Choose the Best Accelerated Technology

Distributed DL/ML Solutions for HPC systems

Shailen Sobhee – Senior AI Software Solutions Engineer
shailen.sobhee@intel.com

14 October 2021

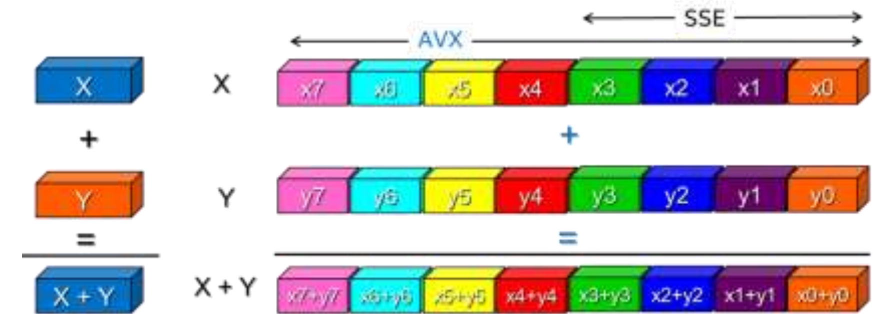
All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

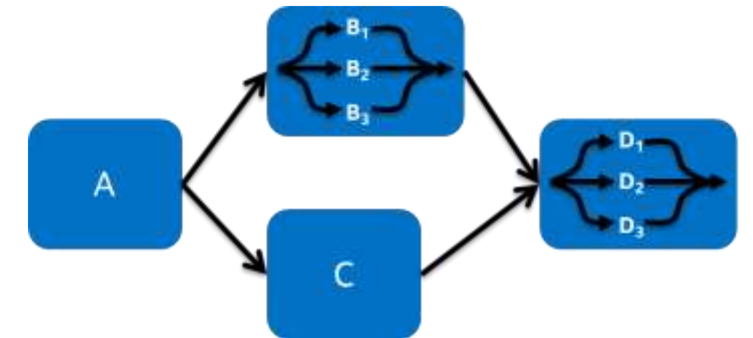
Agenda

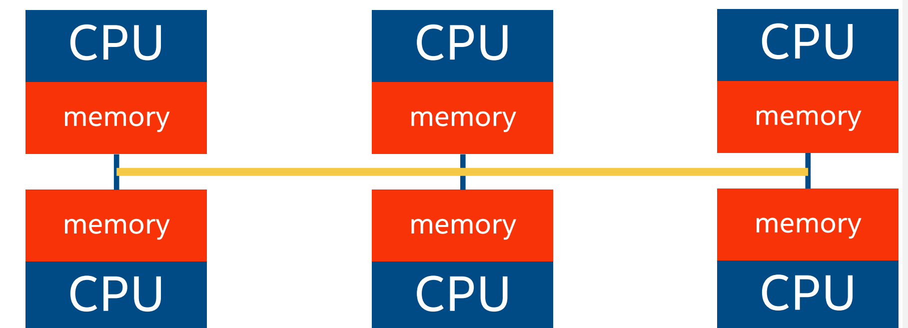
- Types of parallelism
- Distribution strategy for
  - Machine Learning
    - daal4py from oneDAL
  - Deep Learning
    - Horovod with oneCCL

Types of parallelism

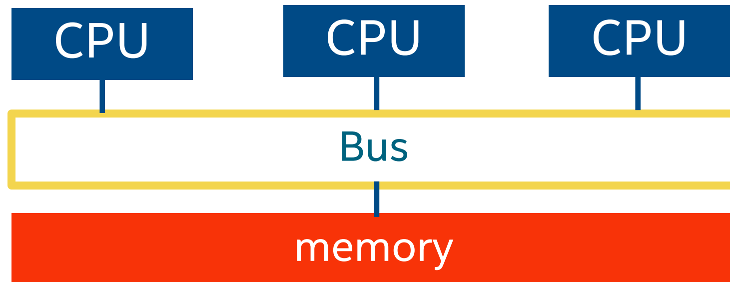
- **SIMD**: Single instruction multiple data (Data Parallel)
  - The same instruction is simultaneously applied on multiple data items
- **MIMD**: Multiple instructions multiple data (Task Parallel)
  - Different instructions on different data
- **SPMD**: Single program multiple data (MPI Parallel)
  - This is the message passing programming on distributed systems



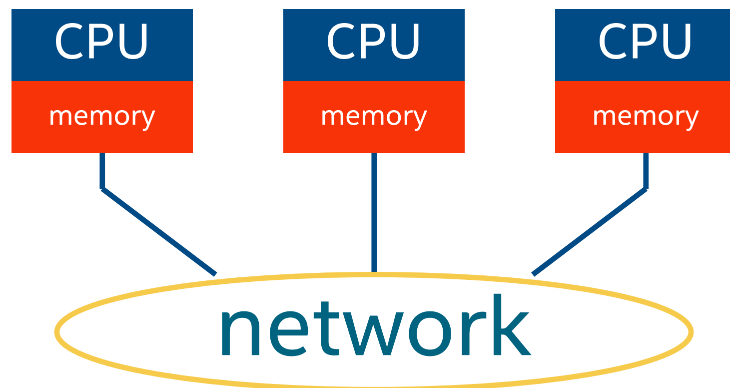




Shared vs distributed memory system



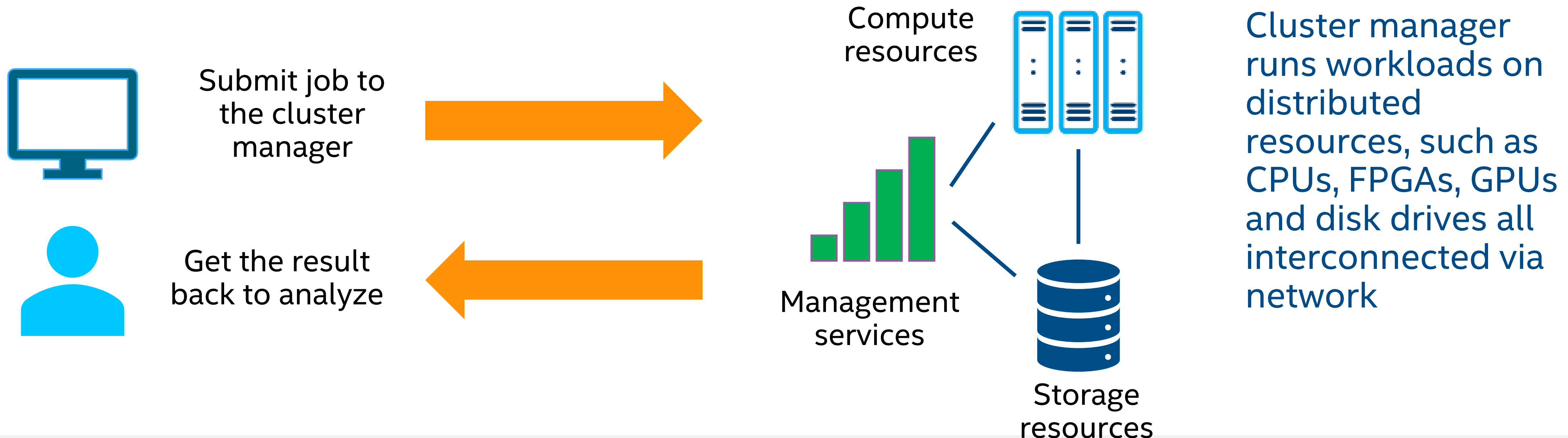
- Shared memory
  - There is a unique address space shared between the processors
  - All the processors can access the same memory



- Distributed memory
  - Each processor has its own local memory
  - Messages are exchanged between the processors to communicate the data

What is high-performance computing (HPC)?

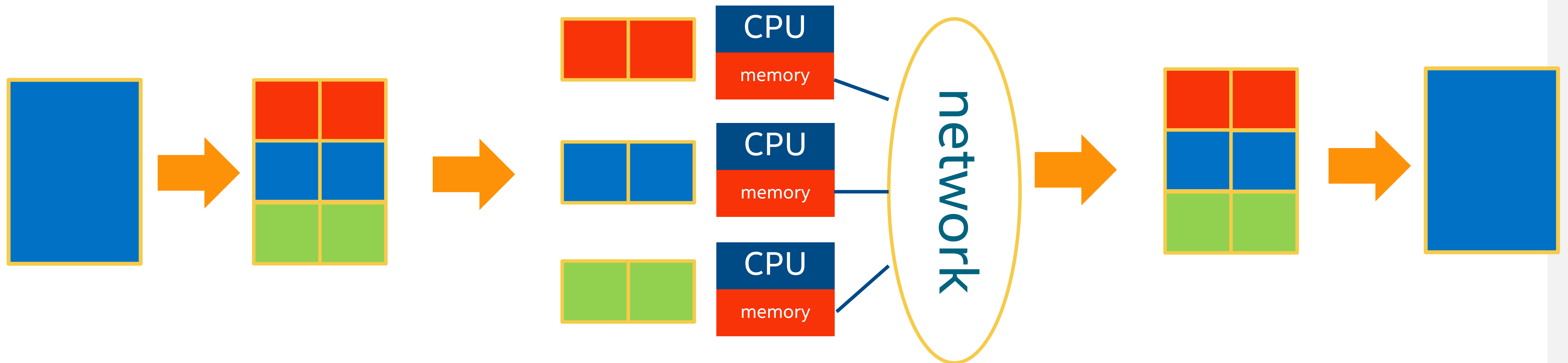
- Leveraging distributed compute resources to solve complex problems with large datasets
  - Terabytes to petabytes to zettabytes of data
  - Results in minutes to hours instead of days or weeks



Cluster manager runs workloads on distributed resources, such as CPUs, FPGAs, GPUs and disk drives all interconnected via network

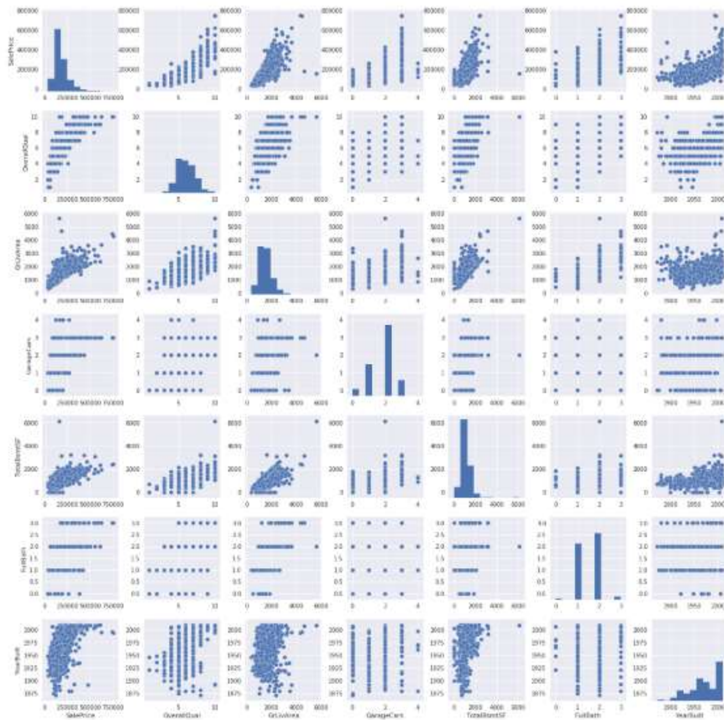
Domain decomposition method for HPC

- The domain decomposition is a technique for dividing a computational problem in several parts (domains) allowing to solve a large problem on the available resources
- *Partition* the data, assign them to each resource and associate the computation
- *Communication* happens to eventually exchange intermediate results
- *Aggregate* the results from the different resources



Distributing strategy for machine learning

From Prototype to Production



PERFORMANCE

https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python

Why distributed ML/DL (1/2)

- Most Machine Learning tasks assume the data can be easily accessible, but:
  - Data loading on a single machine can be a bottleneck in case of large amount of data
  - To run production applications large memory systems is required (data not fitting in the local computer RAM)
  - Traditional sequential algorithms are not suitable in case of distributed memory system
- Time to solution is critical on highly competitive market.

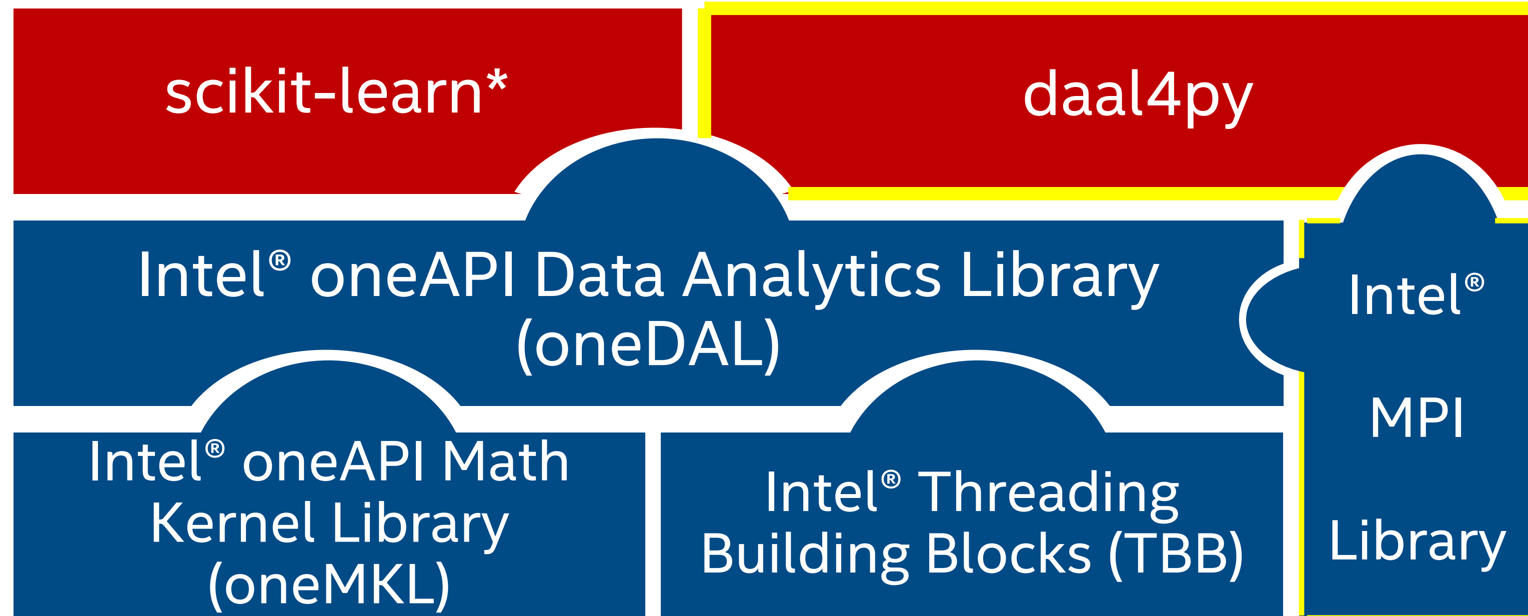
Why distributed ML/DL (2/2)

- Deep Learning training takes time:
  - Computational complexity of DL training can be up to 100+ ExaFLOP (1 ExaFLOP =10^{18} op);
  - Typical single node performance is up-to tens of TeraFLOPS (1 TF = 10^{12} op/sec);
  - Peak performance of most powerful HPC clusters is up-to tens of PetaFLOPS (1 PF = 10^{15} op/sec).
- **Time to solution is critical on highly competitive market.**

Intel® daal4py

- **daal4py** makes your Machine Learning algorithms in Python lightning fast and easy to use
- For scaling capabilities, daal4py also provides the ability to do distributed machine learning using **Intel® MPI library**
- daal4py operates in **SPMD** style (Single Program Multiple Data), which means your program is executed on several processes (e.g. similar to MPI)
- The use of MPI is not required for daal4py's SPMD-mode to work, all necessary communication and synchronization happens under the hood of daal4py
- It is possible to use daal4py and mpi4py in the same program

Scaling Machine Learning Beyond a Single Node



Simple Python* API

Powers scikit-learn*

Powered by Intel® oneDAL

<u>Scalable to multiple nodes</u>

```
> python -m daal4py <your-scikit-learn-script>
```

Monkey-patch any scikit-learn* on the command-line

```
import daal4py.sklearn
daal4py.sklearn.patch_sklearn()
```

Monkey-patch any scikit-learn* programmatically

https://intelpython.github.io/daal4py/sklearn.html#

oneAPI Data Analytics Library (oneDAL)

PCA
Kmeans
LinearRegression
Ridge
SVC
pairwise_distances
Logistic_regression_path

Scikit-Learn* **Equivalents**

USE_DAAL4PY_SKLEARN=YES

Scikit-Learn* **API** Compatible

KNeighborsClassifier
RandomForestClassifier
RandomForestRegressor

Use directly for

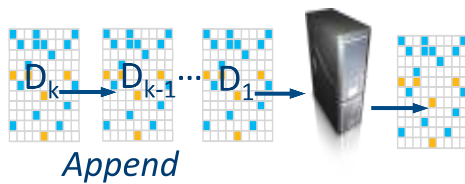
- Scaling to multiple nodes
- Streaming data
- Non-homogeneous dataframes

daal4py

oneDAL

Processing Modes

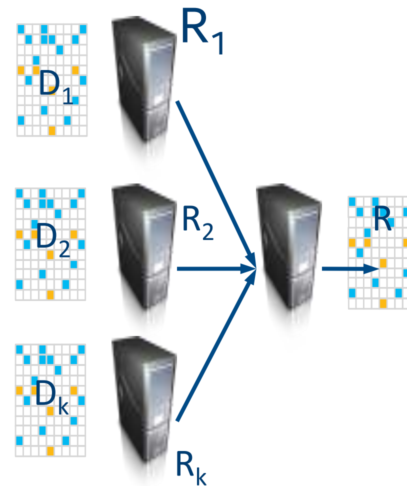
Batch Processing



$R = F(D_1,\ldots,D_k)$

```
d4p.kmeans_init(10, method="plusPlusDense")
```

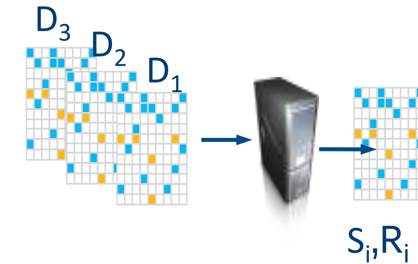
Distributed Processing



$R = F(R_1,\ldots,R_k)$

```
d4p.kmeans_init(10, method="plusPlusDense“, distributed=“True”)
```

Online Processing

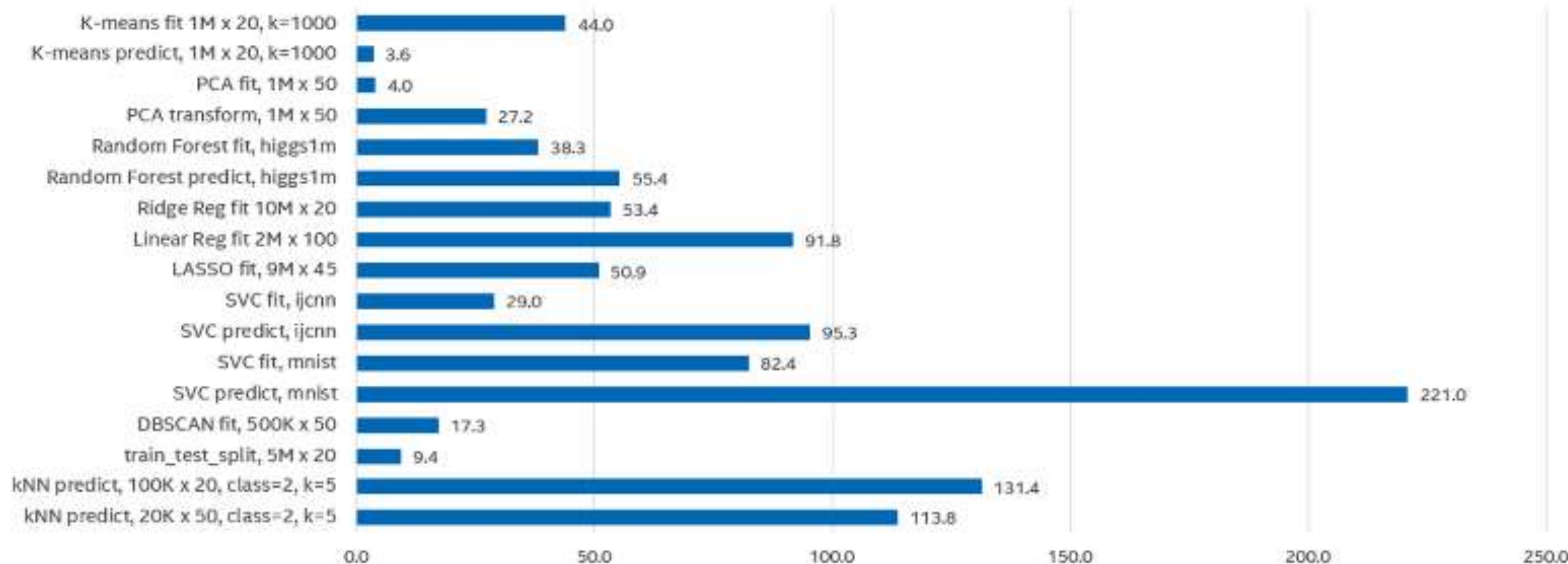


$S_{i+1} = T(S_i,D_i)$

$R_{i+1} = F(S_{i+1})$

```
d4p.kmeans_init(10, method="plusPlusDense“, streaming=“True”)
```

Speedup of oneDAL-Powered Scikit-learn* over Original Scikit-learn



Performance varies by use, configuration, and other factors. Learn more at www.intel.com/Performanceindex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

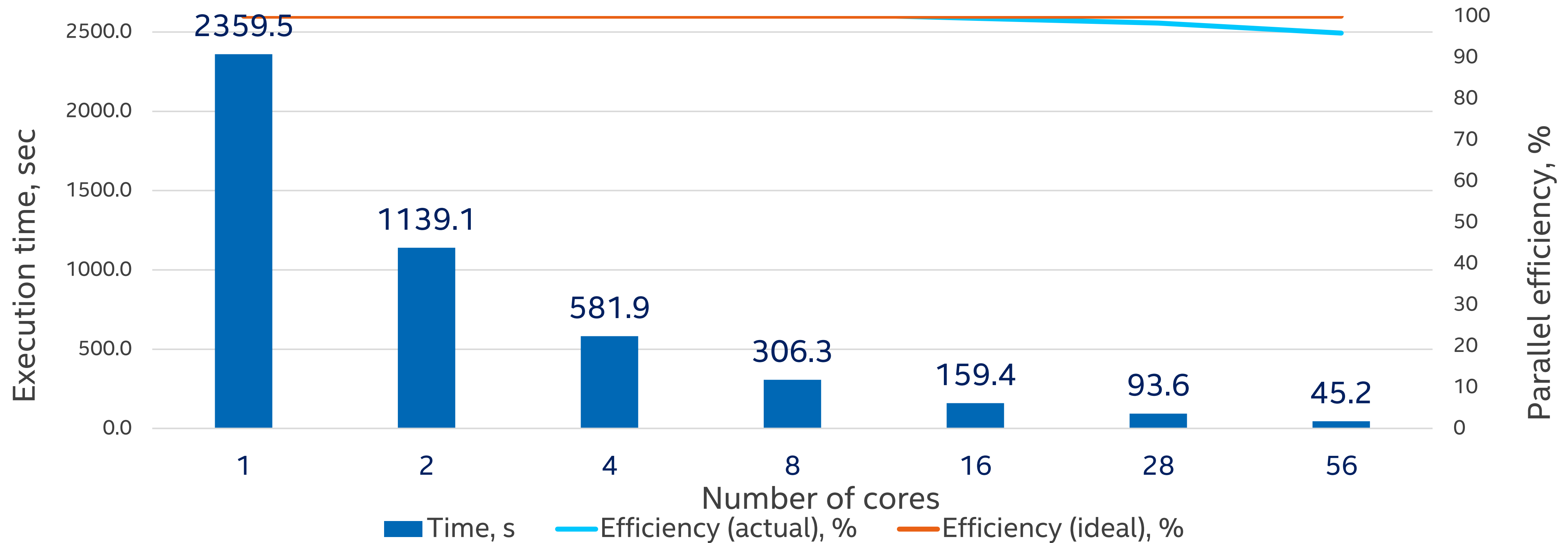
Your costs and results may vary. Intel technologies may require enabled hardware, software, or service activation.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Configuration: Testing by Intel as of 10/23/2020. Intel® oneAPI Data Analytics Library 2021.1 (oneDAL), Scikit-learn 0.23.1, Intel® Distribution for Python 3.8; Intel(R) Xeon(R) Platinum 8280LCPU @ 2.70GHz, 2 sockets, 28 cores per socket, 10M samples, 10 features, 100 clusters, 100 iterations, float32.

oneDAL K-Means Fit, Cores Scaling

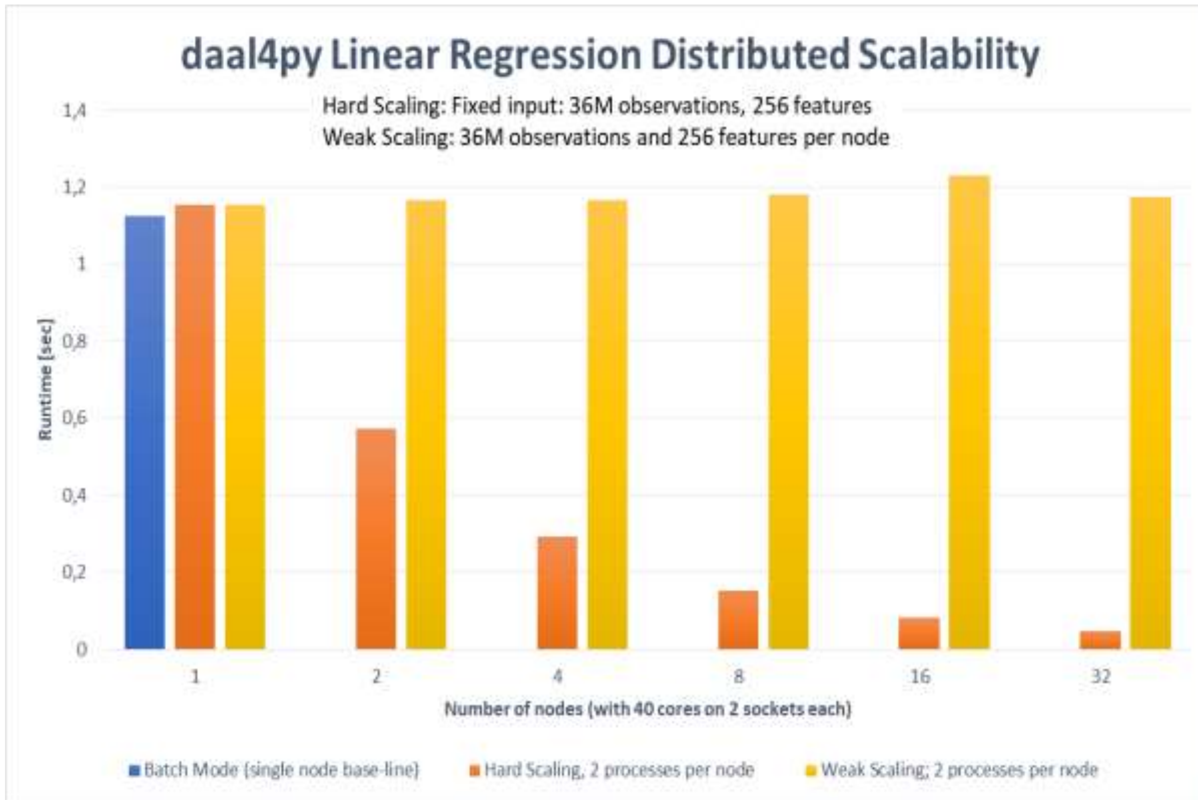
(10M samples, 10 features, 100 clusters, 100 iterations, float32)

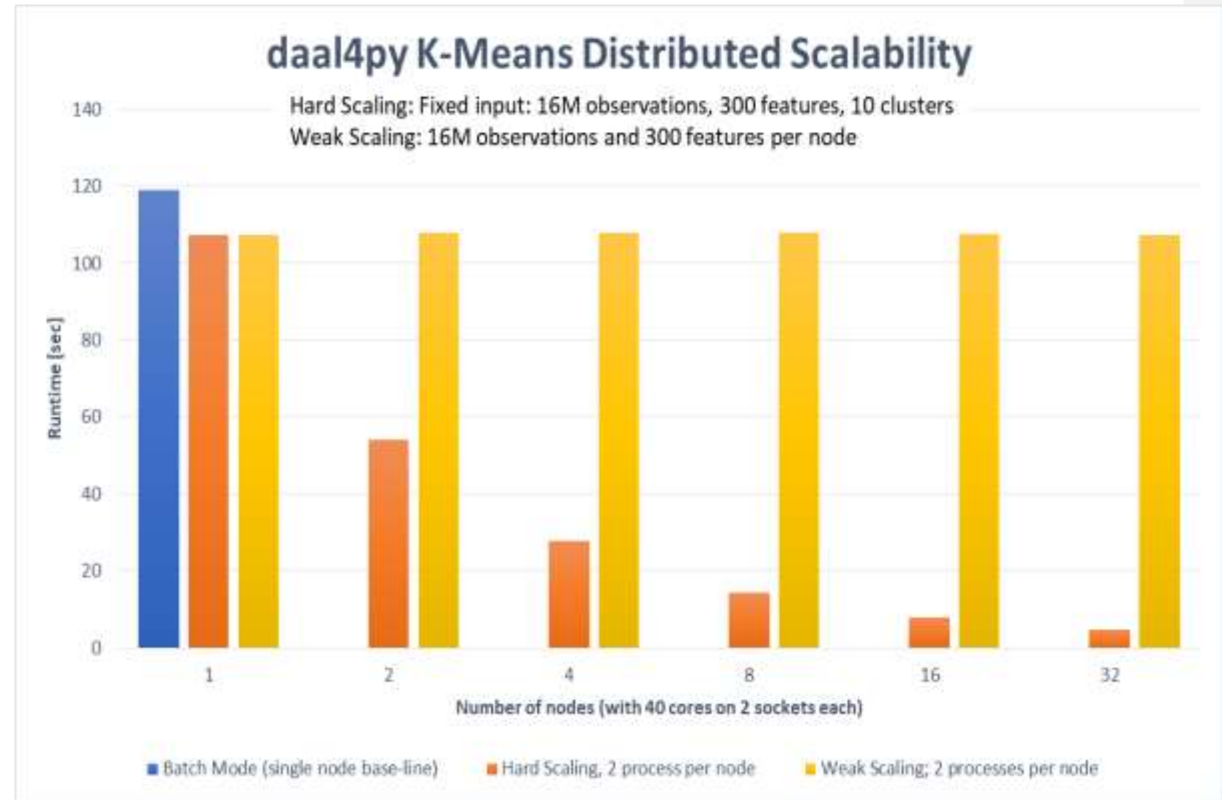


Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.
Your costs and results may vary. Intel technologies may require enabled hardware, software, or service activation.
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
Configuration: Testing by Intel as of 10/23/2020. Intel® oneAPI Data Analytics Library 2021.1 (oneDAL); Intel® Xeon® Platinum 8280LCPU @ 2.70GHz, 2 sockets, 28 cores per socket, 10M samples, 10 features, 100 clusters, 100 iterations, float32.

Strong & Weak Scaling via daal4py

Hardware	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz, EIST/Turbo on
	2 sockets, 20 Cores per socket
	192 GB RAM
	16 nodes connected with Infiniband
Operating System	Oracle Linux Server release 7.4
Data Type	double





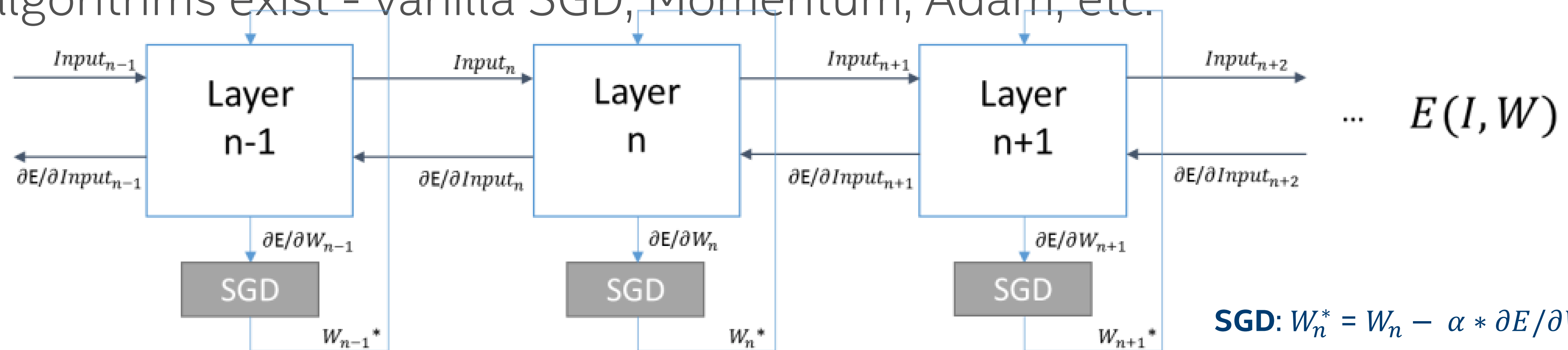
On a 32-node cluster (1280 cores) daal4py computed linear regression of 2.15 TB of data in 1.18 seconds and 68.66 GB of data in less than 48 milliseconds.

On a 32-node cluster (1280 cores) daal4py computed K-Means (10 clusters) of 1.12 TB of data in 107.4 seconds and 35.76 GB of data in 4.8 seconds.

Distribution strategy for deep learning

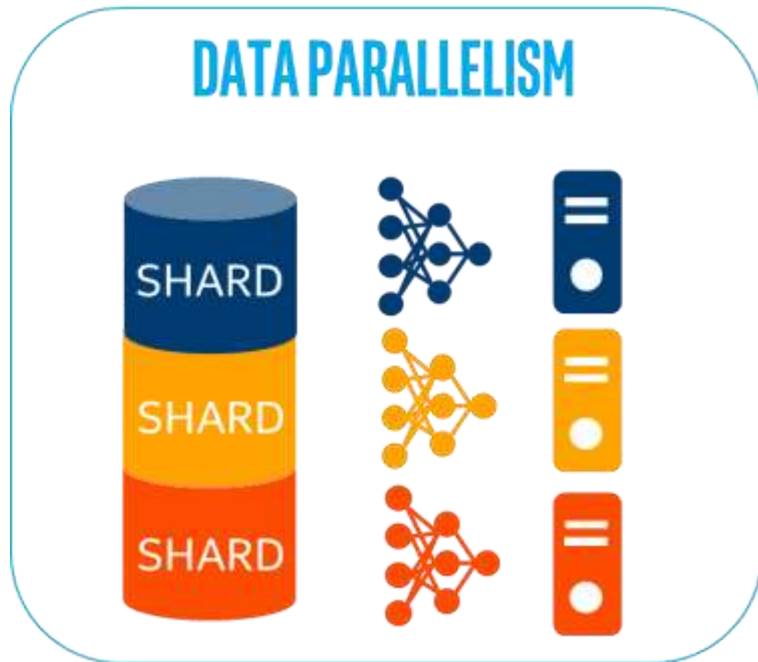
Deep Learning Training procedure

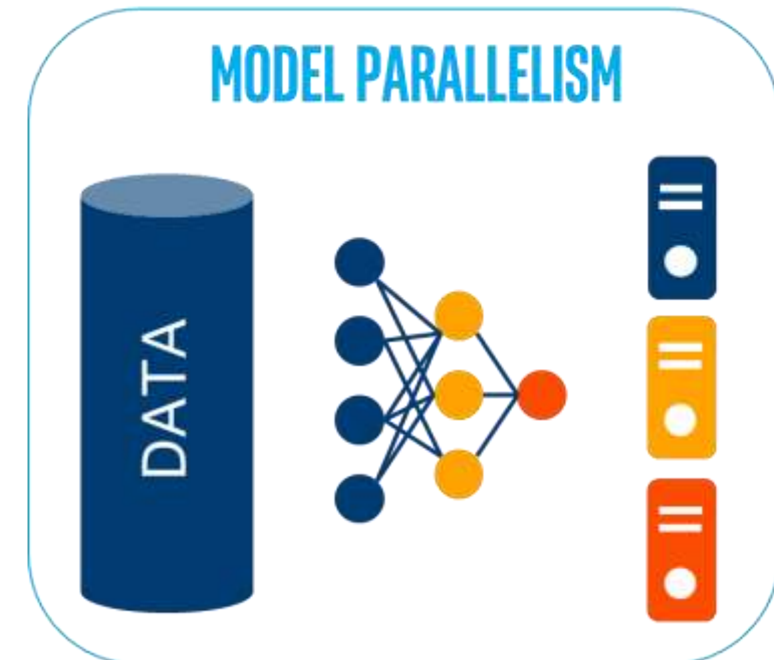
- **Forward propagation**: calculate loss function based on the input batch and current weights;
- **Backward propagation**: calculate error gradients w.r.t. weights for all layers (using chain rule);
- **Weights update**: use gradients to update weights; there are different algorithms exist - vanilla SGD, Momentum, Adam, etc.



SGD: $W_n^* = W_n - \alpha * \partial E / \partial W_n$ or variants

Neural Network parallelism





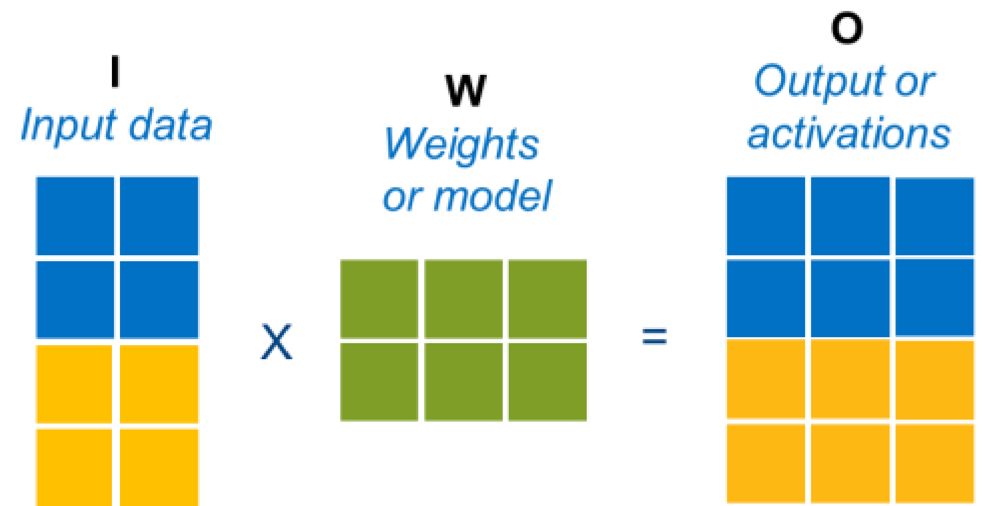
Data is processed in increments of N. Work on minibatch samples and distributed among the available resources.

The work is divided according to the neurons in each layer. The sample minibatch is copied to all processors which compute part of the DNN.

source: https://arxiv.org/pdf/1802.09941.pdf

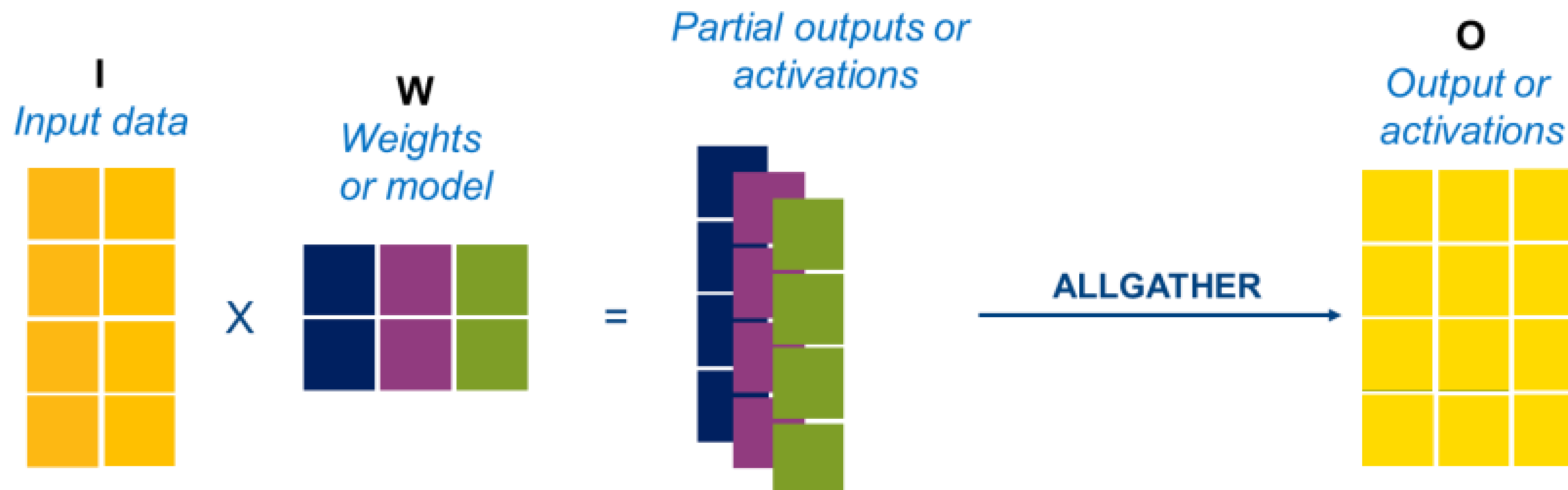
Multi-node parallelization

- **Data parallelism**:
  - Replicate the model across nodes;
  - Feed each node with its own batch of input data;
  - Communication for gradients is required to get their average across nodes;
  - Can be either
    - *AllReduce* pattern
    - *ReduceScatter + AllGather* patterns



Multi-node parallelization

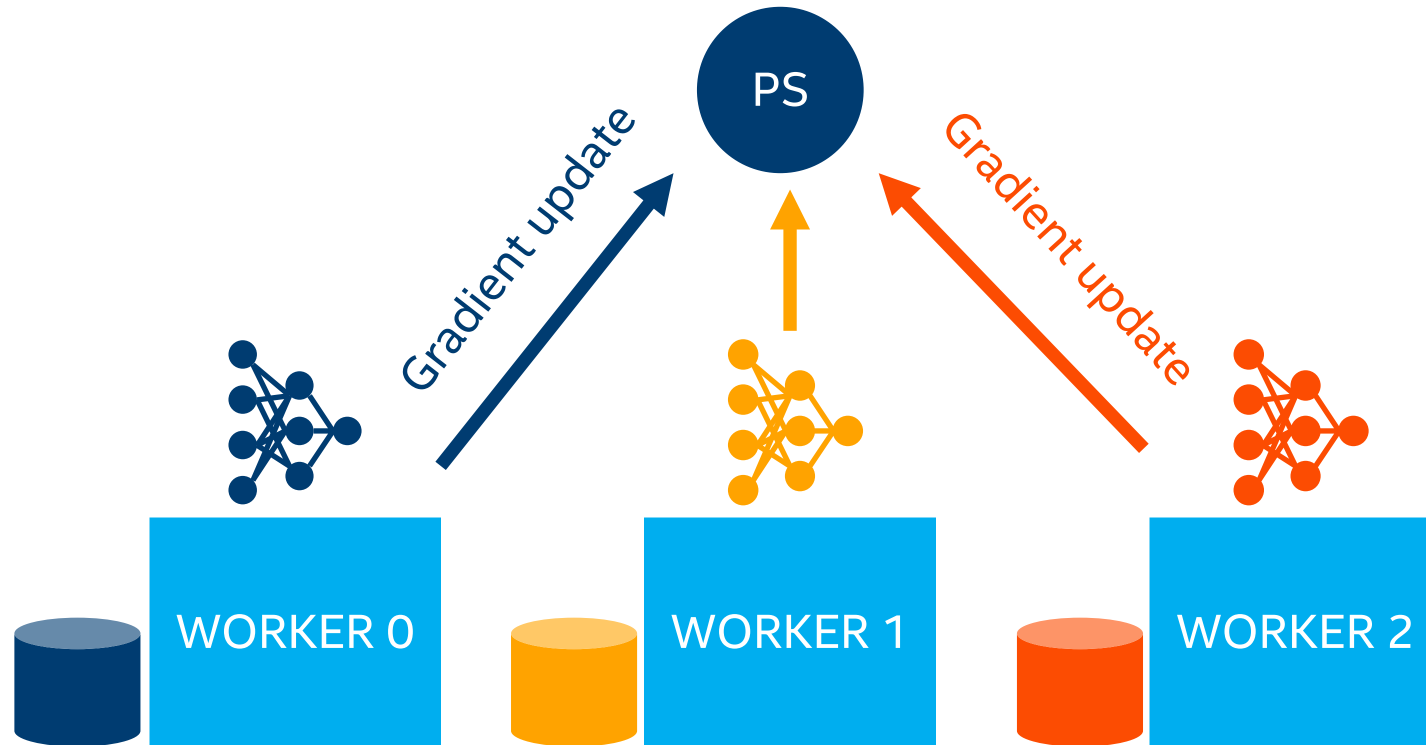
- **Model parallelism**:
  - Model is split across nodes;
  - Feed each node with the same batch of input data;
  - Communication for partial activations is required to gather the result;



Multi-node parallelization

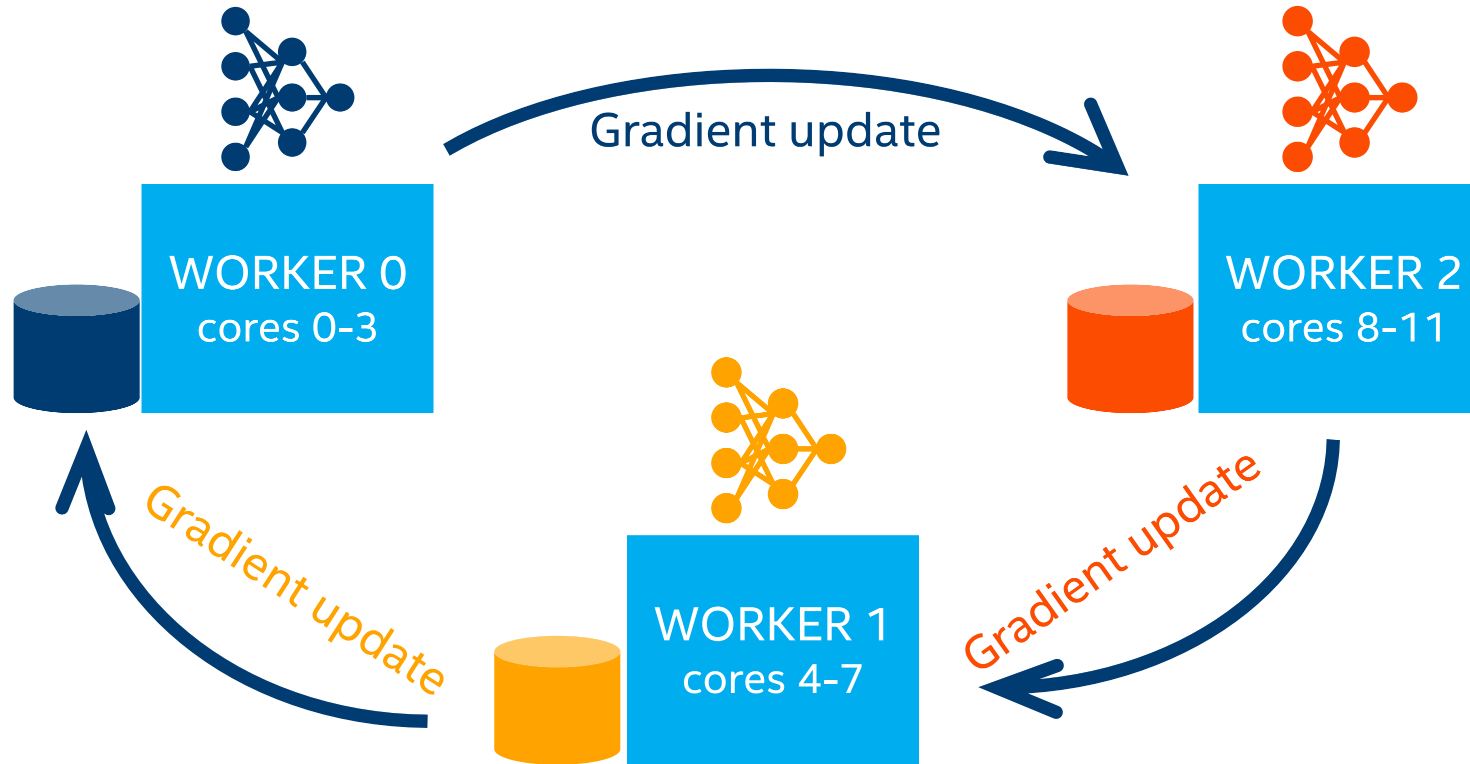
- **What parallelism flavor to use?**
  - Use model parallelism when volume of gradients is much higher than volume of activations or when model doesn't fit memory;
  - Use data parallelism otherwise;
  - Parallelism choice affects activations/gradients ratio
    - Data parallelism at scale makes activations << weights
    - Model parallelism at scale makes weights << activations
  - There're also other parallelism flavors – pipelined, spatial, etc.

Parameter Server



Tree using gRPC calls

Horovod



Ring All-Reduce using MPI

https://arxiv.org/abs/1802.05799v3

Distributed Training for Deep Neural Network

Intel® oneAPI Collective Communications Library (oneCCL)

Intel® oneAPI Collective Communications Library

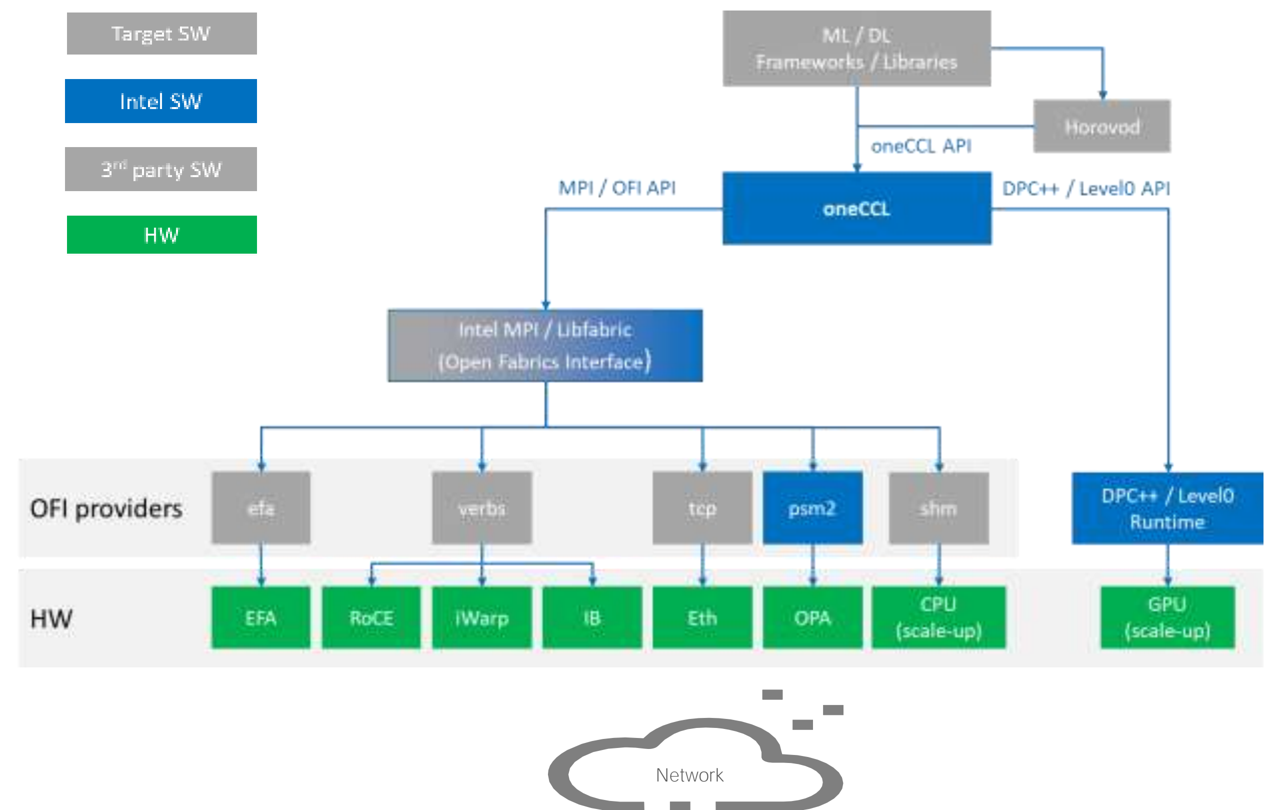
Optimize Communication Patterns

oneCCL provides optimized communication patterns for high performance on Intel CPUs & GPUs to distribute model training across multiple nodes

Transparently supports many interconnects, such as Intel® Omni-Path Architecture, InfiniBand, & Ethernet

Built on top of lower-level communication middleware-MPI & libfabrics

Enables efficient implementations of collectives used for deep learning training-all-gather, all-reduce, & reduce-scatter



Intel® oneAPI Collective Communications Library

Key Features (part 1/2)

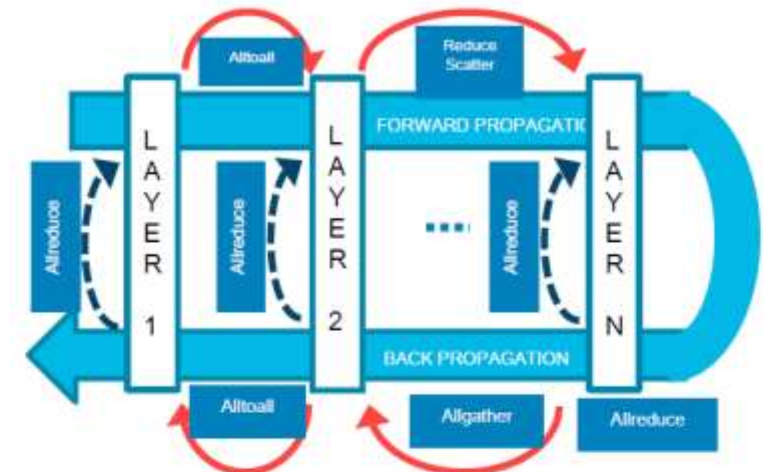
Enables efficient implementations of collectives used for deep learning training – all-gather, all-reduce, and more

oneCCL is designed for easy integration into deep learning (DL) frameworks

Provides C++ API and interoperability with DPC++

Supported Collectives

- Allgatherv
- Allreduce
- Alltoallv
- Broadcast
- Reduce
- ReduceScatter



Intel® oneAPI Collective Communications Library

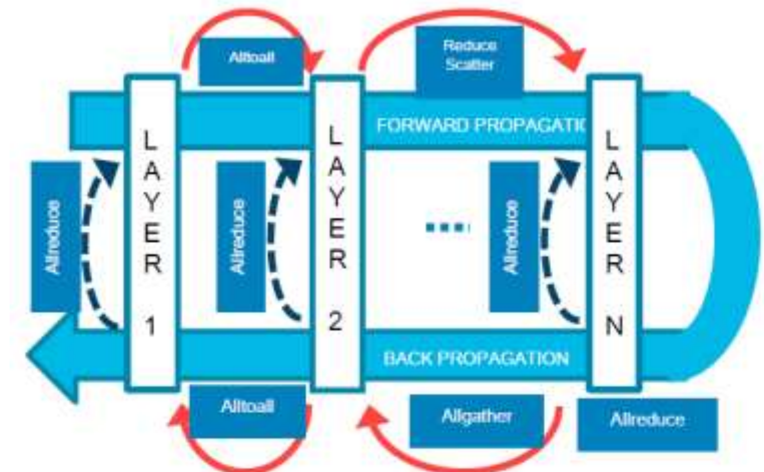
Key Features (part 2/2)

Deep Learning Optimizations include:

- Asynchronous progress for compute communication overlap
- Dedication of cores to ensure optimal network use
- Message prioritization, persistence, and out-of-order execution
- Collectives in low-precision data types

Supported Collectives

- Allgatherv
- Allreduce
- Alltoallv
- Broadcast
- Reduce
- ReduceScatter



Message Passing Interface (MPI)

```
$ mpirun -H 192.168.1.100,192.168.1.105 hostname
aipg-infra-07.intel.com
aipg-infra-09.intel.com
```

```
$ mpirun -H host1,host2,host3 python hello.py
Hello World!
Hello World!
Hello World!
```

Changes to TensorFlow

1

```
import tensorflow as tf

import horovod.tensorflow as hvd
```

2

```
hvd.init()
```

3

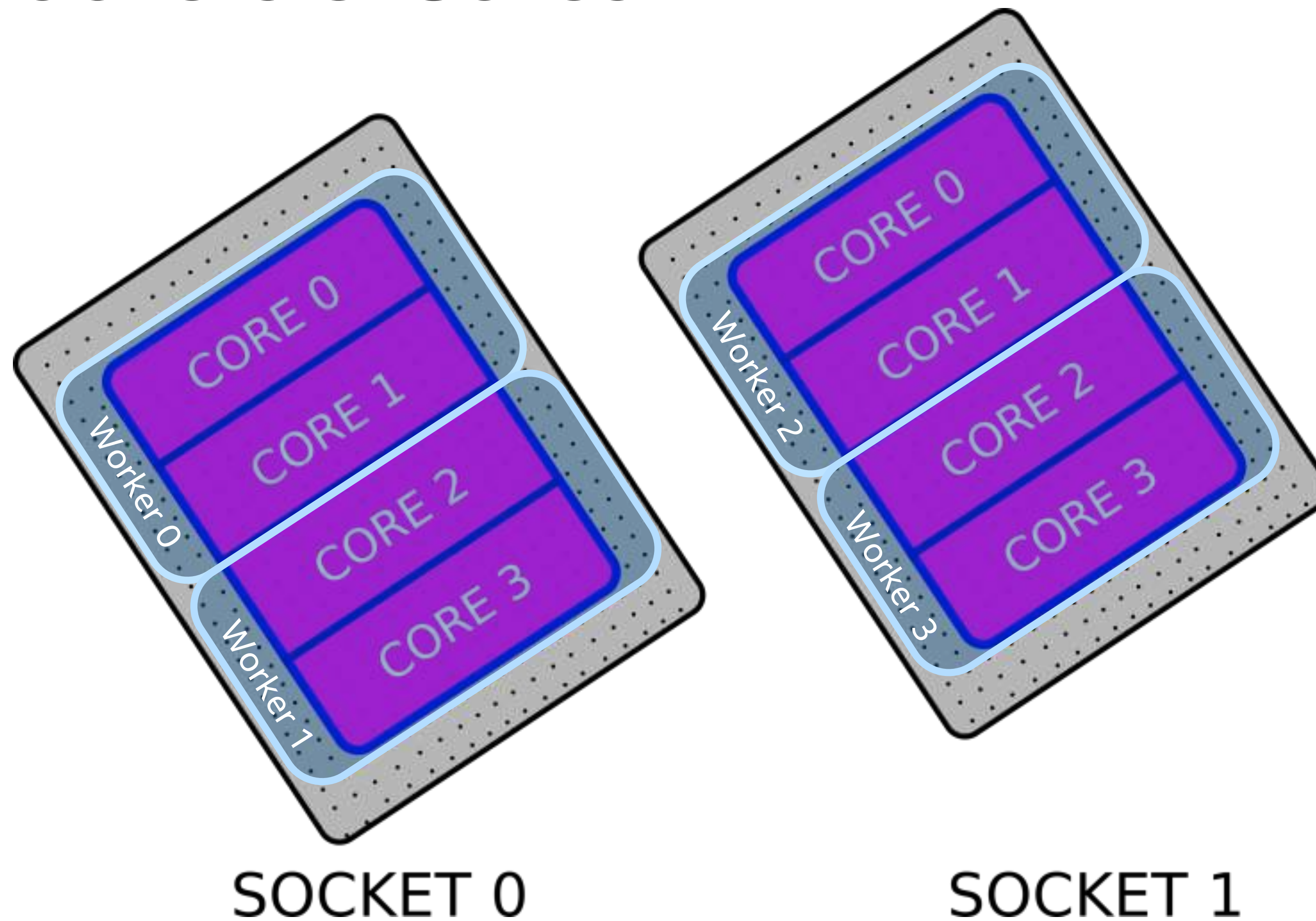
```
opt = tf.train.AdagradOptimizer(0.01 * hvd.size())

opt = hvd.DistributedOptimizer(opt)
```

4

```
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
```

Sockets & Cores



SOCKET

Receptacle on the motherboard for one physically packaged processor.

CORE

A complete private set of registers, execution units, and queues to execute a program.

Multiple workers per CPU with Intel MPI

```
$ mpirun
-H hostA, hostB, hostC
-n 6
-ppn 2
-print-rank-map
-genv I_MPI_PIN_DOMAIN=socket
-genv OMP_NUM_THREADS=24
-genv OMP_PROC_BIND=true
-genv KMP_BLOCKTIME=1
python train_model.py
```

Intel MPI

Distributing with oneAPI AI Analytics toolkit

- Source the oneAPI vars

```
source /opt/intel/oneapi/setvars.sh --ccl-configuration=cpu_icc --force
source /opt/intel/oneapi/mpi/latest/env/vars.sh -i_mpi_library_kind=release_mt
```

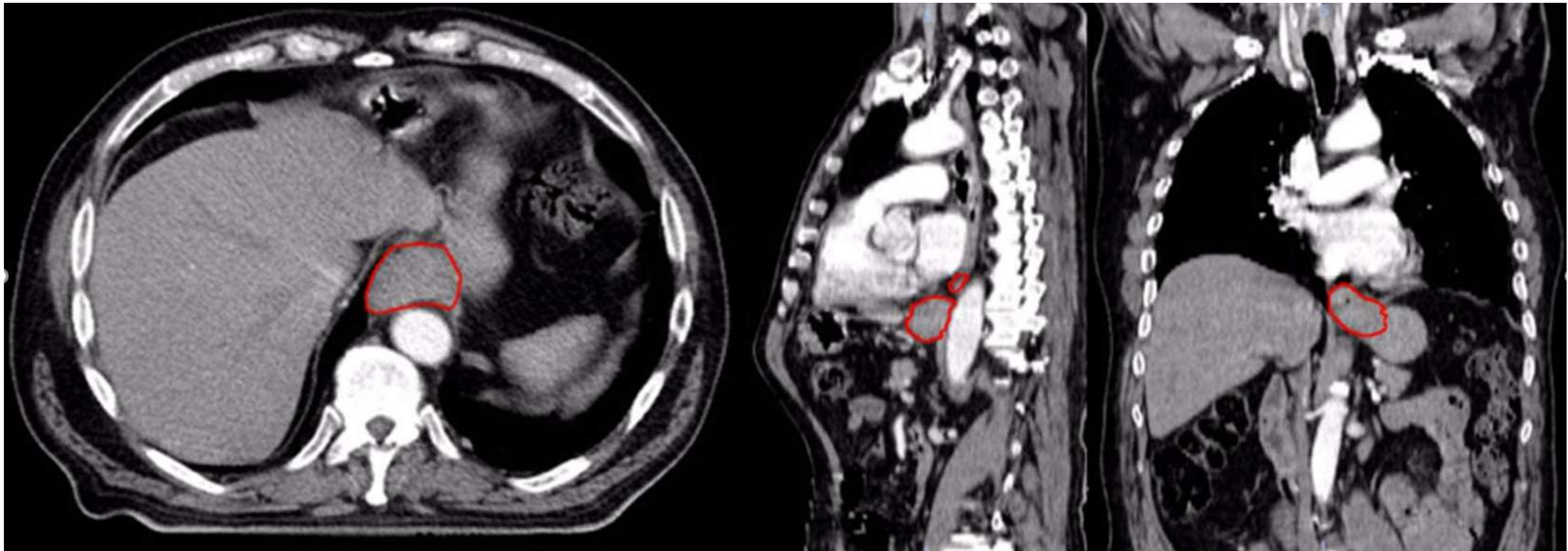
- Activate conda environment
- Use MPI to start distributing.

```
mpiexec –np X –hosts pc1,pc2 python myscript.py
```

CASE-STUDY

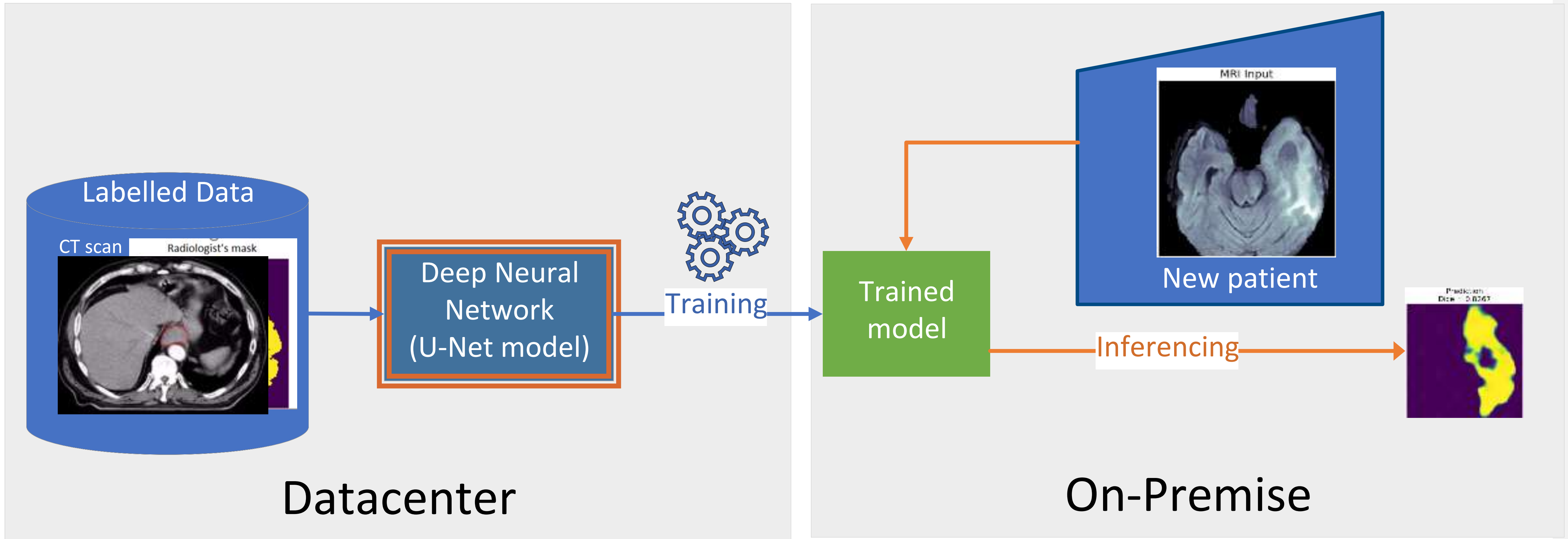
Engagement overview on the ASPIRE project

Problem Statement: AI Algorithm to **segment*** the Metabolic Tumour Volume (MTV) in oesophageal cancer

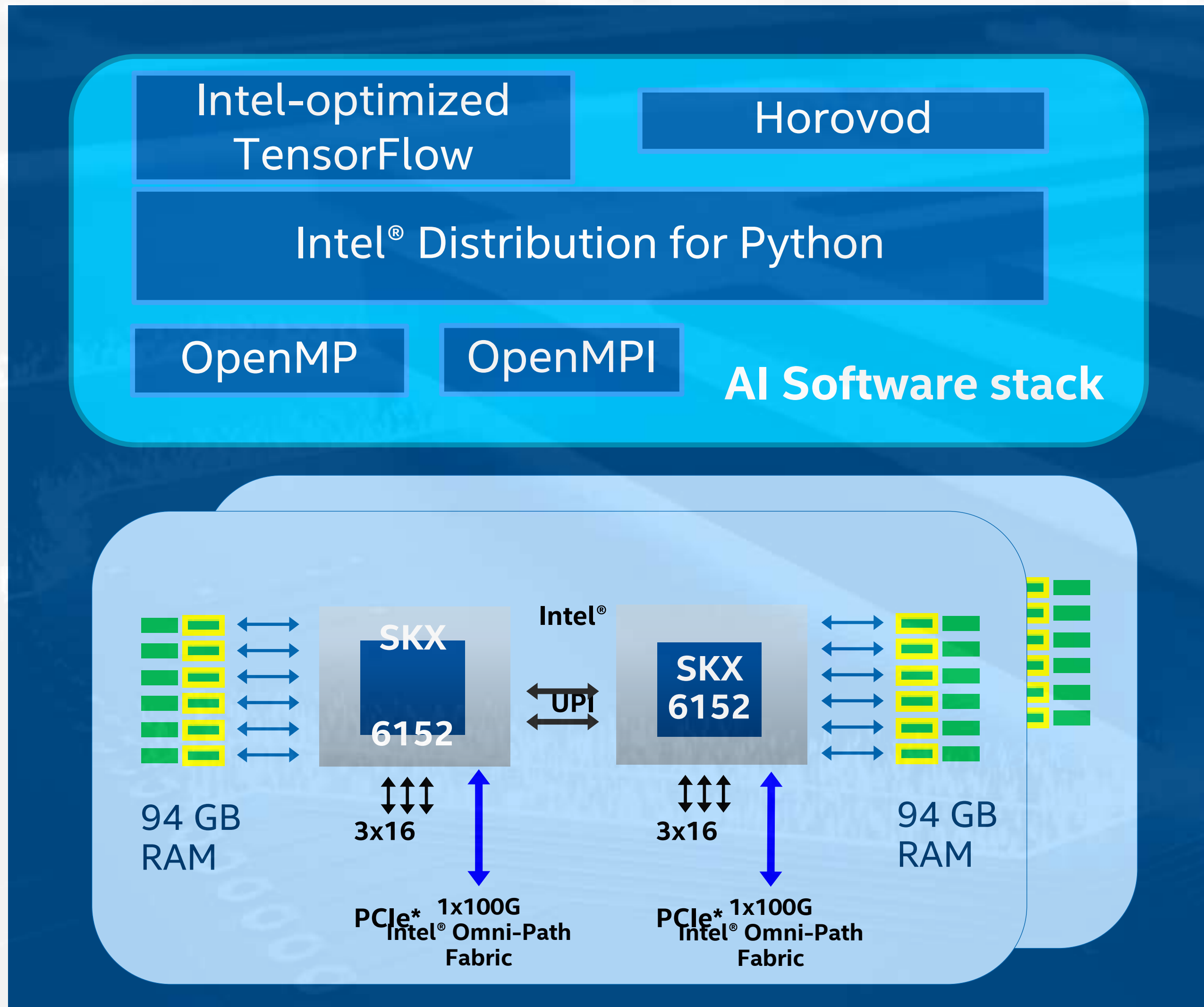


* **Segmentation**: Find the contour of the tumor on the CT scan

AI solution to for tumor segmentation:



Training Platform configuration



AI Software stack Configuration

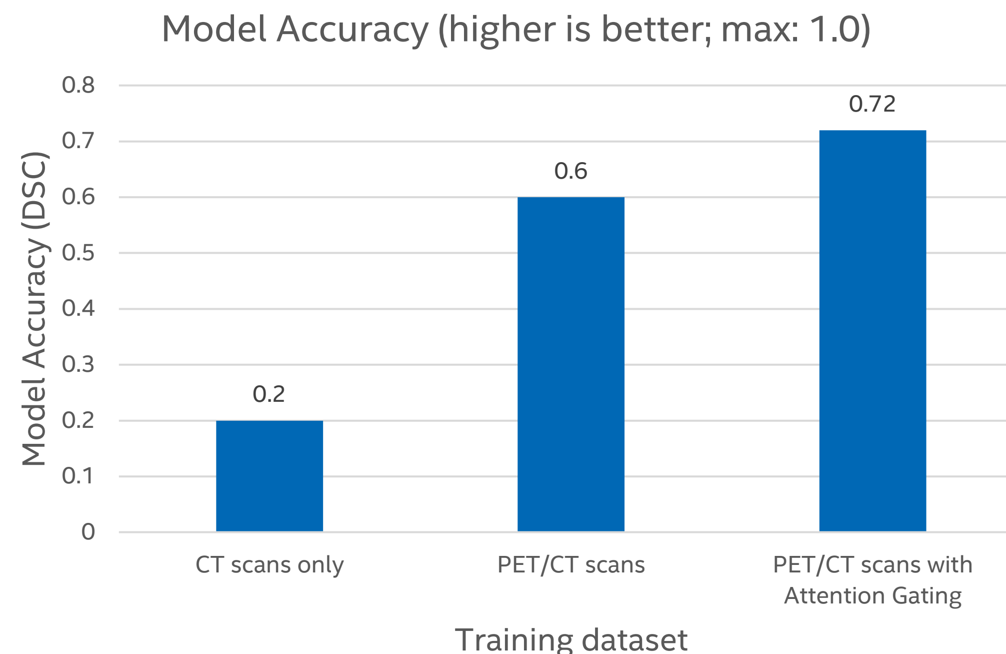
- Intel® Distribution for Python **3.7**
- Intel-optimized TensorFlow **1.12**
- OpenMPI **4.0.2**
- OpenMP **(gcc 4.8.5)**
- Horovod **0.18.2**

Hardware

- 2 Intel® Xeon® 6152 2S nodes (22 cores per socket) => 88 cores total
- 188 GB RAM / node (376 GB RAM total)
- 300 GB SSD / node (600 GB total)
- Omni-Path interconnect fabric

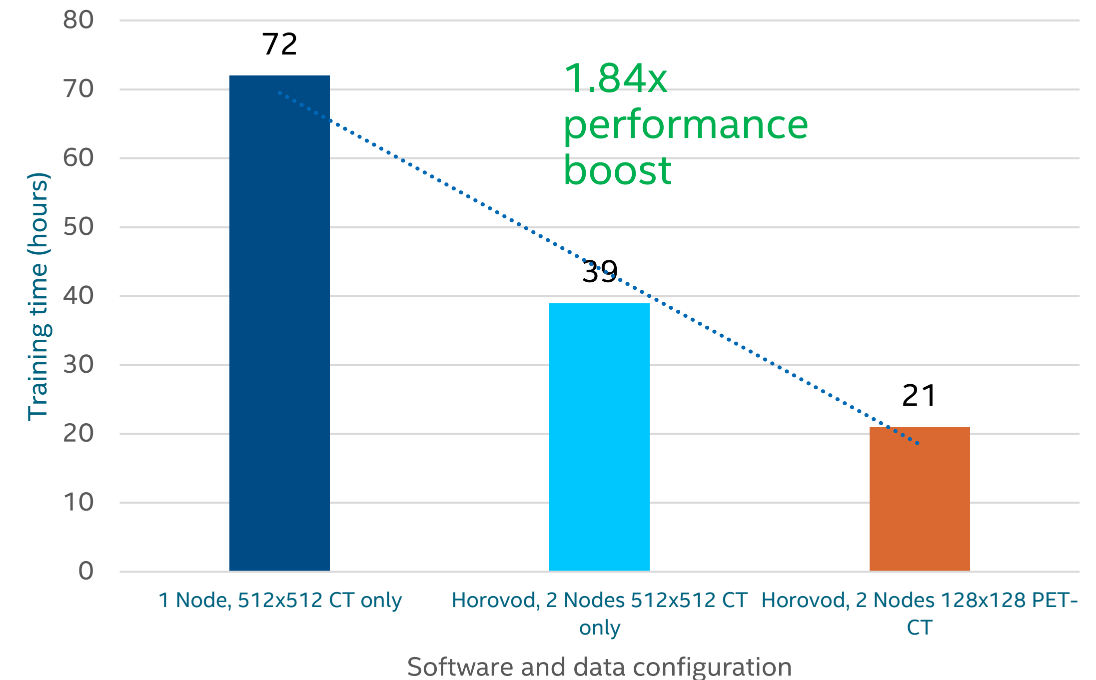
Model accuracy improvement

- Using CT scans alone gave low accuracy*
  - We included an additional PET channel
- U-Net Neural Network optimizations
  - Custom dropout rates on individual layers
  - Custom loss function (Dice Score Metric, Jaccard Metric and Tversky Loss)
  - Attention-Gating (available in TF 2.0)



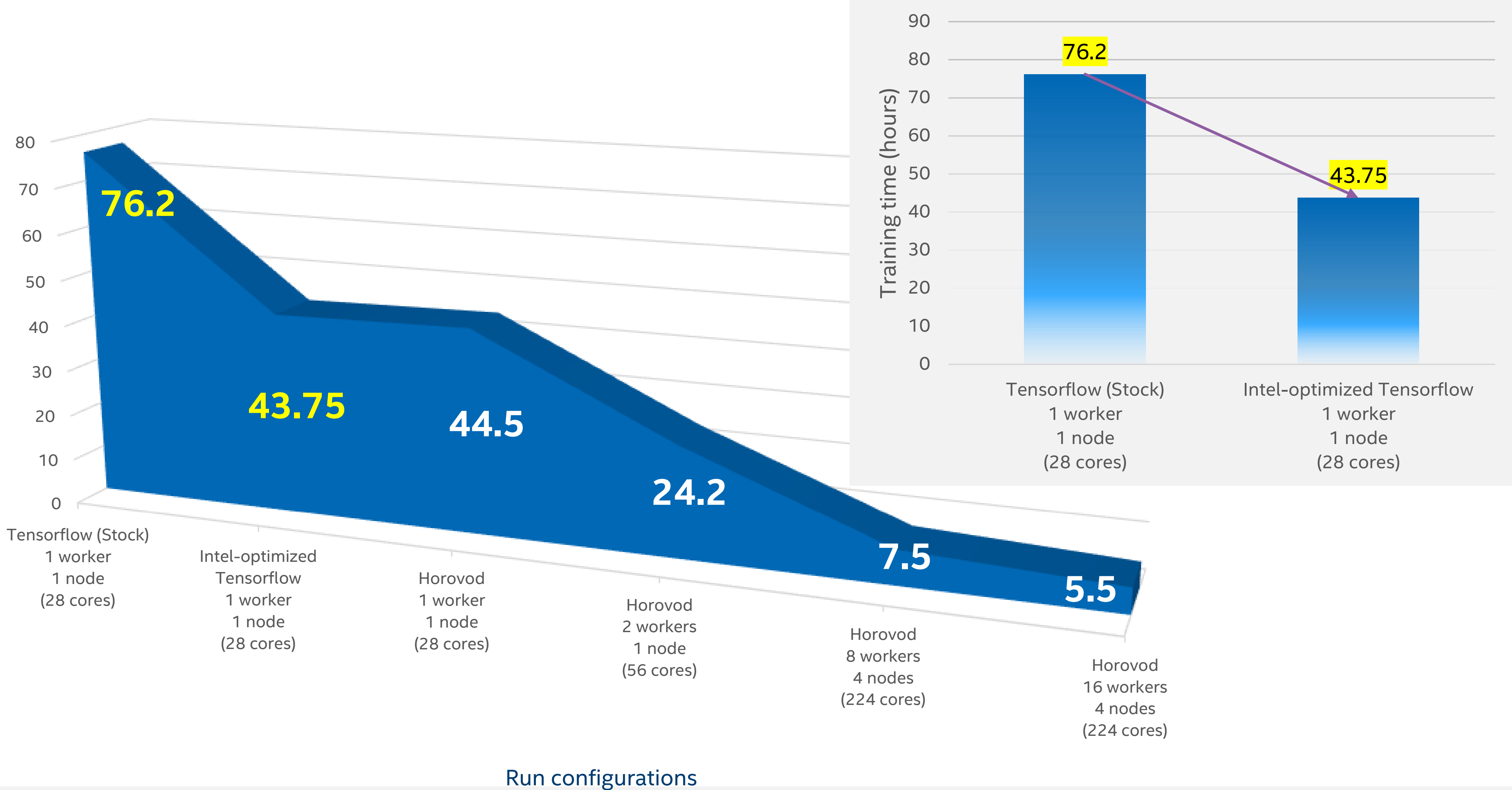
Training performance improvement

- Intel-optimized TensorFlow with node-level optimizations
  - OMP_NUM_THREADS = #physical cores
  - KMP_BLOCKTIME=1
  - KMP_AFFINITY=granularity=fine,compact
  - INTER and INTRA THREADS
- **Scaling-out:** Horovod – MPI parameter optimizations
  - 72 hours -> 39 hours



Training dataset:
3489 CT images, 3489 PET images

Performance results (Brain Tumor)



intel®

Feedback Survey