以下提供 3 個「測試 CUDA 支援」的實驗範例，皆適合在 **Jetson Orin Nano**（或任何已安裝 CUDA 的 NVIDIA GPU）上讓本科生動手操作。每個實驗包含：**教學目標、前置準備、步驟說明、完整程式碼、結果觀察與延伸討論**。學生只需打開終端或 Jupyter Notebook，即可完成並看到實際效能差異。

## 實驗 1：CUDA C「向量相加」(Vector Addition)

**目的**：驗證 NVCC 編譯器與 GPU 核心能否正確運行，並認識 Grid / Block / Thread 基本概念。

### 1. 前置準備

sudo apt update

sudo apt install -y build-essential

# 確認 nvcc 已安裝

nvcc --version

### 2. 建立原始碼 vector\_add.cu

#include <stdio.h>

#include <cuda\_runtime.h>

\_\_global\_\_ void vecAdd(const float\* A, const float\* B, float\* C, int N) {

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

if (idx < N) C[idx] = A[idx] + B[idx];

}

int main() {

const int N = 1 << 20; // 1M elements

const size\_t bytes = N \* sizeof(float);

float \*h\_A, \*h\_B, \*h\_C;

cudaHostAlloc(&h\_A, bytes, cudaHostAllocDefault);

cudaHostAlloc(&h\_B, bytes, cudaHostAllocDefault);

cudaHostAlloc(&h\_C, bytes, cudaHostAllocDefault);

for (int i = 0; i < N; ++i) { h\_A[i] = 1.0f; h\_B[i] = 2.0f; }

float \*d\_A, \*d\_B, \*d\_C;

cudaMalloc(&d\_A, bytes); cudaMalloc(&d\_B, bytes); cudaMalloc(&d\_C, bytes);

cudaMemcpy(d\_A, h\_A, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, bytes, cudaMemcpyHostToDevice);

int threads = 256;

int blocks = (N + threads - 1) / threads;

vecAdd<<<blocks, threads>>>(d\_A, d\_B, d\_C, N);

cudaMemcpy(h\_C, d\_C, bytes, cudaMemcpyDeviceToHost);

printf("Result[0] = %f (should be 3.0)\n", h\_C[0]);

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

cudaFreeHost(h\_A); cudaFreeHost(h\_B); cudaFreeHost(h\_C);

return 0;

}

### 3. 編譯與執行

nvcc -o vector\_add vector\_add.cu

./vector\_add

### 4. 結果觀察

若 Result[0] = 3.000000，代表 GPU 核心正確執行。

可透過 nvprof ./vector\_add（CUDA 11 前）或 nsys profile（CUDA 11+）查看 kernel 執行時間。

### 5. 延伸

改變 N、threads 及 blocks，觀察效能變化。

嘗試加入 **流 (stream)**、**Unified Memory** 或 **C++17 thrust::transform** 實作。

## 實驗 2：PyTorch GPU vs. CPU 矩陣乘 Benchmark

**目的**：驗證 torch.cuda.is\_available()，並對比 CPU / GPU 在大型矩陣乘法上的速度差異。

### 1. 前置準備

# 建議使用 NVIDIA 提供的 PyTorch wheel

pip3 install --upgrade torch torchvision --extra-index-url https://download.pytorch.org/whl/cu118

python3 -c "import torch, platform, os; print(torch.\_\_version\_\_, torch.version.cuda)"

### 2. Jupyter / .py 程式

import torch, time

device\_cpu = torch.device('cpu')

device\_gpu = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

print("CUDA Available:", torch.cuda.is\_available())

print("GPU Name:", torch.cuda.get\_device\_name(0) if torch.cuda.is\_available() else "N/A")

# 4096x4096 矩陣

N = 4096

A = torch.randn(N, N)

B = torch.randn(N, N)

def matmul\_test(device):

A\_d, B\_d = A.to(device), B.to(device)

torch.cuda.synchronize() if device.type == 'cuda' else None

t0 = time.perf\_counter()

C = A\_d @ B\_d

torch.cuda.synchronize() if device.type == 'cuda' else None

t1 = time.perf\_counter()

return t1 - t0

cpu\_time = matmul\_test(device\_cpu)

gpu\_time = matmul\_test(device\_gpu)

print(f"CPU time: {cpu\_time:.4f}s")

print(f"GPU time: {gpu\_time:.4f}s")

print(f"Speed-up: {cpu\_time/gpu\_time:.2f}×")

### 3. 結果觀察

Orin Nano (FP16 預設關閉) 通常可比 CPU 提速數十倍。

學生可自行改用 torch.float16 或 bfloat16 觀察省時與誤差差異。

### 4. 延伸

測試不同矩陣尺寸，畫出 Size–Latency 曲線。

加入 **AMP (Automatic Mixed Precision)** 與 **Tensor Cores** 將 FP32→FP16 自動混合。

## 實驗 3：OpenCV CUDA vs. CPU 影像處理

**目的**：確認 OpenCV 已以 CUDA 模組編譯，並比較 Sobel 邊緣偵測在 GPU / CPU 上的 FPS。

### 1. 前置準備

# JetPack 5.x 內建 OpenCV + CUDA；若自行編譯需開啟 WITH\_CUDA=ON

python3 - <<'PY'

import cv2

print("CUDA support:", cv2.cuda.getCudaEnabledDeviceCount() > 0)

print(cv2.getBuildInformation().split("Parallel framework")[0])

PY

### 2. Python 程式 opencv\_cuda\_sobel.py

import cv2, time

cap = cv2.VideoCapture(0) # USB 攝影機

assert cap.isOpened(), "Camera not found"

gpu\_available = cv2.cuda.getCudaEnabledDeviceCount() > 0

print("OpenCV CUDA available:", gpu\_available)

# Sobel 核心

def sobel\_cpu(frame):

gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

return cv2.Sobel(gray, cv2.CV\_8U, 1, 0, ksize=3)

def sobel\_gpu(frame):

gpu\_frame = cv2.cuda\_GpuMat()

gpu\_frame.upload(frame)

gray = cv2.cuda.cvtColor(gpu\_frame, cv2.COLOR\_BGR2GRAY)

sobel = cv2.cuda.Sobel(gray, cv2.CV\_8U, 1, 0, ksize=3)

return sobel.download()

mode\_gpu = gpu\_available

t\_prev, cnt = time.time(), 0

while True:

ret, frame = cap.read()

if not ret: break

output = sobel\_gpu(frame) if mode\_gpu else sobel\_cpu(frame)

cv2.imshow("Sobel", output)

cnt += 1

if time.time() - t\_prev > 1: # 每秒統計 FPS

print("GPU" if mode\_gpu else "CPU", "FPS:", cnt)

cnt, t\_prev = 0, time.time()

key = cv2.waitKey(1) & 0xFF

if key == ord('m'): mode\_gpu = not mode\_gpu # m 鍵切換

if key == 27: break # ESC 離開

cap.release()

cv2.destroyAllWindows()

### 3. 執行

python3 opencv\_cuda\_sobel.py

以 m 鍵在 GPU / CPU 間切換，終端會每秒列出目前 FPS。

Jetson Orin Nano 的 GPU 版本通常為 CPU 版本的 5–10 倍。

### 4. 延伸

改用 **Gaussian Blur、Canny、Bilateral Filter** 等 GPU API。

試著在 720 p 與 1080 p 解析度比較吞吐量。

觀察 tegrastats 中 GPU Load 與功耗差異，討論能源效率。

## 課後思考與報告建議

| **主題** | **可收錄內容** |
| --- | --- |
| 效能分析 | 各實驗 CPU vs. GPU 的平均時間 / FPS、Speed-up 倍數、GPU 使用率、溫度 |
| 精度影響 | FP16 / INT8 量化後的誤差（需額外實驗） |
| 能源評估 | 透過 jtop 比較不同負載的功耗 (PVA, NVDEC) |
| 教學反思 | 學生學習曲線、遇到的常見錯誤與排解經驗 |

透過這三個循序漸進的實驗，學生將從**低階 CUDA C** 到 **高階 Python 框架 (PyTorch / OpenCV)** 全面理解 Jetson 上的 CUDA 支援與加速優勢。