bit))
+                               continue;
+                       bit_set(job_rgpu_bitstr, i_bit);
```

```
+                               bit_set(step_rgpu_bitstr, i_bit);
+                               remaining -= 1;
+                       }
+               }
+               if (remaining>0)
+                       error("gres_rgpu_step_alloc: remaining: %d > 0 in job: %u step: %u"
        , remaining, job_id, step_id);
+
+               dev_list = NULL;
+               tmp_str = xstrdup(rgpu_list);
+
+               tok = strtok(tmp_str, ",");
+               for (i_bit = 0; (i_bit<bit_size(job_rgpu_bitstr)) && (tok!=NULL); i_bit++)
        {
+                       if (bit_test(step_rgpu_bitstr, i_bit)) {
+                               if (!dev_list)
+                                       dev_list = xmalloc(2048);
+                               else
+                                       xstrcat(dev_list, ",");
+                               xstrcat(dev_list, tok);
+                       }
+                       tok = strtok(NULL, ",");
+               }
+               xstrcat(dev_list, "\0");
+               *output = xstrdup(dev_list);
+               xfree(dev_list);
+               xfree(tmp_str);
+
+               return SLURM_SUCCESS;
+}
--- slurm-2.6.2/src/common/gres.h       2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/gres.h       2014-06-25 09:59:56.033663932 +0200
@@ -53,6 +53,13 @@
        GRES_VAL_TYPE_ALLOC   = 3
 };

+/* gpu scheduling data */
+struct gpu_data {
+       uint32_t weight;                        /* weight of the scheduling */
+       uint32_t node_id;                       /* node hosting the gpu */
+       uint32_t gpu_id;                        /* id of the gpu in this node */
+};
+
 /* Gres state information gathered by slurmd daemon */
 typedef struct gres_slurmd_conf {
        /* Count of gres available in this configuration record */
@@ -71,6 +78,10 @@

        /* Gres ID number */
        uint32_t plugin_id;
+
+       /* rgpu specifications */
+       uint64_t mem;
+        uint32_t cc_version;
 } gres_slurmd_conf_t;

 /* Current gres state information managed by slurmctld daemon */
@@ -95,6 +106,11 @@
        bitstr_t **topo_gres_bitmap;
        uint32_t *topo_gres_cnt_alloc;
        uint32_t *topo_gres_cnt_avail;
+
+       /* Specific information (if gres.conf contains RGPU resources) */
+        uint64_t *mem_rgpu_alloc;              /*memory allocated in each gpu */
+        uint64_t *mem_rgpu_avail;              /*memory of each gpu */
+        uint32_t *cc_version;                  /*version of each gpu */
 } gres_node_state_t;

 /* Gres job state as used by slurmctld daemon */
@@ -111,6 +127,16 @@
```

```
	 * gres_bit_step_alloc is a subset of gres_bit_alloc */
	bitstr_t **gres_bit_step_alloc;
	uint32_t  *gres_cnt_step_alloc;
+
+	/* Quantity of memory requiered in every single rgpu */
+	uint64_t rgpu_cnt_mem;
+	/* For each rgpu in each node, this matrix will
+	 * save the quantity of memory allocated by this
+	 * job in each rgpu of the partition */
+	uint64_t ** rgpu_mem_alloc;
+	 /* Minimum version of compute capability of the rgpus required.
+	  * This value is got by: major * 1000 + minor * 10 */
+	 uint32_t com_cap;
 } gres_job_state_t;

 /* Gres job step state as used by slurmctld daemon */
@@ -127,6 +153,9 @@
	uint32_t node_cnt;
	bitstr_t *node_in_use;
	bitstr_t **gres_bit_alloc;
+
+	uint64_t rgpu_cnt_mem;
+	 uint32_t com_cap;
 } gres_step_state_t;

 /*
@@ -619,4 +648,64 @@
  */
 extern uint32_t gres_get_value_by_type(List job_gres_list, char* gres_name);

+
+/************************** RCUDA *******************************************/
+
+
+extern int gres_plugin_job_state_validate_2(char *req_config, List *gres_list,
   char **rgpu_req, char **new_req);
+
+extern int gres_rgpu_job_state_validate(char *req_config, void **gres_rgpu);
+
+extern List weight_my_gpus(int node_id, List gres_list, uint64_t job_mem);
+
+extern int gres_rgpu_job_alloc_id(struct job_resources *job, List node_gres_list,
+			      int node_offset, uint32_t job_id, char *node_name,
+			      uint32_t * rgpus_remaining, uint32_t gpu_id);
+/*
+ * Allocate rgpus to a job and update node and job rgpus information
+ * IN job_gres_status          - information about the rgpus of the job
+ * IN node_gres_list           - node's gres_list built by
+ *                   gres_plugin_node_config_validate()
+ * IN node_offset              - zero-origin index to the node of interest
+ * IN job_id                   - job's ID (for logging)
+ * IN node_name                - name of the node (for logging)
+ * IN/OUT rgpus_remaining      - number of RGPUs that haven't still been
+ *                   allocated
+ * IN node_cnt                 - total number of nodes originally allocated
+ *                   to the job
+ * RET SLURM_SUCCESS or error code
+ */
+extern int gres_rgpu_job_alloc(struct job_resources *job, List node_gres_list,
+			      int node_offset, uint32_t job_id, char *node_name,
+			      uint32_t * rgpus_remaining);
+/*
+ * Deallocate rgpus to a job and update node and job rgpus information
+ * IN job_gres_status    - information about the rgpus of the job
+ * IN node_gres_list     - node's gres_list built by
+ *          gres_plugin_node_config_validate()
+ * IN node_offset        - zero-origin index to the node of interest
+ * IN job_id             - job's ID (for logging)
+ * IN node_name          - name of the node (for logging)
+ * RET SLURM_SUCCESS or error code
```

```
+ */
+extern int gres_rgpu_job_dealloc(void * job_gres_status, List node_gres_list,
+                                 int node_offset, uint32_t job_id,
+                                 char *node_name);
+
+extern int gres_plugin_step_state_validate_2(char *req_config,
+                                             List *step_gres_list,
+                                             List job_gres_list,
+                                             uint32_t job_id, uint32_t step_id,
+                                             char **rgpu_req, char **new_req);
+
+extern int gres_rgpu_step_state_validate(char *req_config,
+                                        void ** rgpu_gres,
+                                        uint32_t job_rgpu,
+                                        uint64_t job_memrgpu);
+
+extern int gres_rgpu_step_alloc(void *step_rgpu, bitstr_t *step_rgpu_bitstr,
+                               bitstr_t *job_rgpu_bitstr,
+                               char *rgpu_list, uint32_t job_id,
+                               uint32_t step_id, char **output);
+
 #endif /* !_GRES_H */
--- slurm-2.6.2/src/common/job_resources.h      2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/job_resources.h      2014-02-26 20:07:21.145103613 +0100
@@ -124,6 +124,14 @@
        uint32_t        ncpus;
        uint32_t *      sock_core_rep_count;
        uint16_t *      sockets_per_node;
+       bitstr_t *      rgpu_node_bitmap;   //bitmap with the gpus per node
+       uint32_t        rgpus;              //number of needed gpus for the job
+       uint64_t        memrgpu;            //quantity of memory
+       void *          rgpu_job_state;     //struct gres_state
+        uint32_t *      node_offset_list;
+        uint64_t        rgpumem_min;
+        uint64_t        rgpumem_max;
+        uint32_t        rgpucc;
 };

 /*
--- slurm-2.6.2/src/common/read_config.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/read_config.c        2014-07-17 16:47:00.660862465 +0200
@@ -323,7 +323,13 @@
        {"PartitionName", S_P_ARRAY, _parse_partitionname,
         _destroy_partitionname},
        {"DownNodes", S_P_ARRAY, _parse_downnodes, _destroy_downnodes},
-
+
+       {"CudaPath", S_P_STRING},
+       {"RcudaPath", S_P_STRING},
+       {"RcudaModeDefault", S_P_STRING},
+       {"RcudaDistDefault", S_P_STRING},
+       {"RgpuMinMemory", S_P_UINT32},
+
        {NULL}
 };

@@ -2386,7 +2392,7 @@

        /* init hash to 0 */
        conf_ptr->hash_val = 0;
-       if ((_config_is_storage(conf_hashtbl, name) < 0) &&
+       if (( _config_is_storage(conf_hashtbl, name) < 0) &&
            (s_p_parse_file(conf_hashtbl, &conf_ptr->hash_val, name, false)
             == SLURM_ERROR)) {
                fatal("something wrong with opening/reading conf file");
@@ -2603,7 +2609,7 @@
        char *default_storage_loc = NULL;
        uint32_t default_storage_port = 0;
        uint16_t uint16_tmp;
-
```

```
+
        if (s_p_get_string(&conf->backup_controller, "BackupController",
                           hashtbl)
            && strcasecmp("localhost", conf->backup_controller) == 0) {
@@ -3689,7 +3695,31 @@
        if (conf->node_prefix == NULL)
                fatal("No valid node name prefix identified");
 #endif
-
+       if (s_p_get_string(&temp_str, "RcudaModeDefault", hashtbl)) {
+               if (strcmp(temp_str, "shar") == 0 || strcmp(temp_str, "shared") ==
    0)
+                               conf->rcuda_mode = 2;
+               else if (strcmp(temp_str, "excl") == 0 || strcmp(temp_str, "
    exclusive") == 0)
+                               conf->rcuda_mode = 1;
+               else
+                       fatal("Bad RcudaModeDefault: %s", temp_str);
+               xfree(temp_str);
+       } else
+               conf->rcuda_mode = DEFAULT_RCUDA_MODE;
+
+       if (s_p_get_string(&temp_str, "RcudaDistDefault", hashtbl)) {
+               if (strcmp(temp_str, "node") == 0)
+                       conf->rcuda_dist = 2;
+               else if (strcmp(temp_str, "global") == 0)
+                       conf->rcuda_dist = 1;
+               else
+                       fatal("Bad RcudaDistDefault: %s", temp_str);
+               xfree(temp_str);
+       } else
+               conf->rcuda_dist = DEFAULT_RCUDA_DISTRIBUTION;
+
+       if (!s_p_get_uint32(&conf->rgpu_min_mem, "RgpuMinMemory", hashtbl))
+               conf->rgpu_min_mem = DEFAULT_RGPU_MEMORY;
+
        xfree(default_storage_type);
        xfree(default_storage_loc);
        xfree(default_storage_host);
--- slurm-2.6.2/src/common/read_config.h        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/read_config.h        2014-07-17 16:47:00.660862465 +0200
@@ -174,6 +174,12 @@
 #define DEFAULT_UNKILLABLE_TIMEOUT   60 /* seconds */
 #define DEFAULT_MAX_TASKS_PER_NODE   128

+#define DEFAULT_RCUDA_MODE                 1       /* exclusive */
+#define DEFAULT_RCUDA_DISTRIBUTION         1       /* global */
+#define DEFAULT_RGPU_MEMORY                512     /* MB */
+#define DEFAULT_CUDA_PATH                  "/usr/local/cuda/lib64:/usr/local/cuda/lib
    "
+#define DEFAULT_RCUDA_PATH                 "rCUDA/framework/rCUDAl"
+
 typedef struct slurm_conf_frontend {
        char *allow_groups;             /* allowed group string */
        char *allow_users;              /* allowed user string */
--- slurm-2.6.2/src/common/slurm_protocol_pack.c        2013-09-10
    23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/slurm_protocol_pack.c        2014-07-17
    17:12:40.916858623 +0200
@@ -2778,6 +2778,7 @@

        select_g_select_jobinfo_pack(msg->select_jobinfo, buffer,
                                     protocol_version);
+       packstr(msg->rgpu_list, buffer);
 }

 static int
@@ -2822,6 +2823,8 @@
                if (select_g_select_jobinfo_unpack(&tmp_ptr->select_jobinfo,
                                                   buffer, protocol_version))
```

```
                         goto unpack_error;
+
+                safe_unpackstr_xmalloc(&tmp_ptr->rgpu_list, &uint32_tmp, buffer);
        } else if (protocol_version >= SLURM_2_3_PROTOCOL_VERSION) {
                safe_unpack32(&tmp_ptr->error_code, buffer);
                safe_unpack32(&tmp_ptr->job_id, buffer);
@@ -4560,7 +4563,7 @@
                safe_unpackstr_xmalloc(&job->account, &uint32_tmp, buffer);
                safe_unpackstr_xmalloc(&job->network, &uint32_tmp, buffer);
                safe_unpackstr_xmalloc(&job->comment, &uint32_tmp, buffer);
-               safe_unpackstr_xmalloc(&job->gres, &uint32_tmp, buffer);
+
                safe_unpackstr_xmalloc(&job->batch_host, &uint32_tmp, buffer);
                safe_unpackstr_xmalloc(&job->batch_script, &uint32_tmp, buffer);
                safe_unpackstr_xmalloc(&job->qos, &uint32_tmp, buffer);
@@ -4643,6 +4646,9 @@
                        job->ntasks_per_core   = mc_ptr->ntasks_per_core;
                        xfree(mc_ptr);
                }
+
+               safe_unpackstr_xmalloc(&job->gres, &uint32_tmp, buffer);
+               safe_unpackstr_xmalloc(&job->rgpu_list, &uint32_tmp, buffer);
        } else if (protocol_version >= SLURM_2_5_PROTOCOL_VERSION) {
                safe_unpack32(&job->assoc_id, buffer);
                safe_unpack32(&job->job_id, buffer);
@@ -6868,6 +6874,9 @@
                error("_pack_job_desc_msg: protocol_version "
                      "%hu not supported", protocol_version);
        }
+
+       pack16(job_desc_ptr->rcuda_mode, buffer);
+       pack16(job_desc_ptr->rcuda_dist, buffer);
 }

 /* _unpack_job_desc_msg
@@ -7301,6 +7310,9 @@
                goto unpack_error;
        }

+       safe_unpack16(&job_desc_ptr->rcuda_mode, buffer);
+       safe_unpack16(&job_desc_ptr->rcuda_dist, buffer);
+
        return SLURM_SUCCESS;

 unpack_error:
@@ -9545,6 +9557,8 @@

                select_g_select_jobinfo_pack(msg->select_jobinfo, buffer,
                                             protocol_version);
+
+               packstr(msg->rgpu_list, buffer);
        } else if (protocol_version >= SLURM_2_4_PROTOCOL_VERSION) {
                pack32(msg->job_id, buffer);
                pack32(msg->step_id, buffer);
@@ -9691,6 +9705,8 @@
                                                 select_jobinfo,
                                                 buffer, protocol_version))
                        goto unpack_error;
+
+               safe_unpackstr_xmalloc(&launch_msg_ptr->rgpu_list, &uint32_tmp,
    buffer);
        } else if (protocol_version >= SLURM_2_4_PROTOCOL_VERSION) {
                safe_unpack32(&launch_msg_ptr->job_id, buffer);
                safe_unpack32(&launch_msg_ptr->step_id, buffer);
--- slurm-2.6.2/src/common/slurm_step_layout.c  2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/slurm_step_layout.c  2014-07-10 09:01:05.030440544 +0200
@@ -275,6 +275,7 @@
                                  step_layout->tasks[i],
                                  buffer);
                }
```

```
+               packstr(step_layout->rgpu_list, buffer);
        } else {
                error("pack_slurm_step_layout: protocol_version "
                      "%hu not supported", protocol_version);
@@ -315,6 +316,7 @@
                                                    buffer);
                        step_layout->tasks[i] = num_tids;
                }
+               safe_unpackstr_xmalloc(&step_layout->rgpu_list, &uint32_tmp, buffer
    );
        } else {
                error("unpack_slurm_step_layout: protocol_version "
                      "%hu not supported", protocol_version);
--- slurm-2.6.2/src/common/slurm_protocol_defs.h        2013-09-10
    23:44:33.000000000 +0200
+++ slurm-rcuda/src/common/slurm_protocol_defs.h        2014-07-17
    08:30:06.092936822 +0200
@@ -560,6 +560,7 @@
        uint32_t slurm_rc;
        char *node_name;
        uint32_t user_id;       /* user the job runs as */
+        char *rgpu_list;
 } complete_batch_script_msg_t;

 typedef struct step_complete_msg {
@@ -727,6 +728,8 @@
        uint32_t spank_job_env_size;
        dynamic_plugin_data_t *select_jobinfo; /* select context, opaque data */
        char *alias_list;       /* node name/address/hostnamne aliases */
+
+        char *rgpu_list;
 } launch_tasks_request_msg_t;

 typedef struct task_user_managed_io_msg {
@@ -839,6 +842,8 @@
        uint16_t restart_cnt;   /* batch job restart count      */
        char **spank_job_env;   /* SPANK job environment variables */
        uint32_t spank_job_env_size;    /* size of spank_job_env */
+
+       char *rgpu_list;        /* assigned list of rgpus */
 } batch_job_launch_msg_t;

 typedef struct job_id_request_msg {
--- slurm-2.6.2/src/slurmctld/job_mgr.c 2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/job_mgr.c 2014-07-08 13:57:59.770827072 +0200
@@ -4400,10 +4400,25 @@
        /* NOTE: If this job is being used to expand another job, this job's
         * gres_list has already been filled in with a copy of gres_list job
         * to be expanded by update_job_dependency() */
+       char *new_req = NULL;
        if ((job_ptr->details->expanding_jobid == 0) &&
-           gres_plugin_job_state_validate(job_ptr->gres, &job_ptr->gres_list)){
+               gres_plugin_job_state_validate_2(job_ptr->gres,
+                       &job_ptr->gres_list, &job_ptr->gres_rgpu_str,
+                       &new_req))
+       {
                error_code = ESLURM_INVALID_GRES;
                goto cleanup_fail;
+       } else {
+               if(gres_rgpu_job_state_validate(job_ptr->gres_rgpu_str, &job_ptr->
    gres_rgpu)){
+                       info("sched: update_job: invalid gres %s for job %u",
    job_ptr->gres_rgpu_str, job_ptr->job_id);
+                       error_code = ESLURM_INVALID_GRES;
+               } else {
+                       xfree(job_ptr->gres);
+                       job_ptr->gres = new_req;
+                       job_ptr->rgpu_enable = 1;
+                       job_ptr->rcuda_mode = job_desc->rcuda_mode;
+                       job_ptr->rcuda_dist = job_desc->rcuda_dist;
```

```
+                }
         }
         gres_plugin_job_state_log(job_ptr->gres_list, job_ptr->job_id);

@@ -5918,7 +5933,7 @@
 {
         struct job_details *detail_ptr;
         time_t begin_time = 0;
-        char *nodelist = NULL;
+        char *nodelist = NULL, *greslist = NULL;
         assoc_mgr_lock_t locks = { NO_LOCK, NO_LOCK,
                                    READ_LOCK, NO_LOCK, NO_LOCK };

@@ -5987,7 +6002,7 @@
                 packstr(dump_job_ptr->account, buffer);
                 packstr(dump_job_ptr->network, buffer);
                 packstr(dump_job_ptr->comment, buffer);
-                packstr(dump_job_ptr->gres, buffer);
+
                 packstr(dump_job_ptr->batch_host, buffer);
                 if (!IS_JOB_COMPLETED(dump_job_ptr) &&
                     (show_flags & SHOW_DETAIL2) &&
@@ -6050,6 +6065,20 @@
                 else
                         _pack_pending_job_details(NULL, buffer,
                                                   protocol_version);
+
+                if (dump_job_ptr->gres){
+                        greslist = xstrdup(dump_job_ptr->gres);
+                        if (dump_job_ptr->rgpu_enable){
+                                xstrcat(greslist, ",");
+                                xstrcat(greslist, dump_job_ptr->gres_rgpu_str);
+                        }
+                        packstr(greslist, buffer);
+                }
+                else
+                        packstr(dump_job_ptr->gres_rgpu_str, buffer);
+
+                packstr(dump_job_ptr->rgpu_list, buffer);
+
         } else if (protocol_version >= SLURM_2_4_PROTOCOL_VERSION) {
                 pack32(dump_job_ptr->assoc_id, buffer);
                 pack32(dump_job_ptr->job_id, buffer);
@@ -7980,6 +8009,8 @@

         if (job_specs->gres) {
                 List tmp_gres_list = NULL;
+                void *gres_rgpu;
+                char *rgpu_req = NULL, *new_req = NULL;
                 if ((!IS_JOB_PENDING(job_ptr)) || (detail_ptr == NULL) ||
                     (detail_ptr->expanding_jobid != 0)) {
                         error_code = ESLURM_DISABLED;
@@ -7988,13 +8019,28 @@
                               job_specs->job_id);
                         xfree(job_ptr->gres);
                         FREE_NULL_LIST(job_ptr->gres_list);
-                } else if (gres_plugin_job_state_validate(job_specs->gres,
-                                                          &tmp_gres_list)) {
+                } else if (gres_plugin_job_state_validate_2(job_specs->gres,
+                                                        &tmp_gres_list, &rgpu_req,
    &new_req))
+                {
                         info("sched: update_job: invalid gres %s for job %u",
                               job_specs->gres, job_specs->job_id);
                         error_code = ESLURM_INVALID_GRES;
                         FREE_NULL_LIST(tmp_gres_list);
                 } else {
+                        if(rgpu_req){
+                                if(gres_rgpu_job_state_validate(rgpu_req, &
    gres_rgpu)){
```

```
+                                       info("sched: update_job: invalid rgpu %s
    for job %u",
+                                                job_specs->gres, job_specs->job_id
    );
+                                       error_code = ESLURM_INVALID_GRES;
+                               } else{
+                                       job_ptr->rgpu_enable = 1;
+                                       job_ptr->rgpu_list = xstrdup(rgpu_req);
+                                       job_ptr->gres_rgpu = gres_rgpu;
+                               }
+                       } else {
+                               job_ptr->rgpu_list = NULL;
+                               job_ptr->gres_rgpu = NULL;
+                       }
                        info("sched: update_job: setting gres to "
                             "%s for job_id %u",
                             job_specs->gres, job_specs->job_id);
--- slurm-2.6.2/src/slurmctld/proc_req.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/proc_req.c        2014-07-17 17:12:40.940858624 +0200
@@ -925,6 +925,13 @@
                        alloc_msg.cpu_count_reps = NULL;
                        alloc_msg.cpus_per_node  = NULL;
                }
+
+               if (job_ptr->rgpu_enable) {
+                       alloc_msg.rgpu_list = xstrdup(job_ptr->rgpu_list);
+               } else {
+                       alloc_msg.rgpu_list = NULL;
+               }
+
                alloc_msg.error_code     = error_code;
                alloc_msg.job_id         = job_ptr->job_id;
                alloc_msg.node_cnt       = job_ptr->node_cnt;
@@ -951,6 +958,8 @@
                xfree(alloc_msg.cpu_count_reps);
                xfree(alloc_msg.cpus_per_node);
                xfree(alloc_msg.node_list);
+               if (job_ptr->rgpu_list != NULL)
+                       xfree(alloc_msg.rgpu_list);
                select_g_select_jobinfo_free(alloc_msg.select_jobinfo);
                schedule_job_save();    /* has own locks */
                schedule_node_save();   /* has own locks */
@@ -2313,6 +2322,7 @@
                job_info_resp_msg.job_id         = job_info_msg->job_id;
                job_info_resp_msg.node_cnt       = job_ptr->node_cnt;
                job_info_resp_msg.node_list      = xstrdup(job_ptr->nodes);
+               job_info_resp_msg.rgpu_list      = xstrdup(job_ptr->rgpu_list);
                job_info_resp_msg.alias_list     = xstrdup(job_ptr->alias_list);
                job_info_resp_msg.select_jobinfo =
                        select_g_select_jobinfo_copy(job_ptr->select_jobinfo);
@@ -2329,6 +2339,7 @@
                xfree(job_info_resp_msg.cpu_count_reps);
                xfree(job_info_resp_msg.cpus_per_node);
                xfree(job_info_resp_msg.node_list);
+               xfree(job_info_resp_msg.rgpu_list);
        }
 }

@@ -3857,7 +3868,7 @@
         */
        job_step_create_request_msg_t req_step_msg;
        struct step_record *step_rec;
-
+
        /*
         * As far as the step record in slurmctld goes, we are just
         * launching a batch script which will be run on a single
@@ -3987,6 +3998,7 @@
        launch_msg_ptr->select_jobinfo = select_g_select_jobinfo_copy(
                job_ptr->select_jobinfo);
```

```
+        launch_msg_ptr->rgpu_list = xstrdup(job_ptr->rgpu_list);
         /* FIXME: for some reason these CPU arrays total all the CPUs
          * actually allocated, rather than totaling up to the requested
          * CPU count for the allocation.
--- slurm-2.6.2/src/slurmctld/read_config.c     2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/read_config.c     2014-02-26 20:07:21.153103612 +0100
@@ -238,6 +238,63 @@
 #endif
 }

+typedef struct gres_state {
+        uint32_t plugin_id;
+        void *gres_data;
+} gres_state_t;
+
+static void _count_rgpus(void)
+{
+        struct part_record   *part_ptr;
+        struct node_record   *node_ptr;
+        ListIterator part_iterator;
+        int i, total_rgpus;
+        ListIterator node_gres_iter;
+        gres_state_t *node_gres_ptr;
+
+        /* scan partition table and identify nodes in each */
+        part_iterator = list_iterator_create(part_list);
+        while ((part_ptr = (struct part_record *) list_next(part_iterator))) {
+                FREE_NULL_BITMAP(part_ptr->node_bitmap);
+
+                if ((part_ptr->nodes == NULL) || (part_ptr->nodes[0] == '\0')) {
+                        /* Partitions need a bitmap, even if empty */
+                        part_ptr->node_bitmap = bit_alloc(node_record_count);
+                        continue;
+                }
+
+                if (node_name2bitmap(part_ptr->nodes, false,
+                                     &part_ptr->node_bitmap)) {
+                        fatal("Invalid node names in partition %s",
+                              part_ptr->name);
+                }
+
+                total_rgpus = 0;
+                for (i=0; i<node_record_count; i++) {
+                        if (bit_test(part_ptr->node_bitmap, i) == 0)
+                                continue;
+                        node_ptr = &node_record_table_ptr[i];
+                        if (node_ptr->gres_list!=NULL) {
+                                //counting the amount of rgpus per partition
+                                node_gres_iter = list_iterator_create(node_ptr->
    gres_list);
+                                while ((node_gres_ptr = (gres_state_t *)
+                                        list_next(node_gres_iter))) {
+                                        gres_node_state_t *gres_puntero = (
    gres_node_state_t *)
+                                                node_gres_ptr->gres_data;
+                                        if (node_gres_ptr->plugin_id==1970300786) {
+                                                //debug3("read_config: rgpu: %i
    found in this node", gres_puntero->gres_cnt_avail);
+                                                total_rgpus += gres_puntero->
    gres_cnt_avail;
+                                        }
+                                }
+                                list_iterator_destroy(node_gres_iter);
+                        }
+                }
+                part_ptr->total_rgpus = total_rgpus;
+                info("read_config: total rgpu in the partition(%s): %i", part_ptr->
    name, part_ptr->total_rgpus);
+        }
```

```
+        list_iterator_destroy(part_iterator);
+        return;
+}

 /*
  * _build_bitmaps_pre_select - recover some state for jobs and nodes prior to
@@ -272,7 +329,7 @@
                                continue;
                        node_ptr = &node_record_table_ptr[i];
                        part_ptr->total_nodes++;
-                       if (slurmctld_conf.fast_schedule)
+                       if (slurmctld_conf.fast_schedule)
                                part_ptr->total_cpus +=
                                        node_ptr->config_ptr->cpus;
                        else
@@ -752,7 +809,7 @@

        /* initialization */
        START_TIMER;
-
+
        if (reconfig) {
                /* in order to re-use job state information,
                 * update nodes_completing string (based on node bitmaps) */
@@ -892,8 +949,9 @@
        }
        xfree(state_save_dir);
        _gres_reconfig(reconfig);
+       _count_rgpus();
        reset_job_bitmaps();            /* must follow select_g_job_init() */
-
+
        (void) _sync_nodes_to_jobs();
        (void) sync_job_files();
        _purge_old_node_state(old_node_table_ptr, old_node_record_count);
--- slurm-2.6.2/src/slurmctld/slurmctld.h       2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/slurmctld.h       2014-02-26 20:07:21.153103612 +0100
@@ -327,7 +327,11 @@
        uint16_t state_up;      /* See PARTITION_* states in slurm.h */
        uint32_t total_nodes;   /* total number of nodes in the partition */
        uint32_t total_cpus;    /* total number of cpus in the partition */
+
+       uint32_t total_rgpus;   /* total number of shared gpus in the partition*/
+
        uint16_t cr_type;       /* Custom CR values for partition (if supported by
            select plugin) */
+
 };

 extern List part_list;                 /* list of part_record entries */
@@ -636,6 +640,22 @@
        uint32_t wait4switch; /* Maximum time to wait for minimum switches */
        bool     best_switch; /* true=min number of switches met           */
        time_t wait4switch_start; /* Time started waiting for switch       */
+
+        /* RCUDA SPECIFIC */
+       uint16_t rgpu_enable; /* specify if mode rgpus is enabled.
+                              * 0 rgpu NOT enabled,
+                              * 1 rgpu enabled and the job has not been queued,
+                              * 2 rgpu enabled and the job has been queued */
+       char *rgpu_list; /* string with the information of the selected
+                                    * rgpus. The string is composed by pairs of
+                                    * numbers which indicate the node and de rgpu
+                                    * in the node. For example: "name:0,name:1".
   */
+       bitstr_t *rgpu_alloc_list; /* bitstr of the rgpu list, where each bit is a
   rgpu */
+       void *gres_rgpu;        /* gres_state_t of the rgpus requestes */
+       char *gres_rgpu_str;    /* String with the rgpu requested */
+        uint16_t rcuda_mode;    /* 0 for default, 1 for "exclusive", 2 for "shared
```

```
     " mode */
+          uint16_t rcuda_dist;     /* 0 for default, 1 for "global", 2 for "node"
     distribution */
+          uint32_t rgpu_min_mem;   /* MB */
 };

 /* Job dependency specification, used in "depend_list" within job_record */
@@ -704,6 +724,10 @@
        switch_jobinfo_t *switch_job;   /* switch context, opaque */
        time_t time_last_active;        /* time step was last found on node */
        time_t tot_sus_time;            /* total time in suspended state */
+
+       bitstr_t *rgpu_list_step;
+       void *gres_rgpu;                /* gres_state_t of the rgpus requestes */
+       char *gres_rgpu_str;    /* String with the rgpu requested */
 };

 extern List job_list;                  /* list of job_record entries */
--- slurm-2.6.2/src/slurmctld/srun_comm.c       2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/srun_comm.c       2014-01-20 11:46:56.828646810 +0100
@@ -93,6 +93,7 @@
                msg_arg = xmalloc(sizeof(resource_allocation_response_msg_t));
                msg_arg->job_id         = job_ptr->job_id;
                msg_arg->node_list      = xstrdup(job_ptr->nodes);
+               msg_arg->rgpu_list      = xstrdup(job_ptr->rgpu_list);
                msg_arg->alias_list     = xstrdup(job_ptr->alias_list);
                msg_arg->num_cpu_groups = job_resrcs_ptr->cpu_array_cnt;
                msg_arg->cpus_per_node  = xmalloc(sizeof(uint16_t) *
--- slurm-2.6.2/src/slurmctld/step_mgr.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/step_mgr.c        2014-01-20 11:46:56.844646815 +0100
@@ -181,6 +181,7 @@
        step_ptr->time_limit = INFINITE;
        step_ptr->jobacct    = jobacctinfo_create(NULL);
        step_ptr->requid     = -1;
+       step_ptr->rgpu_list_step = bit_alloc(job_ptr->job_resrcs->rgpus);
        (void) list_append (job_ptr->step_list, step_ptr);

        return step_ptr;
@@ -257,6 +258,9 @@
  * the switch_g_job_step_complete() must be called upon completion
  * and not upon record purging. Presently both events occur
  * simultaneously. */
+       struct job_record *job_ptr = step_ptr->job_ptr;
+       int i_first, i_last, i_bit;
+
        if (step_ptr->switch_job) {
                switch_g_job_step_complete(step_ptr->switch_job,
                                           step_ptr->step_layout->node_list);
@@ -265,6 +269,16 @@
        resv_port_free(step_ptr);
        checkpoint_free_jobinfo (step_ptr->check_job);

+       if (step_ptr->gres_rgpu_str) {
+               i_first = 0;
+               i_last = bit_size(job_ptr->rgpu_alloc_list);
+               for (i_bit = i_first; (i_bit<i_last); i_bit++)
+                       if (bit_test(step_ptr->rgpu_list_step, i_bit))
+                               bit_clear(job_ptr->rgpu_alloc_list, i_bit);
+       }
+       FREE_NULL_BITMAP(step_ptr->rgpu_list_step);
+       xfree(step_ptr->gres_rgpu_str);
+
        xfree(step_ptr->host);
        xfree(step_ptr->name);
        slurm_step_layout_destroy(step_ptr->step_layout);
@@ -1763,6 +1777,13 @@
        }
        gres_plugin_step_state_log(step_ptr->gres_list, job_ptr->job_id,
                                   step_ptr->step_id);
+
```

```
+	if (step_ptr->gres_rgpu_str) {
+		gres_rgpu_step_alloc(step_ptr->gres_rgpu, step_ptr->rgpu_list_step,
+			job_ptr->rgpu_alloc_list,
+			job_ptr->rgpu_list, job_ptr->job_id, step_ptr->step_id,
+			&step_ptr->step_layout->rgpu_list);
+	}
 }

 /* Dump a job step's CPU binding information.
@@ -2080,13 +2101,30 @@
		xfree(step_specs->gres);
	else if (step_specs->gres == NULL)
		step_specs->gres = xstrdup(job_ptr->gres);
-	i = gres_plugin_step_state_validate(step_specs->gres, &step_gres_list,
+
+	/* This function creates the gres structure of the steps.
+	 * The rgpu steps are not created here, but they are
+	 * created in step_alloc_rgpu().
+	 */
+	char *rgpu_req = NULL, *new_req = NULL;
+	void *rgpu_gres = NULL;
+	i = gres_plugin_step_state_validate_2(step_specs->gres, &step_gres_list,
					job_ptr->gres_list, job_ptr->job_id,
-					NO_VAL);
+						NO_VAL, &rgpu_req, &new_req);
+
	if (i != SLURM_SUCCESS) {
		if (step_gres_list)
			list_destroy(step_gres_list);
		return i;
+	} else {
+		if (rgpu_req) {
+			xfree(step_specs->gres);
+			step_specs->gres = new_req;
+			i = gres_rgpu_step_state_validate(rgpu_req, &rgpu_gres,
+				job_ptr->job_resrcs->rgpus, job_ptr->job_resrcs->
    memrgpu);
+			if (i!=SLURM_SUCCESS)
+				return i;
+		}
	}

	job_ptr->time_last_active = now;
@@ -2181,6 +2219,8 @@
		break;
	}

+	step_ptr->gres_rgpu_str = rgpu_req;
+	step_ptr->gres_rgpu = rgpu_gres;
	step_ptr->gres       = step_specs->gres;
	step_specs->gres     = NULL;
	step_ptr->gres_list = step_gres_list;
--- slurm-2.6.2/src/slurmctld/job_scheduler.c   2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/slurmctld/job_scheduler.c   2014-07-17 17:19:12.576857646 +0200
@@ -1189,6 +1189,7 @@
	launch_msg_ptr->ntasks = job_ptr->details->num_tasks;
	launch_msg_ptr->alias_list = xstrdup(job_ptr->alias_list);
	launch_msg_ptr->nodes = xstrdup(job_ptr->nodes);
+	launch_msg_ptr->rgpu_list = xstrdup(job_ptr->rgpu_list);
	launch_msg_ptr->overcommit = job_ptr->details->overcommit;
	launch_msg_ptr->open_mode  = job_ptr->details->open_mode;
	launch_msg_ptr->acctg_freq = xstrdup(job_ptr->details->acctg_freq);
--- slurm-2.6.2/src/srun/libsrun/allocate.c     2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/srun/libsrun/allocate.c     2014-02-26 20:07:21.153103612 +0100
@@ -770,6 +770,8 @@
		j->spank_job_env_size = opt.spank_job_env_size;
	}

+	j->rcuda_mode = opt.rcuda_mode;
+	j->rcuda_dist = opt.rcuda_dist;
```

```
        return (j);
 }

--- slurm-2.6.2/src/srun/libsrun/opt.c  2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/srun/libsrun/opt.c  2014-02-26 20:07:21.153103612 +0100
@@ -192,6 +192,9 @@
 #define LONG_OPT_LAUNCH_CMD      0x156
 #define LONG_OPT_PROFILE         0x157

+#define LONG_OPT_RCUDA_MODE     0x160
+#define LONG_OPT_RCUDA_DISTRIBUTION     0x161
+
 extern char **environ;

 /*---- global variables, defined in opt.h ----*/
@@ -917,6 +920,8 @@
                {"uid",              required_argument, 0, LONG_OPT_UID},
                {"usage",            no_argument,       0, LONG_OPT_USAGE},
                {"wckey",            required_argument, 0, LONG_OPT_WCKEY},
+               {"rcuda-mode",       required_argument, 0, LONG_OPT_RCUDA_MODE},
+               {"rcuda-distribution",       required_argument, 0,
    LONG_OPT_RCUDA_DISTRIBUTION},
                {NULL,               0,                 0, 0}
        };
        char *opt_string = "+A:B:c:C:d:D:e:Eg:hHi:I::jJ:kK::lL:m:n:N:"
@@ -937,10 +942,11 @@
        else
                error("opt.progname is already set.");
        optind = 0;
+
        while((opt_char = getopt_long(argc, argv, opt_string,
                                      optz, &option_index)) != -1) {
                switch (opt_char) {
-
+
                case (int)'?':
                        fprintf(stderr,
                                "Try \"srun --help\" for more information\n");
@@ -1561,6 +1567,28 @@
                        xfree(opt.gres);
                        opt.gres = xstrdup(optarg);
                        break;
+               case LONG_OPT_RCUDA_MODE:
+                       opt.rcuda_mode = 0;
+                       if (strcasecmp(optarg, "shar") == 0 || strcasecmp(optarg, "
    shared") == 0)
+                               opt.rcuda_mode = 2;
+                       else if (strcasecmp(optarg, "excl") == 0 || strcasecmp(
    optarg, "exclusive") == 0)
+                               opt.rcuda_mode = 1;
+                       else {
+                               info("Invalid rCUDA mode: %s (exclusive, shared)",
    optarg);
+                               exit(error_exit);
+                       }
+                       break;
+               case LONG_OPT_RCUDA_DISTRIBUTION:
+                       opt.rcuda_dist = 0;
+                       if (strcasecmp(optarg, "node") == 0)
+                               opt.rcuda_dist = 2;
+                       else if (strcasecmp(optarg, "global") == 0)
+                               opt.rcuda_dist = 1;
+                       else {
+                               error("Invalid rCUDA distribution: %s (global, node
    )", optarg);
+                               exit(error_exit);
+                       }
+                       break;
                case LONG_OPT_ALPS:
                        verbose("Not running ALPS. --alps option ignored.");
```

```
                                break;
@@ -1593,7 +1621,7 @@
        char **rest = NULL;

        _set_options(argc, argv);
-
+
        if ((opt.pn_min_memory > -1) && (opt.mem_per_cpu > -1)) {
                if (opt.pn_min_memory < opt.mem_per_cpu) {
                        info("mem < mem-per-cpu - resizing mem to be equal "
--- slurm-2.6.2/src/srun/libsrun/opt.h  2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/srun/libsrun/opt.h  2014-04-15 15:55:29.712364799 +0200
@@ -235,6 +235,11 @@
        int req_switch;         /* Minimum number of switches   */
        int wait4switch;        /* Maximum time to wait for minimum switches */
        bool user_managed_io;   /* 0 for "normal" IO, 1 for "user manged" IO */
+
+        /* RCUDA SPECIFIC */
+        uint16_t rcuda_mode;          /* 0 for default, 1 for "exclusive", 2 for "
    shared" mode */
+        uint16_t rcuda_dist;          /* 0 for default, 1 for "global", 2 for "node
    " distribution */
+        /*****************/
 } opt_t;

 extern opt_t opt;
--- slurm-2.6.2/src/srun/libsrun/srun_job.c     2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/srun/libsrun/srun_job.c     2014-01-20 11:46:57.232646809 +0100
@@ -96,6 +96,7 @@
        uint32_t                num_cpu_groups;
        dynamic_plugin_data_t  *select_jobinfo;
        uint32_t                stepid;
+        char                    *rgpulist;
 } allocation_info_t;

 static int shepard_fd = -1;
@@ -195,6 +196,7 @@
        ai->stepid          = NO_VAL;
        ai->alias_list      = resp->alias_list;
        ai->nodelist        = opt.alloc_nodelist;
+       ai->rgpulist        = resp->rgpu_list;
        hl = hostlist_create(ai->nodelist);
        hostlist_uniq(hl);
        alloc_count = hostlist_count(hl);
@@ -397,10 +399,12 @@
        i->cpus_per_node   = resp->cpus_per_node;
        i->cpu_count_reps  = resp->cpu_count_reps;
        i->select_jobinfo  = select_g_select_jobinfo_copy(resp->select_jobinfo);
-
+       i->rgpulist        = xstrdup(resp->rgpu_list);
+
        job = _job_create_structure(i);

        xfree(i->nodelist);
+       xfree(i->rgpulist);
        xfree(i);

        return (job);
--- slurm-2.6.2/src/salloc/opt.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/salloc/opt.c        2014-04-15 15:55:29.076364805 +0200
@@ -167,6 +167,9 @@
 #define LONG_OPT_REQ_SWITCH      0x143
 #define LONG_OPT_PROFILE         0x144

+#define LONG_OPT_RCUDA_MODE     0x160
+#define LONG_OPT_RCUDA_DISTRIBUTION     0x161
+
 /*---- global variables, defined in opt.h ----*/
 opt_t opt;
 int error_exit = 1;
```

```
@@ -688,6 +691,8 @@
                {"wait-all-nodes",required_argument, 0, LONG_OPT_WAIT_ALL_NODES},
                {"wckey",         required_argument, 0, LONG_OPT_WCKEY},
                {"switches",      required_argument, 0, LONG_OPT_REQ_SWITCH},
+               {"rcuda-mode",       required_argument, 0, LONG_OPT_RCUDA_MODE},
+               {"rcuda-distribution",       required_argument, 0,
    LONG_OPT_RCUDA_DISTRIBUTION},
                {NULL,            0,                 0, 0}
        };
        char *opt_string =
@@ -1159,6 +1164,28 @@
                        xfree(opt.gres);
                        opt.gres = xstrdup(optarg);
                        break;
+               case LONG_OPT_RCUDA_MODE:
+                       opt.rcuda_mode = 0;
+                       if (strcasecmp(optarg, "shar") == 0 || strcasecmp(optarg, "
    shared") == 0)
+                               opt.rcuda_mode = 2;
+                       else if (strcasecmp(optarg, "excl") == 0 || strcasecmp(
    optarg, "exclusive") == 0)
+                               opt.rcuda_mode = 1;
+                       else {
+                               info("Invalid rCUDA mode: %s (exclusive, shared)",
    optarg);
+                               exit(error_exit);
+                       }
+                       break;
+               case LONG_OPT_RCUDA_DISTRIBUTION:
+                       opt.rcuda_dist = 0;
+                       if (strcasecmp(optarg, "node") == 0)
+                               opt.rcuda_dist = 2;
+                       else if (strcasecmp(optarg, "global") == 0)
+                               opt.rcuda_dist = 1;
+                       else {
+                               error("Invalid rCUDA distribution: %s (global, node
    )", optarg);
+                               exit(error_exit);
+                       }
+                       break;
                case LONG_OPT_WAIT_ALL_NODES:
                        opt.wait_all_nodes = strtol(optarg, NULL, 10);
                        break;
--- slurm-2.6.2/src/salloc/opt.h        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/salloc/opt.h        2014-04-15 15:55:29.192364802 +0200
@@ -162,6 +162,10 @@
        char **spank_job_env;   /* SPANK controlled environment for job
                                 * Prolog and Epilog             */
        int spank_job_env_size; /* size of spank_job_env        */
+
+        /* RCUDA SPECIFIC */
+        uint16_t rcuda_mode;         /* 0 for default, 1 for "exclusive", 2 for "
    shared" mode */
+        uint16_t rcuda_dist;         /* 0 for default, 1 for "global", 2 for "node
    " distribution */
 } opt_t;

 extern opt_t opt;
--- slurm-2.6.2/src/salloc/salloc.c     2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/salloc/salloc.c     2014-04-15 15:55:29.412364802 +0200
@@ -748,7 +748,10 @@
                desc->spank_job_env      = opt.spank_job_env;
                desc->spank_job_env_size = opt.spank_job_env_size;
        }
-
+
+       desc->rcuda_mode = opt.rcuda_mode;
+       desc->rcuda_dist = opt.rcuda_dist;
+
        return 0;
```

```
 }

--- slurm-2.6.2/src/sbatch/opt.c        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/sbatch/opt.c        2014-04-15 15:55:29.504364803 +0200
@@ -176,6 +176,9 @@
 #define LONG_OPT_PROFILE         0x154
 #define LONG_OPT_IGNORE_PBS      0x155

+#define LONG_OPT_RCUDA_MODE     0x160
+#define LONG_OPT_RCUDA_DISTRIBUTION     0x161
+
 /*---- global variables, defined in opt.h ----*/
 opt_t opt;
 int error_exit = 1;
@@ -729,6 +732,8 @@
        {"wrap",          required_argument, 0, LONG_OPT_WRAP},
        {"switches",      required_argument, 0, LONG_OPT_REQ_SWITCH},
        {"ignore-pbs",    no_argument,       0, LONG_OPT_IGNORE_PBS},
+       {"rcuda-mode",       required_argument, 0, LONG_OPT_RCUDA_MODE},
+       {"rcuda-distribution",       required_argument, 0,
    LONG_OPT_RCUDA_DISTRIBUTION},
        {NULL,            0,                 0, 0}
 };

@@ -1624,6 +1629,28 @@
                        xfree(opt.gres);
                        opt.gres = xstrdup(optarg);
                        break;
+               case LONG_OPT_RCUDA_MODE:
+                       opt.rcuda_mode = 0;
+                       if (strcasecmp(optarg, "shar") == 0 || strcasecmp(optarg, "
    shared") == 0)
+                               opt.rcuda_mode = 2;
+                       else if (strcasecmp(optarg, "excl") == 0 || strcasecmp(
    optarg, "exclusive") == 0)
+                               opt.rcuda_mode = 1;
+                       else {
+                               info("Invalid rCUDA mode: %s (exclusive, shared)",
    optarg);
+                               exit(error_exit);
+                       }
+                       break;
+               case LONG_OPT_RCUDA_DISTRIBUTION:
+                       opt.rcuda_dist = 0;
+                       if (strcasecmp(optarg, "node") == 0)
+                               opt.rcuda_dist = 2;
+                       else if (strcasecmp(optarg, "global") == 0)
+                               opt.rcuda_dist = 1;
+                       else {
+                               error("Invalid rCUDA distribution: %s (global, node
    )", optarg);
+                               exit(error_exit);
+                       }
+                       break;
                case LONG_OPT_WAIT_ALL_NODES:
                        opt.wait_all_nodes = strtol(optarg, NULL, 10);
                        break;
--- slurm-2.6.2/src/sbatch/opt.h        2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/sbatch/opt.h        2014-04-15 15:55:29.648364803 +0200
@@ -177,6 +177,11 @@
        char **spank_job_env;   /* SPANK controlled environment for job
                                 * Prolog and Epilog             */
        int spank_job_env_size; /* size of spank_job_env         */
+
+        /* RCUDA SPECIFIC */
+        uint16_t rcuda_mode;         /* 0 for default, 1 for "exclusive", 2 for "
    shared" mode */
+        uint16_t rcuda_dist;         /* 0 for default, 1 for "global", 2 for "node
    " distribution */
+        /*****************/
```

```
 } opt_t;

 extern opt_t opt;
--- slurm-2.6.2/src/sbatch/sbatch.c     2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/sbatch/sbatch.c     2014-07-17 16:47:00.660862465 +0200
@@ -461,6 +461,8 @@
        if (opt.wait4switch >= 0)
                desc->wait4switch = opt.wait4switch;

+       desc->rcuda_mode = opt.rcuda_mode;
+       desc->rcuda_dist = opt.rcuda_dist;

        return 0;
 }
--- slurm-2.6.2/src/squeue/print.c      2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/squeue/print.c      2014-06-26 11:37:17.929433883 +0200
@@ -782,6 +781,18 @@
        if (suffix)
                printf("%s", suffix);
        return SLURM_SUCCESS;
+}
+
+int _print_job_rgpu_list(job_info_t * job, int width, bool right,
+               char* suffix)
+{
+       if (job == NULL)        /* Print the Header instead */
+               _print_str("RGPULIST", width, right, false);
+       else
+               _print_str(job->rgpu_list, width, right, false);
+       if (suffix)
+               printf("%s", suffix);
+       return SLURM_SUCCESS;
 }

 int _print_job_node_inx(job_info_t * job, int width, bool right, char* suffix)
--- slurm-2.6.2/src/squeue/print.h      2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/squeue/print.h      2014-06-26 11:37:17.929433883 +0200
@@ -98,6 +98,8 @@
         job_format_add_function(list,wid,right,prefix,_print_job_reason)
 #define job_format_add_reason_list(list,wid,right,prefix) \
        job_format_add_function(list,wid,right,prefix,_print_job_reason_list)
+#define job_format_add_rgpu_list(list,wid,right,prefix) \
+       job_format_add_function(list,wid,right,prefix,_print_job_rgpu_list)
 #define job_format_add_name(list,wid,right,suffix) \
        job_format_add_function(list,wid,right,suffix,_print_job_name)
 #define job_format_add_licenses(list,wid,right,suffix) \
@@ -205,6 +207,8 @@
                        char* suffix);
 int _print_job_reason_list(job_info_t * job, int width, bool right_justify,
                        char* suffix);
+int _print_job_rgpu_list(job_info_t * job, int width, bool right_justify,
+                       char* suffix);
 int _print_job_name(job_info_t * job, int width, bool right_justify,
                        char* suffix);
 int _print_job_licenses(job_info_t * job, int width, bool right_justify,
--- slurm-2.6.2/src/squeue/opts.c       2013-09-10 23:44:33.000000000 +0200
+++ slurm-rcuda/src/squeue/opts.c       2014-06-26 11:37:17.929433883 +0200
@@ -783,6 +783,11 @@
                                                        field_size,
                                                        right_justify,
                                                        suffix );
+                       else if (field[0] == 'Z')
+                               job_format_add_rgpu_list( params.format_list,
+                                                       field_size,
+                                                       right_justify,
+                                                       suffix );
                        else {
                                prefix = xstrdup("%");
                                xstrcat(prefix, token);
```

```
--- slurm-2.6.2/src/plugins/select/cons_res/select_cons_res.c   2013-09-10
    23:44:33.000000000 +0200
+++ slurm-rcuda/src/plugins/select/cons_rgpu/select_cons_rgpu.c 2014-06-25
    09:01:28.341672679 +0200
@@ -124,8 +124,6 @@
 int switch_record_cnt __attribute__((weak_import));
 bitstr_t *avail_node_bitmap __attribute__((weak_import));
 bitstr_t *idle_node_bitmap __attribute__((weak_import));
-uint16_t *cr_node_num_cores __attribute__((weak_import));
-uint32_t *cr_node_cores_offset __attribute__((weak_import));
 #else
 slurm_ctl_conf_t slurmctld_conf;
 struct node_record *node_record_table_ptr;
@@ -137,8 +135,6 @@
 int switch_record_cnt;
 bitstr_t *avail_node_bitmap;
 bitstr_t *idle_node_bitmap;
-uint16_t *cr_node_num_cores;
-uint32_t *cr_node_cores_offset;
 #endif

 /*
@@ -169,9 +165,9 @@
  * as 100 or 1000.  Various SLURM versions will likely require a certain
  * minimum version for their plugins as the node selection API matures.
  */
-const char plugin_name[] = "Consumable Resources (CR) Node Selection plugin";
-const char plugin_type[] = "select/cons_res";
-const uint32_t plugin_id      = 101;
+const char plugin_name[] = "Consumable Resources RGPU Node Selection plugin";
+const char plugin_type[] = "select/cons_rgpu";
+const uint32_t plugin_id = 150;
 const uint32_t plugin_version = 100;
 const uint32_t pstate_version = 7;     /* version control on saved state */

@@ -180,6 +176,8 @@
 uint32_t select_debug_flags;
 uint16_t select_fast_schedule;

+uint16_t *cr_node_num_cores = NULL;
+uint32_t *cr_node_cores_offset = NULL;
 struct part_res_record *select_part_record = NULL;
 struct node_res_record *select_node_record = NULL;
 struct node_use_record *select_node_usage  = NULL;
@@ -192,9 +190,13 @@
 struct select_nodeinfo {
        uint16_t magic;         /* magic number */
        uint16_t alloc_cpus;
-       uint32_t alloc_memory;
 };

+typedef struct gres_state {
+       uint32_t plugin_id;
+       void *gres_data;
+} gres_state_t;
+
 extern select_nodeinfo_t *select_p_select_nodeinfo_alloc(void);
 extern int select_p_select_nodeinfo_free(select_nodeinfo_t *nodeinfo);

@@ -210,8 +212,7 @@
 static int _run_now(struct job_record *job_ptr, bitstr_t *bitmap,
                    uint32_t min_nodes, uint32_t max_nodes,
                    uint32_t req_nodes, uint16_t job_node_req,
-                   List preemptee_candidates, List *preemptee_job_list,
-                   bitstr_t *exc_core_bitmap);
+                   List preemptee_candidates, List *preemptee_job_list);
 static int _sort_usable_nodes_dec(struct job_record *job_a,
                                   struct job_record *job_b);
 static int _test_only(struct job_record *job_ptr, bitstr_t *bitmap,
@@ -220,10 +221,10 @@
```

```
 static int _will_run_test(struct job_record *job_ptr, bitstr_t *bitmap,
                           uint32_t min_nodes, uint32_t max_nodes,
                           uint32_t req_nodes, uint16_t job_node_req,
-                          List preemptee_candidates, List *preemptee_job_list,
-                          bitstr_t *exc_core_bitmap);
+                          List preemptee_candidates, List *preemptee_job_list);

-static void _dump_job_res(struct job_resources *job) {
+static void _dump_job_res(struct job_resources *job)
+{
        char str[64];

        if (job->core_bitmap)
@@ -291,6 +292,50 @@
        return;
 }

+/* (re)set cr_node_num_cores arrays */
+static void _init_global_core_data(struct node_record *node_ptr, int node_cnt)
+{
+       uint32_t n;
+
+       xfree(cr_node_num_cores);
+       cr_node_num_cores = xmalloc(node_cnt*sizeof(uint16_t));
+
+       xfree(cr_node_cores_offset);
+       cr_node_cores_offset = xmalloc((node_cnt+1)*sizeof(uint32_t));
+
+       for (n = 0; n<node_cnt; n++) {
+               uint16_t cores;
+               if (select_fast_schedule) {
+                       cores = node_ptr[n].config_ptr->cores;
+                       cores *= node_ptr[n].config_ptr->sockets;
+               } else {
+                       cores = node_ptr[n].cores;
+                       cores *= node_ptr[n].sockets;
+               }
+               cr_node_num_cores[n] = cores;
+               if (n>0) {
+                       cr_node_cores_offset[n] = cr_node_cores_offset[n-1]+
+                               cr_node_num_cores[n-1];
+               } else
+                       cr_node_cores_offset[0] = 0;
+       }
+
+       /* an extra value is added to get the total number of cores */
+       /* as cr_get_coremap_offset is sometimes used to get the total */
+       /* number of cores in the cluster */
+       cr_node_cores_offset[node_cnt] = cr_node_cores_offset[node_cnt-1]+
+               cr_node_num_cores[node_cnt-1];
+
+}
+
+
+/* return the coremap index to the first core of the given node */
+extern uint32_t cr_get_coremap_offset(uint32_t node_index)
+{
+       return cr_node_cores_offset[node_index];
+}
+
+
 /* Helper function for _dup_part_data: create a duplicate part_row_data array */
 static struct part_row_data *_dup_row_data(struct part_row_data *orig_row,
                                            uint16_t num_rows)
@@ -373,7 +418,8 @@
 }

 /* delete the given row data */
-static void _destroy_row_data(struct part_row_data *row, uint16_t num_rows) {
+static void _destroy_row_data(struct part_row_data *row, uint16_t num_rows)
```

```
+{
        uint16_t i;
        for (i = 0; i < num_rows; i++) {
                FREE_NULL_BITMAP(row[i].row_bitmap);
@@ -424,6 +470,9 @@
        this_ptr = select_part_record;

        part_iterator = list_iterator_create(part_list);
+       if (part_iterator==NULL)
+               fatal("memory allocation failure");
+
        while ((p_ptr = (struct part_record *) list_next(part_iterator))) {
                this_ptr->part_ptr = p_ptr;
                this_ptr->num_rows = p_ptr->max_share;
@@ -451,7 +500,7 @@
 {
        struct job_record *job1_ptr = (struct job_record *) x;
        struct job_record *job2_ptr = (struct job_record *) y;
-       return (int) SLURM_DIFFTIME(job1_ptr->end_time, job2_ptr->end_time);
+       return(int) difftime(job1_ptr->end_time, job2_ptr->end_time);
 }


@@ -582,21 +631,12 @@
                                bit_nclear(this_row->row_bitmap, 0, size-1);
                        }
                } else {
-                       if (job_ptr) { /* just remove the job */
+                       xassert(job_ptr);
                                xassert(job_ptr->job_resrcs);
                                remove_job_from_cores(job_ptr->job_resrcs,
-                                                     &(this_row->row_bitmap),
-                                                     cr_node_num_cores);
-                       } else { /* totally rebuild the bitmap */
-                               size = bit_size(this_row->row_bitmap);
-                               bit_nclear(this_row->row_bitmap, 0, size-1);
-                               for (j = 0; j < this_row->num_jobs; j++) {
-                                       add_job_to_cores(this_row->job_list[j],
-                                                        &(this_row->row_bitmap),
+                               &this_row->row_bitmap,
                                                        cr_node_num_cores);
                                }
-                       }
-               }
                return;
        }

@@ -784,6 +824,37 @@
         */
 }

+static int _call_alloc_rgpu(int *n, int i, int *offset_aux, uint32_t *remaining,
+                              int action, struct job_record *job_ptr)
+{
+       struct job_resources *job = job_ptr->job_resrcs;
+       struct node_record *node_ptr;
+       List gres_list;
+
+       *n += 1;
+       offset_aux[*n] = i;
+       job->node_offset_list[i] = *n;
+
+       if (*remaining>0) {
+               node_ptr = select_node_record[i].node_ptr;
+               if (action!=2) {
+                       if (select_node_usage[i].gres_list)
+                               gres_list = select_node_usage[i].gres_list;
+                       else
+                               gres_list = node_ptr->gres_list;
+                       //allocating gpus
```

```
+                       gres_rgpu_job_alloc(job, gres_list, *n, job_ptr->job_id,
+                               node_ptr->name, remaining);
+                       gres_plugin_node_state_log(gres_list, node_ptr->name);
+               } //if action
+       }// if remaining
+       else {
+               //clearing this node from the bitmap since it won't be used
+               bit_clear(job->rgpu_node_bitmap, i);
+       }
+
+       return SLURM_SUCCESS;
+}

 /* allocate resources to the given job
  * - add 'struct job_resources' resources to 'struct part_res_record'
@@ -799,9 +870,18 @@
        struct node_record *node_ptr;
        struct part_res_record *p_ptr;
        List gres_list;
-       int i, n;
+
        bitstr_t *core_bitmap;

+       int i, n, j;
+       gres_state_t *job_gres_ptr;
+       gres_job_state_t *job_gres_data;
+       char *dev_list = NULL;
+
+       int offset_aux[select_node_cnt];
+       job->node_offset_list = malloc(select_node_cnt * sizeof(int));
+
+

        if (!job || !job->core_bitmap) {
                error("job %u has no select data", job_ptr->job_id);
                return SLURM_ERROR;
@@ -813,12 +893,62 @@
        if (select_debug_flags & DEBUG_FLAG_CPU_BIND)
                _dump_job_res(job);

+       if (job->rgpu_node_bitmap!=NULL) {
+               debug("cons_rgpu: _add_job_to_res: rgpus needed: %u", job->rgpus);
+               uint32_t remaining = job->rgpus;
+
+               /*
+                * First, we try to allocate rgpus from the selected nodes
+                */
+               n = -1;
+               for (i = 0; i<select_node_cnt; i++) {
+                       if (!bit_test(job_ptr->node_bitmap, i) || !bit_test(job->
    rgpu_node_bitmap, i))
+                               continue;
+                       _call_alloc_rgpu(&n, i, offset_aux, &remaining, action,
    job_ptr);
+               }
+
+               /*
+                * Then, if the job needs more rgpus, we will find them through
+                * the rest of the nodes.
+                */
+               for (i = 0; i<select_node_cnt; i++) { //iterating the nodes
+                       if (!bit_test(job->rgpu_node_bitmap, i) || bit_test(job_ptr
    ->node_bitmap, i))
+                               continue;
+                       _call_alloc_rgpu(&n, i, offset_aux, &remaining, action,
    job_ptr);
+               }
+
+               //collecting information about allocated gpus
+               job_gres_ptr = (gres_state_t *) job->rgpu_job_state;
+               job_gres_data = (gres_job_state_t *) job_gres_ptr->gres_data;
```

```
+               for (i = 0, n = -1; i<select_node_cnt; i++) { //recorre los nodos
+                       if (!bit_test(job->rgpu_node_bitmap, i))
+                               continue;
+                       n++;
+                       if (job_gres_data->gres_bit_alloc[n]!=NULL) {
+                               for (j = 0; j<bit_size(job_gres_data->
    gres_bit_alloc[n]); j++) { //recorre las gpus
+                                       if (bit_test(job_gres_data->gres_bit_alloc[
    n], j)) {
+                                               if (!dev_list)
+                                                       dev_list = xmalloc(128);
+                                               else
+                                                       xstrcat(dev_list, ",");
+                                               node_ptr = select_node_record[
    offset_aux[n]].node_ptr;
+                                               xstrfmtcat(dev_list, "%s:%i",
    node_ptr->name, j);
+                                       }
+                               } //for gpus
+                       }// if not null
+               } //for nodes
+               xstrcat(dev_list, "\0");
+               job_ptr->rgpu_list = xstrdup(dev_list);
+               job_ptr->rgpu_alloc_list = bit_alloc(job->rgpus);
+               debug("cons_rgpu: _add_job_to_res: job id: %u - number gpus: %u -
    rgpulist: %s - nodelist: %s",
+                       job_ptr->job_id, job->rgpus, job_ptr->rgpu_list, job_ptr->
    nodes);
+               xfree(dev_list);
+       } //if rgpus mode
+
        for (i = 0, n = -1; i < select_node_cnt; i++) {
                if (!bit_test(job->node_bitmap, i))
                        continue;
                n++;
-               if (job->cpus[n] == 0)
-                       continue;  /* node lost by job resize */

                node_ptr = select_node_record[i].node_ptr;
                if (action != 2) {
@@ -877,25 +1007,18 @@
                        break;
                }
                if (i >= p_ptr->num_rows) {
-                       /* Job started or resumed and it's allocated resources
-                        * are already in use by some other job. Typically due
-                        * to manually resuming a job. */
-                       error("cons_res: ERROR: job overflow: "
-                             "could not find idle resources for job %u",
-                             job_ptr->job_id);
+                       /* ERROR: could not find a row for this job */
+                       error("cons_rgpu: ERROR: job overflow: "
+                               "could not find row for job");
                        /* just add the job to the last row for now */
                        _add_job_to_row(job, &(p_ptr->row[p_ptr->num_rows-1]));
                }
                /* update the node state */
-               for (i = 0, n = -1; i < select_node_cnt; i++) {
-                       if (bit_test(job->node_bitmap, i)) {
-                               n++;
-                               if (job->cpus[n] == 0)
-                                       continue;  /* node lost by job resize */
+               for (i = 0; i<select_node_cnt; i++) {
+                       if (bit_test(job->node_bitmap, i))
                                select_node_usage[i].node_state +=
                                        job->node_req;
                        }
-               }
                if (select_debug_flags & DEBUG_FLAG_CPU_BIND) {
                        info("DEBUG: _add_job_to_res (after):");
```

```
                    _dump_part(p_ptr);
@@ -970,8 +1093,12 @@
        }

        tmp_bitmap = bit_copy(to_job_resrcs_ptr->node_bitmap);
+       if (!tmp_bitmap)
+               fatal("bit_copy: malloc failure");
        bit_or(tmp_bitmap, from_job_resrcs_ptr->node_bitmap);
        tmp_bitmap2 = bit_copy(to_job_ptr->node_bitmap);
+       if (!tmp_bitmap)
+               fatal("bit_copy: malloc failure");
        bit_or(tmp_bitmap2, from_job_ptr->node_bitmap);
        bit_and(tmp_bitmap, tmp_bitmap2);
        bit_free(tmp_bitmap2);
@@ -1135,6 +1262,7 @@
                        struct node_use_record *node_usage,
                        struct job_record *job_ptr, int action)
 {
+       //debug("RCUDA_DEALLOC job:%i (pid:%d - thread:%u): %s(%s,%d)", job_ptr->
    job_id, getpid(), (unsigned int)pthread_self(), __FILE__, __func__, __LINE__);
        struct job_resources *job = job_ptr->job_resrcs;
        struct node_record *node_ptr;
        int first_bit, last_bit;
@@ -1161,12 +1289,31 @@
                last_bit = -2;
        else
                last_bit =  bit_fls(job->node_bitmap);
+
+       if (job->rgpu_node_bitmap!=NULL) { //rgpu mode
+               debug("cons_rgpu: _rm_job_from_res: deallocating: %u rgpus within %
    u nodes",
+                       job->rgpus, bit_set_count(job->rgpu_node_bitmap));
+               for (i=0; i<select_node_cnt; i++){
+                       if (!bit_test(job->rgpu_node_bitmap, i))
+                               continue;
+                       n = job->node_offset_list[i];
+                       node_ptr = node_record_table_ptr+i;
+                       if (action!=2) {
+                               if (node_usage[i].gres_list)
+                                       gres_list = node_usage[i].gres_list;
+                               else
+                                       gres_list = node_ptr->gres_list;
+                               gres_rgpu_job_dealloc(job->rgpu_job_state,
    gres_list,
+                                       n, job_ptr->job_id, node_ptr->name);
+                               //gres_plugin_node_state_log(gres_list, node_ptr->
    name);
+                       }
+               }
+       }//end if rgpu mode
+
        for (i = first_bit, n = -1; i <= last_bit; i++) {
                if (!bit_test(job->node_bitmap, i))
                        continue;
                n++;
-               if (job->cpus[n] == 0)
-                       continue;  /* node lost by job resize */

                node_ptr = node_record_table_ptr + i;
                if (action != 2) {
@@ -1386,8 +1534,8 @@
        }


-       /* some node of job removed from core-bitmap, so refresh CR bitmaps */
-       _build_row_bitmaps(p_ptr, NULL);
+       /* job was found and removed from core-bitmap, so refresh CR bitmaps */
+       _build_row_bitmaps(p_ptr, job_ptr);

        /* Adjust the node_state of the node removed from this job.
```

```
         * If all cores are now available, set node_state = NODE_CR_AVAILABLE */
@@ -1467,24 +1615,10 @@
                          uint32_t req_nodes, uint16_t job_node_req)
 {
        int rc;
-       uint16_t tmp_cr_type = cr_type;
-
-       if (job_ptr->part_ptr->cr_type) {
-               if (((cr_type & CR_SOCKET) || (cr_type & CR_CORE)) &&
-                   (cr_type & CR_ALLOCATE_FULL_SOCKET)) {
-                       tmp_cr_type &= ~(CR_SOCKET|CR_CORE);
-                       tmp_cr_type |= job_ptr->part_ptr->cr_type;
-               } else {
-                       info("cons_res: Can't use Partition SelectType unless "
-                            "using CR_Socket or CR_Core and "
-                            "CR_ALLOCATE_FULL_SOCKET");
-               }
-       }
-
        rc = cr_job_test(job_ptr, bitmap, min_nodes, max_nodes, req_nodes,
-                        SELECT_MODE_TEST_ONLY, tmp_cr_type, job_node_req,
+               SELECT_MODE_TEST_ONLY, cr_type, job_node_req,
                         select_node_cnt, select_part_record,
-                        select_node_usage, NULL);
+               select_node_usage);
        return rc;
 }

@@ -1507,8 +1641,7 @@
 static int _run_now(struct job_record *job_ptr, bitstr_t *bitmap,
                     uint32_t min_nodes, uint32_t max_nodes,
                     uint32_t req_nodes, uint16_t job_node_req,
-                    List preemptee_candidates, List *preemptee_job_list,
-                    bitstr_t *exc_core_bitmap)
+                    List preemptee_candidates, List *preemptee_job_list)
 {
        int rc;
        bitstr_t *orig_map = NULL, *save_bitmap;
@@ -1519,27 +1652,16 @@
        bool remove_some_jobs = false;
        uint16_t pass_count = 0;
        uint16_t mode;
-       uint16_t tmp_cr_type = cr_type;

        save_bitmap = bit_copy(bitmap);
 top:   orig_map = bit_copy(save_bitmap);
-
-       if (job_ptr->part_ptr->cr_type) {
-               if (((cr_type & CR_SOCKET) || (cr_type & CR_CORE)) &&
-                   (cr_type & CR_ALLOCATE_FULL_SOCKET)) {
-                       tmp_cr_type &= ~(CR_SOCKET|CR_CORE);
-                       tmp_cr_type |= job_ptr->part_ptr->cr_type;
-               } else {
-                       info("cons_res: Can't use Partition SelectType unless "
-                            "using CR_Socket or CR_Core and "
-                            "CR_ALLOCATE_FULL_SOCKET");
-               }
-       }
+       if (!orig_map)
+               fatal("bit_copy: malloc failure");

        rc = cr_job_test(job_ptr, bitmap, min_nodes, max_nodes, req_nodes,
-                        SELECT_MODE_RUN_NOW, tmp_cr_type, job_node_req,
+               SELECT_MODE_RUN_NOW, cr_type, job_node_req,
                         select_node_cnt, select_part_record,
-                        select_node_usage, exc_core_bitmap);
+               select_node_usage);

        if ((rc != SLURM_SUCCESS) && preemptee_candidates) {
               /* Remove preemptable jobs from simulated environment */
```

```
@@ -1558,6 +1680,8 @@
                }

                job_iterator = list_iterator_create(preemptee_candidates);
+               if (job_iterator==NULL)
+                       fatal("memory allocation failure");
                while ((tmp_job_ptr = (struct job_record *)
                        list_next(job_iterator))) {
                        if (!IS_JOB_RUNNING(tmp_job_ptr) &&
@@ -1575,10 +1699,9 @@
                        rc = cr_job_test(job_ptr, bitmap, min_nodes,
                                        max_nodes, req_nodes,
                                        SELECT_MODE_WILL_RUN,
-                                       tmp_cr_type, job_node_req,
+                               cr_type, job_node_req,
                                        select_node_cnt,
-                                       future_part, future_usage,
-                                       exc_core_bitmap);
+                               future_part, future_usage);
                        tmp_job_ptr->details->usable_nodes = 0;
                        /*
                         * If successful, set the last job's usable count to a
@@ -1622,9 +1745,13 @@
                         * actually used */
                        if (*preemptee_job_list == NULL) {
                                *preemptee_job_list = list_create(NULL);
+                               if (*preemptee_job_list==NULL)
+                                       fatal("list_create malloc failure");
                        }
                        preemptee_iterator = list_iterator_create(
                                preemptee_candidates);
+                       if (preemptee_iterator==NULL)
+                               fatal("memory allocation failure");
                        while ((tmp_job_ptr = (struct job_record *)
                                list_next(preemptee_iterator))) {
                                mode = slurm_job_preempt_mode(tmp_job_ptr);
@@ -1660,8 +1787,7 @@
 static int _will_run_test(struct job_record *job_ptr, bitstr_t *bitmap,
                         uint32_t min_nodes, uint32_t max_nodes,
                         uint32_t req_nodes, uint16_t job_node_req,
-                        List preemptee_candidates, List *preemptee_job_list,
-                        bitstr_t *exc_core_bitmap)
+                        List preemptee_candidates, List *preemptee_job_list)
 {
        struct part_res_record *future_part;
        struct node_use_record *future_usage;
@@ -1671,27 +1797,16 @@
        bitstr_t *orig_map;
        int action, rc = SLURM_ERROR;
        time_t now = time(NULL);
-       uint16_t tmp_cr_type = cr_type;

        orig_map = bit_copy(bitmap);
-
-       if (job_ptr->part_ptr->cr_type) {
-               if (((cr_type & CR_SOCKET) || (cr_type & CR_CORE)) &&
-                   (cr_type & CR_ALLOCATE_FULL_SOCKET)) {
-                       tmp_cr_type &= ~(CR_SOCKET|CR_CORE);
-                       tmp_cr_type |= job_ptr->part_ptr->cr_type;
-               } else {
-                       info("cons_res: Can't use Partition SelectType unless "
-                            "using CR_Socket or CR_Core and "
-                            "CR_ALLOCATE_FULL_SOCKET");
-               }
-       }
+       if (!orig_map)
+               fatal("bit_copy: malloc failure");

        /* Try to run with currently available nodes */
        rc = cr_job_test(job_ptr, bitmap, min_nodes, max_nodes, req_nodes,
```

```
-                          SELECT_MODE_WILL_RUN, tmp_cr_type, job_node_req,
+                 SELECT_MODE_WILL_RUN, cr_type, job_node_req,
                           select_node_cnt, select_part_record,
-                          select_node_usage, exc_core_bitmap);
+                 select_node_usage);
         if (rc == SLURM_SUCCESS) {
                 FREE_NULL_BITMAP(orig_map);
                 job_ptr->start_time = time(NULL);
@@ -1717,6 +1832,8 @@
         if (!cr_job_list)
                 fatal("list_create: memory allocation error");
         job_iterator = list_iterator_create(job_list);
+        if (job_iterator==NULL)
+                fatal("memory allocation failure");
         while ((tmp_job_ptr = (struct job_record *) list_next(job_iterator))) {
                 if (!IS_JOB_RUNNING(tmp_job_ptr) &&
                     !IS_JOB_SUSPENDED(tmp_job_ptr))
@@ -1745,15 +1862,11 @@
         if (preemptee_candidates) {
                 bit_or(bitmap, orig_map);
                 rc = cr_job_test(job_ptr, bitmap, min_nodes, max_nodes,
-                                 req_nodes, SELECT_MODE_WILL_RUN, tmp_cr_type,
+                        req_nodes, SELECT_MODE_WILL_RUN, cr_type,
                                  job_node_req, select_node_cnt, future_part,
-                                 future_usage, exc_core_bitmap);
-                if (rc == SLURM_SUCCESS) {
-                        /* Actual start time will actually be later than "now",
-                         * but return "now" for backfill scheduler to
-                         * initiate preemption. */
-                        job_ptr->start_time = now;
-                }
+                        future_usage);
+                if (rc==SLURM_SUCCESS)
+                        job_ptr->start_time = now+1;
         }

         /* Remove the running jobs one at a time from exp_node_cr and try
@@ -1761,6 +1874,8 @@
         if (rc != SLURM_SUCCESS) {
                 list_sort(cr_job_list, _cr_job_list_sort);
                 job_iterator = list_iterator_create(cr_job_list);
+                if (job_iterator==NULL)
+                        fatal("memory allocation failure");
                 while ((tmp_job_ptr = list_next(job_iterator))) {
                         int ovrlap;
                         bit_or(bitmap, orig_map);
@@ -1767,16 +1882,16 @@
                         ovrlap = bit_overlap(bitmap, tmp_job_ptr->node_bitmap);
                         if (ovrlap == 0)        /* job has no usable nodes */
                                 continue;       /* skip it */
-                        debug2("cons_res: _will_run_test, job %u: overlap=%d",
+                        debug2("cons_rgpu: _will_run_test, job %u: overlap=%d",
                                 tmp_job_ptr->job_id, ovrlap);
+                        //debug("RCUDA job:%i %s(%s,%d) llamando a rm_job_from_res
    ()", job_ptr->job_id, __FILE__, __func__, __LINE__);
                         _rm_job_from_res(future_part, future_usage,
                                          tmp_job_ptr, 0);
                         rc = cr_job_test(job_ptr, bitmap, min_nodes,
                                          max_nodes, req_nodes,
-                                         SELECT_MODE_WILL_RUN, tmp_cr_type,
+                                SELECT_MODE_WILL_RUN, cr_type,
                                          job_node_req, select_node_cnt,
-                                         future_part, future_usage,
-                                         exc_core_bitmap);
+                                future_part, future_usage);
                         if (rc == SLURM_SUCCESS) {
                                 if (tmp_job_ptr->end_time <= now)
                                         job_ptr->start_time = now + 1;
@@ -1796,8 +1911,12 @@
                  * in selected plugin, but by Moab or something else. */
```

```
                if (*preemptee_job_list == NULL) {
                        *preemptee_job_list = list_create(NULL);
+                       if (*preemptee_job_list==NULL)
+                               fatal("list_create malloc failure");
                }
                preemptee_iterator =list_iterator_create(preemptee_candidates);
+               if (preemptee_iterator==NULL)
+                       fatal("memory allocation failure");
                while ((tmp_job_ptr = (struct job_record *)
                        list_next(preemptee_iterator))) {
                        if (bit_overlap(bitmap,
@@ -1836,7 +1955,8 @@
        select_node_usage = NULL;
        _destroy_part_data(select_part_record);
        select_part_record = NULL;
-       cr_fini_global_core_data();
+       xfree(cr_node_num_cores);
+       xfree(cr_node_cores_offset);

        if (cr_type)
                verbose("%s shutting down ...", plugin_name);
@@ -1914,7 +2034,7 @@
        /* initial global core data structures */
        select_state_initializing = true;
        select_fast_schedule = slurm_get_fast_schedule();
-       cr_init_global_core_data(node_ptr, node_cnt, select_fast_schedule);
+       _init_global_core_data(node_ptr, node_cnt);

        _destroy_node_data(select_node_usage, select_node_record);
        select_node_cnt  = node_cnt;
@@ -1929,7 +2049,6 @@
                        struct config_record *config_ptr;
                        config_ptr = node_ptr[i].config_ptr;
                        select_node_record[i].cpus    = config_ptr->cpus;
-                       select_node_record[i].boards  = config_ptr->boards;
                        select_node_record[i].sockets = config_ptr->sockets;
                        select_node_record[i].cores   = config_ptr->cores;
                        select_node_record[i].vpus    = config_ptr->threads;
@@ -1937,7 +2056,6 @@
                                real_memory;
                } else {
                        select_node_record[i].cpus    = node_ptr[i].cpus;
-                       select_node_record[i].boards  = node_ptr[i].boards;
                        select_node_record[i].sockets = node_ptr[i].sockets;
                        select_node_record[i].cores   = node_ptr[i].cores;
                        select_node_record[i].vpus    = node_ptr[i].threads;
@@ -1997,16 +2115,14 @@
                              uint32_t min_nodes, uint32_t max_nodes,
                              uint32_t req_nodes, uint16_t mode,
                              List preemptee_candidates,
-                             List *preemptee_job_list,
-                             bitstr_t *exc_core_bitmap)
+                             List *preemptee_job_list)
 {
        int rc = EINVAL;
        uint16_t job_node_req;
-       static bool debug_cpu_bind = false, debug_check = false;
+       bool debug_cpu_bind = false, debug_check = false;

        xassert(bitmap);

-       debug2("select_p_job_test for job %u", job_ptr->job_id);
        if (!debug_check) {
                debug_check = true;
                if (slurm_get_debug_flags() & DEBUG_FLAG_CPU_BIND)
@@ -2031,16 +2147,14 @@
        if (mode == SELECT_MODE_WILL_RUN) {
                rc = _will_run_test(job_ptr, bitmap, min_nodes, max_nodes,
                                    req_nodes, job_node_req,
-                                   preemptee_candidates, preemptee_job_list,
```

```
-                                       exc_core_bitmap);
+                               preemptee_candidates, preemptee_job_list);
        } else if (mode == SELECT_MODE_TEST_ONLY) {
                rc = _test_only(job_ptr, bitmap, min_nodes, max_nodes,
                                req_nodes, job_node_req);
        } else if (mode == SELECT_MODE_RUN_NOW) {
                rc = _run_now(job_ptr, bitmap, min_nodes, max_nodes,
                              req_nodes, job_node_req,
-                             preemptee_candidates, preemptee_job_list,
-                             exc_core_bitmap);
+                       preemptee_candidates, preemptee_job_list);
        } else
                fatal("select_p_job_test: Mode %d is invalid", mode);

@@ -2048,8 +2162,8 @@
                if (job_ptr->job_resrcs)
                        log_job_resources(job_ptr->job_id, job_ptr->job_resrcs);
                else {
-                       info("no job_resources info for job %u rc=%d",
-                            job_ptr->job_id, rc);
+                       info("no job_resources info for job %u",
+                               job_ptr->job_id);
                }
        } else if (debug_cpu_bind && job_ptr->job_resrcs) {
                log_job_resources(job_ptr->job_id, job_ptr->job_resrcs);
@@ -2184,12 +2298,7 @@
                                         Buf buffer,
                                         uint16_t protocol_version)
 {
-       if (protocol_version >= SLURM_2_6_PROTOCOL_VERSION) {
-               pack16(nodeinfo->alloc_cpus, buffer);
-               pack32(nodeinfo->alloc_memory, buffer);
-       } else {
                pack16(nodeinfo->alloc_cpus, buffer);
-       }

        return SLURM_SUCCESS;
 }
@@ -2203,12 +2312,7 @@
        nodeinfo_ptr = select_p_select_nodeinfo_alloc();
        *nodeinfo = nodeinfo_ptr;

-       if (protocol_version >= SLURM_2_6_PROTOCOL_VERSION) {
-               safe_unpack16(&nodeinfo_ptr->alloc_cpus, buffer);
-               safe_unpack32(&nodeinfo_ptr->alloc_memory, buffer);
-       } else {
                safe_unpack16(&nodeinfo_ptr->alloc_cpus, buffer);
-       }

        return SLURM_SUCCESS;

@@ -2247,13 +2351,14 @@
 {
        struct part_res_record *p_ptr;
        struct node_record *node_ptr = NULL;
-       int i=0, n=0, start, end;
-       uint16_t tmp, tmp_16 = 0, tmp_part;
+       int i = 0, n = 0, c, start, end;
+       uint16_t tmp, tmp_16 = 0;
        static time_t last_set_all = 0;
        uint32_t node_threads, node_cpus;
+       select_nodeinfo_t *nodeinfo = NULL;

        /* only set this once when the last_node_update is newer than
-        * the last time we set things up. */
+          the last time we set things up. */
        if (last_set_all && (last_node_update < last_set_all)) {
                debug2("Node select info for set all hasn't "
                       "changed since %ld",
@@ -2262,12 +2367,14 @@
```

```
        }
        last_set_all = last_node_update;

-       for (n = 0, node_ptr = node_record_table_ptr;
-            n < select_node_cnt; n++, node_ptr++) {
-               select_nodeinfo_t *nodeinfo = NULL;
-               /* We have to use the '_g_' here to make sure we get the
-                * correct data to work on.  i.e. cray calls this plugin
-                * from within select/cray which has it's own struct. */
+       for (n = 0; n<select_node_cnt; n++) {
+               node_ptr = &(node_record_table_ptr[n]);
+
+               /* We have to use the '_g_' here to make sure we get
+                  the correct data to work on.  i.e. cray calls this
+                  plugin from within select/cray which has it's own
+                  struct.
+                */
                select_g_select_nodeinfo_get(node_ptr->select_nodeinfo,
                                             SELECT_NODEDATA_PTR, 0,
                                             (void *)&nodeinfo);
@@ -2290,16 +2397,20 @@
                for (p_ptr = select_part_record; p_ptr; p_ptr = p_ptr->next) {
                        if (!p_ptr->row)
                                continue;
-                       tmp_part = 0;
                        for (i = 0; i < p_ptr->num_rows; i++) {
                                if (!p_ptr->row[i].row_bitmap)
                                        continue;
-                               tmp = bit_set_count_range(p_ptr->row[i].row_bitmap,
-                                                         start, end);
-                               /* Report row with largest CPU count */
-                               tmp_part = MAX(tmp, tmp_part);
+                               tmp = 0;
+                               for (c = start; c<end; c++) {
+                                       if (bit_test(p_ptr->row[i].row_bitmap,
+                                                    c))
+                                               tmp++;
+                               }
+                               /* get the row with the largest cpu
+                                  count on it. */
+                               if (tmp>tmp_16)
+                                       tmp_16 = tmp;
                        }
-                       tmp_16 += tmp_part;     /* Add CPU counts all parts */
                }

                /* The minimum allocatable unit may a core, so scale
@@ -2308,12 +2419,6 @@
                        tmp_16 *= node_threads;

                nodeinfo->alloc_cpus = tmp_16;
-               if (select_node_record) {
-                       nodeinfo->alloc_memory =
-                               select_node_usage[n].alloc_memory;
-               } else {
-                       nodeinfo->alloc_memory = 0;
-               }
        }

        return SLURM_SUCCESS;
@@ -2341,7 +2446,6 @@
 {
        int rc = SLURM_SUCCESS;
        uint16_t *uint16 = (uint16_t *) data;
-       uint32_t *uint32 = (uint32_t *) data;
        char **tmp_char = (char **) data;
        select_nodeinfo_t **select_nodeinfo = (select_nodeinfo_t **) data;

@@ -2372,9 +2476,6 @@
        case SELECT_NODEDATA_EXTRA_INFO:
```

```
                *tmp_char = NULL;
                break;
-       case SELECT_NODEDATA_MEM_ALLOC:
-               *uint32 = nodeinfo->alloc_memory;
-               break;
        default:
                error("Unsupported option %d for get_nodeinfo.", dinfo);
                rc = SLURM_ERROR;
@@ -2528,6 +2629,8 @@

        /* reload job data */
        job_iterator = list_iterator_create(job_list);
+       if (job_iterator==NULL)
+               fatal("memory allocation failure");
        while ((job_ptr = (struct job_record *) list_next(job_iterator))) {
                if (IS_JOB_RUNNING(job_ptr)) {
                        /* add the job */
@@ -2543,221 +2646,6 @@
        return SLURM_SUCCESS;
 }

 /*
  * select_p_resv_test - Identify the nodes which "best" satisfy a reservation
  *     request. "best" is defined as either single set of consecutive nodes
@@ -2765,60 +2653,35 @@
  *     OR the fewest number of consecutive node sets
  * IN avail_bitmap - nodes available for the reservation
  * IN node_cnt - count of required nodes
- * IN core_bitmap - cores which can not be used for this reservation
- * OUT avail_bitmap - nodes allocated for the reservation
- * OUT core_bitmap - cores which allocated to this reservation
  * RET - nodes selected for use by the reservation
  */
-extern bitstr_t * select_p_resv_test(bitstr_t *avail_bitmap, uint32_t node_cnt,
-                                     uint32_t *core_cnt, bitstr_t **core_bitmap)
+extern bitstr_t * select_p_resv_test(bitstr_t *avail_bitmap, uint32_t node_cnt)
 {
        bitstr_t **switches_bitmap;             /* nodes on this switch */
-       bitstr_t **switches_core_bitmap;        /* cores on this switch */
        int       *switches_cpu_cnt;            /* total CPUs on switch */
        int       *switches_node_cnt;           /* total nodes on switch */
        int       *switches_required;           /* set if has required node */

        bitstr_t  *avail_nodes_bitmap = NULL;   /* nodes on any switch */
-       bitstr_t *sp_avail_bitmap;
-       int rem_nodes, rem_cores = 0;           /* remaining resources desired */
+       int rem_nodes; /* remaining resources desired */
        int i, j;
        int best_fit_inx, first, last;
        int best_fit_nodes;
        int best_fit_location = 0, best_fit_sufficient;
        bool sufficient;
-       int cores_per_node;

        xassert(avail_bitmap);
-
-       /* When reservation includes a nodelist we use sequential_pick code */
-       if (!switch_record_cnt || !switch_record_table || !node_cnt)  {
-               return sequential_pick(avail_bitmap, node_cnt, core_cnt,
-                                      core_bitmap);
-       }
+       if (!switch_record_cnt|| !switch_record_table)
+               return bit_pick_cnt(avail_bitmap, node_cnt);

        /* Use topology state information */
        if (bit_set_count(avail_bitmap) < node_cnt)
                return avail_nodes_bitmap;
-
-       if (core_cnt && (*core_bitmap == NULL))
-               *core_bitmap = _make_core_bitmap_filtered(avail_bitmap, 0);
```

```
-
        rem_nodes = node_cnt;

-       /* Assuming symmetric cluster */
-       if (core_cnt) {
-               rem_cores = core_cnt[0];
-               cores_per_node = core_cnt[0] / MAX(node_cnt, 1);
-       } else if (cr_node_num_cores)
-               cores_per_node = cr_node_num_cores[0];
-       else
-               cores_per_node = 1;
-
        /* Construct a set of switch array entries,
         * use the same indexes as switch_record_table in slurmctld */
        switches_bitmap   = xmalloc(sizeof(bitstr_t *) * switch_record_cnt);
-       switches_core_bitmap = xmalloc(sizeof(bitstr_t *) * switch_record_cnt);
        switches_cpu_cnt  = xmalloc(sizeof(int)        * switch_record_cnt);
        switches_node_cnt = xmalloc(sizeof(int)        * switch_record_cnt);
        switches_required = xmalloc(sizeof(int)        * switch_record_cnt);
@@ -2822,45 +2685,11 @@
        switches_cpu_cnt  = xmalloc(sizeof(int)        * switch_record_cnt);
        switches_node_cnt = xmalloc(sizeof(int)        * switch_record_cnt);
        switches_required = xmalloc(sizeof(int)        * switch_record_cnt);
-
        for (i=0; i<switch_record_cnt; i++) {
-               char str[100];
                switches_bitmap[i] = bit_copy(switch_record_table[i].
                                              node_bitmap);
                bit_and(switches_bitmap[i], avail_bitmap);
                switches_node_cnt[i] = bit_set_count(switches_bitmap[i]);
        }

 #if SELECT_DEBUG
@@ -2881,8 +2710,7 @@
        /* Determine lowest level switch satifying request with best fit */
        best_fit_inx = -1;
        for (j=0; j<switch_record_cnt; j++) {
-               if ((switches_node_cnt[j] < rem_nodes) ||
-                   (core_cnt && (switches_cpu_cnt[j] < core_cnt[0])))
+               if (switches_node_cnt[j]<rem_nodes)
                        continue;
                if ((best_fit_inx == -1) ||
                    (switch_record_table[j].level <
@@ -2890,7 +2718,6 @@
                    ((switch_record_table[j].level ==
                      switch_record_table[best_fit_inx].level) &&
                     (switches_node_cnt[j] < switches_node_cnt[best_fit_inx])))
-                       /* We should use core count by switch here as well */
                        best_fit_inx = j;
        }
        if (best_fit_inx == -1) {
@@ -2911,17 +2738,11 @@
        /* Select resources from these leafs on a best-fit basis */
        avail_nodes_bitmap = bit_alloc(node_record_count);
        while (rem_nodes > 0) {
-               int avail_cores_in_node;
                best_fit_nodes = best_fit_sufficient = 0;
                for (j=0; j<switch_record_cnt; j++) {
                        if (switches_node_cnt[j] == 0)
                                continue;
-                       if (core_cnt) {
-                               sufficient =
-                                       (switches_node_cnt[j] >= rem_nodes) &&
-                                       (switches_cpu_cnt[j] >= core_cnt[0]);
-                       } else
-                               sufficient = switches_node_cnt[j] >= rem_nodes;
+                       sufficient = (switches_node_cnt[j]>=rem_nodes);
                        /* If first possibility OR */
                        /* first set large enough for request OR */
                        /* tightest fit (less resource waste) OR */
```

```
@@ -2943,9 +2764,8 @@
                first = bit_ffs(switches_bitmap[best_fit_location]);
                last  = bit_fls(switches_bitmap[best_fit_location]);
                for (i=first; ((i<=last) && (first>=0)); i++) {
-                       if (!bit_test(switches_bitmap[best_fit_location], i)){
+                       if (!bit_test(switches_bitmap[best_fit_location], i))
                                continue;
-                       }

                        bit_clear(switches_bitmap[best_fit_location], i);
                        switches_node_cnt[best_fit_location]--;
@@ -2956,26 +2776,7 @@
                                continue;
                        }

                        bit_set(avail_nodes_bitmap, i);
-                       if (core_cnt)
-                           rem_cores -= cores_per_node;
                        if (--rem_nodes <= 0)
                                break;
                }
@@ -2984,93 +2785,13 @@
        if (rem_nodes > 0)      /* insufficient resources */
                FREE_NULL_BITMAP(avail_nodes_bitmap);

-fini:  for (i=0; i<switch_record_cnt; i++) {
+fini:  for (i = 0; i<switch_record_cnt; i++)
                FREE_NULL_BITMAP(switches_bitmap[i]);
-               FREE_NULL_BITMAP(switches_core_bitmap[i]);
-       }
-
        xfree(switches_bitmap);
-       xfree(switches_core_bitmap);
        xfree(switches_cpu_cnt);
        xfree(switches_node_cnt);
        xfree(switches_required);

        return avail_nodes_bitmap;
 }
```

```
--- slurm-2.6.2/src/plugins/select/cons_res/select_cons_res.h   2013-09-10
    23:44:33.000000000 +0200
+++ slurm-rcuda/src/plugins/select/cons_rgpu/select_cons_rgpu.h 2014-01-20
    11:46:55.880646814 +0100
+#ifndef _CONS_RGPU_H
+#define _CONS_RGPU_H

 #include <fcntl.h>
 #include <stdio.h>
@@ -85,7 +85,6 @@
 struct node_res_record {
        struct node_record *node_ptr;   /* ptr to the actual node */
        uint16_t cpus;                  /* count of processors configured */
-       uint16_t boards;                /* count of boards configured */
        uint16_t sockets;               /* count of sockets configured */
        uint16_t cores;                 /* count of cores configured */
        uint16_t vpus;                  /* count of virtual cpus (hyperthreads)
@@ -112,4 +111,4 @@
 extern void cr_sort_part_rows(struct part_res_record *p_ptr);
 extern uint32_t cr_get_coremap_offset(uint32_t node_index);

+#endif /* !_CONS_RGPU_H */
```