CUDA & GPU

# Apple Silicon ML — MLX, CoreML & Metal

Run and optimize machine learning on Apple Silicon with MLX, CoreML, and Metal — the frameworks, workflows, and best practices.

Dr. Farshid Pirahansiah · pirahansiah.com

## Numba JIT Tutorial and PyCUDA with Apple Silicon Adaptation

### Numba JIT on Apple Silicon

This tutorial explores using Numba's `@jit(nopython=True)` decorator to optimize Python code for faster execution. The `@jit(nopython=True)` decorator from Numba compiles Python functions into machine code for improved performance, especially for numerical tasks.

#### Basic Example: Sum of Squares

```
from numba import jit

@jit(nopython=True)
def sum_of_squares(n):
    total = 0
    for i in range(n):
        total += i * i
    return total

print(sum_of_squares(10))  # Output: 285
```

On Apple Silicon, Numba can be used to optimize CPU-bound tasks. Although it doesn't directly support GPU via Metal or NPU, you can use it to significantly speed up CPU computations, which Apple's **M1/M2** chips handle efficiently with multiple cores.

#### Transition to Metal for GPU

To offload heavy parallel tasks to the GPU, Apple uses **Metal**, an API for high-performance graphics and computation on macOS. Metal's **Metal Shading Language (MSL)** provides a way to run GPU tasks that would otherwise be written for CUDA in environments like PyCUDA.

### PyCUDA to Metal for Apple Silicon

PyCUDA is typically used for running GPU tasks on NVIDIA hardware using CUDA. However, on Apple Silicon, you can transition from PyCUDA to **Metal** for GPU programming. Metal can handle parallel GPU tasks on macOS or iOS.

### Metal Shading Language (MSL) for Compute Kernels

Here is an example of using MSL for performing an addition operation on two arrays:

```
#include <metal_stdlib>
using namespace metal;

kernel void add_arrays(const device float* inA [[buffer(0)]],
                       const device float* inB [[buffer(1)]],
                       device float* out [[buffer(2)]],
                       uint id [[thread_position_in_grid]]) {
    out[id] = inA[id] + inB[id];
}
```

You can compile and run this code using Swift to dispatch the compute kernel on the GPU.

### Running Metal from Swift

```
import Metal

let device = MTLCreateSystemDefaultDevice()
let commandQueue = device?.makeCommandQueue()

let shader = '''
#include <metal_stdlib>
using namespace metal;
kernel void add_arrays(const device float* inA [[buffer(0)]],
                       const device float* inB [[buffer(1)]],
                       device float* out [[buffer(2)]],
                       uint id [[thread_position_in_grid]]) {
    out[id] = inA[id] + inB[id];
}
'''

let library = try! device?.makeLibrary(source: shader, options: nil)
let addFunction = library?.makeFunction(name: "add_arrays")
let computePipelineState = try! device?.makeComputePipelineState(function: addFunction!)

let commandBuffer = commandQueue?.makeCommandBuffer()
let computeEncoder = commandBuffer?.makeComputeCommandEncoder()
computeEncoder?.setComputePipelineState(computePipelineState!)

computeEncoder?.dispatchThreads(MTLSize(width: data.count, height: 1, depth: 1),
                                threadsPerThreadgroup: MTLSize(width: 32, height: 1, depth:
1))
computeEncoder?.endEncoding()
commandBuffer?.commit()
commandBuffer?.waitUntilCompleted()
```

## CoreML for Apple NPU

**CoreML** is optimized for running machine learning models on Apple Silicon's **Neural Engine (NPU)**. You can train models using **MLX** or other machine learning frameworks like PyTorch or TensorFlow, and convert them to CoreML for inference on iOS and macOS devices.

### Example of CoreML Conversion:

```
import coremltools as ct
import torch

# Convert PyTorch model to CoreML format
model = torch.load('model.pth')
input_shape = ct.Shape([1, 3, 224, 224])
mlmodel = ct.convert(model, inputs=[ct.TensorType(name="input", shape=input_shape)])
mlmodel.save('model.mlmodel')
```

## Using MLX on Apple Silicon

The **MLX** framework is highly adaptable to Apple Silicon. It provides support for training and deploying machine learning models across various platforms and hardware types. On Apple hardware, it can integrate with **Metal Performance Shaders (MPS)** and **CoreML** to run models on the **GPU** or **Neural Engine**.

### Fullmoon iOS Integration with MLX and Metal

The **Fullmoon iOS** project by **Mainframe Computer** demonstrates how to run large language models like **LLaMA** on Apple devices using **MLX**, **CoreML**, and **Metal**. It leverages Apple's GPU for model inference, showing how you can deploy ML models optimized for Apple's **M1/M2 chips** using the **MLX** framework. You can explore more about **Fullmoon iOS** at its GitHub repository.

## Summary

- **Numba JIT**: Optimize CPU-bound Python code on Apple Silicon.
- **Metal**: Replace PyCUDA with Metal for GPU programming on macOS and iOS, using **MSL** for kernel code.
- **CoreML**: Run machine learning models on the NPU or GPU, converting them from TensorFlow or PyTorch using **coremltools**.
- **MLX**: Use **MLX** to develop and deploy machine learning models efficiently across Apple's hardware accelerators.
- **Fullmoon iOS**: Demonstrates the use of **MLX** and **Metal** for running large models like **LLaMA** on Apple Silicon.

Relevant links: - MLX GitHub Repository - Fullmoon iOS GitHub Repository