GPU Programming

Yazeed Balasmeh
Arpan Ghosal

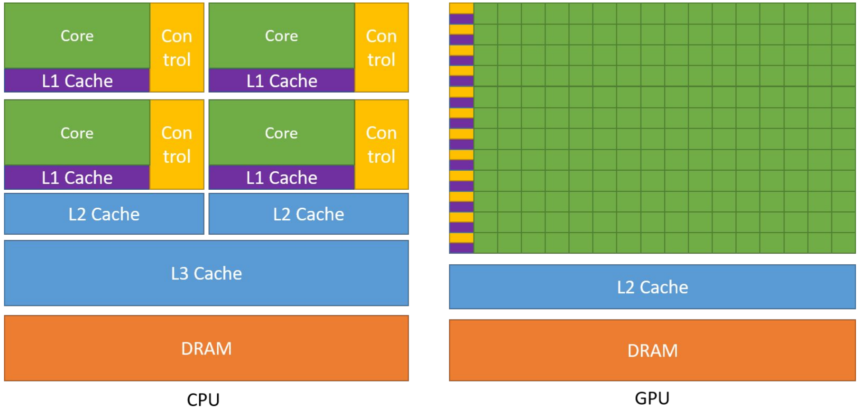
Universität Siegen

CPPS Center for Particle Physics Siegen

Accelerator Optimized Programming

Softwares can leverage hardware accelerators like:

- **GPUs** (Graphics Processing Units)
  - effective for parallel tasks; used heavily in video renderings, ML, simulations etc.
- **TPUs** (Tensor Processing Units)
  - accelerate tensor operations, work seamlessly with TensorFlow; very efficient ML training and inference.
- **ASICs** (Application-Specific Integrated Circuits)
  - usually custom-designed, efficient for specific tasks; can be used for example in cameras.
  - More customizable versions: FPGAs (Field-Programmable Gate Arrays).



These accelerators are designed to enhance performance by handling tasks like **parallel processing** and **matrix operations**, enabling faster and more efficient computations.

Using accelerators in Python

Many Libraries Available for Hardware Acceleration:

- **CuPy**: Serves as a simple drop-in replacement for NumPy.
  - Runs exclusively on GPUs.
  - Easy and direct GPU control.
- **JAX, TensorFlow, PyTorch**:
  - Provide NumPy-like interfaces.
  - Run on CPUs, GPUs, and TPUs.
  - Offer advanced features like Just-In-Time (JIT) compilation and automatic differentiation.
- **Numba**:
  - Uses JIT compilation to enable GPU acceleration.
  - Allows for fast execution of Python code.
- **CUDA (Nvidia)**:
  - Ideal for low-level control and customization.
  - Enables writing custom GPU kernels using features like warps and threads.

Tutorials will be provided if you are interested.

We will mainly discuss this part.

Introduced already

Possible in future (?)

TensorFlow

https://www.tensorflow.org

Google's Deep Learning Library: TensorFlow

- **Key Idea**: A powerful mathematical library that focuses on **dataflow graphs** to perform operations on multi-dimensional arrays (tensors) efficiently.
- **Open-source and highly popular**: Over 185k ★ on GitHub (ranked 12th overall).
- **Scalability:** Big plus for large-scale applications.
- **Two main components**:
  - **High-level API**: Keras for building neural networks easily.
  - **Low-level API**: Offers NumPy-style operations (e.g., `tf.sqrt`, `tf.random.uniform`).

```
#-------> ILLUSTRATIVE ONLY !!

# Running simple operations
with tf.Session() as sess:
        print("a + b =", sess.run(c))
        print("a * b =", sess.run(d))

# Using TensorFlow for matrix operations
mat1 = tf.constant([[3., 3.]])
mat2 = tf.constant([[2.],[2.]])
product = tf.matmul(mat1, mat2)

# Execute the matrix operation
with tf.Session() as sess:
        result = sess.run(product)
        print("Matrix multiplication result:", result)

# Building a neural network using high-level Keras API
model = tf.keras.Sequential([
        tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

# Generate random data to simulate training
inputs = tf.random.normal([1000, 784])
targets = tf.random.uniform([1000], maxval=10, dtype=tf.int32)

# Train the model
model.fit(inputs, targets, epochs=5)
```

PyTorch

PyTorch

https://www.pytorch.org

Excellent support by Facebook: PyTorch

- **More Pythonic**: No need for graph building. Easier for beginners to ML.
- **Dynamic:** Changes can be made on-the-fly during executions.
- **Behaves like NumPy**: Easy to use and intuitive for Python developers.
- **Key Advantage**: Allows the use of standard Python for control flow, making it flexible and powerful.

```
#-------> ILLUSTRATIVE ONLY !!

# Building a neural network with PyTorch's nn module
class SimpleNN(nn.Module):
        def __init__(self):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(784, 512)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(512, 10)
        self.softmax = nn.Softmax(dim=1)

        def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.softmax(self.layer2(x))
        return x

model = SimpleNN()

# Generate random data to simulate input
inputs = torch.randn(1000, 784)
targets = torch.randint(0, 10, (1000,))

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
for i in range(5):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, torch.nn.functional.one_hot(targets, num_classes=10))
        loss.backward()
        optimizer.step()
        print(f"Epoch {i+1}, Loss: {loss.item()}")

# Demonstrating dynamic computation adjustments
for i in range(5):
        if i % 2 == 0:
        optimizer.param_groups[0]['lr'] *= 0.1  # Adjust learning rate dynamically
        outputs = model(inputs)
        loss = criterion(outputs, torch.nn.functional.one_hot(targets, num_classes=10))
        print(f"Adjusted learning rate, Epoch {i+1}, Loss: {loss.item()}")
```

TensorFlow vs PyTorch

Oh no, we're too **static**.
Let's be more **dynamic**!

Oh no, we're too **dynamic**.
Let's be more **static**!

TensorFlow

PyTorch

JAX

A Scientific Python-Focused Package

- Combines the functionality of **NumPy/SciPy** with automatic differentiation.
- Optimized for speed with GPU/TPU support and JIT compilation.

https://jax.readthedocs.io/en/latest/quickstart.html

Quickstart

JAX is a library for array-oriented numerical computation (*à la* NumPy), with automatic differentiation and JIT compilation to enable high-performance machine learning research.

This document provides a quick overview of essential JAX features, so you can get started with JAX quickly:

- JAX provides a unified NumPy-like interface to computations that run on CPU, GPU, or TPU, in local or distributed settings.
- JAX features built-in Just-In-Time (JIT) compilation via Open XLA, an open-source machine learning compiler ecosystem.
- JAX functions support efficient evaluation of gradients via its automatic differentiation transformations.
- JAX functions can be automatically vectorized to efficiently map them over arrays representing batches of inputs.

Comparison

- 2016: TensorFlow

Heavy, multiple features, but not as sleek and clean as JAX

- 2017: PyTorch

Good for scientific computing, very pythonic but less straightforward for productions and deployments

- 2019: JAX

The go-to these days

GPU Programming

Check the Google collab notebook.