Intel® oneAPI AI Analytics Toolkit on Data Center

GPUs

Yuning Qiel
Orel Yehuda
Louie Tsai
Jeff Rodgers

@Intel AI Customer Engineering Team
Agenda

- oneAPI - AI Analytics Toolkit
- Hands-on Environment Setup
- Intel Optimizations for PyTorch on XPU
- Intel Optimizations for TensorFlow on XPU
- Distributed DL on XPU
- Intel Distribution for Python (IDP) on XPU
OneAPI - AI Analytics Toolkit Overview
Modern Applications Demand Increased Processing

Diverse accelerators needed to meet today’s performance requirements:
48% of developers target heterogeneous systems
that use more than one kind of processor or core\(^1\)

CPU  GPU  FPGA  Other Accelerators

Developer Challenges: Multiple Architectures, Vendors, and Programming Models

Open, Standards-based, Multiarchitecture Programming
**oneAPI Industry Initiative**

**Break the Chains of Proprietary Lock-in**
- C++ programming model for multiple architectures and vendors
- Cross-architecture code reuse for freedom from vendor lock-in

**Realize all the Hardware Value**
- Performance across CPU, GPUs, FPGAs, and other accelerators
- Expose and exploit cutting-edge features of the latest hardware

**Develop & Deploy Software with Peace of Mind**
- Open industry standards provide a safe, clear path to the future
- Interoperable with familiar languages and programming models including Fortran, Python, OpenMP, and MPI
- Powerful libraries for acceleration of domain-specific functions

---

Application Workloads Need Diverse Hardware

Middleware & Frameworks

Direct Programming

- SYCL (C++)

API-Based Programming
- Math
- Threading
- Parallel STL
- Ray Tracing
- Analytics/ML
- DNN
- MLComm
- Volumetric Rendering
- Video Processing
- Signal Processing
- Image Processing
- Image Denoise

Low-Level Hardware Interface (oneAPI Level Zero)

- CPU
- GPU
- FPGA
- Other Accelerators

---

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models
oneAPI Industry Momentum

<table>
<thead>
<tr>
<th>End Users</th>
<th>National Labs</th>
<th>ISVs &amp; OSVs</th>
<th>OEMs &amp; SIs</th>
<th>Universities &amp; Research Institutes</th>
<th>CSPs &amp; Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accrad</td>
<td>Argonne</td>
<td>codeplay</td>
<td>Dell Technologies</td>
<td>Berkeley</td>
<td>Google Cloud</td>
</tr>
<tr>
<td>@rchanan</td>
<td>Oak Ridge National</td>
<td>Red Hat</td>
<td>Atos</td>
<td>UCDavis</td>
<td>Microsoft Azure</td>
</tr>
<tr>
<td>Brightskies</td>
<td>Laboratory</td>
<td>Canonical</td>
<td>Lenovo</td>
<td>University of Illinois</td>
<td>Alibaba Cloud</td>
</tr>
<tr>
<td>kt</td>
<td></td>
<td>CATI</td>
<td>MEGWARE</td>
<td>University of Maryland</td>
<td>TensorFlow</td>
</tr>
<tr>
<td>Philips</td>
<td></td>
<td>Codeplay</td>
<td>Hewlett Packard</td>
<td>University of Nebraska</td>
<td>Taboola</td>
</tr>
<tr>
<td>Verizon</td>
<td></td>
<td>Canonical</td>
<td>HCL</td>
<td>University of Saskatchewan</td>
<td>Tencent</td>
</tr>
<tr>
<td>MediaKind</td>
<td></td>
<td>CERTIFACE</td>
<td>Lenovo</td>
<td>University of Washington</td>
<td>DataRobot</td>
</tr>
<tr>
<td>Autodesk</td>
<td></td>
<td>CGG</td>
<td>IBM</td>
<td>University of Wisconsin</td>
<td>Naver CLOVA</td>
</tr>
<tr>
<td>ARNOLD</td>
<td></td>
<td>Eurecom</td>
<td>Lenovo</td>
<td>UCR</td>
<td>PyTorch</td>
</tr>
<tr>
<td>Reuters</td>
<td></td>
<td>Illumination MacGuff</td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>SABREX</td>
<td></td>
<td>Allegro.ai</td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>SAMSUNG MEDISON</td>
<td></td>
<td>TANGENIMATION</td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>EURECOM</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>BENTLEY</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Chinese Petroleum Corporation</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>CINESITE</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Laika</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Max Planck Research Laboratory</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>MEDION</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Illumination MacGuff</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Chaordata</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Veranka</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Oak Ridge National Laboratory</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Peraton Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Sankhya Sutra</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Laboratories National de</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Computations Scientifiques</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Universidad Nacional de</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Colombia</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Fundación de la Ciencia y</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Tecnología</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Oak Ridge National Laboratory</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
<tr>
<td>Lenovo Labs</td>
<td></td>
<td></td>
<td>Lenovo</td>
<td>University of Wisconsin</td>
<td></td>
</tr>
</tbody>
</table>

These organizations support the oneAPI initiative for a single, unified programming model for cross-architecture development. It does not indicate any agreement to purchase or use of Intel's products. *Other names and brands may be claimed as the property of others.
Accelerating Choice with SYCL
Khronos Group Standard

- Open, standards-based
- Multiarchitecture performance
- Freedom from vendor lock-in
- Comparable performance to native CUDA on Nvidia GPUs
- Extension of widely used C++ language
- Speed code migration via open source SYCLomatic or Intel® DPC++ Compatibility Tool

Architectures

<table>
<thead>
<tr>
<th>Intel</th>
<th>Nvidia</th>
<th>AMD CPU/GPU</th>
<th>RISC-V</th>
<th>ARM Mali</th>
<th>PowerVR</th>
<th>Xilinx</th>
</tr>
</thead>
</table>

Testing Date: Performance results are based on testing by Intel as of April 15, 2023 and may not reflect all publicly available updates.

Configuration Details and Workload Setup: Intel® Xeon® Platinum 8360Y CPU @ 2.4GHz, 2 socket, Hyper Thread On, Turbo On, 256GB Hynix DDR4-3200, ucode 0xd000363. GPU: Nvidia A100 PCIe 80GB GPU memory. Software: SYCL open source/CLANG 17.0.0, CUDA SDK 12.0 with NVIDIA NVCC 12.0.76, cuMath 12.0, cuDNN 12.0, Ubuntu 22.04.1. SYCL open source/CLANG compiler switches: -fsycl-targets=nvptx64-nvidia-cuda, NVIDIA NVCC compiler switches: -O3 -gencode arch=compute_80, code=sm_80. Represented workloads with Intel optimizations.

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. Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary.
Intel® oneAPI Toolkits
A complete set of proven developer tools expanded from CPU to Accelerators

Intel® oneAPI Base Toolkit
A core set of high-performance libraries and tools for building C++, SYCL and Python applications

Add-on Domain-specific Toolkits

Intel® oneAPI Tools for HPC
Deliver fast Fortran, OpenMP & MPI applications that scale

Intel® oneAPI Tools for IoT
Build efficient, reliable solutions that run at network’s edge

Intel® oneAPI Rendering Toolkit
Create performant, high-fidelity visualization applications

Toolkits powered by oneAPI

Intel® AI Analytics Toolkit
Accelerate machine learning & data science pipelines end-to-end with optimized DL frameworks & high-performing Python libraries

Intel® Distribution of OpenVINO™ Toolkit
Deploy high performance inference & applications from edge to cloud

Latest version available 2022.2
Intel® AI Analytics Toolkit

Accelerate end-to-end AI and data analytics pipelines with libraries optimized for Intel® architectures

Who needs this product?
Data scientists, AI researchers, ML and DL developers, AI application developers

Top Features/Benefits
- Deep learning performance for training and inference with Intel optimized DL frameworks and tools
- Drop-in acceleration for data analytics and machine learning workflows with compute-intensive Python packages

Top Features/Benefits

Deep Learning
- Intel® Optimization for TensorFlow
- Intel® Optimization for PyTorch
- Intel® Neural Compressor
- Model Zoo for Intel® Architecture

Machine Learning
- Intel® Extension for Scikit-learn
- Intel-optimized XGBoost

Data Analytics
- Intel® Distribution of Modin
- OmniSci Backend

Intel-optimized Python
- NumPy
- SciPy
- Numba
- Pandas
- Data Parallel Python

Hardware support varies by individual tool. Architecture support will be expanded over time.

Get the Toolkit [HERE](#) or via these locations

- Intel Installer
- Docker
- Apt, Yum
- Conda
- Intel® DevCloud

Back to Domain-specific Toolkits for Specialized Workloads
Intel Data Center GPU Architecture Terminology

**Xe-core**
- Compute Building Block of Xe HPC-based GPUs
- Up to:
  - 4 Slices
  - 64 Xe-cores
  - 64 Ray Tracing Units
  - 4 Hardware Contexts
- L2 Cache
- 4 HBM2e controllers
- 1 Media Engine
- 8 Xe Links

**Xe HPC Stack**

**Xe HPC Slice**
- 16 Xe-cores
- 8MB L1 Cache
- 16 Ray Tracing Units
- Ray Traversal
- Triangle Intersection
- Bounding Box Intersect.
- 1 Hardware Context

**Execution Unit (EU)**
- Xe Vector Engine
- Vector Engine
- XVE

**Systolic/DPAS part of EU**
- Xe Matrix Extension
- Matrix Engine
- XMX

**Subslice (SS) or Dual Subslice (DSS)**
- Xe-core
- NA
- XC

**Slice**
- Render Slice / Compute Slice
- Slice
- SLC

**Tile**
- Stack
- Stack
- STK
Intel® Data Center GPU Flex Series

Arctic Sound M
Super Flexible Data Center GPU for Visual Cloud, Media & Inference

150W
Maximum Peak Performance

30+ 1080p Streams
32 X∞ Cores & Ray Tracing Units
4 X∞ Media Engines

40+ Game Streams

Up to 62 Virtualized Functions
Up to 150 AI TOPS

Industry First
AV1 HW Encode & Decode
XMX AI Acceleration Built in

PCIe Gen 4 Cards - Available Q3’22

75W
High Density Multipurpose
Intel® Data Center GPU Max Series

- Up to 408 MB of L2 Cache
- AI-Boosting Intel® Xe Matrix Extensions (XM X)
# Intel Data Center GPU Max Series Products & Form Factor

<table>
<thead>
<tr>
<th></th>
<th>Max 1550 GPU (800W QAM)</th>
<th>Max 1350 GPU (450W QAM)</th>
<th>Max 1100 GPU (300W PCIe)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X’ Cores</td>
<td>128</td>
<td>112</td>
<td>56</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>HBM2E 128 GB</td>
<td>HBM2E 96 GB</td>
<td>HBM2E 48 GB</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>L1 64 MB</td>
<td>L1 48 MB</td>
<td>L1 28 MB</td>
</tr>
<tr>
<td></td>
<td>L2 408 MB</td>
<td>L2 216 MB</td>
<td>L2 108 MB</td>
</tr>
<tr>
<td><strong>Max TDP</strong></td>
<td>600W</td>
<td>450W</td>
<td>300W</td>
</tr>
<tr>
<td><strong>Form Factor</strong></td>
<td>QAM</td>
<td></td>
<td>PCIe AIC</td>
</tr>
<tr>
<td><strong>Host Interconnect</strong></td>
<td></td>
<td></td>
<td>PCIe Gen5</td>
</tr>
<tr>
<td><strong>Physical Ports</strong></td>
<td>X’ Link 53 GB/s</td>
<td></td>
<td>X’ Link 53 GB/s</td>
</tr>
<tr>
<td></td>
<td>16 ports</td>
<td></td>
<td>6 ports</td>
</tr>
<tr>
<td>Architecture</td>
<td>Xe-LP (TGL)</td>
<td>Xe-HPG (Arc A770)</td>
<td>Xe-HPG (Data Center GPU Flex 170)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
<td>-------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Slice count</td>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>XC (DS/SS) count</td>
<td>6</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>XEVE (EU) / XC</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>XEVE count</td>
<td>96</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>Threads / XEVE</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Thread count</td>
<td>672</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>FLOPs / clk - single precision, MAD</td>
<td>1536</td>
<td>6192</td>
<td>6192</td>
</tr>
<tr>
<td>FLOPs / clk - double precision, MAD</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FLOPs / clk - FP16 DP445</td>
<td>NA</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>LL cache size</td>
<td>3.84MB</td>
<td>16MB</td>
<td>8MB</td>
</tr>
<tr>
<td>SLM size</td>
<td>6 x 128KB</td>
<td>32 x 128KB</td>
<td>16 x 128KB</td>
</tr>
<tr>
<td>FMA, SP (ops / XEVE / clk)</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>SORT, SP (ops / XEVE / clk)</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
**Software Stack for Intel® Data Center Flex Series GPU**

<table>
<thead>
<tr>
<th>Orchestration</th>
<th>Middleware Framework &amp; Runtimes</th>
<th>Low-level Libraries &amp; Tools</th>
<th>Drivers</th>
<th>OS</th>
<th>Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes</td>
<td>Kubernetes</td>
<td>oneDNN</td>
<td>Level Zero</td>
<td>Linux</td>
<td>VMWare</td>
</tr>
<tr>
<td></td>
<td>Open Visual Cloud</td>
<td>oneVPL</td>
<td>OpenGL</td>
<td>Media UMD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel® VTune™ Profiler</td>
<td></td>
<td></td>
<td>3rd Party Closed Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel XPU Manager</td>
<td></td>
<td></td>
<td>Open Source Components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Capture &amp; Stream SDK</td>
<td></td>
<td></td>
<td>Open Source Software created by Intel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DirectX &amp; Indirect Display</td>
<td></td>
<td></td>
<td>Intel Software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vulkan / OpenGL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Media UMD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AI Visual Inference†**

- DL Streamer
- OpenVINO™ toolkit
- TensorFlow*
- PyTorch*
- GStreamer
- FFmpeg

**Media Delivery**

- Open Visual Cloud
- Intel® VTune™ Profiler
- Intel® XPU Manager
- Intel Capture & Stream SDK
- DirectX & Indirect Display
- Vulkan / OpenGL
- Media UMD
- Linux

**Cloud Gaming**

- Android Gaming
- Windows Gaming†
- Android Container AIC
- Intel Bridge Technology
- Open WebRTC Toolkit Game Streaming SDK

**Virtual Desktop Infrastructure (VDI)†**

- Horizon
- Xen App & Desktop

---

Note: oneDNN is the oneAPI Deep Neural Network Library. oneDAL is the oneAPI Data Analytics Library. oneVPL is the oneAPI Video Processing Library. oneVPL, oneDNN, oneDAL, and Intel VTune Profiler are in the Intel® oneAPI Base Toolkit (individual tools can be downloaded separately). Intel-optimized TensorFlow & PyTorch are in Intel® AI Analytics Toolkit. †Reflects capabilities of Intel Data Center GPU Flex Series that will be available when product is fully mature.
Intel ® XPU Manager Product Suite

• A free and open-source suite of solutions built on top of the oneAPI Level Zero interface for monitoring and managing Intel data center XPU.

  • Intel XPU System Management Interface (SMI)
    • A command line utility for local XPU management

  • Intel XPU Manager
    • A full-fledged solution with a daemon for aggregate telemetry collection, RESTful APIs for remote XPU management, and a local library for 3rd party solutions integration, and more.

https://github.com/intel/xpumanager
Environment Setup
Intel® Developer Cloud

a service platform for developing and running workloads in Intel®-optimized deployment environments with the latest Intel® processors

• Landing page:

• Please follow the instructions to get started:
  http://tinyurl.com/ReadmeIDC
  • https://github.com/bjodom/idc
TensorFlow and PyTorch Environments

**TensorFlow Environment**
- Activate the prepared tensorflow env
  - `conda activate tensorflow_xpu`
- Launch an Interactive session to pvc node
  - `$srun -p pvc-shared --pty bash`
  - `$source /opt/intel/oneapi/setvars.sh`
- Pip install necessary python packages
  - `$pip install torchvision --no-deps`
  - `$pip install pillow --no-deps`

**PyTorch Environment**
- Activate the prepared pytorch env
  - `conda activate pytorch_xpu`
- Launch an Interactive session to pvc node
  - `$srun -p pvc-shared --pty bash`
  - `$source /opt/intel/oneapi/setvars.sh`
- Pip install necessary python packages
  - `$pip install torchvision --no-deps`
  - `$pip install pillow --no-deps`

**Validation for both TensorFlow and PyTorch**
- `wget https://raw.githubusercontent.com/oneapi-src/oneAPI-samples/master/AI-and-Analytics/version_check.py`
- `$python version_check.py`

```bash
$ lspci | grep -i display

1:0.0: Display controller: Intel Corporation Device 0bb0 (rev 2f)
2:0.0: Display controller: Intel Corporation Device 0bb0 (rev 2f)
3:0.0: Display controller: Intel Corporation Device 0bb0 (rev 2f)
```

(cpu1: 0x3, cpu2: 0x3, cpu3: 0x3, ...)
<table>
<thead>
<tr>
<th>Nodes and GPU cards</th>
<th>How to login the node</th>
<th>How to use one of 4 GPU card</th>
<th>User Assigned Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 9. cards 0-3</td>
<td><code>srun -p pvc-shared -w idc-beta-batch-pvc-node-9 -- pty /bin/bash</code></td>
<td><code>ZE_AFFINITY_MASK=0</code></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=1</code></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=2</code></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=3</code></td>
<td>3</td>
</tr>
<tr>
<td>Node 10. cards 0-3</td>
<td><code>srun -p pvc-shared -w idc-beta-batch-pvc-node-10 -- pty /bin/bash</code></td>
<td><code>ZE_AFFINITY_MASK=0</code></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=1</code></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=2</code></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=3</code></td>
<td>7</td>
</tr>
<tr>
<td>Node 11. cards 0-3</td>
<td><code>srun -p pvc-shared -w idc-beta-batch-pvc-node-11 -- pty /bin/bash</code></td>
<td><code>ZE_AFFINITY_MASK=0</code></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=1</code></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=2</code></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=3</code></td>
<td>11</td>
</tr>
<tr>
<td>Node 12. cards 0-3</td>
<td><code>srun -p pvc-shared -w idc-beta-batch-pvc-node-12 -- pty /bin/bash</code></td>
<td><code>ZE_AFFINITY_MASK=0</code></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=1</code></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=2</code></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ZE_AFFINITY_MASK=3</code></td>
<td>15</td>
</tr>
</tbody>
</table>
Intel Optimizations for PyTorch on XPU
Major Optimization Methodologies

- Enabled functionality and performance optimizations on Intel GPUs.
- Additional performance boost and early adoption of aggressive optimizations through Intel® Extension for PyTorch*
Overview of Intel® Extension for PyTorch*

- **Eager Mode (Default)**
  - Focus on operators
  - For development and debugging

- **Graph Mode (TorchScript)**
  - Fuse operators and use constant folding to modify and merge the model structure to reduce time loss on invalid operations
  - For deployment
  - To switch to graph mode use TorchScript: `torch.jit.trace()` or `torch.jit.script()`

- oneDNN available in both PyTorch and IPEX
- AMX automatically enabled with oneDNN v2.6 and newer.
- Loaded dynamically in Python* script
- Dynamically linked in CPP executables

Documentation (XPU)

GitHub
https://github.com/intel/intel-extension-for-pytorch
Memory Layout

- Used mainly in image workloads
- NCHW (PyTorch default)
  - `torch.contiguous_format`
- NHWC (IPEX only default 1.13+)
  - `torch.channels_last`
  - NHWC format yields higher performance on Intel® hardware
PyTorch to IPEX – Getting Started

GPU FP32 Inference Example

```python
import torch
import torchvision.models as models
import intel_extension_for_pytorch as ipex

model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
model.eval()
data = torch.rand(1, 3, 224, 224)

model = model.to('xpu')
data = data.to('xpu')
model = ipex.optimize(model)

with torch.no_grad():
    model(data)
```

- Import Intel® Extension for PyTorch* package
- Set model and data to xpu (or 'xpu:id')
  - `model.to('xpu')`
  - `data.to('xpu')`
- `torch.xpu.optimize()` is an alternative of `ipex.optimize()` in Intel® Extension for PyTorch*, to provide identical usage for XPU device only

- Code sample can be found here: [https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/examples.html](https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/examples.html)
Enable Float32 using IPEX

- `ipex.optimize` function applies optimizations against the model object, as well as an optimizer object.
- In function, set `dtype` parameter to customize data type

```python
model = model.to("xpu")
model, optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.float32)
```
Low-precision Optimization – BF16

BF16 has the **same range** as FP32 but **less precision** due to 16 less mantissa bits. Running with 16 bits can give significant performance speedup.

Enable BFloat16 using IPEX

- Similar to Float32, the `optimize` function also works for BFloat16
- Set `dtype` parameter to `torch.bfloat16` instead

```python
model = model.to("xpu")
model, optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)
```

- Auto Mixed Precision (AMP) needed to run in BFloat16

```python
with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
```
Training with Intel® Extension for PyTorch

```python
import torch
import torchvision

# Code changes
import intel_extension_for_pytorch as ipex

LR = 0.001
DOWNLOAD = True
DATA = 'datasets/cifar10/

transform = torchvision.transforms.Compose[
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
]

train_dataset = torchvision.datasets.CIFAR10(
    root=DATA,
    train=True,
    transform=transform,
    download=DOWNLOAD,
)

train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=128
)

model = torchvision.models.resnet50()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=0.9)
model.train()

# Code changes
model = model.to("xpu")
criterion = criterion.to("xpu")
model, optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)

for batch_idx, (data, target) in enumerate(train_loader):
    data = data.to("xpu")
    target = target.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        print(batch_idx)

    torch.save(
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
    , 'checkpoint.pth')
```

*The .to("xpu") is needed for GPU only
**Use torch.cpu.amp.autocast() for CPU
***Channels last format is automatic
Inference with Intel® Extension for PyTorch

**Resnet50**

```python
import torch
import torchvision.models as models
# code changes
import intel_extension_for_pytorch as iex
# code changes

model = models.resnet50(pretrained=True)
model.eval()
data = torch.randn(1, 3, 224, 224)

# code changes
model = model.to("xpu")
data = data.to("xpu")
model = iex.optimize(model, dtype=torch.bfloat16)
# code changes

with torch.no_grad():
    d = torch.randn(1, 3, 224, 224)
    d = d.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
        # code changes
        model = torch.jit.trace(model, d)
        model = torch.jit.freeze(model)
        model(data)
```

**BERT**

```python
import torch
from transformers import BertModel
# code changes
import intel_extension_for_pytorch as iex
# code changes

model = BertModel.from_pretrained(args.model_name)
model.eval()

vocab_size = model.config.vocab_size
batch_size = 1
seq_length = 512
data = torch.randint(vocab_size, size=(batch_size, seq_length))

# code changes
model = model.to("xpu")
data = data.to("xpu")
model = iex.optimize(model, dtype=torch.bfloat16)
# code changes

with torch.no_grad():
    d = torch.randint(vocab_size, size=(batch_size, seq_length))
    d = d.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
        # code changes
        model = torch.jit.trace(model, (d,), strict=False)
        model = torch.jit.freeze(model)
        model(data)
```

*The .to("xpu") is needed for GPU only
**Use torch.cpu.amp.autocast() for CPU
***Channels last format is automatic
Low-precision Optimization – INT8

- What is Quantization?
  - An approximation method
  - The process of mapping values from a large set (e.g. continuous, FP64/FP32) to those with smaller set (e.g. countable, BF16, INT8)

- Why Quantization?
  - Significant performance increase with similar accuracy

- How to Quantize?
  - PyTorch quantization
  - IPEX quantization (with or w/o INC integration)
  - Inter Neural Compressor (INC)

![Diagram showing comparison between FP32 and INT8 formats]
Static vs Dynamic Quantization

• Static (Preferred)
  • Quantizes weights and activations of model
  • Fuses activations into preceding layers
  • Requires calibration dataset to determine optimal quantization parameters for activations
  • Used when both memory bandwidth and compute savings are important
  • Only works on inputs with fixed sizes; typically used for CNNs

• Dynamic
  • Weights are quantized ahead of time, but activations are quantized during inference
  • Used when model execution time is dominated by memory bandwidth
  • Can work on inputs with variable sizes; typically used for LSTM and Transformer models with small batch size

NOTE: Some models are not traceable, and therefore cannot be statically quantized.
Quantization Workflow and API

### Static Quantization:

1. Import `intel_extension_for_pytorch as ipex`.
2. Import `prepare` and convert from `intel_extension_for_pytorch.quantization`
3. Instantiate a config object from `torch.nn.quantization.config` to save configuration data during calibration.
4. Prepare model for calibration.
5. Perform calibration against dataset.
6. Invoke `ipex.quantization.convert` function to apply the calibration config object to the fp32 model object to get an INT8 model.
7. Save the INT8 model into a .pt file.

```python
import torch
from intel_extension_for_pytorch.quantization import prepare, convert

model = Model()
model.eval()
data = torch.rand((shape))

config = ipex.quantization.default_static_config
# Alternatively, define your own config
# from torch.nn.quantization import QuantConfig, Quantization
# config = QuantConfig(activation=QuantConfig.default(), weight=QuantConfig.default())
# config = QuantConfig(activation=QuantConfig.default(), weight=QuantConfig.default())
# config = ipex.quantization.config(config)

for d in calibration_data_loader():
    prepared_model(d)

converted_model = convert(prepared_model)

traced_model = torch.jit.trace(converted_model, data)
traced_model = torch.jit.freeze(traced_model)

traced_model.save("quantized_model.pt")
```

### Dynamic Quantization:

1. Import `intel_extension_for_pytorch as ipex`.
2. Import `prepare` and convert from `intel_extension_for_pytorch.quantization`
3. Instantiate a config object from `torch.nn.quantization.config` to save configuration data during calibration.
4. Prepare model for quantization.
5. Convert the model.
6. Run inference to perform dynamic quantization.
7. Save the INT8 model into a .pt file.

```python
import torch
from intel_extension_for_pytorch.quantization import prepare, convert

model = Model()
model.eval()
data = torch.rand((shape))

dynamic_config = ipex.quantization.default_dynamic_config
# Alternatively, define your own config
# from torch.nn.quantization import QuantConfig, Quantization
# config = QuantConfig(activation=QuantConfig.default(), weight=QuantConfig.default())
# config = ipex.quantization.config(config)

for d in calibration_data_loader():
    prepared_model(d)

converted_model = convert(prepared_model)

traced_model = torch.jit.trace(converted_model, data)
traced_model = torch.jit.freeze(traced_model)

traced_model.save("quantized_model.pt")
```
Quantization Workflow and API (GPU)

Static Quantization:

```python
import Intel®_extension_for_pytorch

def MyModel(torch.nn.Module):
    ...

    # construct the model
    model = MyModel(...)  
    model = model.to('xpu')
    model = torch.jit.trace(model, input)  
    model.qconfig = torch.quantization.QConfig(...)
    model = torch.quantization.quantize_jit.prepare_jit(model, {"": qconfig}, True)

for images in calibration_data_loader():
    images = images.to('xpu')
    model(images)
    model = torch.quantization.quantize_jit.convert_jit(model, True)

    # run the model
    with torch.inference_mode():
        input = input.to('xpu')
        output = model(input)
```

```python
# define the model
model = MyModel(...)  
model = model.to('xpu')
model = torch.jit.trace(model, input)  
model.qconfig = torch.quantization.QConfig(...)
model = torch.quantization.quantize_jit.prepare_jit(model, {"": qconfig}, True)

for images in calibration_data_loader():
    images = images.to('xpu')
    model(images)
    model = torch.quantization.quantize_jit.convert_jit(model, True)

    # run the model
    with torch.inference_mode():
        input = input.to('xpu')
        output = model(input)
```
TorchScript and `torch.compile()`

- **TorchScript**
  - Converts PyTorch `model` into a graph for faster execution
  - `torch.jit.trace()` traces and records all operations in the computational graph; requires a sample input
  - `torch.jit.script()` parses the Python source code of the model and compiles the code into a graph; sample input not required

- **`torch.compile()` – in BETA**
  - Makes PyTorch `code` run faster by just-in-time (JIT)-compiling PyTorch code into optimized kernels
TorchScript

• A method to run PyTorch in graph mode
• *Invoke script mode with `torch.jit.trace` (requires sample input) or `torch.jit.script`*
• `torch.xpu.amp.autocast` *can be used with `torch.jit.trace` to apply graph optimizations*

```python
model = torch.jit.trace(model, d)
model = torch.jit.freeze(model)
```
Hands-on Demo

• Examples: https://github.com/intel/intel-extension-for-pytorch/tree/xpu-master/examples/gpu

• Steps
  • `git clone https://github.com/intel/intel-extension-for-pytorch.git`
  • `cd intel-extension-for-pytorch`
  • `git checkout xpu-master`  # must use this branch for GPUs
  • `cd examples/gpu`  # contains inference and training samples
  • Navigate into inference or training
  • Create a jupyter notebook (.ipynb) file and open it
  • Run different cases by copy/pasting the code and note the runtime differences from the code changes. Be sure to call `torch.xpu.synchronize()` before measuring time. This waits for all kernels to finish before proceeding
# Runtime Configuration for GPU

<table>
<thead>
<tr>
<th>Launch Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPEX_VERBOSE</td>
<td>0</td>
<td>Verbose level in integer. Set to 1 to print verbose output for Intel® Extension for PyTorch® GPU customized kernel. Set to other value is not supported so far.</td>
</tr>
<tr>
<td>IPEX_SIMPLE_TRACE</td>
<td>OFF</td>
<td>Simple trace functionality. If set to ON, enable simple trace for all operators. Set to other value is not supported.</td>
</tr>
<tr>
<td>IPEX_TILE_AS_DEVICE</td>
<td>ON</td>
<td>Device partition. If set to OFF, tile partition will be disabled and map device to physical device. Set to other value is not supported.</td>
</tr>
<tr>
<td>IPEX_XPU_SYNC_MODE</td>
<td>OFF</td>
<td>Kernel Execution mode. If set to ON, use synchronized execution mode and perform blocking wait for the completion of submitted kernel. Set to other value is not supported.</td>
</tr>
<tr>
<td>IPEX_FP32_MATH_MODE</td>
<td>FP32</td>
<td>Floating-point math mode. Set to TF32 for using TF32 math mode, BF32 for using BF32 math mode. Set to other value is not supported. Refer to <a href="https://github.com/oneapi-src/oneDNN/tree/rfcs/rfcs/20210301-computation-datatypes">https://github.com/oneapi-src/oneDNN/tree/rfcs/rfcs/20210301-computation-datatypes</a> for the definition of TF32 and BF32 math mode.</td>
</tr>
</tbody>
</table>
OneDNN Verbose

- Generate oneDNN Verbose logs using guide and parser
- To enable verbosity, set environment variables:
  - export DNNL_VERBOSE=1
  - export DNNL_VERBOSE_TIMESTAMP=1
- Set a Python breakpoint RIGHT AFTER one iteration of training/inference
PyTorch to IPEX – Validating Output
Getting Started with Model Zoo

- Model Zoo for Intel® Architecture: contains Intel optimizations for running deep learning workloads on Intel® Xeon® Scalable processors
- GitHub: [https://github.com/IntelAI/models](https://github.com/IntelAI/models)
ImageNet Data Prep

- Download ImageNet2012 training and validation sets
- Extract files from each tar file
- Place files into respective folder
  - Run the `valprep.sh` script to organize validation data
  - See script on left to prepare training data
- *NOTE: no need to run scripts. Pre-processed dataset located here:
  - `/gpfs/jlse-fs0/projects/intel_anl_shared/imagenet`
Model Zoo w/IPEX

- **Resnet50v1.5 Inference README**
  - `$git clone https://github.com/IntelAI/models.git`

- Commands to run Resnet50v1.5 inference (with Dummy dataset)
  - `$cd models`
  - `$mkdir output_resnet50v1_5_inference`
  - `export OUTPUT_DIR=/home/uXXXXXX/models/output_resnet50v1_5_inference`
  - `export PRECISION="int8"`
  - `./quickstart/image_recognition/pytorch/resnet50v1_5/inference/gpu/inference_block_format.sh`

- Alternatively, run this script with input argument “inference”
  - Copy modelzoo_resnet.sh into MODEL_DIR
  - `source modelzoo_resnet.sh inference`
  - *NOTE: this script will not be maintained*
Model Zoo w/IPEX Output

- Inference (Results with ImageNet validation)

```
resnet50 int8 inference block nchw
oneclBindings_for_pytorch not available!
Use XPU: 0
⇒ using pre-trained model 'resnet50'
model to xpu
doing int8 jit calibration
doing int8 inference

<table>
<thead>
<tr>
<th>Test</th>
<th>Time</th>
<th>Loss</th>
<th>Acc @ 1</th>
<th>Acc @ 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/30</td>
<td>25.014</td>
<td>5.2118e-01</td>
<td>87.30 (87.30)</td>
<td>96.68 (96.68)</td>
</tr>
<tr>
<td>2/30</td>
<td>12.551</td>
<td>1.0375e+00</td>
<td>74.61 (80.96)</td>
<td>92.58 (94.63)</td>
</tr>
<tr>
<td>3/30</td>
<td>8.397</td>
<td>4.6560e-01</td>
<td>87.11 (83.01)</td>
<td>97.46 (95.57)</td>
</tr>
<tr>
<td>4/30</td>
<td>6.320</td>
<td>6.7351e-01</td>
<td>84.77 (83.45)</td>
<td>95.12 (95.46)</td>
</tr>
<tr>
<td>5/30</td>
<td>5.073</td>
<td>5.9476e-01</td>
<td>83.98 (83.55)</td>
<td>96.48 (95.66)</td>
</tr>
<tr>
<td>6/30</td>
<td>4.243</td>
<td>8.3799e-01</td>
<td>76.46 (82.37)</td>
<td>96.00 (95.72)</td>
</tr>
<tr>
<td>7/30</td>
<td>3.649</td>
<td>6.5648e-01</td>
<td>82.71 (82.42)</td>
<td>96.58 (95.84)</td>
</tr>
<tr>
<td>8/30</td>
<td>3.264</td>
<td>6.5710e-01</td>
<td>79.59 (82.07)</td>
<td>97.35 (96.03)</td>
</tr>
<tr>
<td>9/30</td>
<td>2.858</td>
<td>6.2756e-01</td>
<td>79.69 (81.89)</td>
<td>95.21 (95.94)</td>
</tr>
<tr>
<td>10/30</td>
<td>2.581</td>
<td>6.4543e-01</td>
<td>83.98 (82.02)</td>
<td>96.39 (95.99)</td>
</tr>
</tbody>
</table>

Quantization Evaluation performance: batch size:1024, throughput:11519.11 image/sec, Acc@1:82.02, Acc@5:95.99
```
Intel Optimizations for TensorFlow on XPU
**Intel® Extension for TensorFlow**

- Intel® Extension for TensorFlow* is a heterogeneous, high performance deep learning extension plugin based on TensorFlow PluggableDevice interface to bring Intel XPU (GPU, CPU, etc) devices into TensorFlow.

- Good performance using default ITEX setting with no code change

- More performance optimizations with minor code change using simple frontend Python API

- GitHub: [https://github.com/intel/intel-extension-for-tensorflow](https://github.com/intel/intel-extension-for-tensorflow)
Intel® Extension for TensorFlow* - GPU Features

• **Features:**
  - Auto Mixed Precision (AMP)
    - support of AMP with BFloat16 and Float16 operations
  - Channels Last
    - support of channels_last (NHWC) memory format
  - DPC++ Extension
    - mechanism to create operators with custom DPC++ kernels running on the XPU device
  - Optimized Fusion
    - support of SGD/AdamW fusion for both FP32 and BF16 precision
    - a set of fusion patterns for inference
Intel® Extension for TensorFlow® - Optimization Methods

- Operator optimizations
- Memory/data layout optimizations
- Graph optimizations
- Mixed Precision
HOWTO: Intel® Extension for TensorFlow*(GPU)

- No code changes, the default backend will be Intel GPU after installing
  - `intel-extension-for-tensorflow[gpu]`

Or

```python
import intel_extension_for_tensorflow as itex

#CPU, GPU or AUTO
backend = "GPU"
itex.set_backend(backend)
```
TensorFlow to ITEX – Getting Started

Setting Backend Example

```python
import numpy as np
import tensorflow as tf
import intel_extension_for_tensorflow as itex
print(itex.__version__)

backend = "CPU"  # CPU, GPU or AUTO
itex.set_backend(backend)

# Conv + ReLU activation + Bias
N = 1
num_channel = 3
input_width, input_height = (5, 5)
filter_width, filter_height = (2, 2)

x = np.random.rand(N, input_width, input_height, num_channel).astype(np.float32)
weight = np.random.rand(filter_width, filter_height, num_channel, num_channel).astype(np.float32)
bias = np.random.rand(num_channel).astype(np.float32)

conv = tf.nn.conv2d(x, weight, strides=[1, 1, 1, 1], padding='SAME')
activation = tf.nn.relu(conv)
result = tf.nn.bias_add(activation, bias)

print(result)
```

- The backend can be set to CPU, GPU or AUTO using the `set_backend` API to run the workload on desired hardware.

- Code sample can be found here: https://github.com/intel/intel-extension-for-tensorflow/blob/main/examples/quick_example.md
ITEX – Verbose mode

- The output shows the oneDNN verbose output process running on GPU.
- Export ONEDNN_VERBOSE=1

```plaintext
onednn_verbose,info,oneDNN v3.1 (commit e008c47c7f2e839ff64c206a21c82059a227717c)
onednn_verbose,info,cpu,runtime:DPC++
onednn_verbose,info,cpu,isa:Intel 64
onednn_verbose,info,cpu,isa:Intel 64
onednn_verbose,info,cpu,isa:Intel 64
onednn_verbose,info,cpu,backend:OpenCL,name:Genuine Intel(R) CPU $0000%
driver_version:2021.13.11
onednn_verbose,info,cpu,backend:Level Zero,name:Intel(R) Graphics [0x020a],driver_version:1.1.20495
onednn_verbose,info,cpu,backend:Level Zero,name:Intel(R) Graphics [0x020a],driver_version:1.1.20495
onednn_verbose,info,cpu,prim_template:operation,engine,primitive,implementation,prop_kind,memory_descriptors,attributes,auxiliary
onednn_verbose,exec,gpu:18446744073709551615,convolution,ocl:gen9:blocked,forward_training,src_f32::blocked:acdb:f0
onednn_verbose,exec,gpu:18446744073709551615,eltwise,ocl:ref:any,forward_training,data_f32::blocked:abcd:f0
onednn_verbose,exec,gpu:18446744073709551615,diff,ocl:ref:any,forward_training,data_f32::blocked:abcd:f0
```

Software and Advanced Technologies Group | SATG/AIA
Intel Corporation Confidential/Proprietary
Mixed precision (BF16 & FP16)

- Use Keras mixed precision API in \textit{Stock TensorFlow}
  - ITEX is compatible

- mixed_precision.set_global_policy('mixed_float16')
  OR
- mixed_precision.set_global_policy('mixed_bfloat16')

Use Advanced Auto Mixed Precision provided by ITEX for better performance
  - 2 modes of activation
  - Can be run from frozen graph
  - Support for fused operations

<table>
<thead>
<tr>
<th></th>
<th>FP16</th>
<th>BF16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel CPU</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Intel GPU</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
ITEX – Advanced Auto Mixed Precision: Python API

• ITEX advanced AMP can be set from code:

```python
import intel_extension_for_tensorflow as itex

auto_mixed_precision_options = itex.AutoMixedPrecisionOptions()
auto_mixed_precision_options.data_type = itex.BFLOAT16 (or itex.FLOAT16)

graph_options = itex.GraphOptions()
graph_options.auto_mixed_precision_options = auto_mixed_precision_options
graph_options.auto_mixed_precision = itex.ON

config = itex.ConfigProto(graph_options=graph_options)
itex.set_backend("gpu", config) [in ITEX v1.0.0 and ITEX v1.1.0]

(NOTE) --> itex.set_config(config) [latest master branch]
```
**ITEX – Advanced Auto Mixed Precision**

**Environment Variable**

- ITEX advanced AMP can also be set via env variables:

  ```
  export ITEX_AUTO_MIXED_PRECISION=1
  export ITEX_AUTO_MIXED_PRECISION_DATA_TYPE="BFLOAT16" (or "FLOT16")
  ```
Enable BF 16 capabilities using ITEX

- ./infer_fp32_vs_amp.sh gpu bf16
- The output of enabling auto mixed precision is shown.
- Code sample can be found here: [https://github.com/intel/intel-extension-for-tensorflow/tree/main/examples/infer_inception_v4_amp](https://github.com/intel/intel-extension-for-tensorflow/tree/main/examples/infer_inception_v4_amp)
Customized ITEX operators

- `itex.ops.ItexLSTM`
  - Has same semantic with `tf.keras.layers.LSTM`.
  - Based on available runtime / hardware, layer will choose ITEX or TF implementation to maximize performance.

- Other such operators are:
  - `itex.ops.gelu`
  - `itex.ops.LayerNormalization`
  - `itex.ops.AdamWithWeightDecayOptimizer`

For more details refer [here](#)
Getting Started with Model Zoo

• Model Zoo for Intel® Architecture: contains Intel optimizations for running deep learning workloads on Intel® Xeon® Scalable processors and Intel Data Center GPU's

• GitHub: https://github.com/IntelAI/models
Model Zoo w/ITEX (Env Setup)

Clone Intel Model Zoo to work directory:

- $ git clone https://github.com/IntelAI/models.git

Download the Intel oneAPI sample (ResNet50_Inference):


Activate conda env and run an interactive session with PVC:

- $ conda activate tensorflow_xpu
- $ srun -p pvc-shared -w idc-beta-batch-pvc-node-11 --pty bash
- $ source /opt/intel/oneapi/setvars.sh
Model Zoo w/ITEX (Jupyter setup)

Get the IP of your interactive session:

- $ echo $(ip a | grep -v -e "127.0.0.1" -e "inet6" | grep "inet" | awk '{print($2)}' | sed 's/\./.*//')

Launch jupyter lab from interactive session:

- $ jupyter-lab --ip 10.10.10.X
  - Fill in the X with the IP you got from step above (i.e 10.10.10.14)

Take note of the IP and the port that jupyter launches on, it will look something like this:

- [http://10.10.14:8888/lab?token=9d83e1d8a0eb3ffed84fa3428aae01e592cab170a4119130](http://10.10.14:8888/lab?token=9d83e1d8a0eb3ffed84fa3428aae01e592cab170a4119130)

From local terminal, port forward IP:

- $ ssh idc -L PORT:10.10.10.X:PORT
  - Fill in the IP (X), and the PORT according to the last step
  - In this example it would look like this: ssh idc -L 8888:10.10.10.14:8888
Model Zoo w/ITEX (ResNet50 Inference Demo)

Getting Started with Intel Model Zoo

This code sample will serve as a sample use case to perform TensorFlow ResNet50 v1.5 inference on a synthetic data implementing a FP32/PFP and its trained model. The pre-trained model published as part of Intel Model Zoo will be used in this sample.

Select precision and download model

Select the precision that you would like to run on the model with -fp32, -fp16 or int8

```bash
precision = "fp32" # or "fp16" or "int8"
```

Choose data precision type [fp32/fp16/int8]

Choose batch size

Modify line to direct where Model Zoo is installed

```bash
# os.environ['MODEL_ZOO'] = '/path/to/model_zoo'
```

Batch and Online Inference

```bash
# Run Inference using --batch-size=24 for throughput, or --batch-size=1 for latency
Free model/Inference/batch_size/benchmark.py \
  --model_name resnet50_3 \
  --framework tensorflow \
  --precision %s \
  --mode Inference \
  --batch-size batch_size \
  --output-frac outputfrac \
  --gp
```
Distributed DL on XPU
Multi-cards DL inference via Horovod on TensorFlow

- **Pre-requisite:**
  - `$source /opt/intel/oneapi/setvars.sh`
  - `$source activate tensorflow_xpu`
  - `$pip install intel-optimization-for-horovod`
  - Find out number of root devices (GPU cards) by “sycl-ls”

- **Example usage:**
  - Set `$NUM_RANKS` as the number of root devices
  - `horovodrun -np $NUM_RANKS -p 22 python tensorflow2_keras_synthetic_benchmark.py`

- **Details:**
  - Need to specific port 22 due to firewall settings in Developer cloud
  - The script used SYCL backend to do distributed training and inference
  - Data Parallelism distributes data across GPUs while using the same model
  - The codes are from
Example output and oneDNN verbose logs

- 4 root devices, 4 GPUs
- 4 ranks and one rank per GPU

Distribute oneDNN computation among gpu 0-3
How to enable Horovod for TF on PVC

- Follow below official guide but replace device name **from GPU to XPU**
  - https://github.com/horovod/horovod/blob/master/docs/tensorflow.rst

For Tensorflow v2:

```python
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpust[hvd.local_rank()], 'XPU')
```
Horovod timeline

- horovodrun -np 4 -p 22 --timeline-filename ./timeline.json python tensorflow2_keras_synthetic_benchmark.py

```plaintext
<table>
<thead>
<tr>
<th>Name</th>
<th>Wall Duration</th>
<th>Self time</th>
<th>Average Wall Duration</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>0.000 ms</td>
<td>1</td>
</tr>
<tr>
<td>NEGOTIATE_ALLREDUCE</td>
<td>20.993 ms</td>
<td>20.993 ms</td>
<td>20.993 ms</td>
<td>1</td>
</tr>
<tr>
<td>ALLREDUCE</td>
<td>0.088 ms</td>
<td>0.015 ms</td>
<td>0.088 ms</td>
<td>1</td>
</tr>
<tr>
<td>WAIT FOR DATA</td>
<td>0.002 ms</td>
<td>0.002 ms</td>
<td>0.002 ms</td>
<td>1</td>
</tr>
<tr>
<td>WAIT FOR OTHER TENSOR DATA</td>
<td>0.056 ms</td>
<td>0.056 ms</td>
<td>0.056 ms</td>
<td>1</td>
</tr>
<tr>
<td>MPI_ALLREDUCE</td>
<td>0.015 ms</td>
<td>0.015 ms</td>
<td>0.015 ms</td>
<td>1</td>
</tr>
<tr>
<td>Totals</td>
<td>21.154 ms</td>
<td>21.081 ms</td>
<td>2.350 ms</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Multi-cards DL inference via DDP on PyTorch

- **Pre-requisite**:
  - `source /opt/intel/oneapi/setvars.sh`
  - `source activate pytorch_xpu`
  - Find out number of root devices (GPU cards) by “sycl-ls”

- Example usage:
  - Set `$NUM_RANKS` as the number of root devices
  - `I_MPI_PORT_RANGE=50000:50500 mpirun --launcher ssh -n 4 -l python demo.py`

- Details:
  - Need to use ssh launcher due to SLURM limitation and assign MPI port due to firewall settings
  - The script used SYCL backend to do distributed training and inference
  - Data Parallelism distributes data across GPUs while using the same model
  - The codes are from
    - https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Getting-Started-Samples/Intel_oneCCL_Bindings_For_PyTorch_GettingStarted
Example output and oneCCL verbose logs

- 4 root devices, 4 GPUs
- 4 ranks and one rank per GPU

CCL_LOG_LEVEL=info I_MPI_PORT_RANGE=50000:50500 mpirun --launcher ssh -n 4 -l python demo.py

Spent 40% of time on XPU allreduce
How to enable DDP for PT on PVC

- Follow below official guide but replace device name from GPU or CPU to XPU and change backend to ccl
  - https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

```python
# Initialize the process group with ccl backend
dist.init_process_group(backend='ccl')

device = "cpu" # xpu:{dist.get_rank()}
+device = "xpu:{dist.get_rank()}
model = Model().to(device)
if dist.get_world_size() > 1:
    model = DDP(model, device_ids=[device] if (device != 'cpu') else None)
```
Back Up