CUDA Dummy Kernel Tuning 實驗報告

一、實驗目的

本實驗利用 PyCUDA 實作一個 dummyKernel（模擬神經網路運算負載的簡單加總迴圈），藉由測試不同 block/grid 組合來分析其對 GPU 執行效能的影響，並找出最適配置。

二、實驗方法

- kernel 設計：模擬矩陣乘法計算（每個 thread 執行一個 N×N 的矩陣元素乘加）
- 每組 block/grid 配置重複執行 5 次並平均其執行時間（純 kernel 時間）
- thread 數 = blockDim.x × blockDim.y × gridDim.x × gridDim.y
- 固定 N=8192 threads，block_sizes = [32, 64, 128, 256]，grid_sizes = [16, 32, 64, 128]

三、程式說明

```
import pycuda.autoinit
import pycuda.driver as drv
from pycuda.compiler import SourceModule
import numpy as np
import csv
import argparse
```

- 匯入必要模組，pycuda.autoinit 會自動初始化 GPU 裝置。
- SourceModule 用來編譯 CUDA C 程式碼。

```
mod = SourceModule(kernel_code)
dummy_kernel = mod.get_function("dummyKernel")
```

- 編譯上面定義的 dummyKernel，並取得 callable 函數

```
kernel_code = """
__global__ void dummyKernel(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    if (row < N && col < N) {
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
"""
```

- dummyKernel 是一個標準的矩陣乘法 Kernel，輸入矩陣 A、B，輸出 C，維度為 N×N。
- 每個 thread 計算 C 的一個元素。

```
def measure_time(N, block_size_x, block_size_y, grid_size_x, grid_size_y, repeat=5):
```

- 分配 GPU 記憶體。
- 設定 block/grid 結構。
- 使用 CUDA Event 測量純運算時間。
- 重複執行 repeat 次取平均時間。
- 回傳平均執行時間（單位：毫秒）。

```
def run_tuning(N, output_csv="dummy_tuning.csv"):
```

- 設定測試參數範圍（如 block size 與 grid size）。
- 對每一種組合：
    - 檢查 thread 數不超過 1024。
    - 計算 grid 配置。
    - 呼叫 measure_time 測試。
    - 記錄並更新最佳組合。
- 最終將所有測試結果寫入 CSV。
- 印出最佳結果。

四、實驗結果（矩陣 1000x1000 為例）

以下為直覺化呈現的執行組態：

Block (X×Y)	**Threads per Block**	**Grid (X×Y)**	**Number of Blocks**	**Total Threads**	**Time (ms)**
128×4	512	8×250	2000	1,024,000	0.01500
128×8	1024	8×125	1000	1,024,000	0.01500
256×1	256	4×1000	4000	1,024,000	0.01500
256×2	512	4×500	2000	1,024,000	0.02141

這些配置皆產生相同數量的 threads（1,024,000），但執行時間略有差異，說明即便總運算量相同，不同配置的並行效率與資源調度仍會影響實際效能。

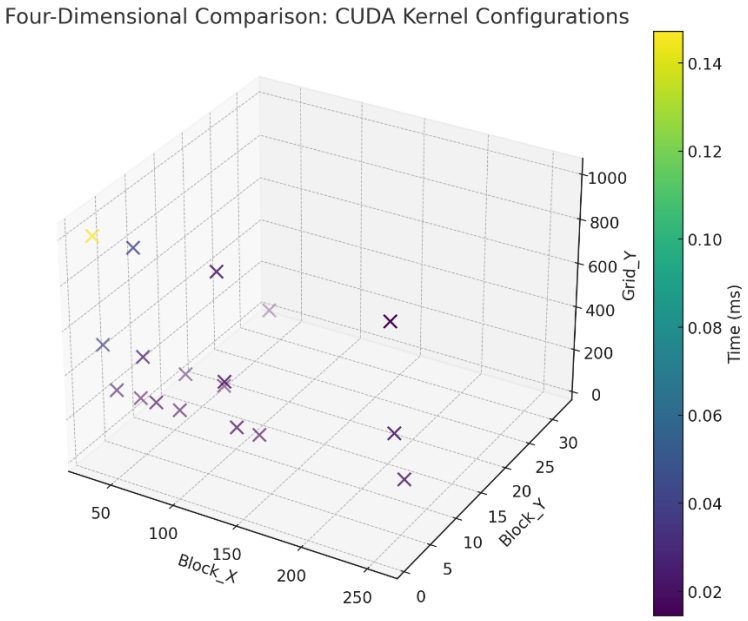
Block X	**Block Y**	**Grid X**	**Grid Y**	**Time (ms)**
32	1	32	1000	0.14723
32	2	32	500	0.04155
32	4	32	250	0.02353
32	8	32	125	0.01500
32	16	32	63	0.01530
32	32	32	32	0.01533
64	1	16	1000	0.04186
64	2	16	500	0.02339
64	4	16	250	0.01564
64	8	16	125	0.01457
64	16	16	63	0.01497
128	1	8	1000	0.02400

128	2	8	500	0.01484
128	4	8	250	0.01451
128	8	8	125	0.01492
256	1	4	1000	0.01485
256	2	4	500	0.02141
256	4	4	250	0.02158

五、分析與結論

最佳配置：

- 當 Block Size = 128x4，Grid Size = 8x250，執行時間為 0.01451 ms，這是所有組合中最快的。



軸	意義
X 軸	Block_X（每個 block 的 X 維度 thread 數）
Y 軸	Block_Y（每個 block 的 Y 維度 thread 數）
Z 軸	Grid_Y（每個 grid 的 Y 維度 block 數量）

顏色	執行時間（Time, ms） → 越深顏色越快

趨勢與分析：

組合	**結論**
Block_Y 1 ~ 2，Grid_Y 1000	效能最差，顏色偏黃，執行時間高。
Block_Y 8 ~ 16，Grid_Y 125	效能最佳，顏色偏紫，執行時間最低。
Block_X 變化（32→256）	顏色分布變化小，顯示 Block_X 影響較小。
Grid_Y 從 1000→125	顏色明顯變深，表示 Grid_Y 減少效果顯著。
Block_Y 過高（>16）	效能提升有限，顏色變化趨於平緩。

結論：

本次 CUDA 核心配置效能分析顯示，Grid_Y 的縮減對效能提升最為顯著，當 Grid_Y 從 1000 降至 125 時，執行時間由 0.14723 ms 降至 0.01500 ms，效能提升近十倍。此外，適度增加 Block_Y（約 8~16）可進一步改善效能，但超過此範圍後效能提升趨於平緩，顏色分布變化不大，顯示已達資源飽和。相比之下，Block_X 的變化（32→256）對效能影響有限，顯示主要瓶頸來自於 Grid_Y 的設定。

六、後續建議

- 測試不同運算密度的 kernel，例如混入更多 memory 操作
- 使用更真實的卷積、矩陣乘法 kernel 做調參比較

七、心得

自從在 meeting 中報告這篇使用機器學習進行自動參數調整的論文（*Machine Learning-Based Auto-Tuning for Enhanced OpenCL Performance Portability*）後，我便產生了一個想法：既然論文中是透過人工神經網路（ANN）預測執行時間

來協助調參，那是否能進一步將這個模型替換為強化學習中的 DQN（Deep Q-Network）？這樣或許能讓模型在探索與利用之間取得更佳平衡，進而自動學習出在不同硬體裝置上能獲得高效能的參數組合。

原本我嘗試以強化學習（Reinforcement Learning, RL）進行自動參數調整，但實驗結果顯示效率不佳。可能是由於強化學習本身的特性——需要大量的試探與學習迭代，導致光是模型訓練就耗時超過一小時。我推測這種方法的優勢或許要在計算量極大、參數空間複雜的情境下才會顯現出來。

```
0.01
(ml_autotune_env) 613K0007C@a120b42d4da1:~/ml_autotune_env$ ./kernel_exec 128 64
0.01
(ml_autotune_env) 613K0007C@a120b42d4da1:~/ml_autotune_env$ python test.py
開始測試 RL 模型自動選擇 CUDA 參數...

[Step 1] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.11 ms
[Step 2] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.10 ms
[Step 3] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.08 ms
[Step 4] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.11 ms
[Step 5] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.11 ms
[Step 6] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.11 ms
[Step 7] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.08 ms
[Step 8] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.08 ms
[Step 9] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.08 ms
[Step 10] 選擇參數：Block Size=128, Grid Size=64，執行時間=0.08 ms
```

透過強化學習中的 DQN 模型測試所得到的參數組合，雖然在探索過程中展現出一定的學習能力，但最終產出的參數在效能表現上並未顯著優於傳統方法，可能仍需更長時間的訓練或更大規模的計算任務才能充分發揮其優勢。

後來我設計了一套自動參數調整系統，只需輸入矩陣大小，即可自動產出對應的最佳參數組合，特別適合應用在神經網路等需大量運算的場景。

在實作過程中，我原本打算在 Windows 本機直接執行 CUDA 的 C 程式，因此嘗試安裝 WSL 和 Ubuntu，卻發現我的電腦僅支援 WSL1，無法升級至支援 CUDA 的 WSL2。無奈之下轉而嘗試使用虛擬機執行 CUDA，結果又碰上 VMware 不支援 GPU 的 CUDA 資訊讀取，導致執行結果始終顯示為 0 秒。最後只好改用實驗室的 server 才順利完成實驗。總之，這份報告的產出過程可說是歷經重重波折，但也因此學到了不少實務經驗。