Transparent Checkpoint and Restart Technology for CUDA® applications

Taichiro Suzuki, Akira Nukada, Satoshi Matsuoka

Tokyo Institute of Technology

Taichiro, SUZUKI

2010.4 ~ 2014.3 Bachelor course at Tokyo Tech

(2013.2 ~ in our laboratory)

2014.4 ~ 2016.3 Master course at Tokyo Tech.

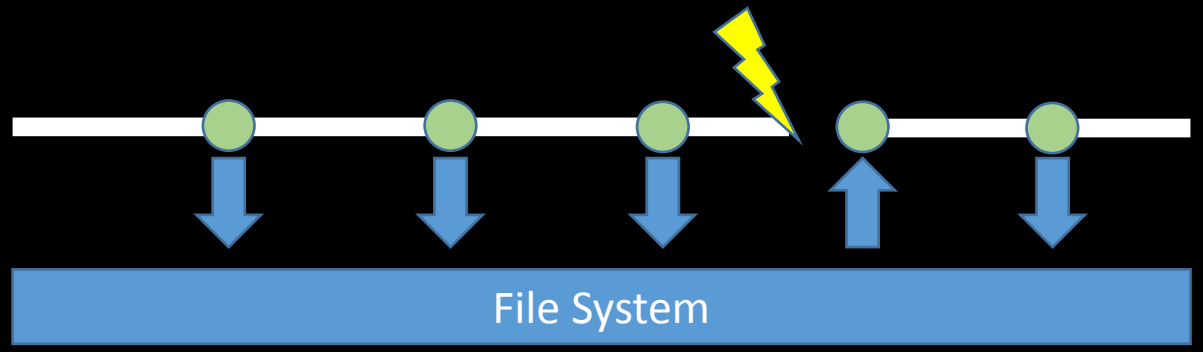
2016.4 ~ Working at an IT company

Checkpoint Technology

[Checkpoint]
Save process state of running application
to checkpoint file.

[Restart]
Restart the execution of the application
from saved checkpoint file.

Fault-Tolerance
By saving checkpoint periodically
we can restart if unexpected fault happens



History of Checkpoint support for CUDA

2002. BLCR

Berkley Lab Checkpoint / Restart [Duell, 2002]

- Popular system-level checkpoint library for Linux systems

BLCR doesn’t work for CUDA applications because data and state on GPU-side cannot be saved and restored.

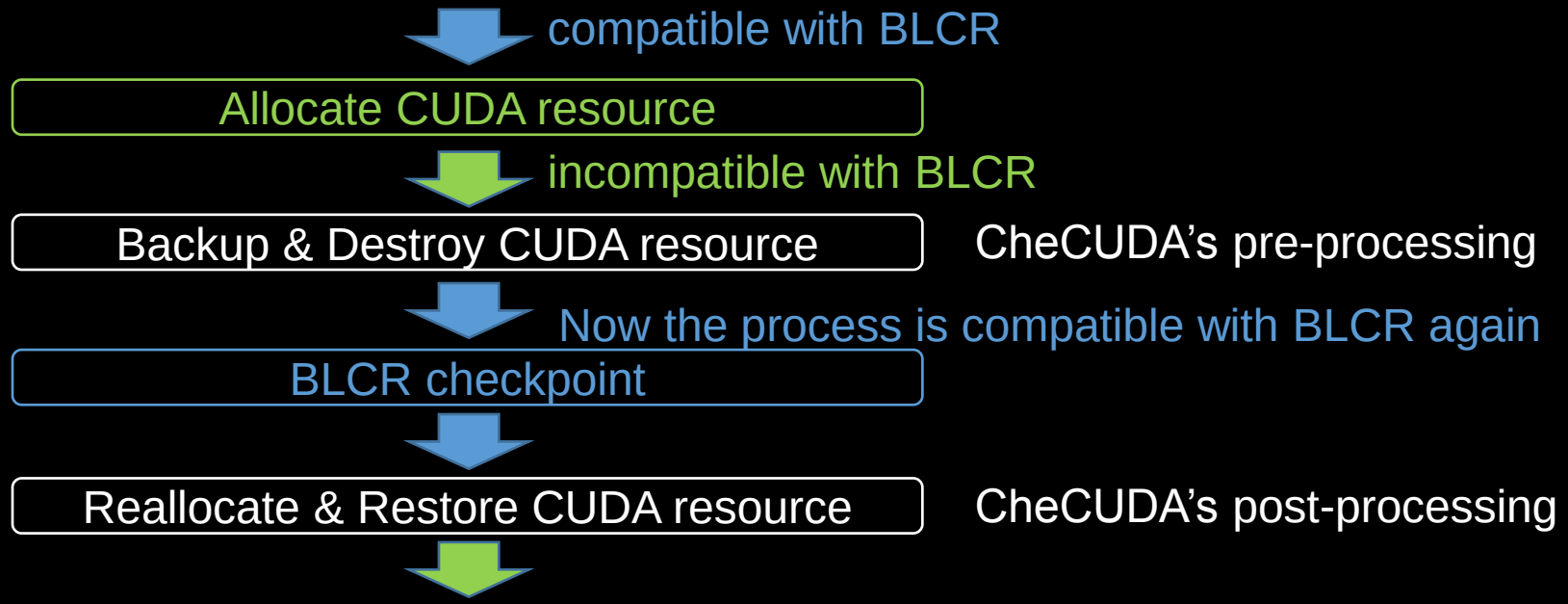
CUDA runtime library uses special device files /dev/nvidia* to control GPUs.

History of Checkpoint support for CUDA

2002. BLCR
2009. CheCUDA

CheCUDA [Takizawa, 2009]

The first success of Checkpoint/Restart of CUDA applications.



When reallocating, the device memory may change.

CheCUDA uses address translation for arguments of CUDA API, but this doesn't work correctly when device address is passed by other way.

History of Checkpoint support for CUDA

2002. BLCR
2009. CheCUDA
2011. NVCR

NVCR [Nukada, 2011]

NVCR's approach is same as CheCUDA, but NVCR works transparently.

NVCR employs *replay* technique to reallocate CUDA resource exactly on same device addresses. NVCR records all CUDA API calls which allocate or free resources.

```
cudaMalloc(&devptr1, size1);        {MALLOC, devptr1, size1 }
cudaMalloc(&devptr2, size2);        {MALLOC, devptr2, size2 }
cudaMalloc(&devptr3, size3);        {MALLOC, devptr3, size3 }
```

Record API, args, and returned addr.

History of Checkpoint support for CUDA

2002. BLCR
2009. CheCUDA
2011. NVCR
2011. CheCL

CheCL [Takizawa, 2011]

CheCL is not for CUDA but OpenCL applications.

CheCL employs a server process which accesses OpenCL devices.

Migration between different platform is also possible.

Goal of CRCUDA : do our best!

Highest availability and compatibility

- transparent
- replay technique (= no address translation)

Work on latest CUDA

- dedicated server process
  - CheCUDA's approach no longer works with CUDA 4.0 and later.
  - BLCR's checkpoint fails, or restart fails.

Minimum overhead

- CUDA pinned memory support
  - Very important feature for fast data transfer between host and device.

How to work transparently

```
$ ldd ./pmemd.cuda
        linux-vdso.so.1 =>  (0x00007ffffe15d000)
        libcurand.so.7.0 => /usr/apps.sp3/cuda/7.0/lib64/libcurand.so.7.0 (0x00002b309585e000)
        libcufft.so.7.0 => /usr/apps.sp3/cuda/7.0/lib64/libcufft.so.7.0 (0x00002b30990c1000)
        libcudart.so.7.0 => /usr/apps.sp3/cuda/7.0/lib64/libcudart.so.7.0 (0x00002b309e11d000)
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00002b309e3db000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b309e6e1000)
        libifport.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libifport.so.5 (0x00002b309e8fe000)
        libifcore.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libifcore.so.5 (0x00002b309eb2e000)
        libimf.so => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libimf.so (0x00002b309ee6e000)
        libsvml.so => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libsvml.so (0x00002b309f331000)
        libm.so.6 => /lib64/libm.so.6 (0x00002b309ff2d000)
        libintlc.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libintlc.so.5 (0x00002b30a01a6000)
        libc.so.6 => /lib64/libc.so.6 (0x00002b30a03fc000)
        /lib64/ld-linux-x86-64.so.2 (0x00002b309563d000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00002b30a0776000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00002b30a098c000)
        librt.so.1 => /lib64/librt.so.1 (0x00002b30a0b90000)
```

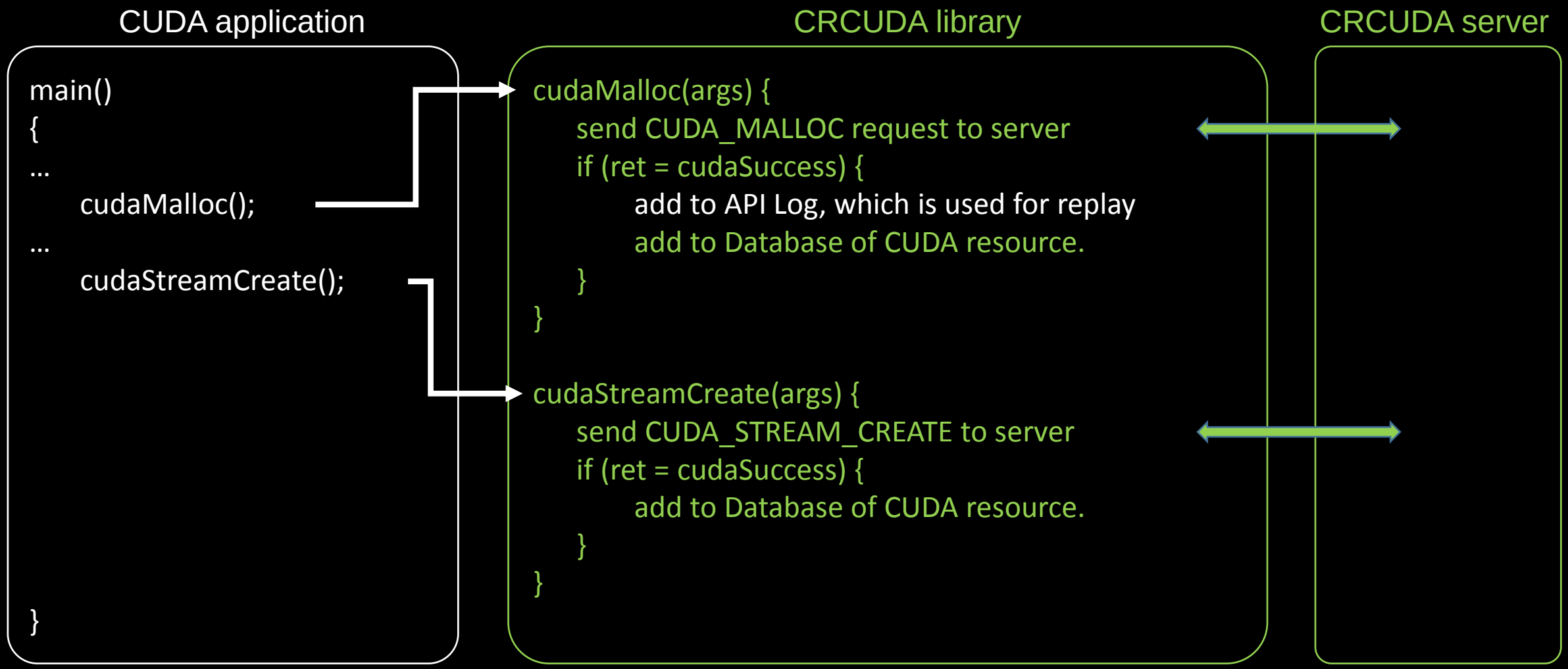
Dynamic libraries related to CUDA

How to work transparently

All we need to do is setting an environment variable. Binary file is not modified.
$ export LD_LIBRARY_PATH=/path/to/crcuda/lib64:$LD_LIBRARY_PATH

```
$ ldd ./pmemd.cuda
        linux-vdso.so.1 =>  (0x00007ffffe15d000)
        libcurand.so.7.0 => /path/to/crcuda/lib64/libcurand.so.7.0 (0x00002b309585e000)
        libcufft.so.7.0 => /path/to/crcuda/lib64/libcufft.so.7.0 (0x00002b30990c1000)
        libcudart.so.7.0 => /path/to/crcuda/lib64/libcudart.so.7.0 (0x00002b309e11d000)
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00002b309e3db000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b309e6e1000)
        libifport.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libifport.so.5 (0x00002b309e8fe000)
        libifcore.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libifcore.so.5 (0x00002b309eb2e000)
        libimf.so => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libimf.so (0x00002b309ee6e000)
        libsvml.so => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libsvml.so (0x00002b309f331000)
        libm.so.6 => /lib64/libm.so.6 (0x00002b309ff2d000)
        libintlc.so.5 => /usr/apps.sp3/isv/intel/xe2013.1.046/composer_xe_2013_sp1.2.144/compiler/lib/intel64/libintlc.so.5 (0x00002b30a01a6000)
        libc.so.6 => /lib64/libc.so.6 (0x00002b30a03fc000)
        /lib64/ld-linux-x86-64.so.2 (0x00002b309563d000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00002b30a0776000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00002b30a098c000)
        librt.so.1 => /lib64/librt.so.1 (0x00002b30a0b90000)
```

How to work transparently



Data transfer between host and device

Two memory type for host

(1) CUDA pinned memory

(2) pageable memory

CRCUDA employs client-server model for BLCR to work correctly.

We have to minimize the overhead of this extra inter-process communication.

Data transfer (CUDA pinned memory)

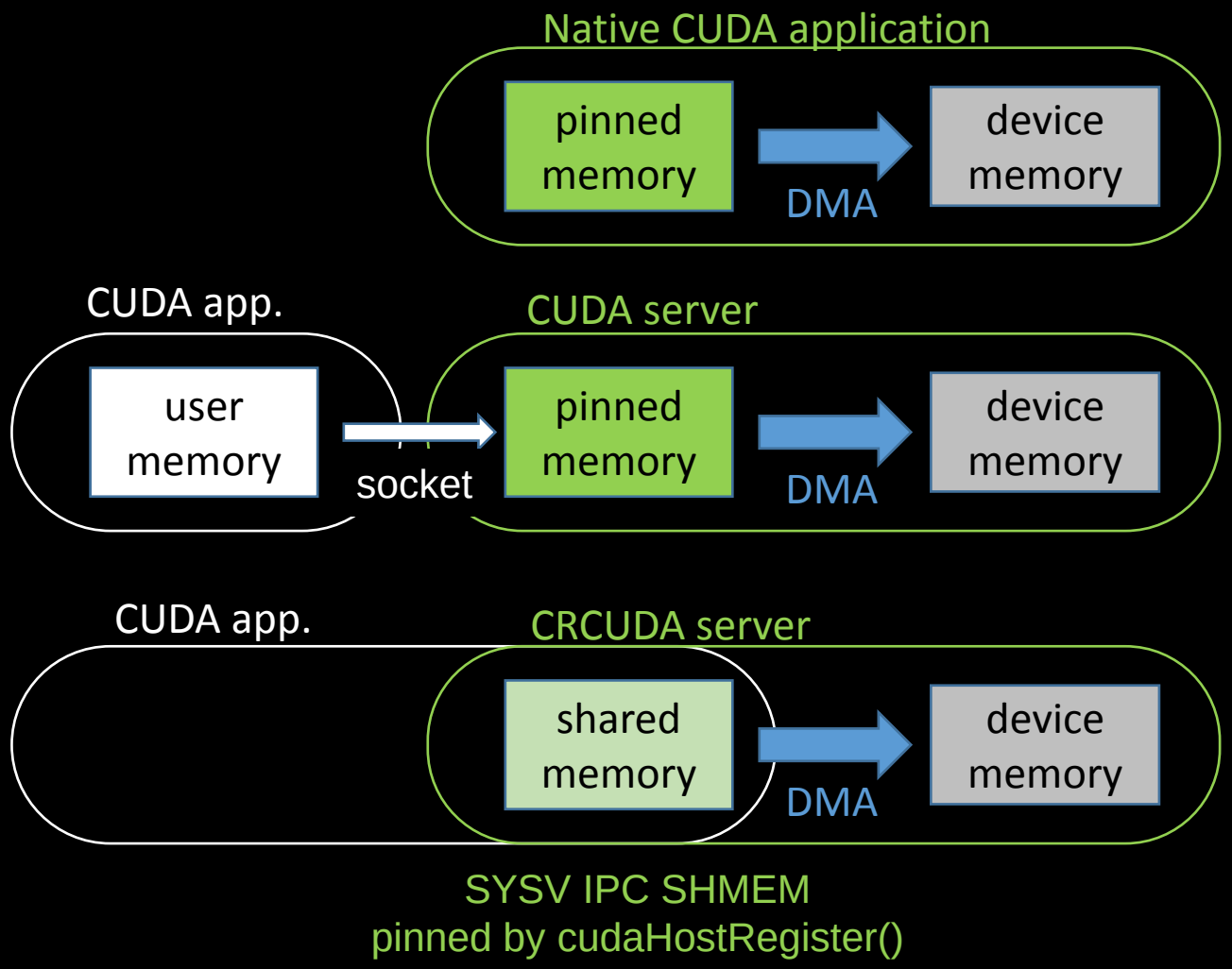
Each circle indicates the border of memory space of each process.

Three implementations are compared.

(1) Native CUDA execution

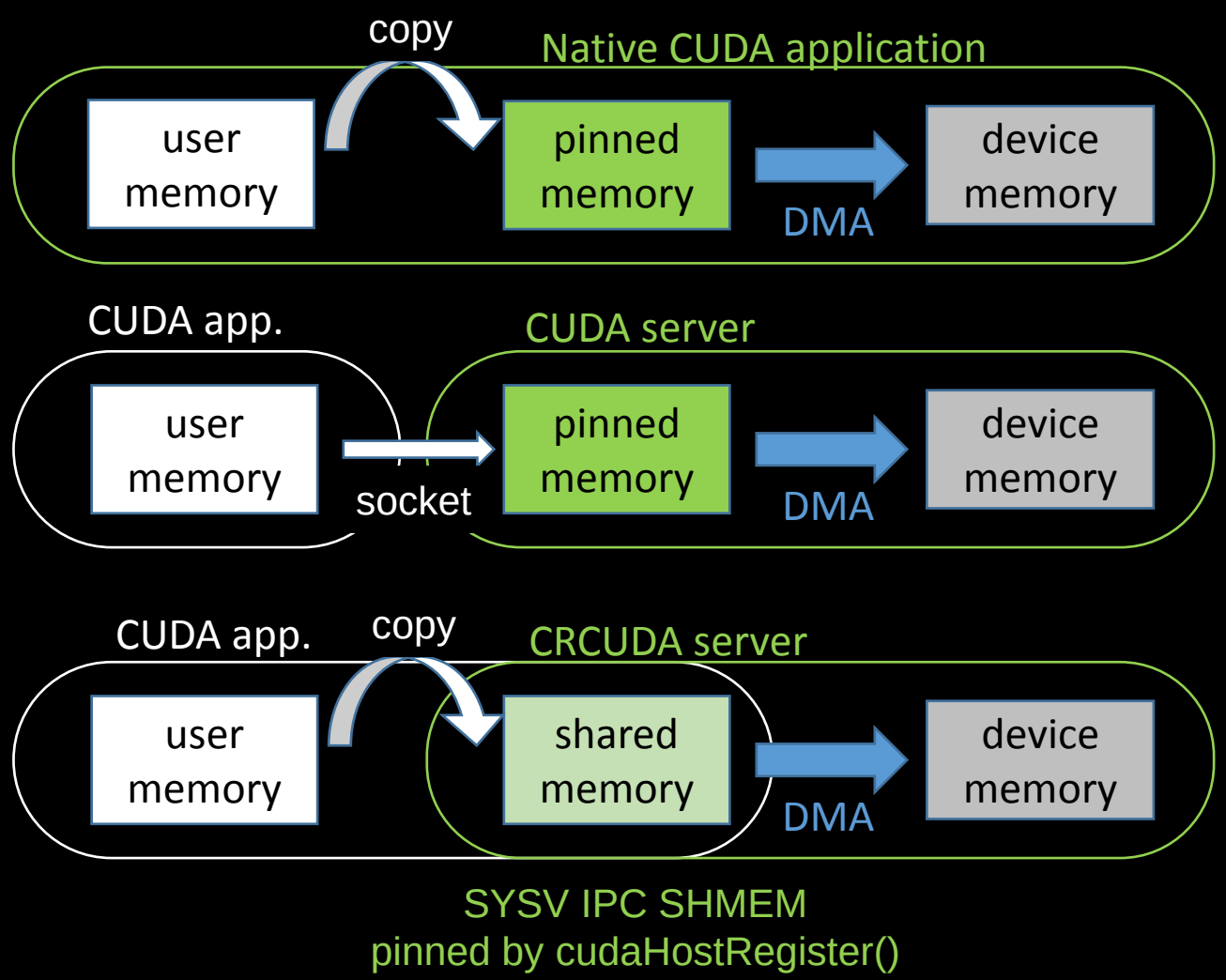
(2) CRCUDA using socket for inter-process

(3) CRCUDA using shmem



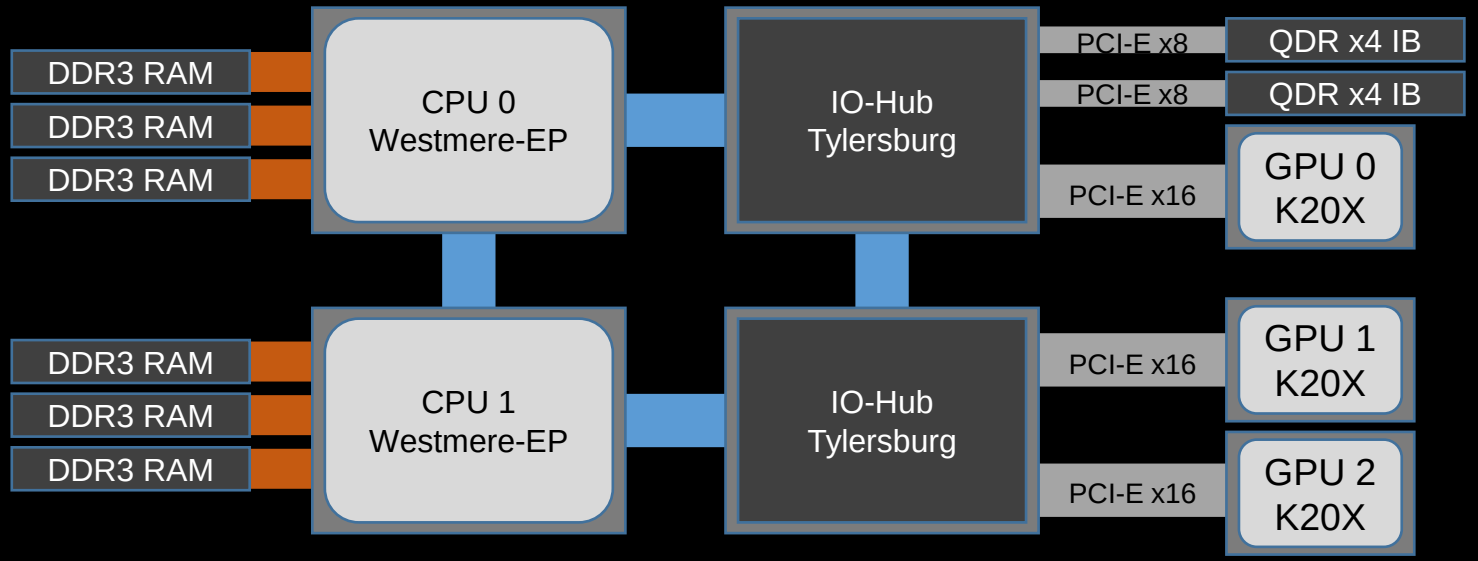
Data transfer (pageable memory)

When pageable memory is specified, CUDA runtime internally copies data to pinned memory buffer, then perform DMA transfer.



Performance Evaluation on TSUBAME 2.5

CPU	Intel Xeon X5670 2.93GHz (6cores) x 2
Memory	DDR3 PC3-10600, 56GB
GPU	NVIDIA Tesla K20X 6GB x 3
OS	SUSE Linux Enterprise Server 11 SP3
CUDA	Version 7.0 (Driver 346.46)
Storage	Lustre filesystem (/work1)

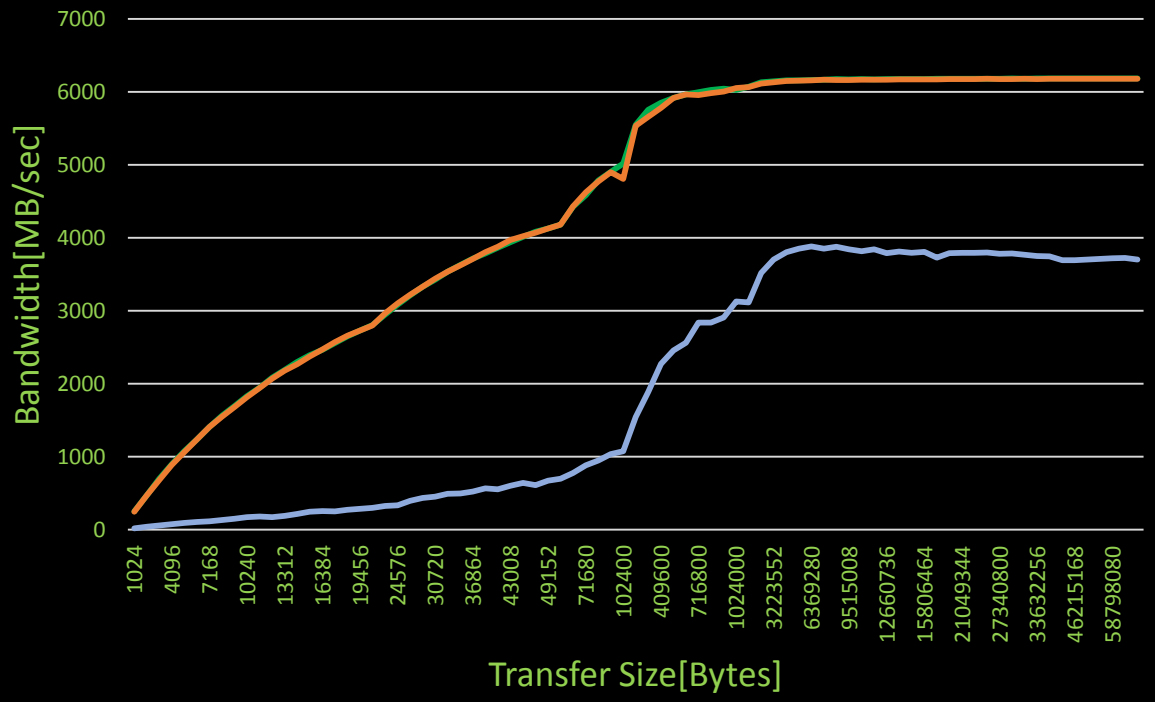


HP ProLiant SL390s G7

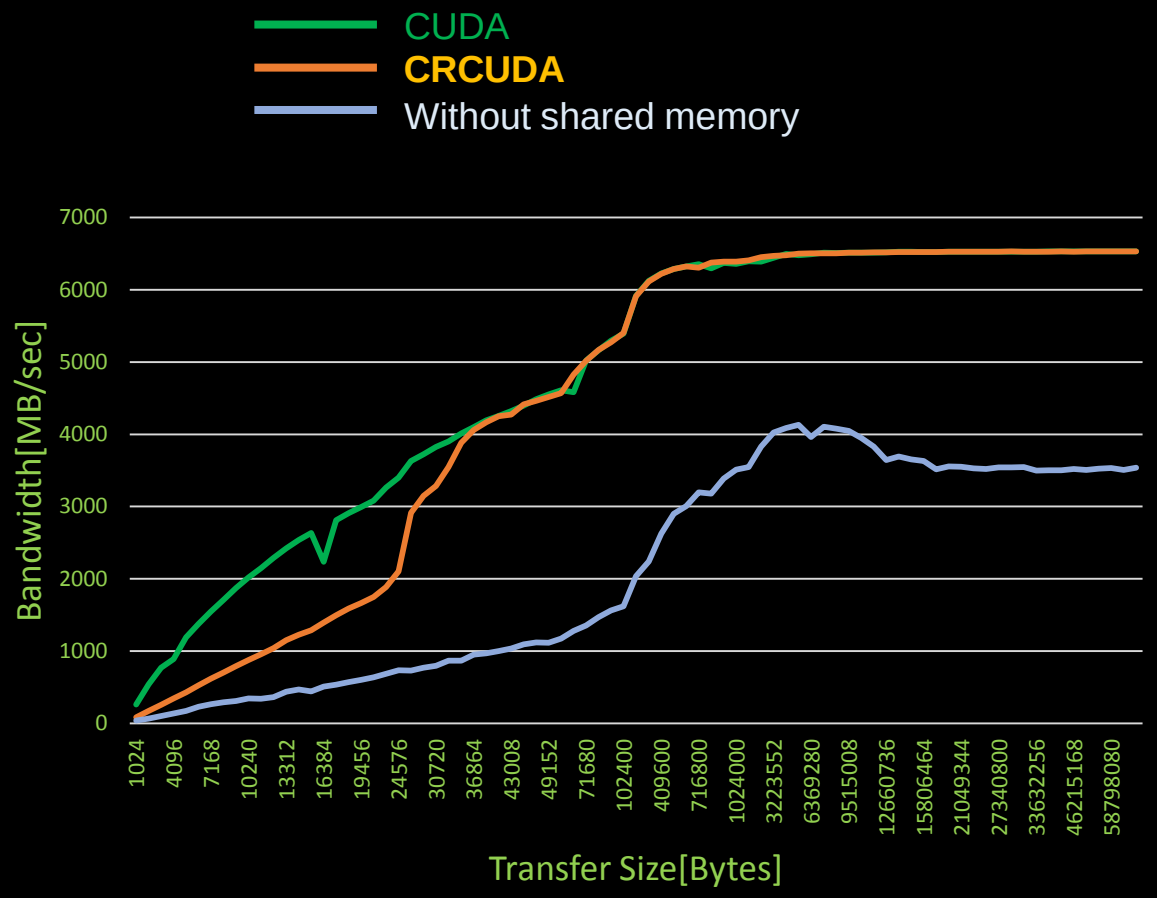
Data transfer bandwidth (pinned)

CRCUDA is as fast as native CUDA

because DMA transfer is performed in both cases



Host to Device

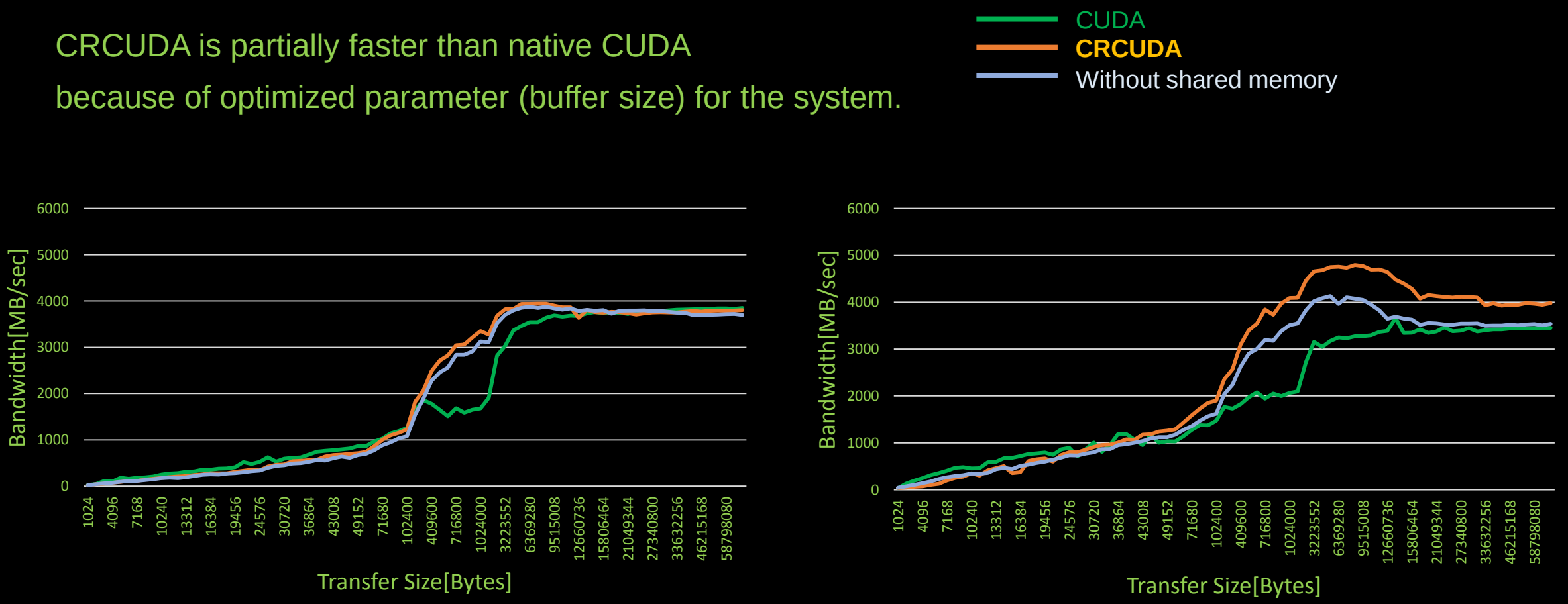


Device to Host

Data transfer bandwidth (pageable)

CRCUDA is partially faster than native CUDA

because of optimized parameter (buffer size) for the system.



Host to Device

Device to Host

Rodinia benchmark suite

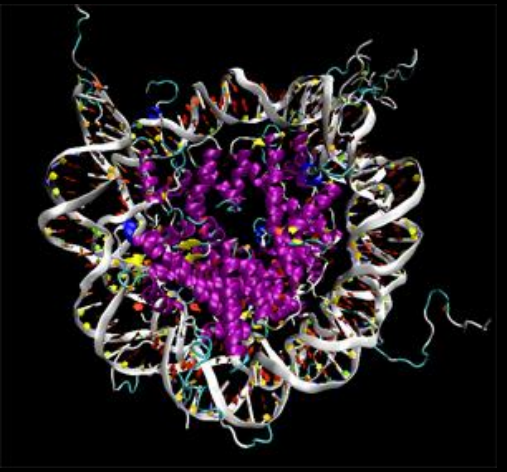
Selected three benchmark programs from Rodinia benchmark suite.

All the others are excluded due to their very short execution time.

Benchmarks	**cfd**	**hotspot**	**Srad_v1**
Native CUDA [sec]	3417	1959	3070
CRCUDA (no ckpt) [sec]	3418	1970	3164
Overhead	0.01 %	0.56 %	**3.03 %**
CRCUDA(interval=5 min)[sec]	3421	1971	3178
Overhead	**0.09 %**	0.62 %	3.49 %
# of checkpoints	11	6	10
File size	30 MB	6 MB	204 MB
Write speed	120 MB/s	28 MB/s	144 MB/s

cfd: iterations = 1260000
hotspot: total_iterations = 33000000
srad_v1: niter = 730000, Nr = 2510, Nc = 2290

AMBER : Molecular Dynamics Simulation



http://ambermd.org/gpus/benchmarks.htm

AMBER 14 update 12, pmemd.cuda binary installed on TSUBAME 2.5

Only one GPU is assigned

Data from AMBER Benchmark Suite 14

Benchmarks	**nucleosome**	**myoglobin**	**TRPCage**	**Cellulose**	**FactorIX**	**JAC**
# of atoms	**25095**	2492	**304**	408609	90906	23558
Native CUDA [sec]	3525	3467	3344	3371	3400	3087
CRCUDA (no ckpt) [sec]	3530	3480	4108	3380	3484	3301
Overhead	0.14 %	0.37 %	**23 %**	0.27 %	2.5 %	6.9 %
CRCUDA(interval=5 min)[sec]	3534	3483	4204	3432	3500	3334
Overhead	**0.26 %**	0.46 %	25 %	1.8 %	2.9 %	8.0 %
# of checkpoints	11	11	14	11	11	11
File size[MB]	49	19	9	861	184	59

Limitations

CRCUDA cannot support following features

1) memory allocation inside CUDA kernel
2) cudaMallocManaged()
3) cudaHostRegister()

In practice, most of HPC applications don’t use these features.

Summary

- CRCUDA enables checkpoint for CUDA applications
  - transparent
  - low overhead data transfer using IPC share memory
  - high overhead when CUDA APIs are called frequently

TODO:

- multi-thread
- multi-GPU
- MPI