Lab5: OpenACC

什麼是 OpenACC

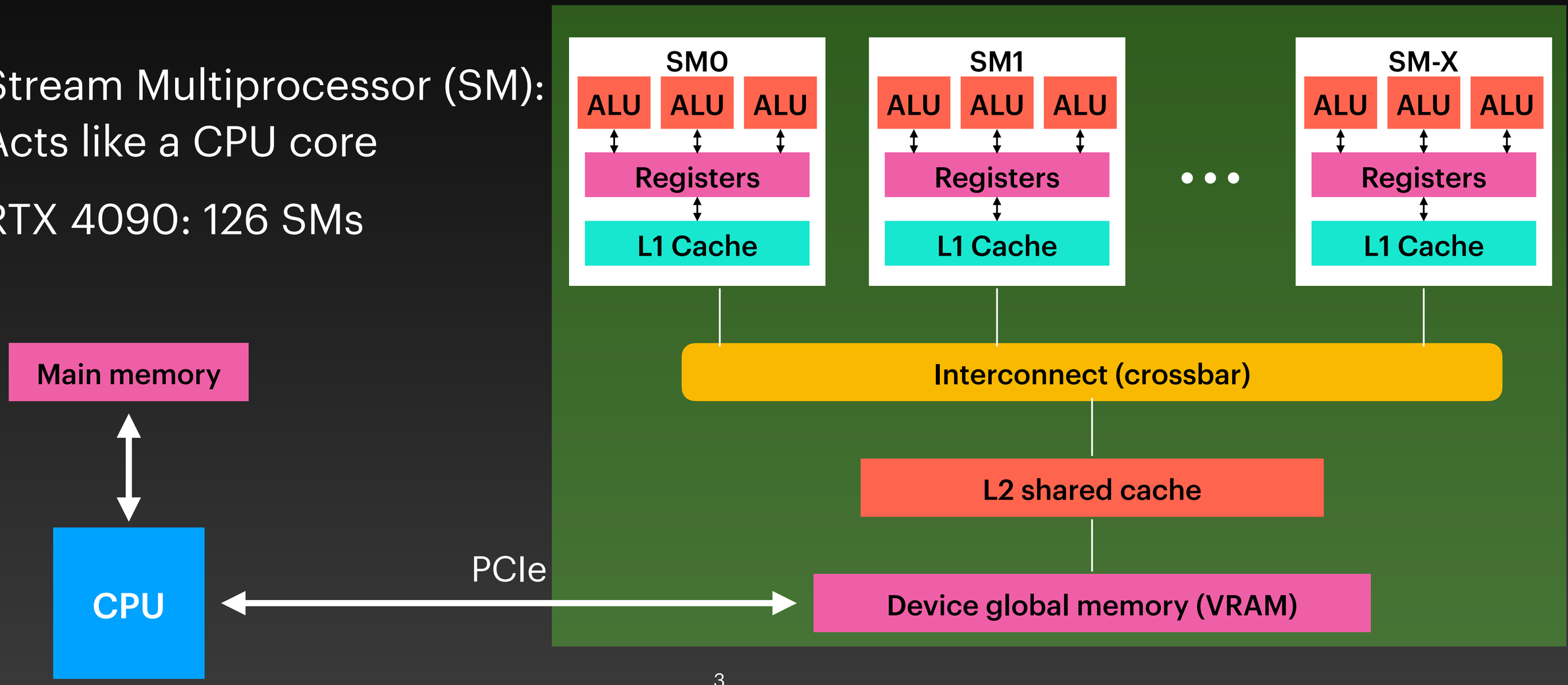
- **Open Acc**elerators 的簡稱
- 透過各種的 compiler 指示（compiler directive）來撰寫 GPU 程式碼
- 降低 GPU 程式設計的技術門檻
- 近年來也增加對 CPU 多執行緒的支持 (multi-threading)

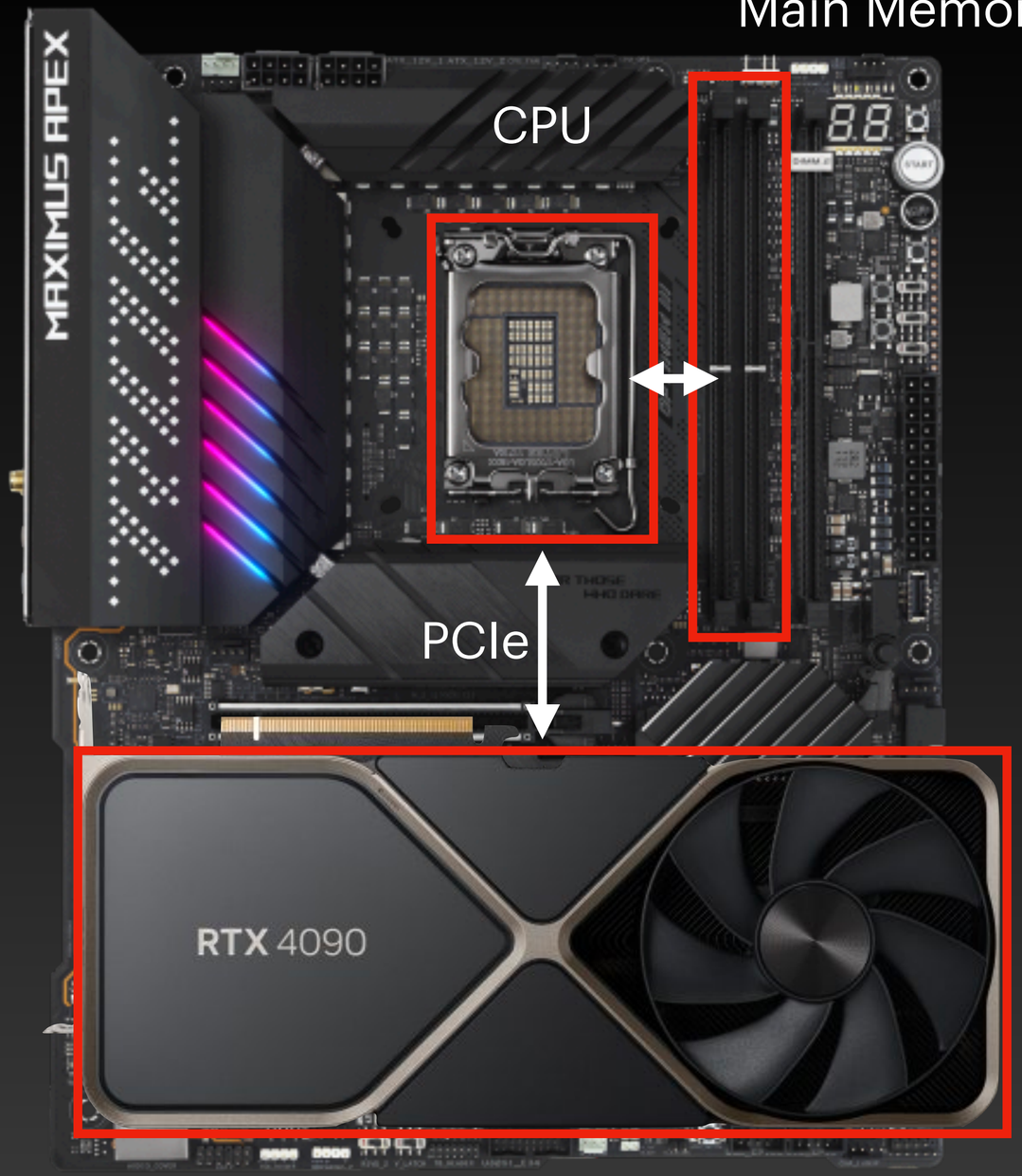
```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels
  {
#pragma acc loop independent collapse(2) reduction(max:error)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                  A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
    ...
  }
}
```

https://www.openacc.org/

GPU Architecture

- Stream Multiprocessor (SM): Acts like a CPU core
- RTX 4090: 126 SMs




Main Memory
CPU
PCIe
GPU
MAXIMUS APEX
RTX 4090

Why OpenACC?

- CUDA 程式設計過程較煩瑣：
  - 需額外管理 CPU device 的記憶體
    - cudaMalloc: 宣告一塊在 GPU 上的記憶體
    - cudaMemcpy: 搬移 CPU & GPU 資料
  - 手動撰寫 CUDA kernel function

```
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Porting to GPU

```
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId ≥ n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz ≠ 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

Why OpenACC?

- CUDA 程式設計過程較煩瑣：
  - 需額外管理 CPU device 的記憶體
    - cudaMalloc: 宣告一塊在 GPU 上的記憶體
    - cudaMemcpy: 搬移 CPU & GPU 資料
  - 手動撰寫 CUDA kernel function

```
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Porting to GPU

```
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId ≥ n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz ≠ 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

Why OpenACC?

- CUDA 程式設計過程較煩瑣：
  - 需額外管理 CPU device 的記憶體
    - cudaMalloc: 宣告一塊在 GPU 上的記憶體
    - cudaMemcpy: 搬移 CPU & GPU 資料
  - 手動撰寫 CUDA kernel function

```
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Porting to GPU

```
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId ≥ n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz ≠ 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

Why OpenACC?

- CUDA 程式設計過程較煩瑣：
  - 需額外管理 CPU device 的記憶體
    - cudaMalloc: 宣告一塊在 GPU 上的記憶體
    - cudaMemcpy: 搬移 CPU & GPU 資料
  - 手動撰寫 CUDA kernel function

```
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Porting to GPU

```
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId ≥ n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz ≠ 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

Why OpenACC?

- CUDA 程式設計過程較煩瑣：
  - 需額外管理 CPU device 的記憶體
    - cudaMalloc: 宣告一塊在 GPU 上的記憶體
    - cudaMemcpy: 搬移 CPU & GPU 資料
  - 手動撰寫 CUDA kernel function

```
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Porting to GPU

```
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId ≥ n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz ≠ 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

Why OpenACC? (Cont'd)

- 一般使用者寫的 CUDA program 不一定擁有最佳的效能
  - CUDA 程式需針對 GPU 硬體設計架構最佳化
- OpenACC 透過編譯器（compiler），協助使用者
  1. 移植部分程式至 GPU 上
  2. 最佳化 GPU 程式

```
#define N 10000000
int main() {
    float x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

A Simple OpenACC Program

1. `#pragma acc data`
   - copyin: 進入 loop 前先將 x 陣列複製至 GPU
   - copyout: 離開 loop 時將 y 陣列複製回 CPU
2. `#pragma acc parallel loop`
   - 平行執行下方的 for loop

- 只用兩行程式碼即完成 GPU 程式!

```
#define N 10000000
int main() {
    float x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Parallelizing Loops with OpenACC

OpenACC Directives

- 語法：`#pragma acc <directive> <clauses>`
  - **#pragma**: Compiler hint (指示詞)
  - **acc**: 告知 compiler 這是 OpenACC 的指示詞
  - **<directive>**: OpenACC 的指令，告知其如何運作
  - **<clauses>**: 更細微的 OpenACC 操作指令

Pragma 指示詞

- 只會套用在 #pragma 的下方一行程式碼
  - 如果要包含多行程式碼，請使用 {}
- 開頭一定是 #pragma acc

```
#define N 10000000
int main() {
    float x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

```
float x[N], y[N];
#pragma acc data copyin(x[0:N]) copyout(y[0:N])
{
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = x[i] + y[i];
    }
}
```

Parallel and Loop Directives

- `parallel`
  - 多餘地平行執行下方程式
    "redundantly parallelize" the code
- `loop`
  - 標記 loop 為「可平行」
- `parallel loop`
  - 平行地執行下方 loop (parallel + loop)

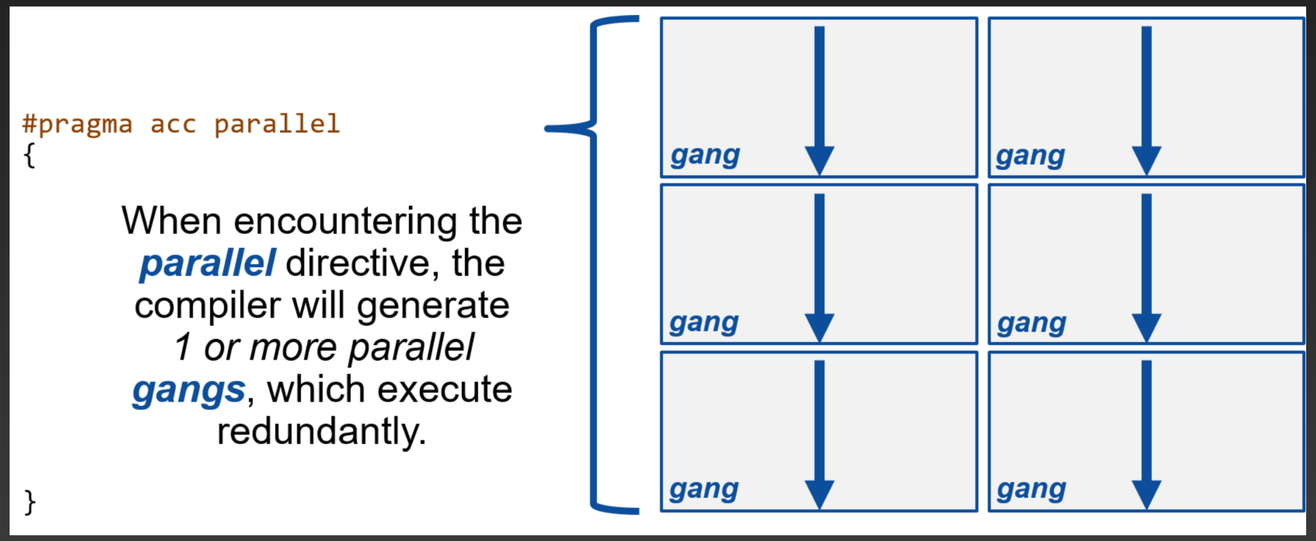
```
#pragma acc parallel
{
    #pragma acc loop
    for (int i = 0; i < N; i++ )
    {
        < loop code >
    }
}
```

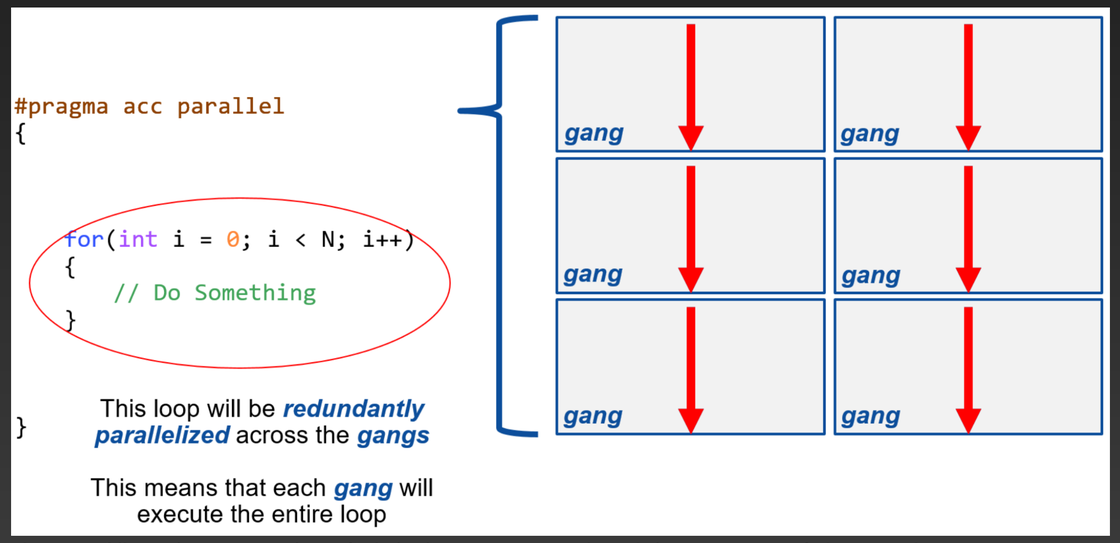
```
#pragma acc parallel loop
for (int i = 0; i < N; i++ )
{
    < loop code >
}
```

Parallel & Gangs

- `#pragma acc parallel` 會產生多個 gang
  - GPU 上的數量通常為上千個
- 每個 gang 均會執行 {...} 的程式碼

6 個由 OpenACC 建立的 gangs 均執行 for loop N 次 ➡️ 實質上沒有平行



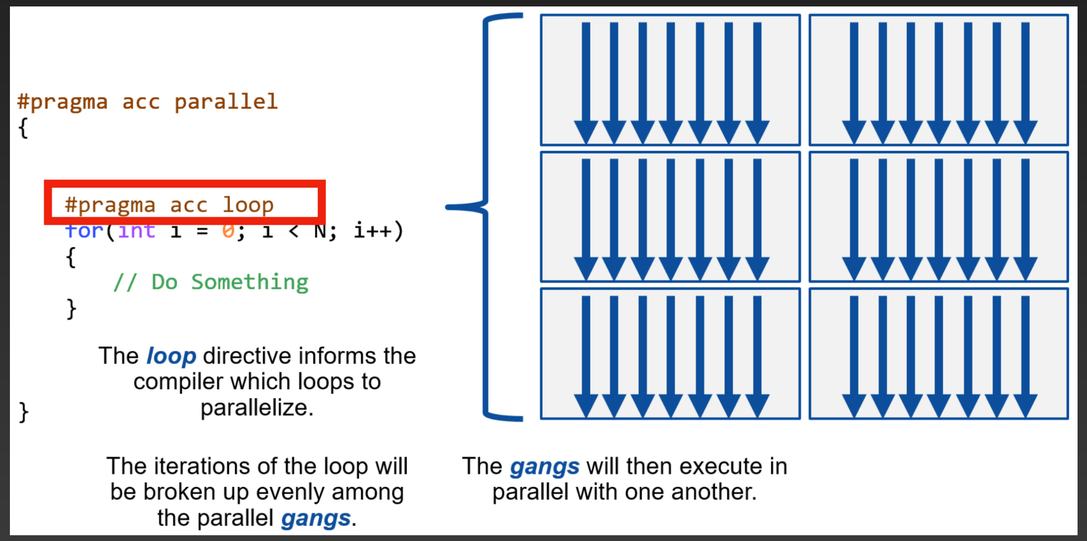


https://github.com/openhackathons-org/gpubootcamp/blob/e858a5e53b981dfd0d07a1f7c493488f45b346ef/hpc/openacc/English/C/jupyter_notebook/openacc_c_lab1.ipynb

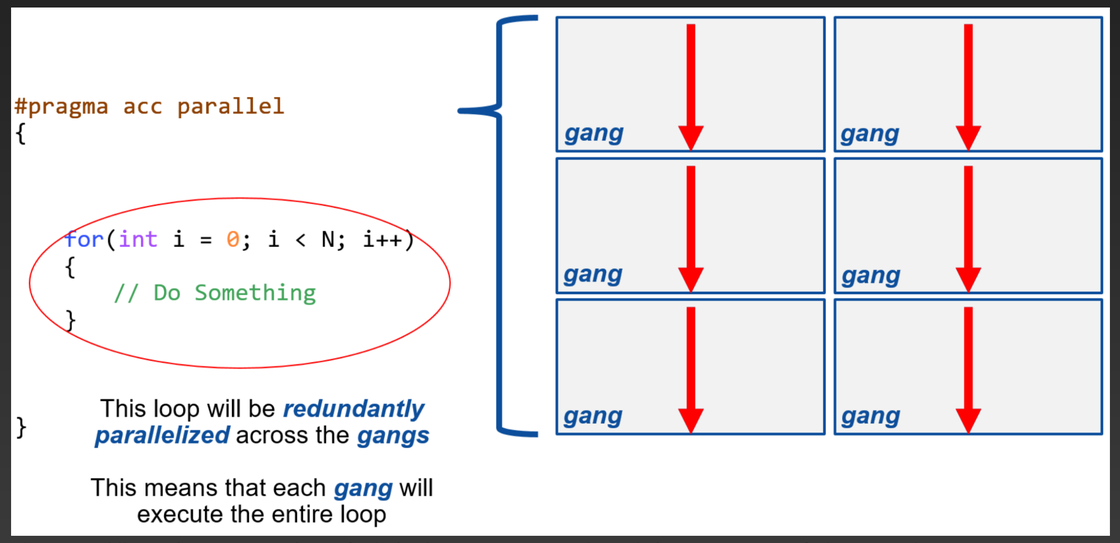
Parallel & Gangs

- 藉由 `#pragma acc loop`，要求 compiler 將迴圈的每個 iteration 分配到不同 gang 上。推薦直接使用 `#pragma acc parallel loop`

6 個由 OpenACC 建立的 gangs 共執行 for loop N 次



6 個由 OpenACC 建立的 gangs 均執行 for loop N 次 ➡️ 實質上沒有平行



https://github.com/openhackathons-org/gpubootcamp/blob/e858a5e53b981dfd0d07a1f7c493488f45b346ef/hpc/openacc/English/C/jupyter_notebook/openacc_c_lab1.ipynb

Reduction

- Reduction：把 N 筆資料 reduce （減少）成 1 筆
- `#pragma acc parallel loop` 只會將 loop 拆開獨立平行執行，無法處理有共用寫入資源的問題
  - 例如：計算總和、乘積、最大最小值...
- 在此情況下，需搭配 reduction 的 clause
  - reduction(***<operation>*** : ***<variable>***)
  - reduction(min:mn)
  - reduction(max:mx)
  - reduction(+:sum)

```
int main() {
    float sum = 0;
    float mn = 999999;
    float x[N];
    #pragma acc parallel loop reduction(min:mn) reduction(+:sum)
    for (int i = 0; i < N; i++) {
        float tmp = sin(x[i]);
        sum = sum + tmp;
        mn = min(mn, tmp);
    }
}
```

Reduction (cont'd)

- Reduction 運作原理：
  1. 每個 gang 均計算自己的 local 結果
  2. Loop 結束後，OpenACC 會透過 parallel reduction 的形式，把 gangs 的結果合併起來

Gang 0:
Sum = 1+2+3+4

+

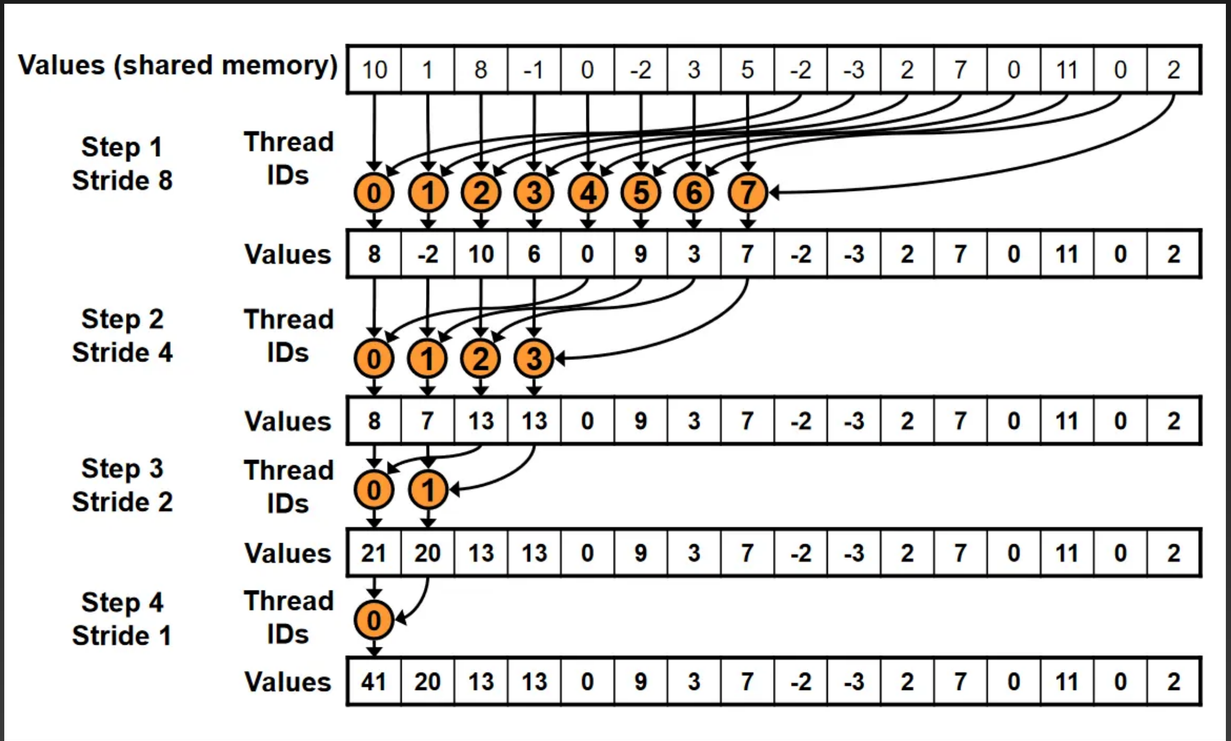
Gang 1:
Sum = 5+6+7+8

```
int main() {
    float sum = 0;
    float mn = 999999;
    float x[N];
    #pragma acc parallel loop reduction(min:mn) reduction(+:sum)
    for (int i = 0; i < N; i++) {
        float tmp = sin(x[i]);
        sum = sum + tmp;
        mn = min(mn, tmp);
    }
}
```

Parallel Reduction

- 傳統的 reduction 無法平行進行。Why?
  - 資料之間如果沒有相依性 ➡️ 才可以平行
  - 但是：第 i 迴圈的結果仰賴 第 (i-1) 迴圈 的資料
- 傳統 reduction
  - Pros: 方便實作
  - Cons: 無法被平行，複雜度 O(N)
- 平行 reduction
  - Pros: 平行後複雜度較低， O(logN)
  - Cons: 實作較複雜。OpenACC > CUDA

```
for (int i = 0; i < N; i++) {
    float tmp = sin(x[i]);
    sum = sum + tmp;
}
```



OpenACC's Kernels Directive

Kernels Directive

- `#pragma acc kernels`
- Kernels v.s. Parallel loop:
  - 如果你很清楚哪個 for loop 可以被平行：使用 `acc parallel loop`
  - 如果不清楚，可以使用 `acc kernels` 交由 compiler 來判斷
    - 缺點： compiler 可能無法自動判別，仍須由使用者來告知

Kernels Directive

```
#pragma acc kernels
for (int i = 0; i < N; i++)
{
    < loop code >
}
```

```
#pragma acc kernels loop independent
for (int i = 0; i < N; i++ )
{
    < loop code >
}
```

```
#pragma acc kernels
{
    for (int i = 0; i < N; i++)
    {
        < loop code >
    }

    < some other sequential code >

    #pragma acc loop independent
    for (int j = 0; j < M; j++)
    {
        < loop code >
    }
}
```

- 如果發現使用 acc kernels 沒有使用到 GPU，很可能是因為 compiler 無法判斷能否被平行
- 根據 compiler 靜態分析的結果，如果 acc kernels 的範圍內不同 for loop 之間沒有資料相依性，OpenACC 可以將不同 loop 合併成一個 kernel function

Common *Kernels* Usage

- `#pragma acc kernels loop independent`
  - 使用 kernels 方式將程式轉移至 GPU 執行
  - 使用 `loop independent` 告知 compiler 將此迴圈平行化

```
#pragma acc kernels loop independent
for (int i = 0; i < N; i++ ) {
    < loop code >
}
```

- `#pragma acc kernels` {...}
  - 程式比較複雜的時候（包含了多個 loops）
  - 由 compiler 做執行上的優化（例如：同時執行）
  - 仍可使用 `loop independent` 提示 compiler

```
#pragma acc kernels
{
    for (int i = 0; i < N; i++)
    {
        < loop code >
    }

    < some other sequential code >

    #pragma acc loop independent
    for (int j = 0; j < M; j++)
    {
        < loop code >
    }
}
```

Kernels v.s. Parallel loop

- 常用語法：
  - `#pragma acc parallel loop` （手動平行）
  - `#pragma acc kernels loop independent` （自動平行，必要時仍為手動）
- 兩種均可以達成相同目的，在寫程式的時候可以都試試看，有些情況下會有效能上的差別。

```
#pragma acc parallel loop
for (int i = 0; i < N; i++ )
{
    < loop code >
}
```

```
#pragma acc kernels loop independent
for (int i = 0; i < N; i++ )
{
    < loop code >
}
```

OpenACC's Data Management

Why Managing Data?

- CPU 的 memory 跟 GPU 是分開的
  - CUDA: cuMalloc -> cuMemcpy(H2D) -> kernel call -> cuMemcpy(D2H)
- 前面的 OpenACC 教學忽略了 data 搬移部分（只包含了 kernel call）

- Managed memory (unified memory)：不需手動處理資料搬移的情況
  - 計算效能相對較差一點
  - 有時間才會再介紹

Data Clauses

	在 GPU 分配記憶體空間	從 CPU 複製至 GPU (進入區塊時)	從 GPU 複製回 CPU (離開區塊時)
copyin(<array>)	✔	✔	
copyout(<array>)	✔		✔
copy(<array>)	✔	✔	✔
create(<array>)	✔		

Example

(1) x 陣列在 GPU 同時讀取&寫入，需使用 copy

```
int main() {
    float x[N];
    #pragma acc data copy(x[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        x[i] = sin(x[i]);
    }
}
```

(2) x 陣列只會被讀取，y 陣列只會被寫入使用 copyin & copyout

```
int main() {
    float x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

(3) 作為暫存的 tmp 在 CPU 端無用處，可以使用 create 來減少資料搬移數

```
int main() {
    float x[N], y[N];
    float tmp[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc data create(tmp[0:N])
    #pragma acc kernels
    {
        for (int i = 0; i < N; i++) {
            tmp[i] = sin(x[i]);
        }
        < ... some seqential code ...>
        for (int i = 0; i < N; i++) {
            y[i] = cos(tmp[i]);
        }
    }

}
```

	在 GPU 分配記憶體空間	從 CPU 複製至 GPU (進入區塊時)	從 GPU 複製回 CPU (離開區塊時)
copyin(<array>)	✔	✔	
copyout(<array>)	✔		✔
copy(<array>)	✔	✔	✔
create(<array>)	✔		

Data Clauses 使用細節

- Array shape
  - `copy(x)`: compiler 可以自動判斷 array 大小
  - `copy(x[0:N])`: compiler 無法自動判斷 array 大小時需指定長度 (N 可以為變數)
- Present Clause
  - #pragma acc parallel loop 在比較複雜的情況無法區分記憶體是否已在 GPU。
  - `#pragma acc data present(<array1>,<array2>)`

```
float x[N], y[N];
float tmp[N];
#pragma acc data copyin(x[0:N]) copyout(y[0:N])
{
    #pragma acc data present(x, y)
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

```
NVC++-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol - B (nn.cpp: 32)
NVC++-F-0704-Compilation aborted due to previous errors. (nn.cpp)
NVC++/x86-64 Linux 23.1-0: compilation aborted
make: *** [Makefile:10: nn] Error 2
```

Data Clauses 使用細節

- create(<array>) 雖然只會在 GPU 上分配空間，沒有 CPU/GPU 搬運的操作，但仍須在 CPU 有對應的陣列。

```
int main() {
    float x[N], y[N];
    float tmp[N];
    #pragma acc data create(tmp[0:N])
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc kernels
    {
        for (int i = 0; i < N; i++) {
            tmp[i] = sin(x[i]);
        }
        for (int i = 0; i < N; i++) {
            y[i] = sin(tmp[i]);
        }
    }

}
```

✔

```
int main() {
    float x[N], y[N];
    float *tmp;
    #pragma acc data create(tmp[0:N])
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc kernels
    {
        for (int i = 0; i < N; i++) {
            tmp[i] = sin(x[i]);
        }
        for (int i = 0; i < N; i++) {
            y[i] = sin(tmp[i]);
        }
    }

}
```

✘

```
int main() {
    float x[N], y[N];

    #pragma acc data create(tmp[0:N])
    #pragma acc data copyin(x[0:N]) copyout
    #pragma acc kernels
    {
        for (int i = 0; i < N; i++) {
            tmp[i] = sin(x[i]);
        }
        for (int i = 0; i < N; i++) {
            y[i] = sin(tmp[i]);
        }
    }

}
```

✘

Data Clauses

- 因為 CPU / GPU 的記憶體是分開的，仍須手動處理資料搬移的問題
- 省去 CUDA 管理另一份 GPU 記憶體的手續 (cudaMalloc, cudaMemcpy, cudaFree)

OpenACC Loop Optimizations

Loop Collapse

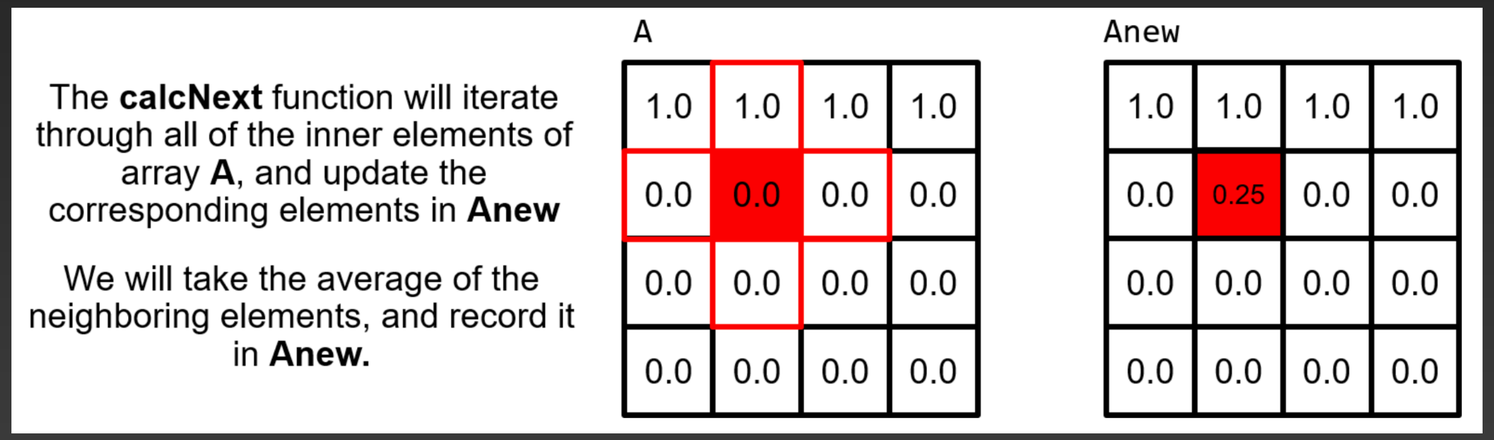
- `#pragma acc parallel loop` 預設只是對下方單一 for loop 做平行
- 某些情況下，不止一個迴圈可以被平行
- 使用 collapse(N) 將 N 層迴圈攤平，平行執行 ➡️ 增加資料平行度，提升 GPU 使用率

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < M; j++)
    {
        for(int k = 0; k < Q; k++)
        {
            x[i][j][k] += 1;
        }
    }
}
```

```
#pragma acc parallel loop collapse(2)
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < M; j++)
    {
        for(int k = 0; k < Q; k++)
        {
            x[i][j][k] += 1;
        }
    }
}
```

Tile Clause

- 將多層迴圈分成不同的 blocks，增加 memory locality（善用 cache）
- 適合優化場景：矩陣乘法、CNN 卷積運算、熱量傳導模擬
  - 共通點：會取用到隔壁矩陣的資料
- 細節請參照 https://github.com/openhackathons-org/gpubootcamp/blob/master/hpc/openacc/English/C/jupyter_notebook/openacc_c_lab3.ipynb



Summary

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    int x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = x[i] * x[i] + cos(x[i]);
    }
}
```

先複製資料

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    int x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = x[i] * x[i] + cos(x[i]);
    }
}
```

再做運算

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    int x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = x[i] * x[i] + cos(x[i]);
    }
}
```

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    double x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    {
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            y[i] = x[i] * x[i] + cos(x[i]);
        }
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            y[i] = y[i] * 2;
        }
    }
}
```

#pragma acc data 下方可以用 “{}”

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    double x[N], y[N];
    #pragma acc data copyin(x[0:N])
    #pragma acc data copyout(y[0:N])
    {
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            y[i] = x[i] * x[i] + cos(x[i]);
        }
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            y[i] = y[i] * 2;
        }
    }
}
```

也可以拆成多個 `#pragma acc data` 使程式碼更好讀

```c
int main() {
    double x[N], y[N];
    #pragma acc data copyin(x[0:N])
    #pragma acc data copyout(y[0:N])
    {
        #pragma acc kernels
        {
            #pragma acc loop independent
            for (int i = 0; i < N; i++) {
                y[i] = x[i] * x[i] + cos(x[i]);
            }
            #pragma acc loop independent
            for (int i = 0; i < N; i++) {
                y[i] = y[i] * 2;
            }
        }
    }
}
```

```c
int main() {
    double x[N], y[N];
    #pragma acc data copyin(x[0:N])
    #pragma acc data copyout(y[0:N])
    {
        #pragma acc kernels loop independent
        for (int i = 0; i < N; i++) {
            y[i] = y[i] * 2;
        }
    }
}
```

也可以試試看 kernels + loop independent

```
#include <iostream>
#include <cmath>
#define N 1000
using namespace std;

int main() {
    double x[N], y[N];
    double sum = 0;
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop reduction(+:sum)
    for (int i = 0; i < N; i++) {
        y[i] = x[i] * x[i] + cos(x[i]);
        sum = sum + y[i];
    }
}
```

要注意有無 reduction 操作

編譯指令:

1. module purge && module load nvhpc
2. nvc++ -fast -acc -gpu=cc70 -Minfo=accel -o <執行檔名稱> <檔案.cpp>

```
[samuel21119@ln01-twnia2 ~]$ nvc++ -fast -acc -gpu=cc70 -Minfo=accel -o acc_exe acc.cpp
main:
      9, Generating copyin(x[:]) [if not already present]
         Generating NVIDIA GPU code
         12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
      9, Generating implicit copy(sum) [if not already present]
         Generating copyout(y[:]) [if not already present]
[samuel21119@ln01-twnia2 ~]$ ./acc_exe
sum: 3.32834e+08
[samuel21119@ln01-twnia2 ~]$
```

編譯

執行

Managed memory (Unified Memory)

Why Unified Memory?

- 傳統 CUDA 與先前介紹的 OpenACC 需在 CPU/GPU 之間手動搬移資料
  - 使用者要自行辨認哪些變數需要被搬移，以及陣列的長度為何
  - 手動的資料搬移有時候會造成不必要的步驟，降低程式效率
- Unified memory 則不需要同時維護 CPU 與 GPU 端的記憶體，只用針對同一塊的位置做寫入，當作 CPU 與 GPU 可以存取相同的內容。

How Unified Memory Works?

- 在硬體上仍存在 CPU 與 GPU 不同區塊的記憶體
- 透過 OS 的 page fault，由 NVIDIA driver 在後台搬移資料：
  - CPU ➡️ GPU： GPU 讀取陣列的時候，發現該位置在 CPU 端有較新的資料
  - GPU ➡️ CPU： 反之亦然

Unified Memory 的缺點

- 需要額外的系統程式來 monitor 兩端的 memory 資料
- 不會事先搬移整塊資料，所以 GPU 很可能算到一半就要停下來等待資料搬移，增加 GPU 的執行時間

References

- NVIDIA GPU Bootcamp: OpenACC 推薦閱讀
  https://github.com/openhackathons-org/gpubootcamp/tree/master/hpc/openacc/English/C/jupyter_notebook
- OpenACC 官網
  https://www.openacc.org/
- OpenACC Tutorial - Data Movement
  https://docs.alliancecan.ca/wiki/OpenACC_Tutorial_-_Data_movement