Challenges in Transition

Keynote talk at International Workshop on Software Engineering Methods for Parallel and High Performance Applications (SEM4HPC 2016)

Kazuaki Ishizaki

IBM Research – Tokyo
kiszk@acm.org

IBM

What is this talk about?

- How we make a HPC platform consumable for non-HPC people?
  - For machine learning (ML) and deep learning (DL)

- This talk is not a solid research proposal, but what I am recently thinking about.

Takeaways

- Users, applications, and HWs are always in transition
  - Programming is becoming hard
- Let us build an end-to-end runtime system for ML and DL
  - Leave each layer to the specialist to do the best
  - Each layer should know everything for optimizations
    - Should not be isolated
- How we can make state-of-the-art technologies consumable in the system?
  - Our research is here!

My History (mostly commercial, sometimes HPC)

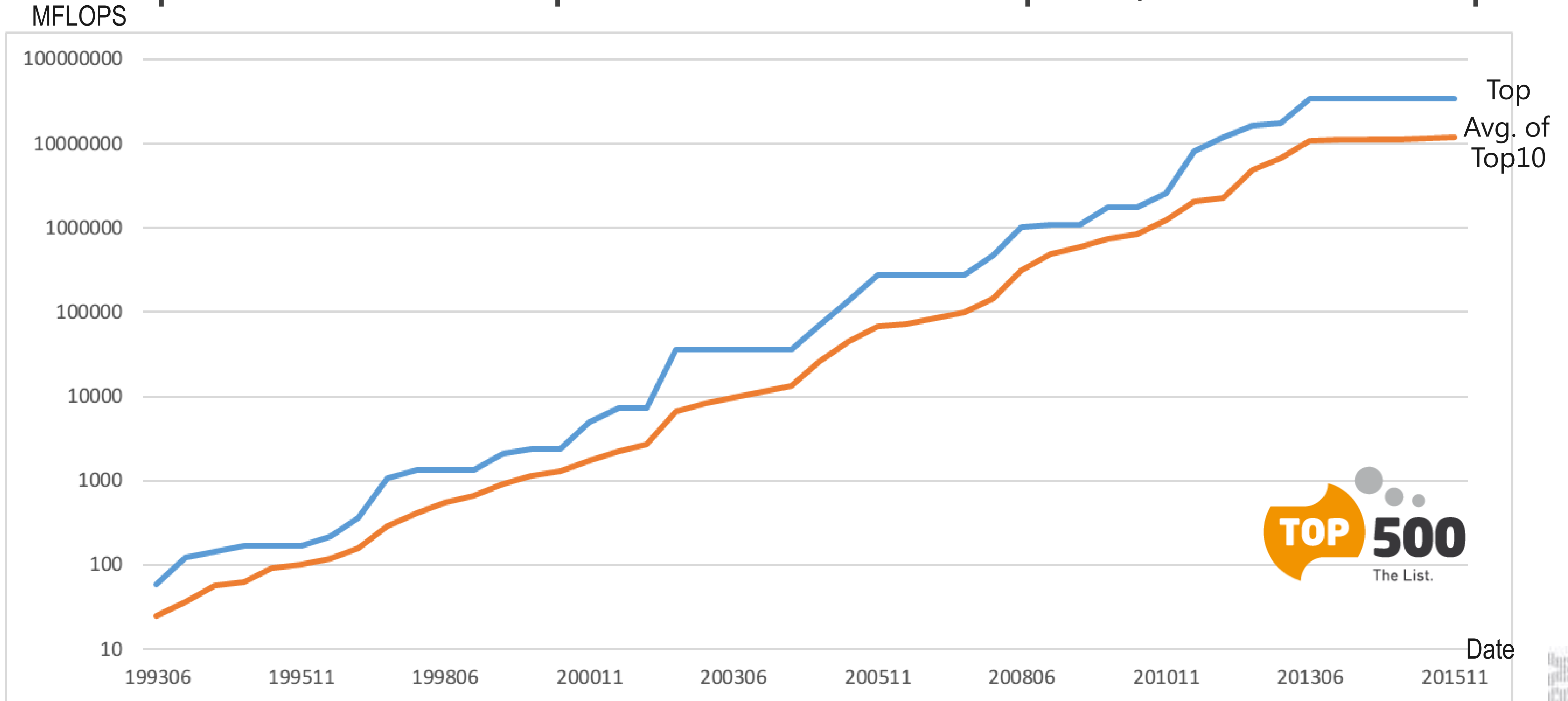
- 1990-1992 Network HW interface for parallel computer
- 1992-1995 Static compiler for High Performance Fortran
- 1996-now Just-in-time compiler for IBM Developers Kit for Java
  - 1996-2000 Benchmark and GUI applications
  - 2000-2010 Web and Enterprise applications
  - 2012- Analytics applications
    - 2014- Java language with GPUs
    - 2015- Apache Spark (in-memory data processing framework) with GPUs

Outline of this talk

- Review transition in HPC
- What are problems in this transition?
- How we will address these problems?

Performance Trend of TOP500

- Great performance improvements for Linpack, at 33.86PFlops



Source: TOP500

TOP #1 Systems in TOP500

Tianhe-2 (2013)

Tianhe-1A (2010)

RoadRunner (2008)

BlueGene/L (2004)

ASCI Red (1997)

CM-5 (1993)

Cray-1 (1975)

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

Source: TOP500

TOP #1 Systems in TOP500

Vector processor

1024 processors

7264 processors

65536 processors

Cell processor

GPU

Xeon Phi

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

Three Eras in HPC

Accelerator Era (2008 -)

MPP Era (1990-)

Vector Era (-1993)

Vector processor

1024 processors

7264 processors

65536 processors

Cell processor

GPU

Xeon Phi

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

*MPP: Massively Parallel Processing

Review for Each Era

- What applications were executed?
- Who wrote these applications?
- What research we did?
- What was commodity HW?

Vector Era

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

Vector Era (-1993)

*MPP: Massively Parallel Processing

Vector processor

Vector Era (- 1993)

- How we can exploit a vector machine for specific applications

Hardware	Slow scalar processor with vector facility
Applications	Weather, wind, fluid, and physics simulations
Programmers	Limited # of programmers who are well-educated for HPC (Ninja programmers)
Research	Automatic vectorization techniques Enhancement of vector HW features (e.g. sparse array support)
Commodity HW	Slow scalar processor

MPP Era

100000000
10000000
1000000
100000
10000
1000
100
10

MPP Era (1990-)

1024 processors

7264 processors

65536 processors

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

*MPP: Massively Parallel Processing

MPP Era (1990 -)

- How we can hide latencies between nodes

Hardware	Massive commodity processors with special network I/F
Applications	Simulations for wider areas (e.g. chemical synthesis)
Programmers	Limited # of programmers who are well-educated for HPC
Research	Improvements on MPI implementations Parallelization and optimization of given applications by hand
Commodity HW	Fast scalar processors

Accelerator Era (2008 -)

Accelerator Era (2008 -)

Cell processor

GPU

Xeon Phi

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

*MPP: Massively Parallel Processing

Innovations in System Software

- CUDA/OpenCL make powerful computing resource accessible

MPI 1.0 (1994)

CUDA (2006)

OpenCL (2008)

NVIDIA

OpenCL

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

Accelerator Era (2008 -)

- How we can exploit GPUs in our applications

Hardware	Massive commodity processors with HW accelerators
Applications	Simulations for wider areas (e.g. chemical synthesis)
Programmers	Limited # of programmers who are well-educated for HPC
Research	GPU-friendly rewriting of given applications by hand GPU-oriented algorithms
Commodity HW	Desktop PC with GPU cards

Innovations in Programming Environment

- MapReduce makes parallel programming easy

100000000
10000000
1000000
100000
10000
1000
100
10

199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

NVIDIA
OpenCL

MapReduce (2004)
Google

Hadoop (2007)
hadoop

Spark (2013)
Spark

Innovations in Infrastructure

- Cloud makes a cluster of machines easily accessible

100000000
10000000
1000000
100000
10000
1000
100
10

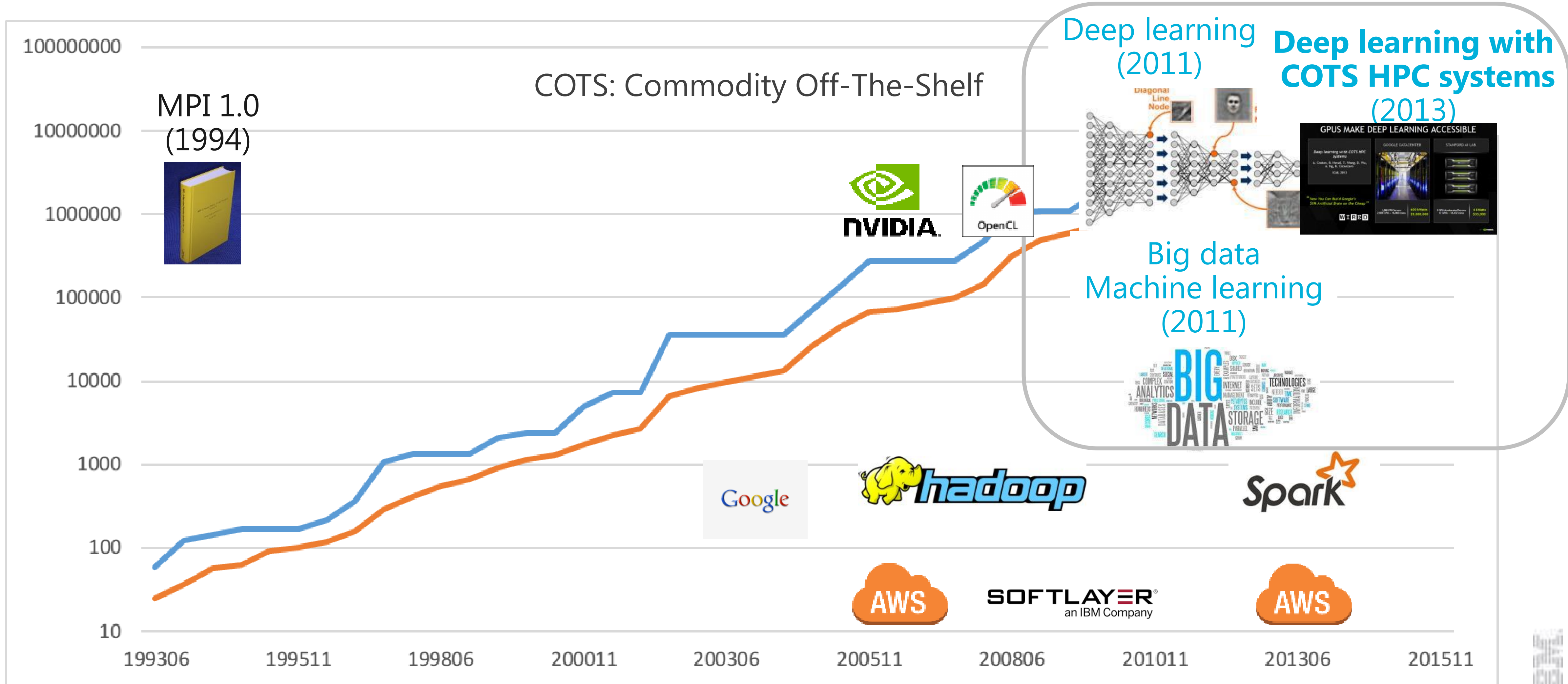
199306 199511 199806 200011 200306 200511 200806 201011 201306 201511

NVIDIA
OpenCL
GPU
Google
hadoop
Spark

AWS EC2 (2006)
AWS
CloudLayer (2009)
SOFTLAYER
an IBM Company
GPU instance (2013)
AWS

Big innovations in Applications

- Machine learning and deep learning are big FP consumers



Source: The analytic store, Deep Learning with GPUs

Accelerator Era 2.0 (2012 -)

- HPC meets machine learning and deep learning with big data

Hardware	Massive commodity processors with HW accelerators
Applications	Machine learning (ML) and deep learning (DL) with big data
Programmers	Data scientists who are non-familiar with HPC
Research	How we can effectively use GPUs? How about accuracy of a new ML/DL algorithm with big data?
Commodity HW	A cluster of machines with GPUs on cloud

Summary of Transition

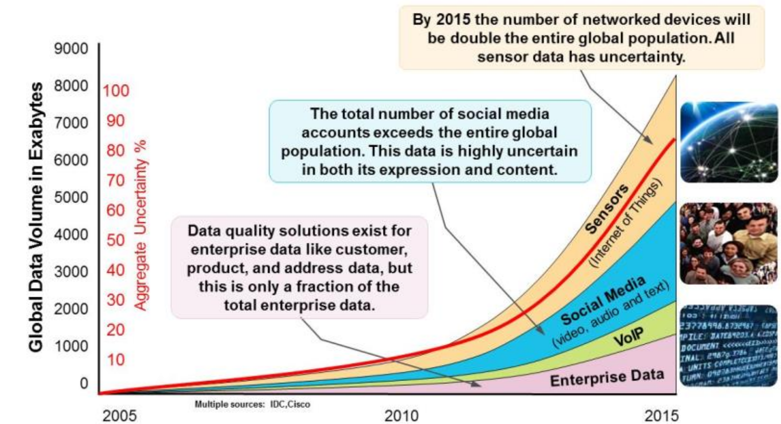
- Majorities of applications are changing
  - From simulations to machine learning (ML)/deep learning (DL) with big data
- HPC HW is becoming commodity
  - GPUs are available on desktop and cloud
  - Cloud provides a cluster of GPUs as a commodity
- Programmers are changing
  - From Ninja programmers to data scientists

Outline of this talk

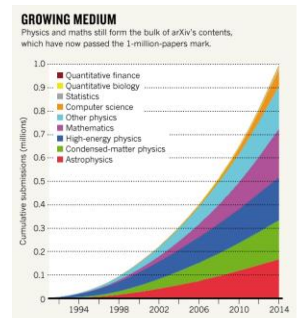
- Review transition in HPC
- What are problems in this transition?
- How we will address these problems?

Details in Application

- Data is becoming rapidly larger
  - 1000x from 2010 to 2015



- The number of applications is rapidly growing
  - arXiv.org is hosting many papers
    - On 2014, hit 1 million articles
    - On 2015, 105,000 new submission and over 139 million downloads
  - github.com is hosting many programs
    - On 2014 4Q, more than 15M updates (pushes) to 2.2M repositories



Source: IDC, CISCO, IBM
Optimization is ready For Big Data
The arXiv preprint server hits 1 million articles
arXiv Update - January 2016
Githut.info

Details in Programming Languages

- Data Scientists love Python and R (e.g. high level languages)
  - Python and R make programming easy
    - Scientific computing operations and libraries (e.g. Numpy)
  - Programs do not scale to a cluster of machines
    - Perform pre-filterings to reduce data size for a machine
    - Spend much time to rewrite it for a cluster
  - It is not easy to write a program optimized for a target architecture

Details in Infrastructure

- Accelerators that matter
  - Processing units
    - GPU, FPGA, ASIC (e.g. Tensorflow Processing Unit), …
  - Storage
    - Non-volatile memory, phase change memory, …
  - Communication
    - Communication between accelerators (e.g. NVLINK), optical interconnect, …

Problems in Future

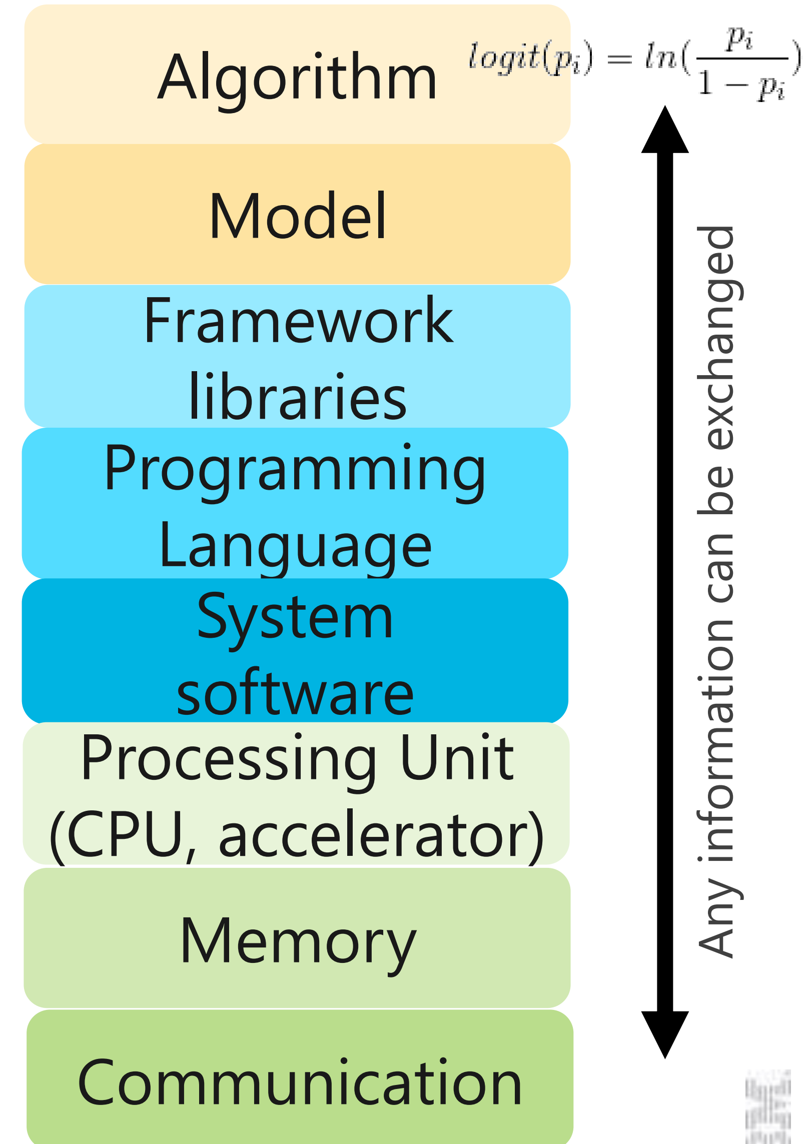
- Data will be too large to store on fast memory
  - Memory hierarchy is becoming deep
- Programming will be hard
  - Hard to program HW accelerators
  - New applications rapidly appear
- Optimization and deployment will be hard
  - Emerging HW accelerators will appear

Outline of this talk

- Review transition in HPC
- What are problems in this transition?
- How we will address these problems?

My Proposal: Build an End-to-End System

- From an algorithm to hardware
- Leave each layer to the specialist to do the best
  - Easy to develop new algorithms
  - Easy to exploit parallelism from the algorithm
  - Easy to generate accelerator code
    - We should avoid complex tasks (e.g. analysis)
- Each layer should know everything
  - What parts of the algorithm are parallel?
  - What happens at hardware
    - We should not make each layer isolated



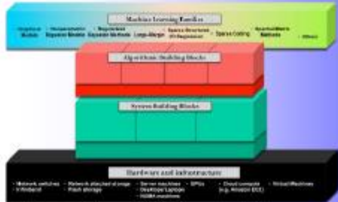
Similar Research 1

The Big-ML "Stack" - More than just software

- **Theory:** Degree of parallelism, convergence analysis, sub-sample complexity ···

- **Representation:** Compact and informative features
- **Model:** Generic building blocks: loss functions, structures, constraints, priors ···
- **Algorithm:** Parallelizable and stochastic MCMC, VI, Opt, Spectrum ···

- **Programming model & Interface:** High: Matlab/R; Medium: C/JAVA; Low: MPI
- **System:** Distributed architecture: DFS, KV-store, task scheduler···
- **Hardware:** GPU, flash storage, cloud ···



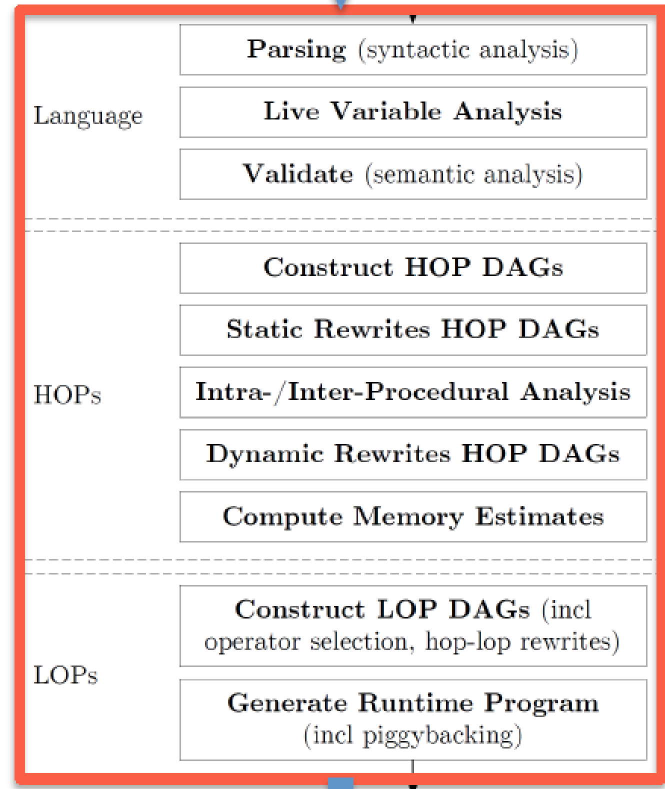
Source: A New Look at the System, Algorithm and Theory Foundations of Distributed Machine Learning

Similar Research 2

- System ML
  - An algorithm written in R subset is translated to an optimized Apache Spark program with information

High-Level Algorithm

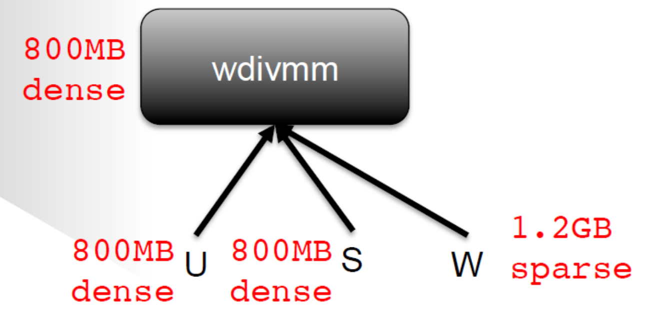
```
HS = t(U) %*% (W * (U %*% S)) + lambda * S;
```



Dynamic Recompile

- Construct HOPs
- Propagate stats
- Determine distributed operations
- **Rewrites**

All operands fit into heap → **use one node**



Parallel Spark Program

Source: Inside SystemML

Our Recent Research: Exploit GPUs at High Level

- Compile a Java Program for GPUs [PACT2015, http://ibm.com/java/jdk/]
  - A parallel stream loop, which explicitly expresses a parallelism, can be offloaded to GPUs by our just-in-time compiler without any GPU specific code

```
IntStream.range(0, N).parallel().forEach(i -> {
  b[i] = a[i] * 2.0;
});
```

Our Recent Research: Exploit GPUs at High Level

- Apache Spark with GPUs [http://github.com/IBMSparkGPU]
  - Drive GPU code from an Apache Spark program transparently from a user

```
// rdd: resilient distributed dataset is distributed over nodes
rdd  = sc.parallelize(1 to 10000, 2) // node0: 1-5000, node1: 5001-10000
rdd1 = rdd.map(i => i * 2)
sum  = rdd1.reduce((x, y) => (x + y))
```

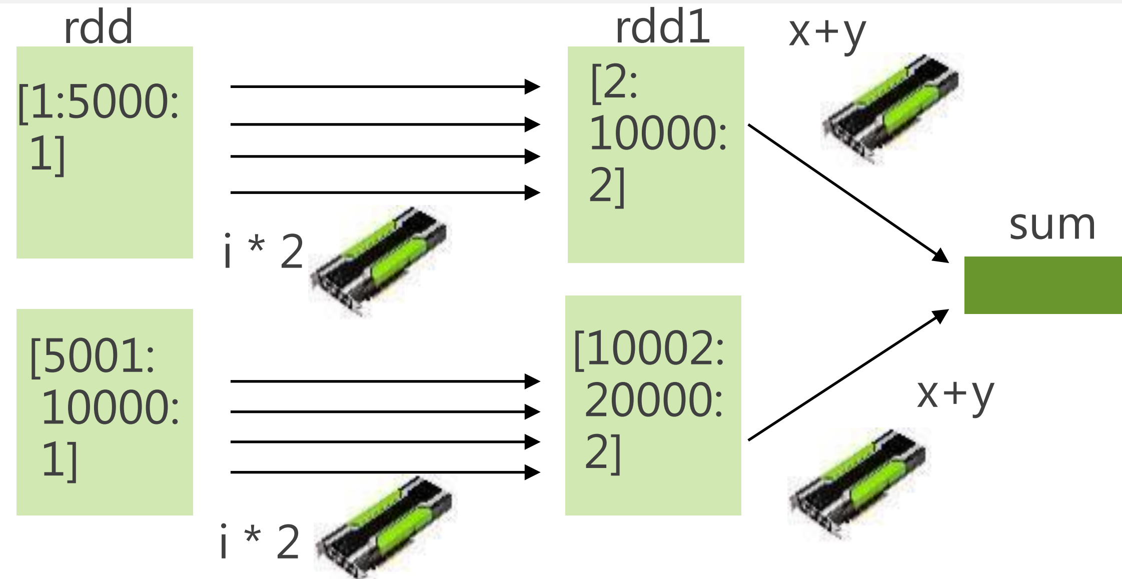


Image Source: NVIDIA

How We Create this Proposal?

- Will we just pile up existing products?

Algorithm

Model

Framework libraries

Programming Language

System software

Processing Unit (CPU, accelerator)

Memory

Communication

How We Create this Proposal?

- Will we just pile up existing products?
  - No, it would invent a naïve FAT stack

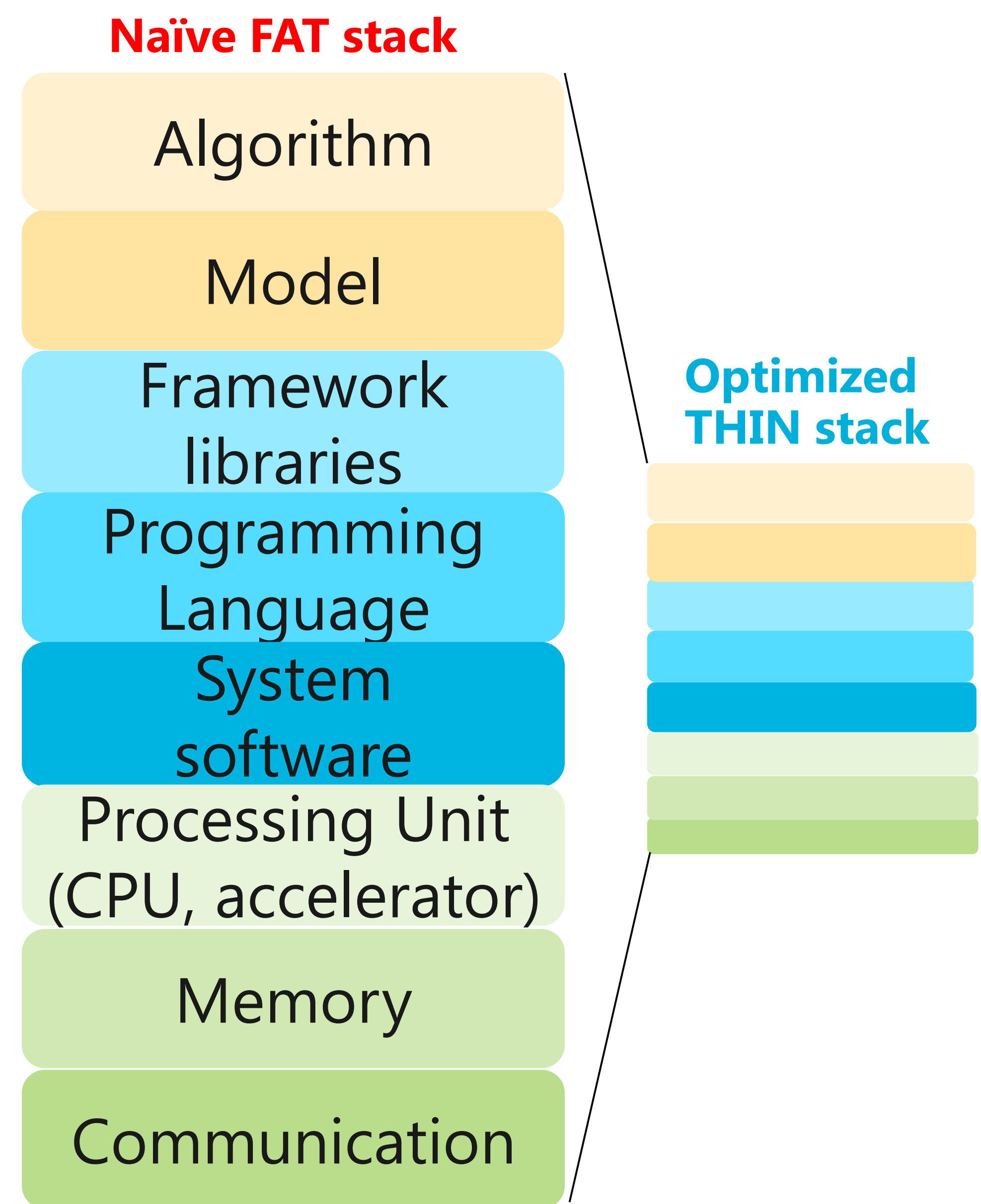
Naïve FAT stack

- Algorithm
- Model
- Framework libraries
- Programming Language
- System software
- Processing Unit (CPU, accelerator)
- Memory
- Communication

How We Create this Proposal?

- Will we just pile up existing products?
  - No, it would invent a naïve FAT stack

- I like an abstraction, but do not like to execute it "as-is"
  - Run as an optimized THIN stack with end-to-end optimizations
    - before an execution
    - during an execution
    - among executions



- Do not guess: Each layer should know everything

Our Research Challenges (1/2)

- Programming environment
  - Algorithm should be written declaratively without losing high level information

- Framework / libraries
  - Resource scheduling
  - Communication-avoiding algorithm
  - Loosely-synchronized execution model
  - Localization (e.g. tiling)

Current ML/DL frameworks have not optimized than HPC software stacks yet

Our Research Challenges (2/2)

- Programming languages / system software
  - Make HW accelerators consumable without specific code
  - Dynamic compilation or deployment for new HW accelerators
  - Automatic tuning
    - Deep learning may help too many tuning knobs in system
  - Appropriate feedbacks from HW to programming
- Debugging
  - Reproduce a bug for some converged algorithms

Recap: Takeaways

- Users, applications, and HWs are always in transition
  - Programming is becoming hard
- Let us build an end-to-end runtime system for ML and DL
  - Leave each layer to the specialist to do the best
  - Each layer should know everything for optimizations
    - Should not be isolated
- How we can make state-of-the-art technologies consumable in the system
  - Our research is here!