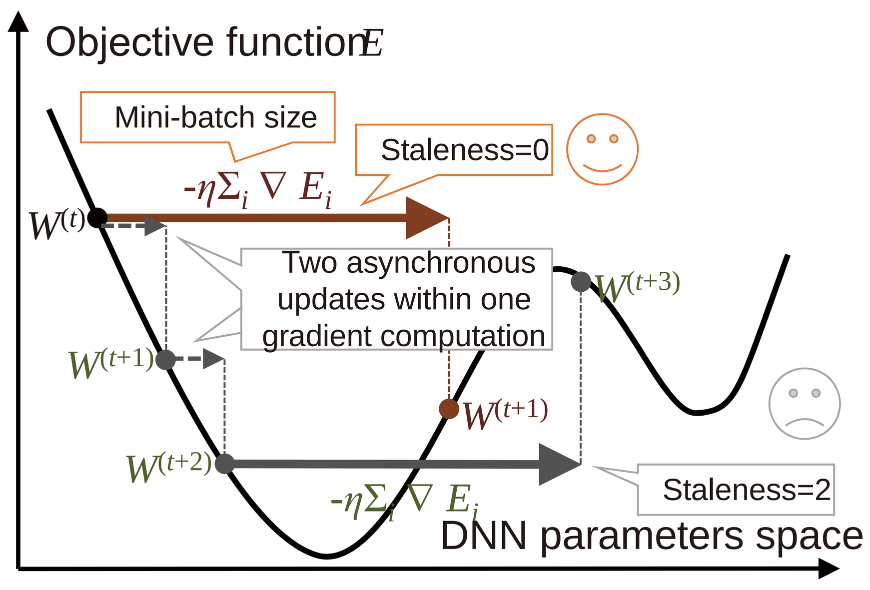
Extreme Big Data and Deep Learning Algorithm Platform

These Researches are Supported by JST CREST Grant Numbers JPMJCR1303 and JPMJCR1687

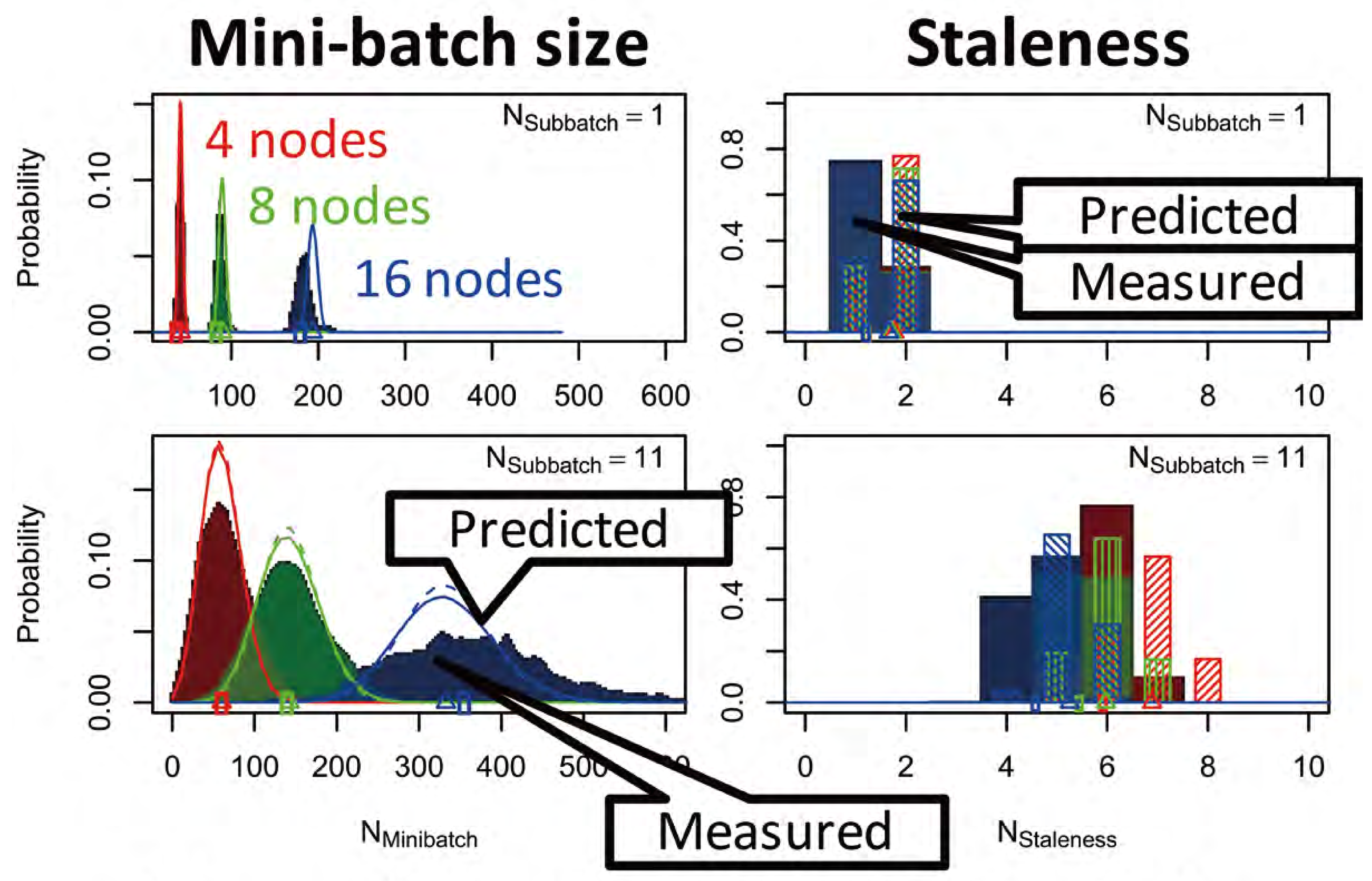
Predicting Statistics of ASGD

Collaborative work with DENSO CORPORATION and DENSO IT LABORATORY, INC

In large-scale Asynchronous Stochastic Gradient Descent (ASGD), mini-batch size and gradient staleness tend to be large and unpredictable, which increase the error of trained DNN



We propose a empirical performance model for an ASGD deep learning system SPRINT which considers probability distribution of mini-batch size and staleness



($N_{Subbatch}$: # of samples per one GPU iteration)

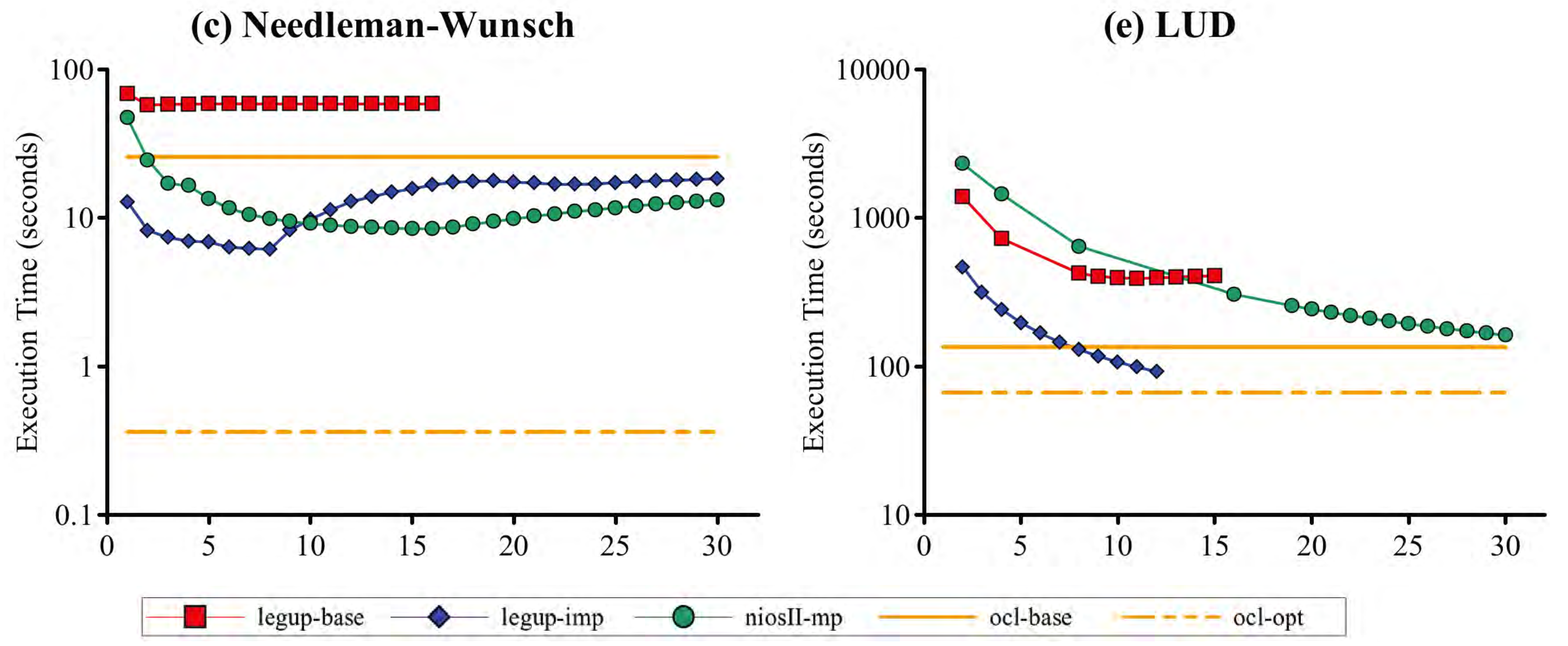
Reference: Yosuke Oyama, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, and Satoshi Matsuoka, "Predicting Statistics of Asynchronous SGD Parameters for a Large-Scale Distributed Deep Learning System on GPU Supercomputers", IEEE BigData 2016

Evaluating Apps on FPGAs

Nowadays, FPGA can rival CPU/GPU performance and energy efficiency, but also known for its hardness for programming.
We compared three high-level programming approaches for FPGAs

- 30-core many-core system (reps. for programmability)
- LegUp High-Level Synthesis (reps for multiple custom accelerators)
- Intel OpenCL for FPGA (reps for Deep-pipeline designs)

We evaluated using Rodinia Benchmark Suite on Stratix V FPGA.
We improved memory hierarchy for many-core and multi-accelerator designs through cache multi-banking.



- Intel OpenCL for FPGA shown highest average performance
- LegUp can remain competitive for good performance and spatial/temporal locality, even without improvement.
- Many-core system offers good programmability, but often does not perform well compared to other approaches

Reference: "Evaluating High-Level Design Strategies on FPGAs for High-Performance Computing", A. Podobas, H.R. Zohouri, N. Maruyama, S. Matsuoka, IEEE FPL 2017

Fast SpGEMM on GPU

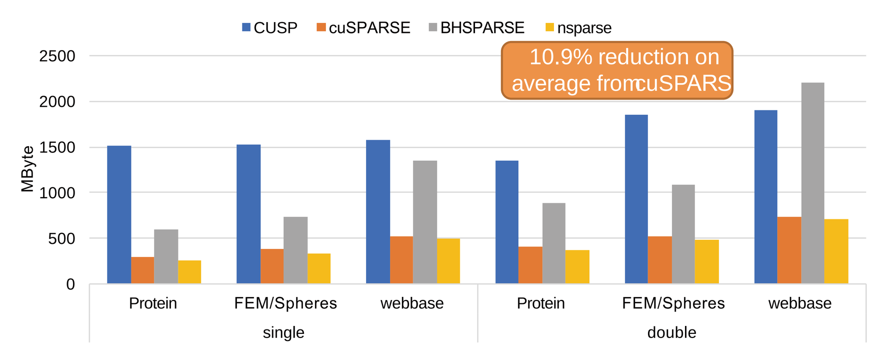
We have devised new Sparse General Matrix-Matrix Multiplication algorithm on GPU, which achieves further speedups and reduces memory usage so that various matrix data can be applied by utilizing GPU's on-chip shared memory and appropriate assigning of GPU resources.

Two Phases Algorithm : 1st phase counts the number of non-zero elements of output matrix, and 2nd phase calculates the output matrix
→ Reduce memory usage
Grouping rows (1, 2, 6)
→ Better utilization of GPU resources
Two ways threads assignments
→ Improve the load-balance
Hash table on fast shared memory
→ Accelerate counting part (3) and calculation part (7)

(1) Count #intermediate products
(2) Divide the rows into groups by #intermediate products
(3) Count #non-zero elements
(4) Set row pointers of output matrix
(5) Memory allocation of output matrix
(6) Divide the rows into groups by #non-zero elements
(7) Compute the output matrix
a. Calculate values and column indices on hash table
b. Shrink the hash table
c. Store to the memory with sorting

Double Precision Performance

Memory Usage



[1] Dalton et al., "CUSP: Generic parallel algorithms for sparse matrix and graph computations ver.0.5.1"
[2] NVIDIA, "Nvidia cuda sparse matrix library (cuSPARSE)"
[3] Liu et al., "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data", IPDPS2014

Code available at:
https://github.com/EBD-CREST/nsparse

Reference: Yusuke Nagasaka, Akira Nukada, Satoshi Matsuoka, "High-performance and Memory-saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU", ICPP 2017.

Increasing GPU Occupancy

Multi-GPU batch-queue systems usually experience large number of idle GPUs due to the scattered idle-GPU problem (Fig.1). We addressed this problem by allowing jobs to utilize remote GPUs and migrating execution on a remote GPU back to a local GPU as soon as one becomes available. This method enables the systems to serve more GPU jobs concurrently while minimizing execution time penalty caused by remote GPU communication.

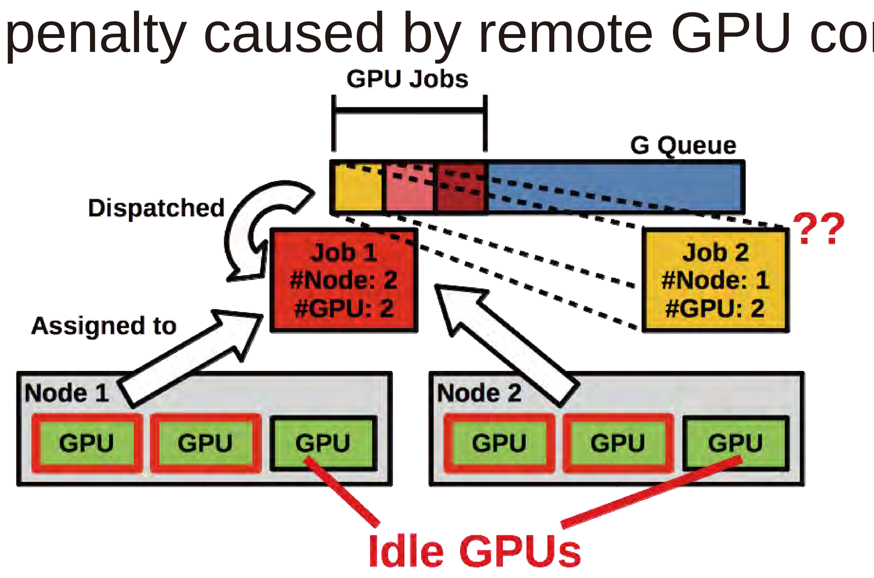


Fig.1: Job 1 and Job 2 cannot run concurrently as Job 2 wants two unoccupied GPUs on the same node.

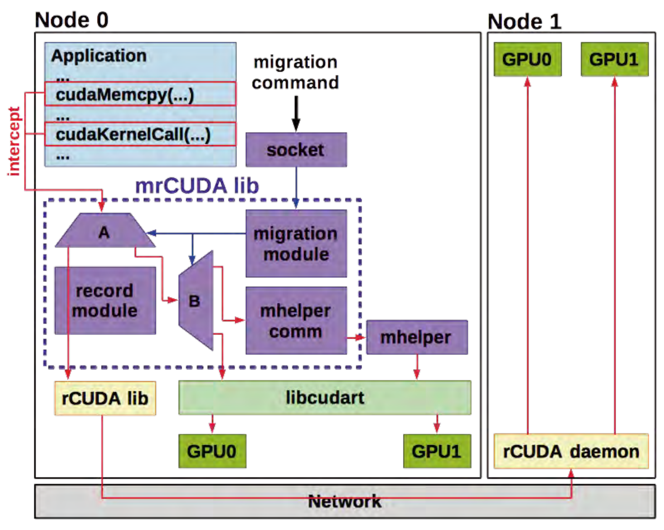


Fig.2: The architecture of mrCUDA, our middleware for handling remote GPU migration on top of rCUDA.

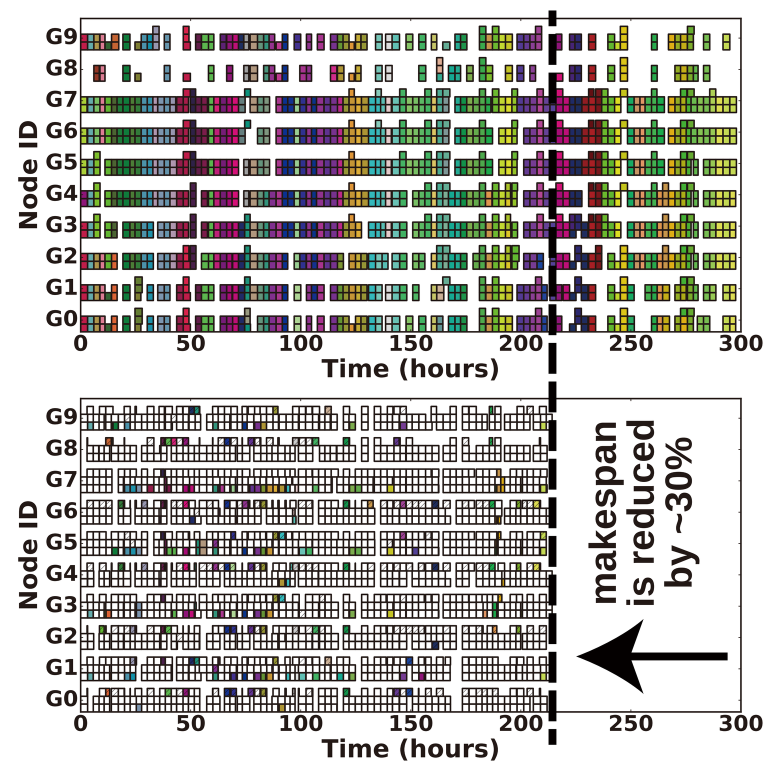


Fig.3: GPU occupancy patterns when using FCFS (top) and our method (bottom).

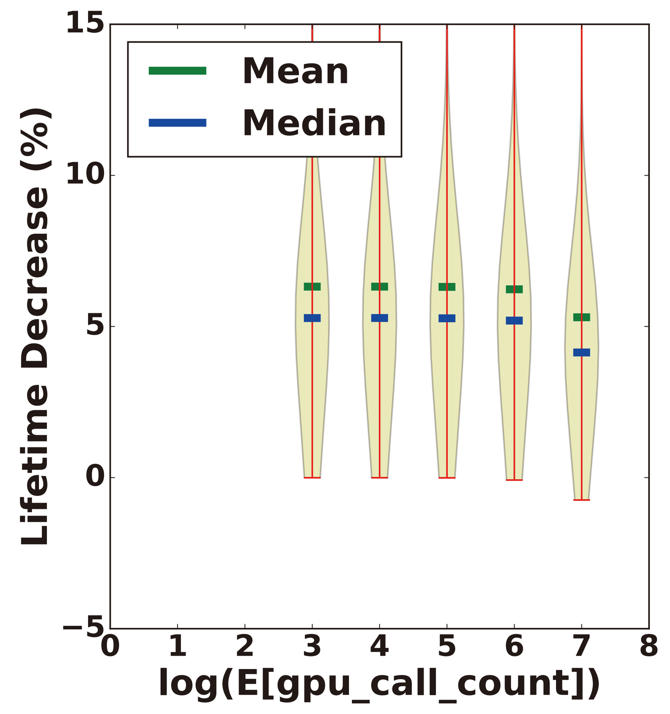


Fig.4: Distribution of jobs' lifetime (waiting + execution time) decrease when using our method compared with FCFS.

Reference: P.Markthub, A.Nomura, and S.Matsuoka, "Serving More GPU Jobs, with Low Penalty, using Remote GPU Execution and Migration," IEEE Cluster 2016.