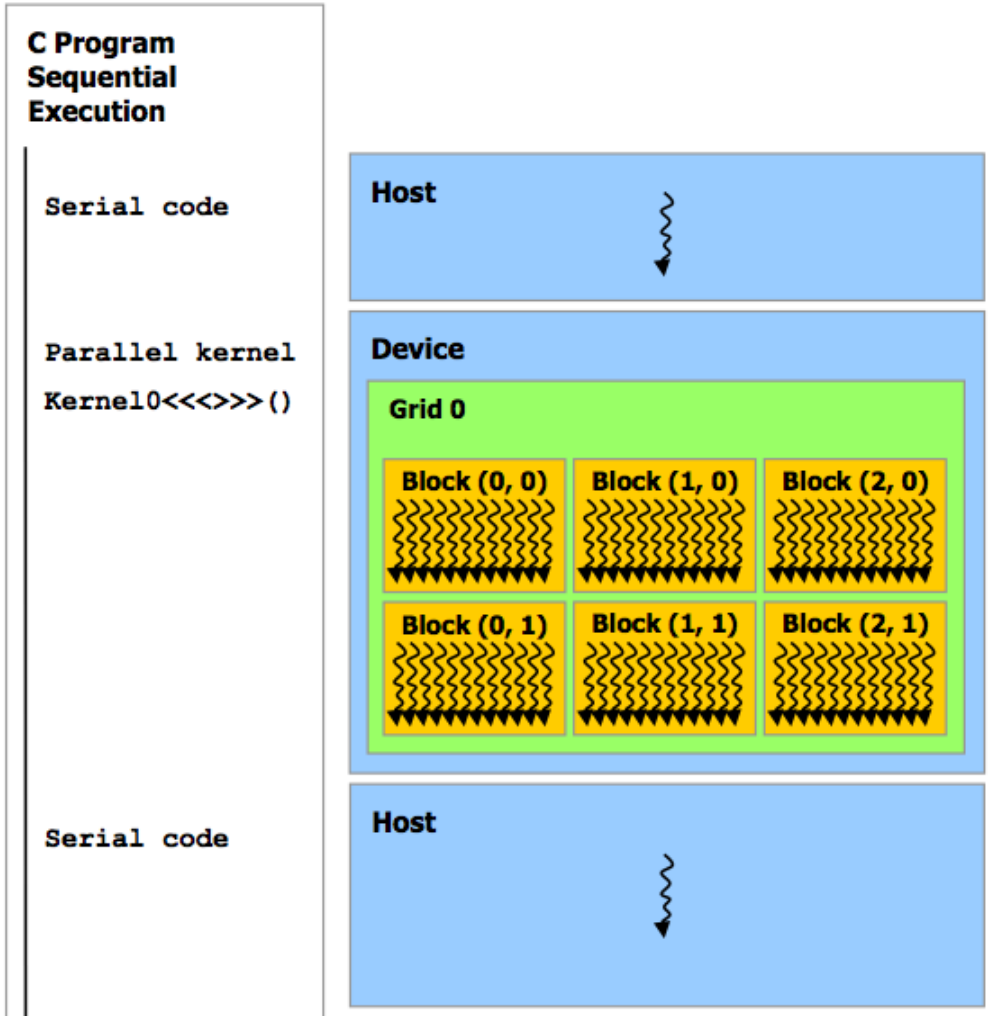
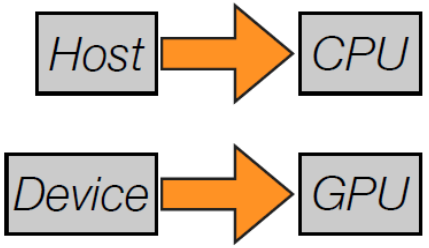
CUDA 程式設計模型

CUDA 程式執行流程圖 (Flow Chart)



Heterogeneous programming model:
- Operates in CPU (host) and GPU (device)



Host 指 CPU ，使用單一執行緒來執行程式。(Host is CPU)

Device 指 GPU ，使用大量執行緒來執行程式。 (Device is GPU)

由 device 執行的程式碼稱為核心函式 (**kernel function**: a function is executed on GPU)

核心函式的 Index System

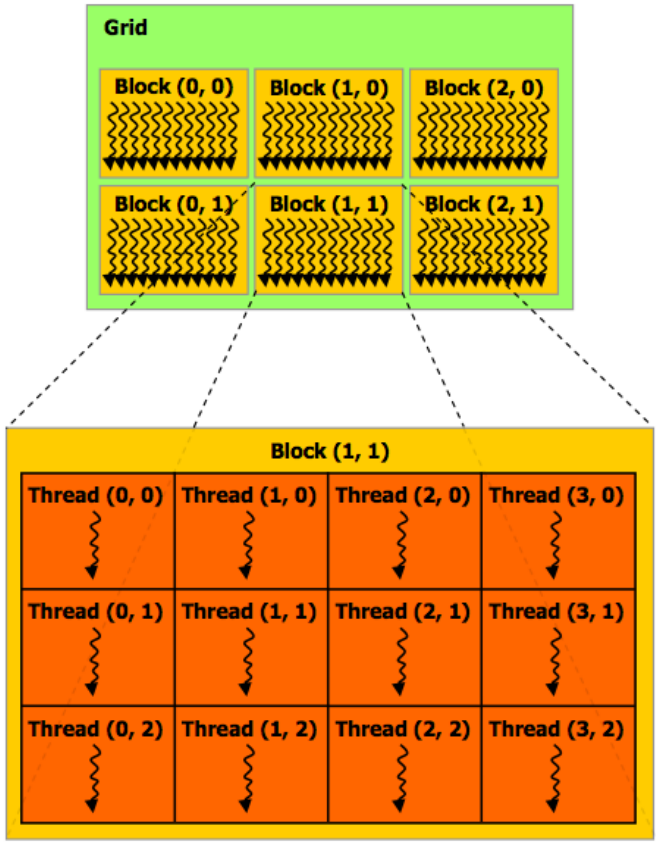


Figure 2-1. Grid of Thread Blocks

核心函式每次執行時會有超大量的執行緒，為了管理所有執行緒的行為，CUDA 核心函式由三階層架構所組成。(Kernel function is executed on a three-level hierarchy)

Grid : 網格，對應到整個 GPU。
Block : 區塊，對應到一個多處理器(SM)
Thread:執行緒，對應到一個核心/串流處理器(core/SP)。

CUDA - Threads

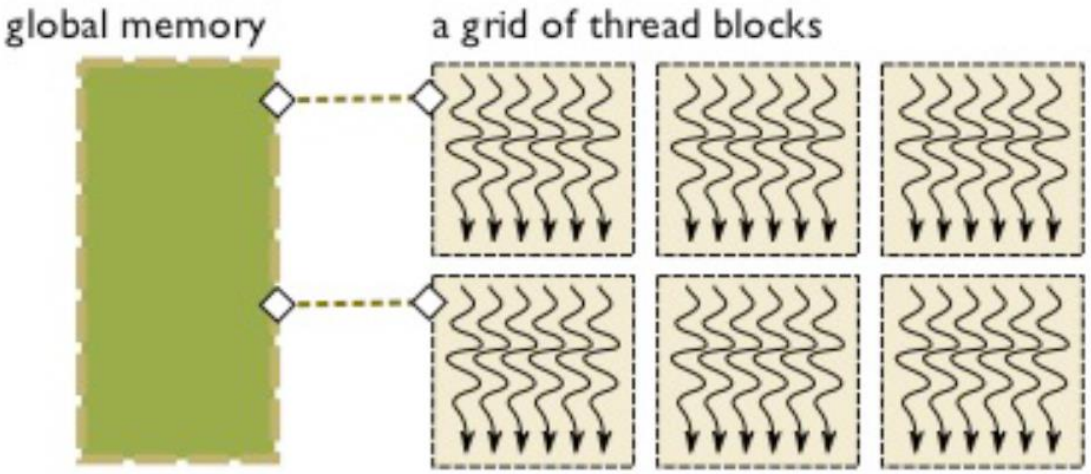
- In CUDA, a **kernel** is executed by many **threads**
- A thread is a sequence of executions
- Multi-thread: many threads will be running at the same time

```
__global__ void  vec_add (float *A, float *B, float *C, int N){
        int i = threadIdx.x + blockDim.x*blockIdx.x;

        if (i>=N) {return;}
        C[i] = A[i] + B[i];
}


void  vec_add (float *A, float *B, float *C, int N){
  for (int i=0; i<N; i++)
        C[i] = A[i] + B[i];
}
```

0_Simple/vectorAdd

CUDA - Threads

- Threads are grouped into **thread blocks**
  - Programming abstraction
  - All threads within a thread block run in the same SM
  - Threads of the same block can communicate
- Thread blocks conform a **grid**

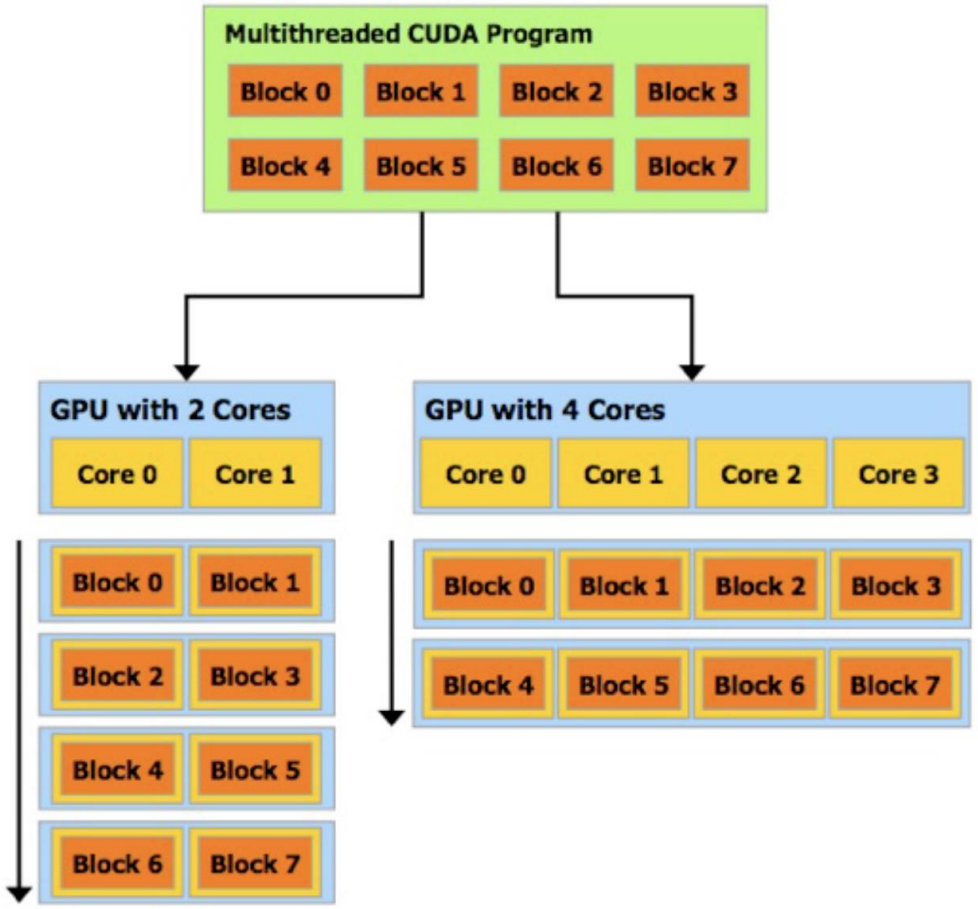


CUDA - Threads

- CUDA virtualizes the physical hardware
  - **Thread** is a virtualized scalar processor
  - **Thread blocks** is a virtualized multiprocessors
- Thread blocks need to be **independent**
  - They run to completion
  - No idea in which order they run (nVidia沒有公開)

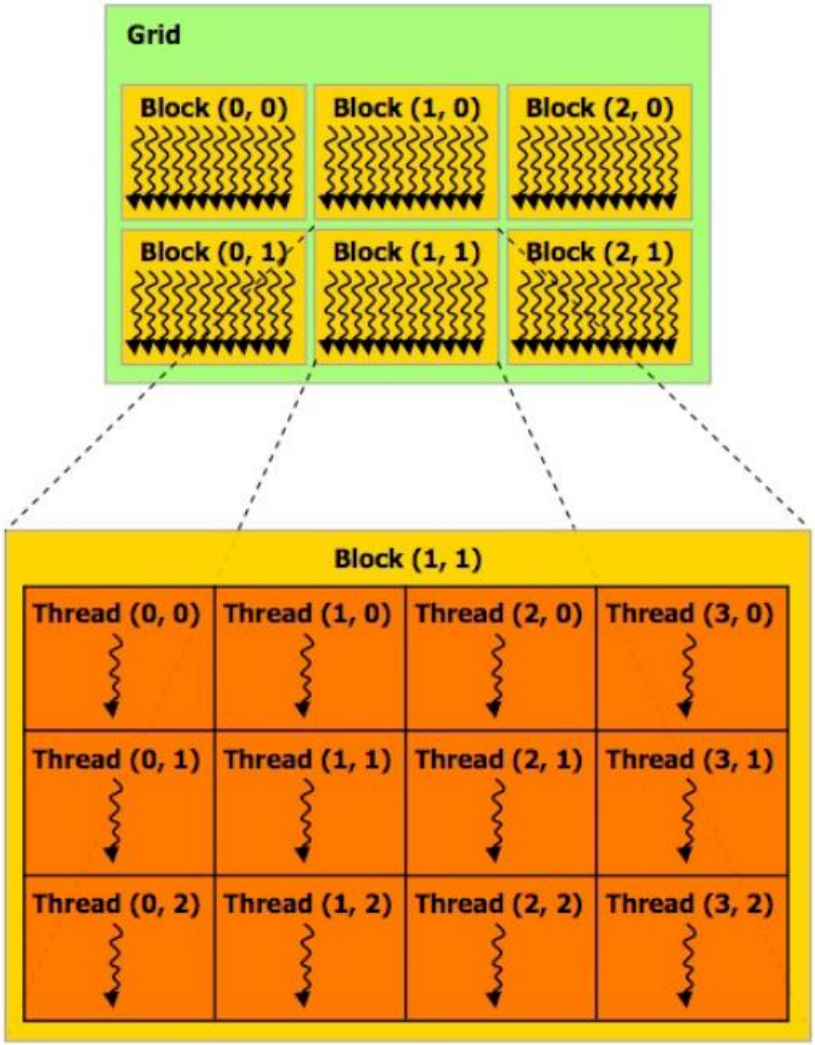
CUDA - Threads

- Provides automatic scalability across GPUs
  - Thread hierarchy
  - Shared memories
  - Barrier synchronization



CUDA - Threads

- Threads have a unique combination of block ID and thread ID
  - We can operate in different parts of the data
  - SIMT: Single Instruction Multiple Threads
  - Threads: 1D, 2D or 3D
  - Blocks: 1D or 2D(GRID 的大小雖然是 3D 的結構，但只能使用 2D 而已(其 z 成員只能是 1))



CUDA - Extensions to C

- Minimal effort from C to CUDA
- Function declarations

  __global__ void kernel()

  __device__ float function()
- Variable qualifiers

  __shared__ float array[5]

  __constant__ float array[5]
- Execution configuration

  dim3 dim_grid(100,100); // 10000 blocks

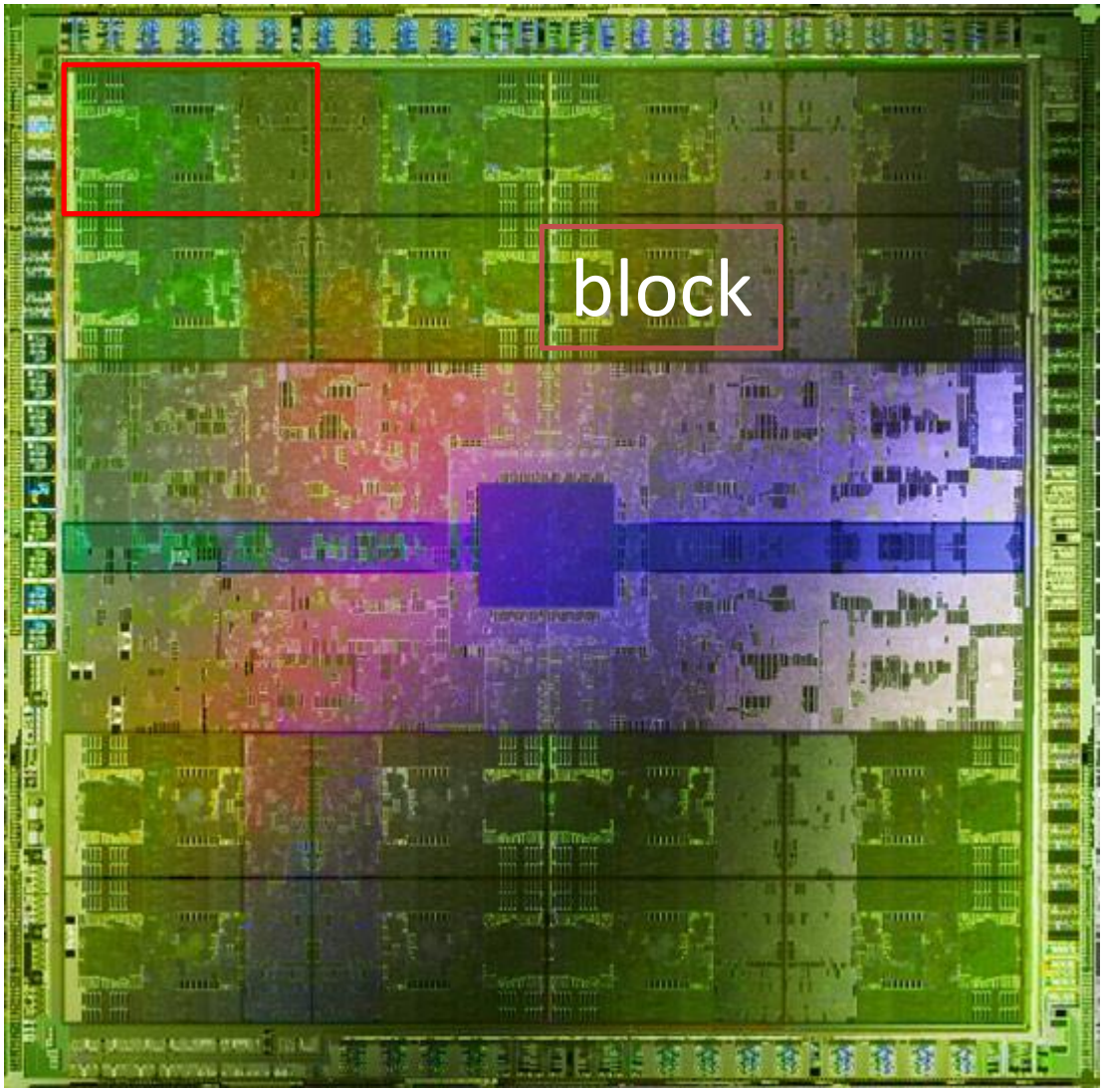
  dim3 dim_block(16, 16); // 256 threads per block in 2D

  kernel <<<dim_grid, dim_block>>> (); // Launch kernel
- Others
  - blockDim.x
  - threadIdx.x
  - blockIdx.x

進階

- 運算子對kernel函式完整的執行配置引數形式是<<<Dg, Db, Ns, S>>>
  - 引數Dg用於定義整個grid的維度和尺寸，即一個grid有多少個block。為dim3型別。Dim3 Dg(Dg.x, Dg.y, 1)表示grid中每行有Dg.x個block，每列有Dg.y個block，第三維恆為1(目前一個核函式只有一個grid)。整個grid中共有Dg.x*Dg.y個block，其中Dg.x和Dg.y最大值為65535。
  - 引數Db用於定義一個block的維度和尺寸，即一個block有多少個thread。為dim3型別。Dim3 Db(Db.x, Db.y, Db.z)表示整個block中每行有Db.x個thread，每列有Db.y個thread，高度為Db.z。Db.x和Db.y最大值為512，Db.z最大值為62。 一個block中共有Db.x*Db.y*Db.z個thread。計算能力為1.0,1.1的硬體該乘積的最大值為768，計算能力為1.2,1.3的硬體支援的最大值為1024。
  - 引數Ns是一個可選引數，用於設定每個block除了靜態分配的shared Memory以外，最多能動態分配的shared memory大小，單位為byte。不需要動態分配時該值為0或省略不寫。
  - 引數S是一個cudaStream_t型別的可選引數，初始值為零，表示該核函式處在哪個流之中。

GPU 晶片圖



Grid

多處理器 (SM)

每個多處理器擁有 :

1. 一個指令快取 (one I-cache)
2. 兩個 Warp 排程器 (two Warp scheduler)
3. 四個SFU(特殊運算單元) (four special function units)
4. 32768 個 32位元存器 (32768 32-bit registers)
5. 16+48 或 48+16 KB 的L1快取+共享記憶體 (64KB L1 cache and 64 KB shred memory)
6. 材質快取及常數快取
7. 16 個 載入/儲存單元

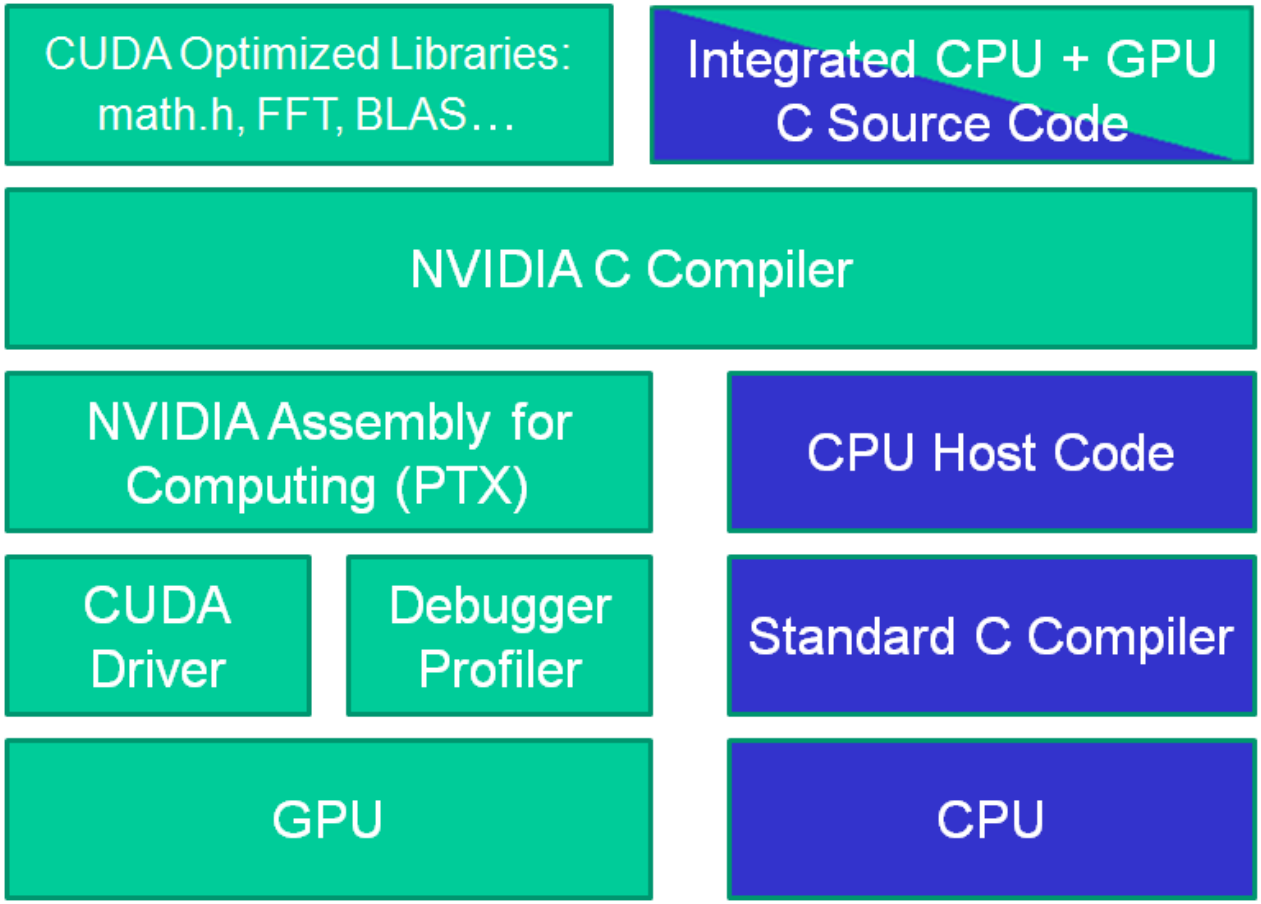
CUDA C

nvcc - NVidia C Compiler

C Language Extensions

CUDA C Runtime

nvcc – nvidia C 語言編譯器 (compiler)

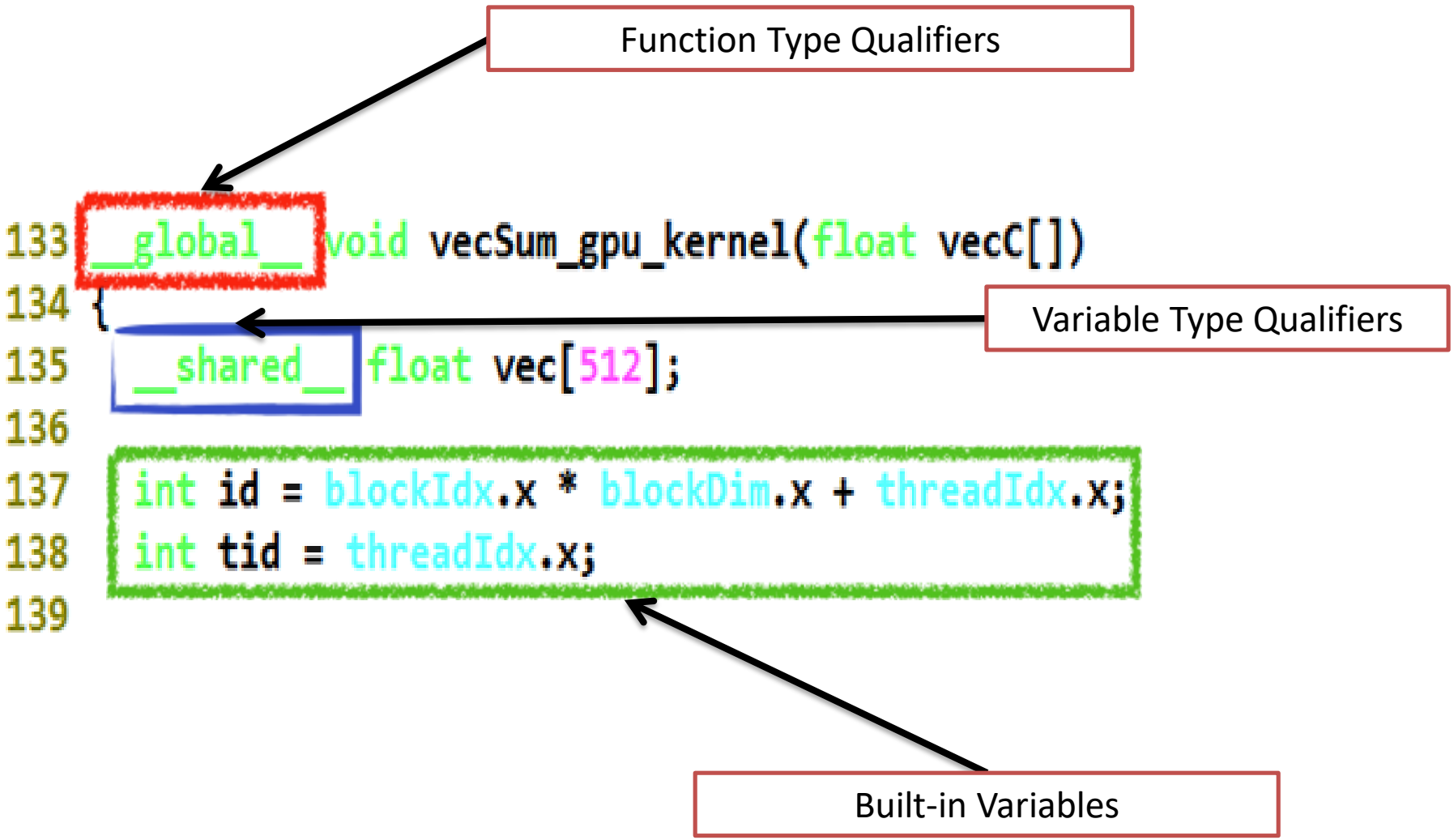


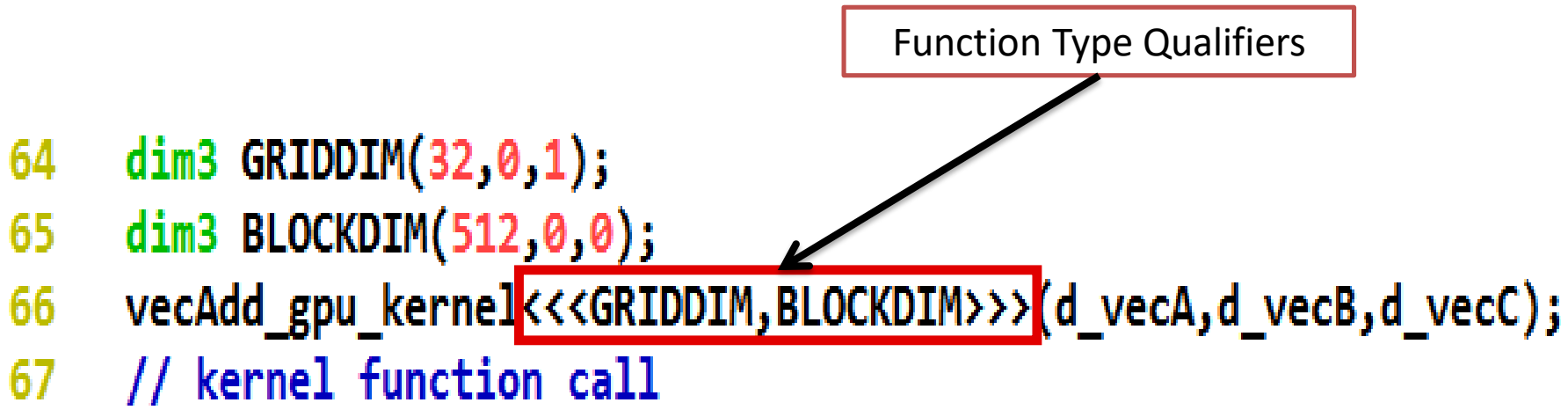
PTX code

- Nvcc (nVidia CUDA compiler)編譯原始碼時，會將和GPU相關的程式分離並轉譯成PTX code((Parallel Thread eXecution) 環境的組合語言 (assembly))
  - 這種設計的好處是不用在編譯時就決定圖形處理器核心類型，可以相容不同型號的GPU核心，但也使得device程式沒辦法直接直接呼叫現成的函式庫
- PTX code在執行時會再透過CUDA Driver解譯成GPU執行的
  - CUDA平行程式在編譯階段不會產生給GPU執行的機器碼
- 原始碼內交由CPU執行的程式碼仍會交由C語言的編譯器進行編一轉成目標執行檔

C 語言延伸 (C Language Extension)

1. Function Type Qualifiers (函式類別標籤)
2. Variable Type Qualifiers (變數類別標籤)
3. Execute Directive (執行指令)
4. Built-in Variables (內建變數)


Function Type Qualifiers
Variable Type Qualifiers
Built-in Variables
133 __global__ void vecSum_gpu_kernel(float vecC[])
134 {
135 __shared__ float vec[512];
136
137 int id = blockIdx.x * blockDim.x + threadIdx.x;
138 int tid = threadIdx.x;
139


Function Type Qualifiers
64 dim3 GRIDDIM(32,0,1);
65 dim3 BLOCKDIM(512,0,0);
66 vecAdd_gpu_kernel<<<GRIDDIM,BLOCKDIM>>>(d_vecA,d_vecB,d_vecC);
67 // kernel function call

Function Type Qualifiers (函式類別標籤)

- 指定函式可以被CPU或是GPU用來呼叫以及執行的標籤，直接寫在函式宣告之前
  - 如前例中的__golobal__表示此函式是被CPU呼叫而允許被GPU執行的程式區段 (__golobal__ is for the functions executing on CPU)
  - __device__表示此函式只能被GPU呼叫且執行於GPU的程式區段 (__device__ is for the functions executing and invoked on GPU)
  - __host__表示此函式只能被CPU呼叫且執行於CPU的程式區段 (__host__ is for the functions executing on GPU and invoked by CPU)

Variable Type Qualifiers (變數類別標籤)

- GPU硬體有複雜的的記憶體架構，不同的記憶體有不同的特性與效能
- 程式開發者在宣告變數時可以利用變數類別標籤來指定該變數要配置在哪一種記憶體上
  - __shared__表示該變數配置在GPU的block共享記憶體中 (variables with __shared__ are on the shared memory of a thread block)
    - 生命週期和block相同
    - 由同一個block中的執行緒共用
  - __device__表示該變數配置在全域記憶體中 (variables with __device__ are on the global memory)
    - 生命週期和應用程式相同
    - 可以由格網中的所有執行緒或是經由特定函式可以被CPU看到
  - __constant__表示該變數配置在常數記憶體中 (variables with __constant__ are on the constant memory)
    - 生命週期和應用程式相同
    - 可以由格網中的所有執行緒或是經由特定函式可以被CPU看到
  - 無設標籤則配置在GPU的暫存器中，如tid與id (variables without type qualifiers are on registers)

CUDA的SIMT運作

- 為有效管理執行時的執行續運作，CUDA採用階層架構方式
- 當一個核心函式被GPU執行時，該函式的運作會被對應一個虛擬的執行緒格網(Grid)之中
  - 格網中包含數個執行緒區塊(Block)，而一個區塊中含有多個執行緒

Grid→Block→Thread

核心函式被呼叫方式

- 函式名稱<<<區塊數目,區塊中的執行緒數目>>>

  function_name<<<block number, thread number in a block>>>

  - 核心函式所能使用的執行緒數量為上述兩個參數的乘積 (The total number of threads is the product of the block number and the thread number in a block)

Built-in Variables (內建變數)

- CUDA程式在執行時透過編譯器會自動產生一些內建變數 (nvcc automatically generates some built-in variables during compile time)
  - 可以用來識別不同的執行緒區塊及執行緒區塊中的執行緒 (for thread identity)
  - 藉由內建變數的組合運用，在CUDA程式內可以指定特定執行緒去處理對應資料
- 內建變數，如
  - gridDim: 儲存格網的維度資料
  - blockDim: 儲存執行緒區塊的維度資料
  - blockIdx: 儲存格網中執行緒區塊的索引值
  - threadIdx: 儲存執行緒區塊中的執行緒索引值
  - wrapSize: 一個wrap中的執行緒個數

CUDA C Runtime API

- Memory Management (記憶體管理)
- Error Handling (錯誤訊息處理)
- Event Management (事件管理)
- Thread Management (執行緒管理), etc...

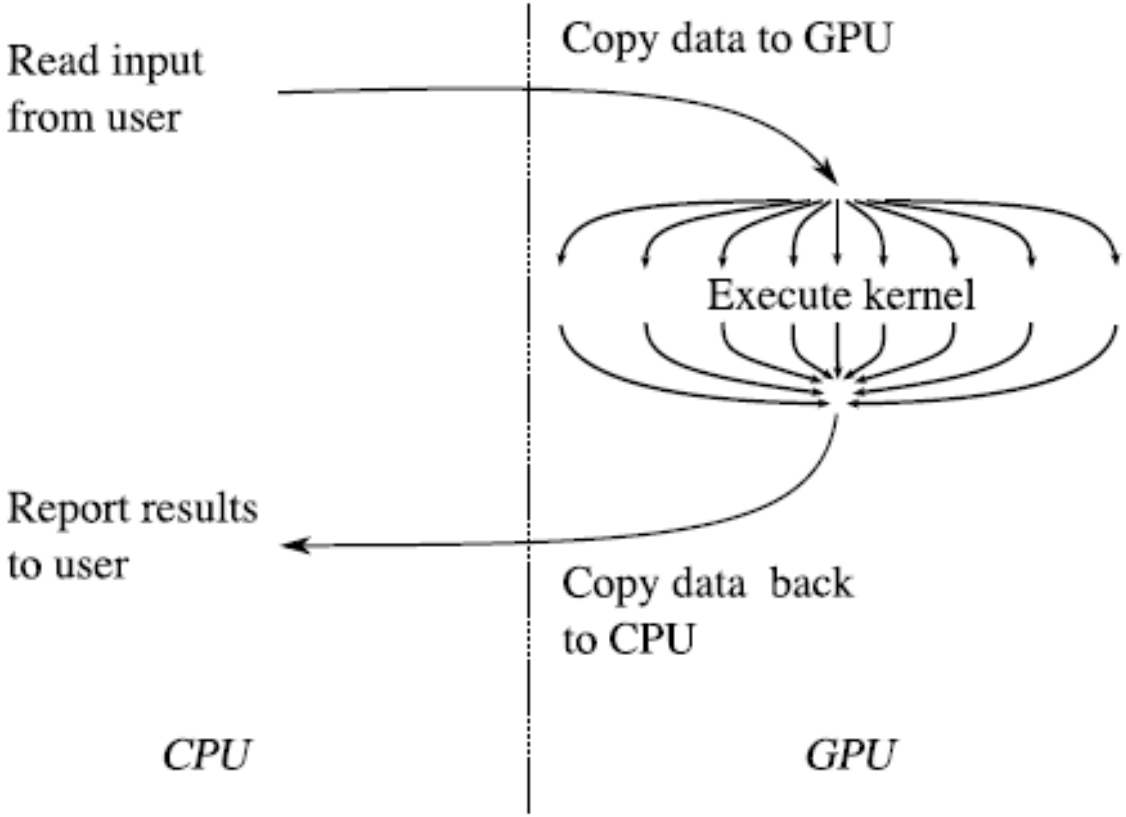


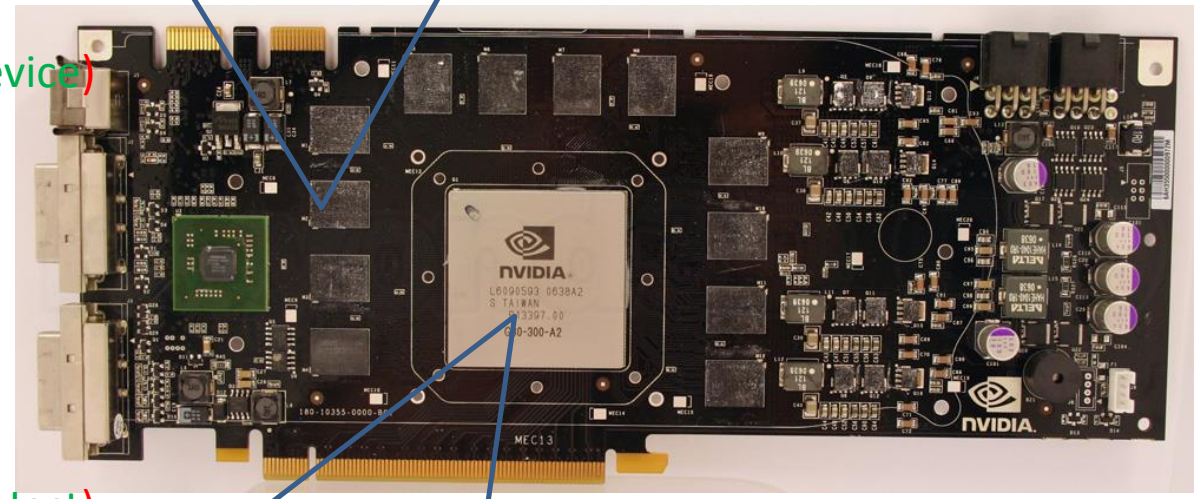
Figure 3
Typical GPGPU application flow. Input data for a GPGPU application must be copied to the GPU's memory along with a pre-allocated output buffer prior to invoking the GPU-based kernel. Output from the kernel is read back into main memory and reported to the user.

(*from Schatz et al. BMC Bioinformatics* 2007)

記憶體管理

cudaMalloc((void**) &arg1, sizeof(int));

cudaMemcpy(&d_arg1 ,&h_arg1,
sizeof(int), cudaMemcpyHostToDevice)



cudaMemcpy(&h_arg1 ,&d_arg1,
sizeof(int), cudaMemcpyDeviceToHost)

function_name<<<GridSize,BlockSize>>>(d_arg1);

CUDA 運算 (1)

- CPU + GPU 的共同運算架構。
- 利用 GPU 的超多核心及超多執行緒架構來輔助 CPU 。

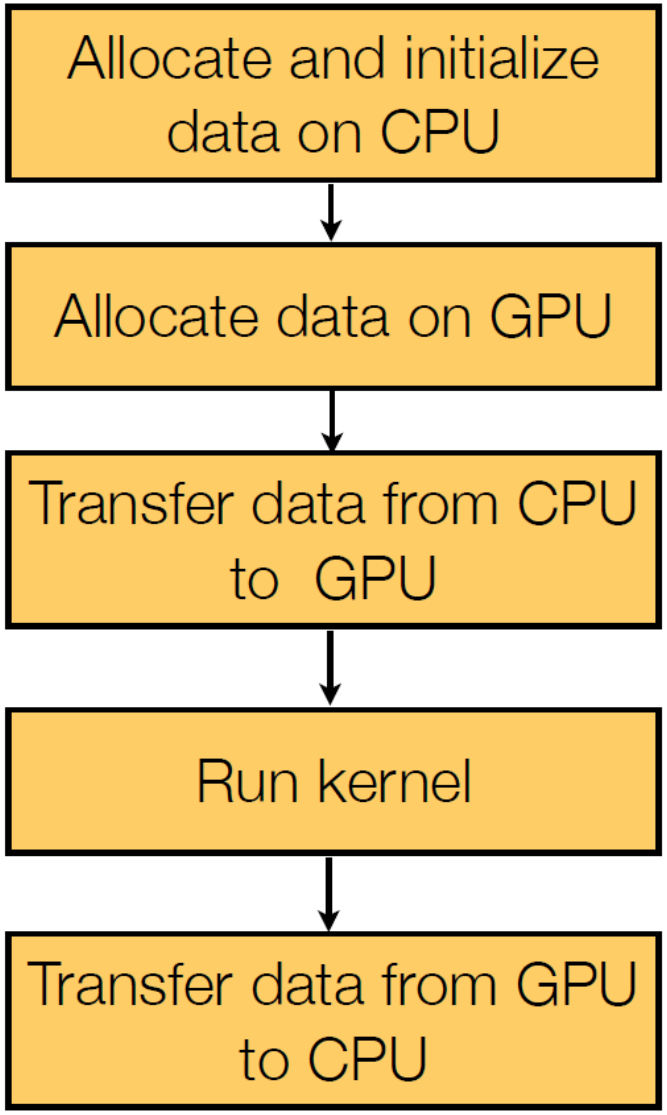
CUDA 運算 (2)

- 學習門檻低:
  1. C 語言
  2. 熟悉 CUDA 的 C 語言延伸，Runtime 函式庫
  3. 學習使用內建變數管理執行緒.
- 效能調校:
  1. GPU 運作的特性
  2. 各種記憶體的特性、限制

CUDA 運算 (3)

- 撰寫 CUDA 程式時：
  1. CUDA 的精神在於輔助 CPU 做運算
  2. 修改部分 CPU 的程式碼達到改善效能的目的
  3. 鎖定欲修改的程式碼區段後，進行修改。

CUDA - Program execution



CUDA 程式設計模型 (1)

一、配置Device Memory (allocate memory on device)

- Device為GPU端 (Device is GPU)
- 使用cudaMalloc API

二、將Host端資料Copy至Device端的Memory (copy data from host to device)

- Host端CPU端 (Host is CPU)
- 使用cudaMemcpy API
- 參數設定：cudaMemcpyHostToDevice

三、執行Kernel (修改的程式區段) (execute the kernel function with multiple threads)

- 可執行多個Kernel
- Kernel所指為GPU所處理的函式

CUDA 程式設計模型 (2)

四、將Device端運算完之資料Copy回Host端 (copy data from device to host)

- 使用cudaMemcpy API
- 參數設定：cudaMemcpyDeviceToHost

五、釋放所分配之記憶體 (free allocated memory)

- 使用cudaFree API

Memory Management (記憶體管理)

- cudaMalloc()
  - 在裝置記憶體中配置一塊記憶體空間 (allocated memory on device)
- cudaMemcpy()
  - 在Device與Host之間的資料傳輸 (copy data)
- cudaFree()
  - 釋放裝置記憶體上的記憶體空間 (free memory)

Error Handling (錯誤訊息處理)

- CUDA的run time library有許多函式在執行完畢後會回傳錯誤代碼
- 可使用cudaGetErrorString()來解析出正確的錯誤訊息
- 如：

```
cudaError_ t R;
R=cudaMalloc((void**)&d_vecA,sizeof(float)*1024);
printf("error is %s\n", cudaGetErrorString(R));
```

Event Management (事件管理)

- CUDA的run time library提供一組精確的計時方式來測量程式碼區段的執行時間 (estimate the execution time of a code segment)
- 如：

```
cudaEvent_t start,stop;
float CPU_execution_time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);
.....
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&CPU_execution_time, start, stop);
printf(" Excuetion Time on GPU: %3.20f s\n",elapsedTime/1000);
```

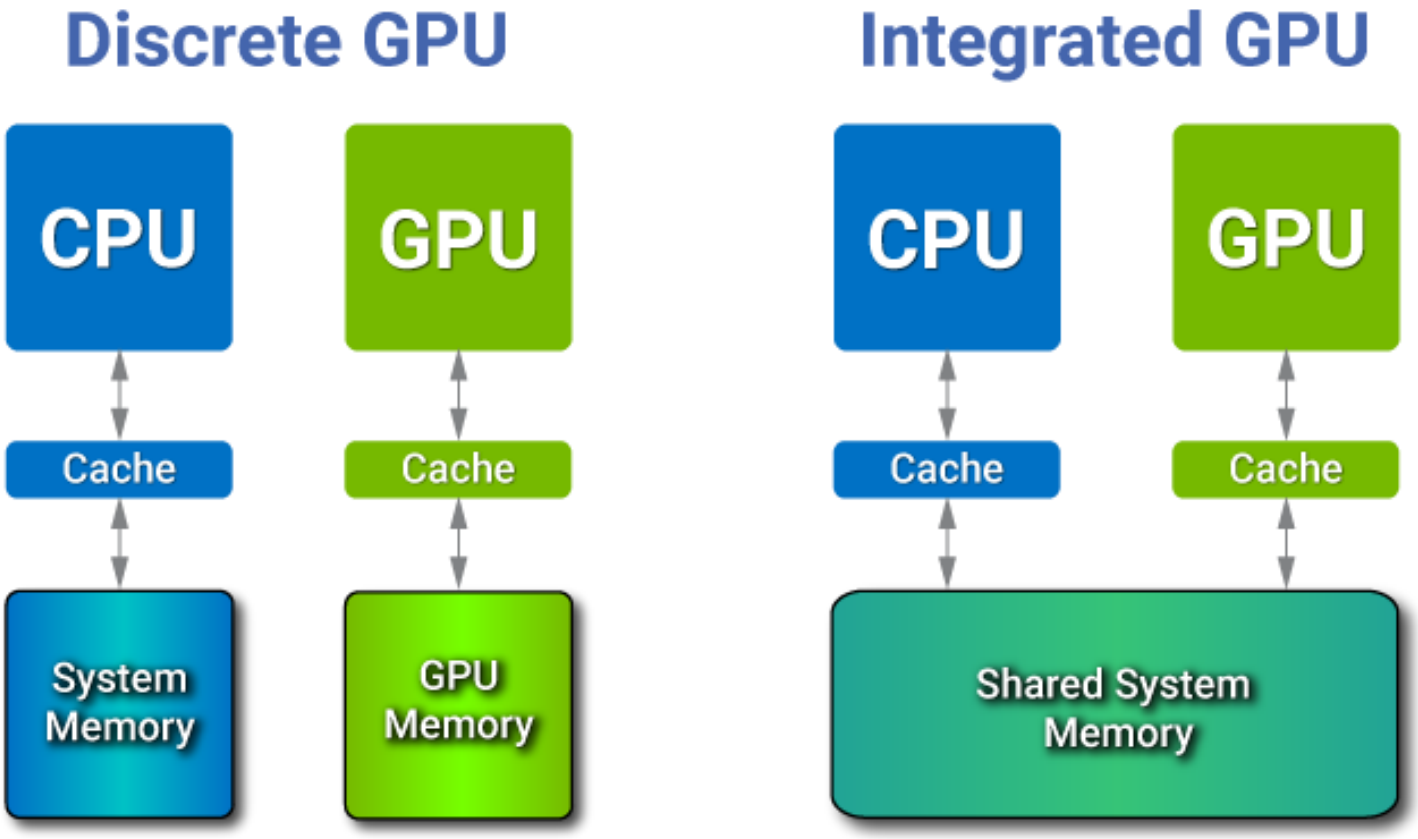
Thread Management (執行緒管理)

- CUDA程式執行時先以CPU的執行緒為主(Host Thread)，當執行核心函式時會發出指令將工作交給GPU處理，然後等待核心函式結束
- 當有兩個核心函式必須依序執行時，必須使用cudaDeviceSynchronize()/cudaThreadSynchronize()函式來指定CPU的執行緒要等待核心函式結束後，才能繼續執行下一步驟的程式碼

Thread Management (執行緒管理)

- __syncthreads(): 在assignment完後，呼叫此函式可以讓區塊中的所有執行緒看到新資料，常用在使用shared memory時。

兩種不同的架構



獨立顯卡，各自擁有記憶體

整合晶片，共享記憶體

不同硬體架構程式下的程式結構

- 獨立顯卡與整合晶片之程式結構並無差別
- 但整合晶片使用共享記憶體，其實是不需要做記憶體間的資料複製，因此，硬體底層以zero-copy機制排除了記憶體間的資料複製