ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatikdienste

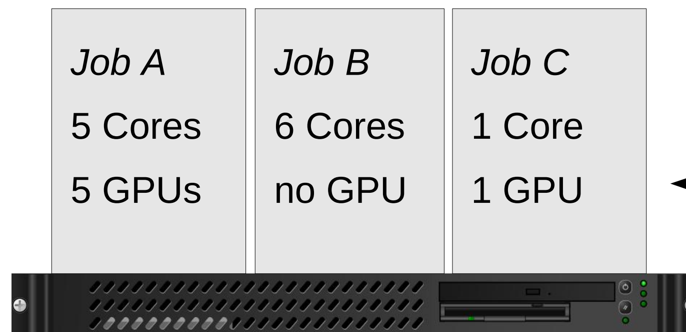
'Enhancing' commercial software

Adrian Ulrich - <adrian.ulrich@id.ethz.ch>

Using GPUs on Brutus

- Fully integrated into Brutus – not a separate cluster
- Not exclusive – shared with other jobs on same node
- … yet another resource (just like cores, RAM, storage...)

```
$ module load cuda
$ bsub -R gpu ./a.out      # more on that later...
```



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Timeline

Login nodes
2 GPUs per Login node

Tesla nodes
4 nodes, each with:
→ 6 GPUs
→ 12 Cores

Fermi nodes
18 nodes, each with:
→ 2 GPUs
→ 24 Cores

ATI nodes

Jan 2009
Jun 2009
Jan 2010
Jun 2010
Jan 2011
Jun 2011

CUDA 2.1
CUDA 2.3
CUDA 3.0
CUDA 3.1*
CUDA 3.2
CUDA 4.0

* CUDA 3.1 adds support for CUDA_VISIBLE_DEVICES

Integration - Goals

- Flexibility
  - Must support multiple generations of Hardware
  - Plan for non NVIDIA GPUs
- Maximizing resource utilization
  - Run independent GPU jobs on same node
  - Available for non GPU jobs
- Easy to use
  - Submitting GPU jobs should be easy

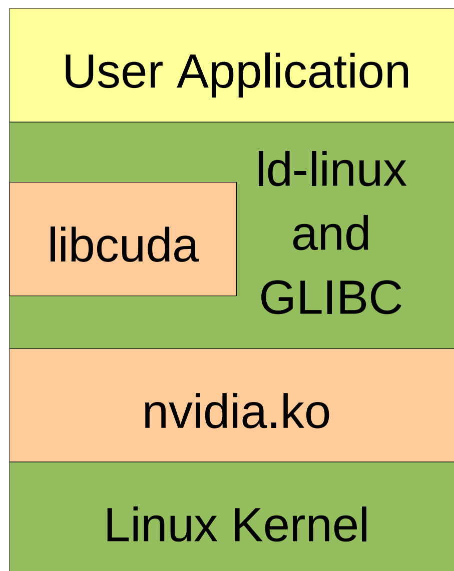
Integration - Solution

- Flexibility
  - Easy with Platform LSF: add new resources (gputype, gpuvendor, cudaversion)
- Maximizing resource utilization
  - Doable with CUDA 3.1 (released in Jun 2010)
  - Not possible with older releases
  - We started with CUDA 2.3

  → Needed to build our own solution

Our own solution: 'enhancing' libcuda:

- CUDA is closed-source (libcuda + nvidia.ko)
- ...but we have full controll about certain parts:



→ Solution: Intercept calls to libcuda with our own library

libfairydust.so was born

- Uses LD_PRELOAD
- Hijacks only 'device related' functions
- Implements all 3 APIs (CUDAAPI, CUDA**RT**API, OpenCL)

Example code (using the driver API):

```
cuDeviceGetCount(&devcount); // fixup &devcount
for(i=0;i<devcount;i++) {
        cuDeviceGet(&dev,i); // fixup i, fake &dev
        cuCtxCreate(&ctx, 0, dev); // fixup dev
        cuMemGetInfo(&mem_free, &mem_total); // untouched
       ......
}
```

Download: https://github.com/adrian-bl/libfairydust

User Application
libcuda
ld-linux
GLIBC
nvidia.ko
Linux-Kernel

libfairydust + fairyd.pl

a.out	`cuDeviceGet(&dev, 0);`
libfairydust	`cuDeviceGet` is hijacked
	Connect to fairyd.pl via localhost:6680
	Sends 'allocate $pid'
fairyd.pl	Searches master pid (LSF launcher) via /proc
	Sends back physical GPU numbers `[2,5]`
libfairydust	calls: `(CUresult)cu_cuDeviceGet(dev, 2);`
CUDA	writes result (2) to **dev** pointer
libfairydust	`*dev = get_vdev(*dev) /* = 0 */`
a.out	`/* back in a.out */`

Only done on first libcuda call

Upgrade to CUDA 3.1

- Drop own API-Wrappers: Use native CUDA_VISIBLE_DEVICES method
- Problem: LSF sets ENV at submission time
  → need a way to set it at startup
- Solution: Hijack first ioctl() call
  → driver ↔ userspace communication works via ioctl()'s on **`/dev/nvidia*`**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ioctl()

```
# strace output:
open("/dev/nvidiactl", O_RDWR)              = 3
ioctl(3, 0xc04846d2, 0xb4dc0d3)             = 0
...
```

Hijacked by libfairydust

Call libc ioctl() and go back to application

```
if(newenv == NULL && rq == 0xc04846d2) {
        __fdust_cuvisible_init(); // rewrites *ENV
}
return (long int)libc_ioctl(fd,rq,argp);
```

libcuda + glibc

libfairydust.so

What about AMD/ATI?

- AMD Stream only implements OpenCL
- Using pre CUDA 3.1 approach

  → much simpler: Only 1 wrapper needed: **clGetDeviceIDs**
- User can select GPU type at submission time
- Also runs on CPUs

Joining forces: LSF and fairyd

Platform LSF

- Allocates resources at cluster level
- Dispatches job to correct host (matching gputype, enough free GPUs, etc..)
- Does not know/care about physical GPUs

fairyd

- Only knows about local compute node (even for MPI)
- Allocates physical GPUs to a job
- Communicates with LSF via environment
- Communicates with libfairydust via TCP/IP

Submission examples

```
$ ssh brutus.ethz.ch
$ module load cuda
$ ./cuda
```

- Runs on the login node (each login node has 2 T10 GPUs)

```
$ bsub -n 2 -R gpu ./cuda
```

- Allocates 2 cores + 1 GPU **per core**
- Runs on default GPU model (Fermi with CUDA 4.0)

Submission examples

```
$ bsub -n 24 -R 'rusage[gpu=0.25]' \
     mpirun ./cuda
```

- Allocates 24 cores + 6 Fermi GPUs (ceil(24*0.25)== 6)

```
$ bsub -R 'select[gputype==Tesla]' \
   -R 'rusage[gpu=6]' ./cuda
```

- Allocates 1 core + 6 Tesla GPUs

```
$ module purge ; module load stream/2.4
$ bsub -R 'select[gputype==Cypress]' ./ocl
```

- Allocates 1 core + 1 AMD/ATI GPU

Operational issues - Software

Nvidia CUDA

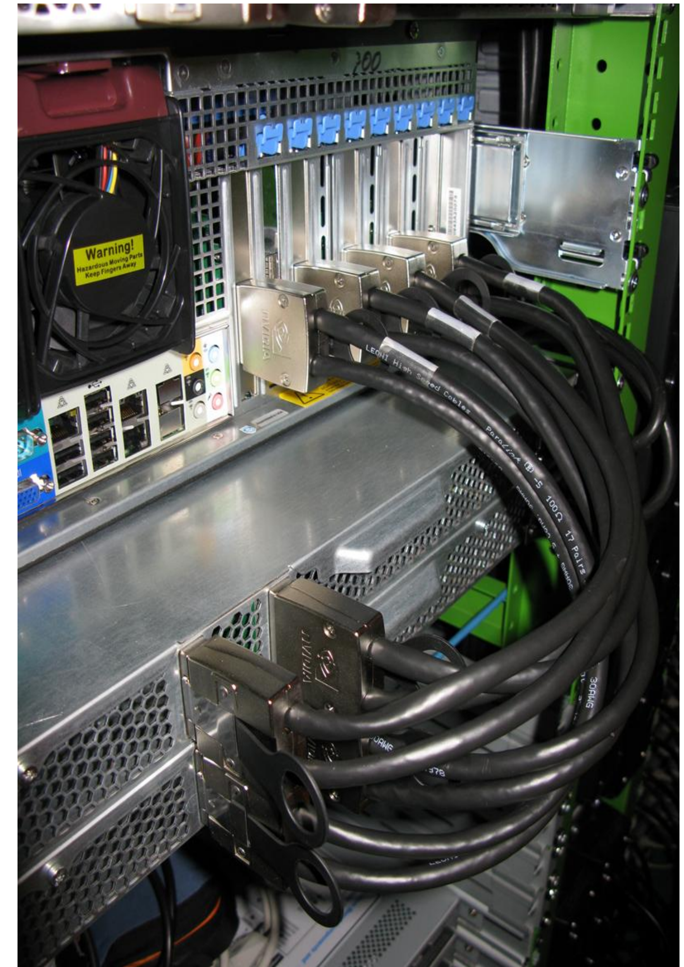
- New releases installed on request
  - Kernel driver has to match userspace release
  - ABI breakage: Users must recompile
  - User can select a specific version (just a resource, defaults to 4.0)

AMD/ATI Stream

- Needs Xorg running!
- Unstable kernel driver (too many kernelpanics)
- Only implements OpenCL: ABI seems to be stable

Operational issues - Hardware

- Memory errors on Tesla GPUs
- No errors on Fermi (so far)
- Tesla T10: 'Interesting' Rails
- PCI Cabling is an adventure of its own



Backup slides

T10 Frontview

Login nodes

Get version of nvidia.ko

```
#define OUR_VERSION "260.19.21"

typedef struct n_version {
  char unk0; char unk1; char unk2; char unk3;
  char reply; char unk5; char unk6; char unk7; char vstring[14];
} n_version;

int main() {
  int fd;
  int r;
  struct n_version nvx;
  memset(&nvx, 0, sizeof(nvx));
  strcpy(nvx.vstring, OUR_VERSION);
  fd = open("/dev/nvidiactl", O_RDWR);
  r = ioctl(fd, 0xC04846D2, &nvx);
  close(fd);
  if(r)
      printf("ioctl() failed. Different version? -> *%s*\n", OUR_VERSION);

  if(nvx.reply)
      printf("Reply from nvidia driver: version=%s\n", nvx.vstring);
  exit(0);
}
```