Computer Organization and Assembly Language
Final Project Report

GPU Architecture and Programming:
From GPGPU to CUDA

GPU結構與程式架構的發展：從GPGPU到CUDA

工管四 B93701214 黃鈞鈺

Contents
目錄

Abstract
內容概要

這個學期以來，在很多的課堂上都聽到教授們提到多核CPU的興起，和同樣採用平行處理概念的GPU對於電腦運算能力的幫助，因此興起了研究的興趣。從GPU最開始的圖形處理開始了解，逐漸推展到GPU應用在一般的電腦運算，從硬體的結構衍生出對應的程式執行架構，也讓我對於平行處理的概念逐漸的清晰，也感受到平行處理的強大之處。

在上一個世代，CPU的發展速度一直隨著Moore's law的預言迅速的發展，每十八個月體積縮小一倍，效能也有一倍以上的提升，使得電腦的運算能力不斷的向上提高。由於硬體的發展太過迅速，使得很多程式的效能提升，都趨向尋求硬體的解決方法。由於硬體支援的方便性和開發迅速，使得使用者對於運算的需求不斷膨脹，但硬體的發展卻逐漸面臨到了極限：矽晶體間的距離無法小於原子距離，這代表單位面積下可以容納的電晶體數量有所極限，而這個數量和運算能力息息相關，迫使CPU的發展走向了另一個方向—多核，也使得平行處理成為新一代電腦運算的顯學。在硬體的發展速度逐漸收斂，而平行處理的新概念非常需要程式的配合，因此以平行處理的方式撰寫程式，是下一個世代的Programmer必要的能力。

以下將由GPU的原始架構，也就是在圖形運算器開始寫起，從傳統的圖形處理流程衍生出的原始GPU結構，到第二代Programmable GPU的出現，逐漸推演出GPU對於CPU運算能力的取代性，以及因應而生的GPGPU(General Purpose GPU)概念。接著說明在第二代硬體結構上的缺失，以及GPGPU程式概念的不足，而有新一代unified shader概念的GPU出現，以及配合unified shader，新的平行處理程式架構—CUDA(Computed Unified Device Achitecture)。最後深入介紹CUDA programming運作的原理，以及程式範例，來說明這個新的平行運算結構在程式效能的提升。

Evolution of GPU Architecture
圖形處理器在結構上的演進

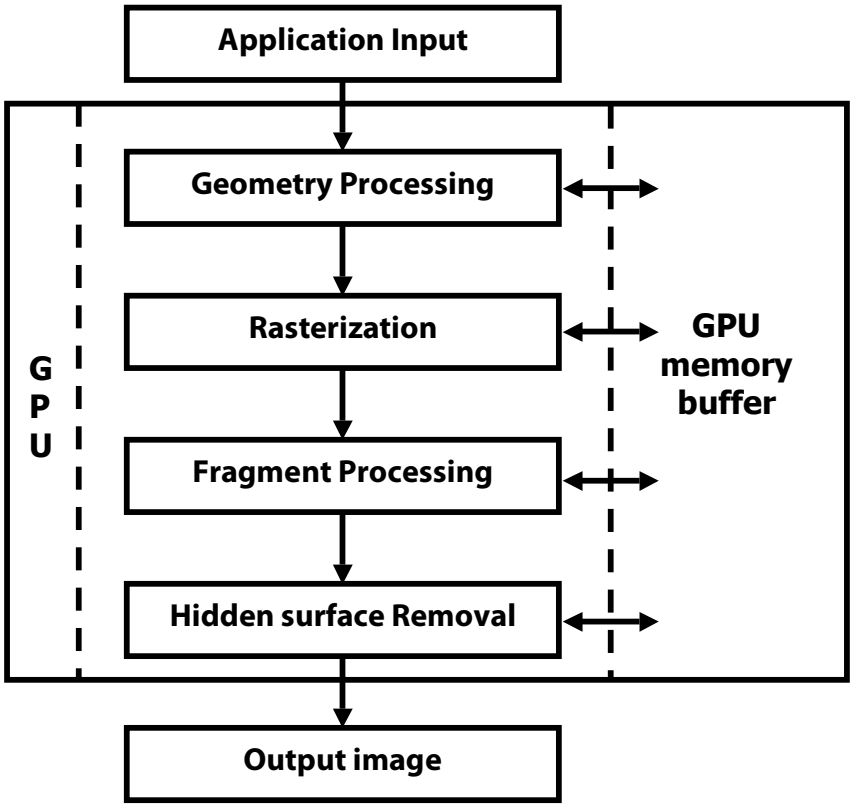
I. Evolution of The base of GPU Architecture

GPU(Graphics Processing Unit)，從字面上來解釋，它最初的功能便是滿足電腦顯示圖形方面的需求而產生的元件。而它從發展之初，為了要達到real-time顯示圖形的目的，因此便相當著重運算能力上的發展。和CPU重視單一個instruction執行時間不同，因為圖形必須整張顯示，GPU的效能重點在於從第一個到最後一個instruction處理的時間不能太長，所以GPU採用提升運算效能的方式，是提升同一時間可以處理的instruction數目，也就是所謂的平行處理(Parallel Computing)。

以下將由圖形顯示所需要處理的工作來看最初的GPU結構設計。

i. 傳統的圖形處理流程與圖形處理器結構
Traditional Graphics Pipeline and GPU Architecture

一般電腦要顯示一張圖片，需要以下的流程：

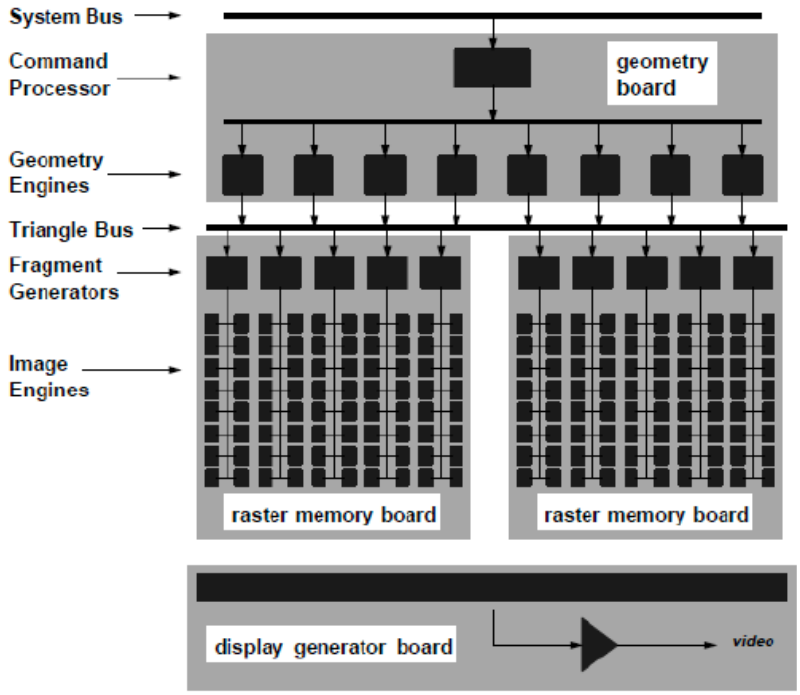


圖一、傳統的圖形輸出流程

1. Application input: 從應用程式輸入想要顯示的圖形資料及command。

2. Geometry processing: 又稱為Vertex processing，因為主要處理的是圖形的頂點資料。輸入的資料會有組成畫面結構的三角形頂點的3D座標，此時這個單元要把他轉成電腦顯示的2D座標，當作下一個單元的input。此時還要計算lighting對於三角形的著色效果和觀察點的位置，並把資料往下傳。
3. Rasterization: 因為電腦是以掃描線的方式逐點顯示圖形，因此要將上一個輸出的2D三角形使用內插法算出所有構成三角形點的座標。
4. Fragment processing: 使用rasterise 的結果，去填上每個點的texture。
5. Pixel processing: 使用z buffer的資料，檢查投影後的每個點是否被其他點的投影擋到，來決定該點有沒有在螢幕上顯示，決定最後的輸出結果。

因此產生的以下的GPU結構:



圖二、Kurt Akeley. RealityEngine Graphics. In *Proc. SIGGRAPH '93*. ACM Press, 1993.

在這個最初的設計中，有以下幾個特點：

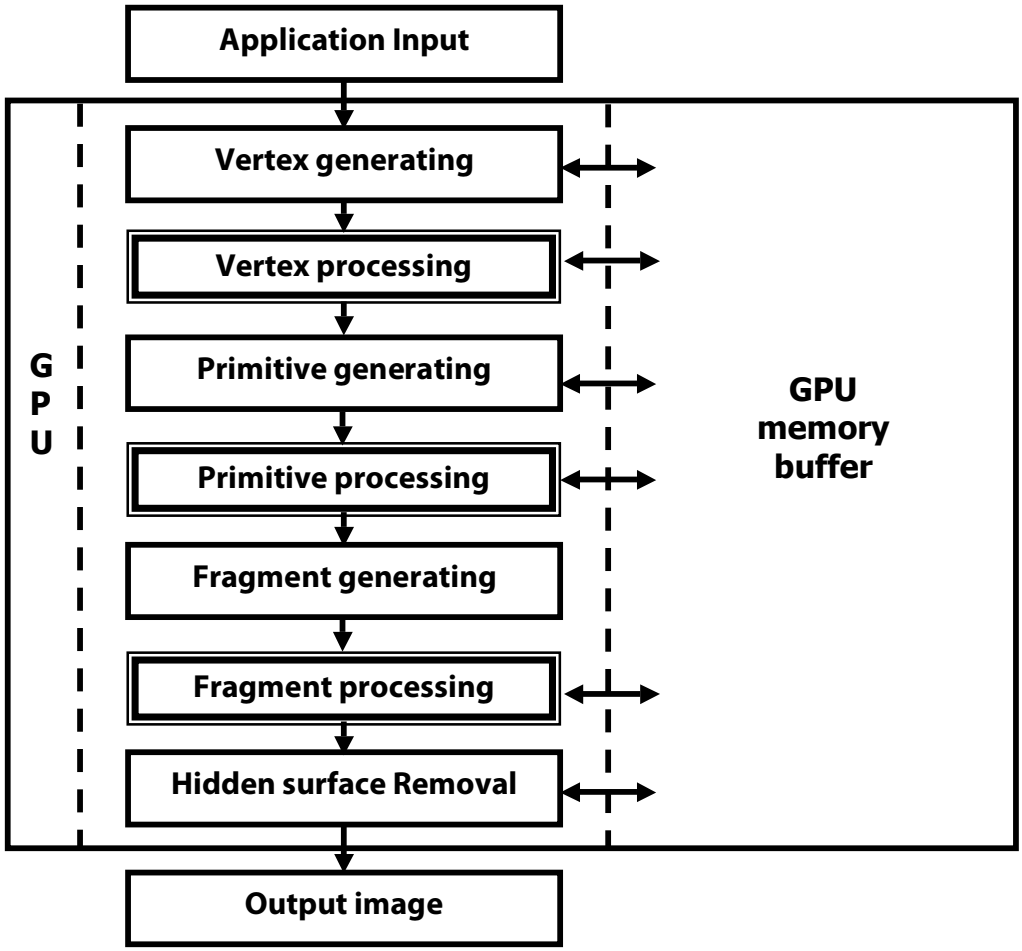
1. 已經採用的平行處理的方式來構成元件：從上面的GPU結構圖中可以看到Geometry Engine, Fragment generator, 都是平行並列的結構。
2. 結構採用fixed function設計，資料進入元件後只能進行固定的操作，沒有很大的彈性：雖然採用了平行處理，但是可以進行的運算是固定的，因此這個時期的GPU還沒有具有取代CPU的能力。

因為運算能力的限制，因此有了第二代GPU的改良：

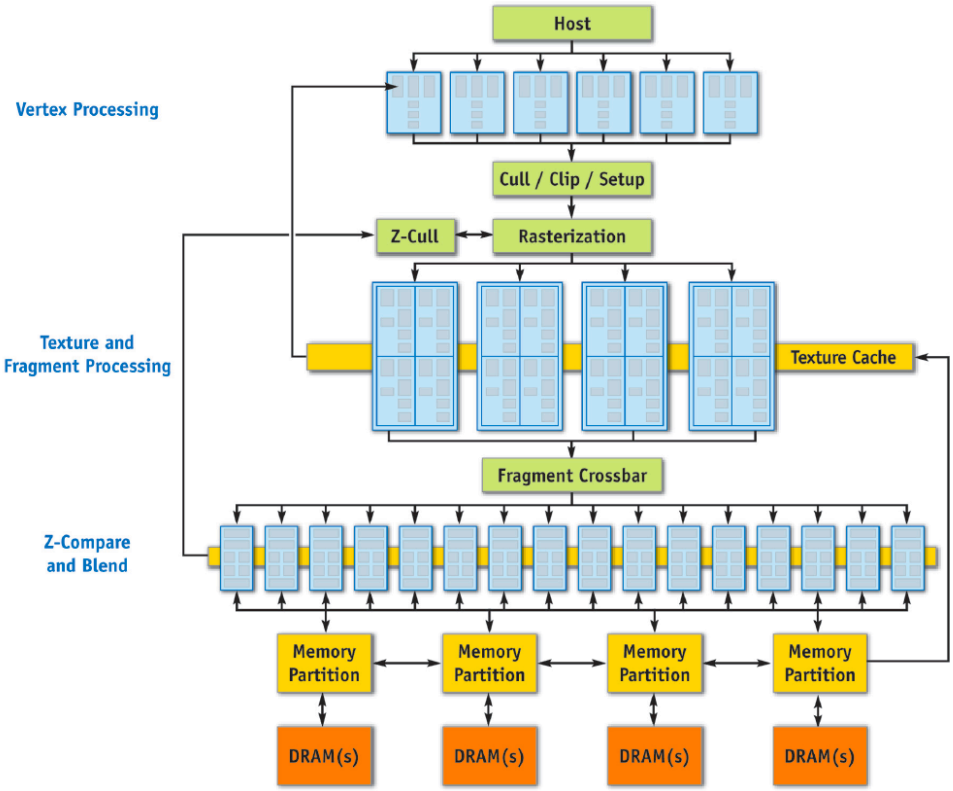
ii. 可程式化的圖形處理流程和圖形處理器的演進 (Programmable Graphics Pipeline and GPU Evolution)

第二代所作的改良主要是將原先固定功能的vertex 和 pixel processing unit改為programmable unit，稱為shader ，讓程式可以進行的操作變多，這樣的改變也增加了GPU內部結構的ALU能力，導致General Purpose GPU的概念產生。而pipeline在任務處理的本身也進行了改良：

1. Vertex 的處理分成Vertex Generating(VG)和Vertex Processing(VP)兩個步驟，VG是將構成圖形的lines, points, triangles以vertex record的形式表示，將這些recored再傳給VP處理。VP收到資料後，就將這些點轉換成投影到2D的座標，作成新的record再傳出去。所謂programmable的部份指的projection、model和view transform的運算。
2. Primitive Generating 和 primitive processing就是處理原先的geometry特性，必須要將同一個三角形的頂點group起來傳出去。
3. Fragment generating(FG)就是原先的Rasterization，將primitive sample在螢幕上，在將這些pixel點記錄成新的record傳出去給Fragment processing(FP)。FP則是從memory中讀出texture的資訊，並且將原始以vertex processing的lighting工作延到這個時候來作，這樣以來可以利用三角形每個點的特性，做到phong shading等較為真實的著色效果。
4. 最後的pixel operation則是做hidden surface removal。



GPU的實際結構則如下圖：



此時，GPU的結構已經具有很高的運算能力。

iii. 從圖形顯示到一般運算：圖形處理器對中央處理器的取代性(From Graphics to General Computation: CPU and GPU)

有了與CPU相近的運算能力，GPU如何處理一般的computing task可以從下表了解：

1. GPU pixel Shader task和CPU computing task的對應關係：

CPU	GPU
Data Array	Texture
Memory Read	Texture Lookup
Loop body	Shader Program
Memory Write	Render to Frame Buffer

從上表中可以看到，GPU要進行一般的運算，可以用和CPU運算類似的模式帶入，例如：原先存放texture資料的memory buffer可以用來存放data array。開始讀入資料時和texture look up進行的動作相同，要進行的運算的程式主體則和用shader language(openGL, cg等等)寫出的shader program類似，最後完成的運算則可以寫入frame buffer，等同於寫回記憶體的動作。因為這樣的相似性，而有了Genernal purpose GPU的概念產生。

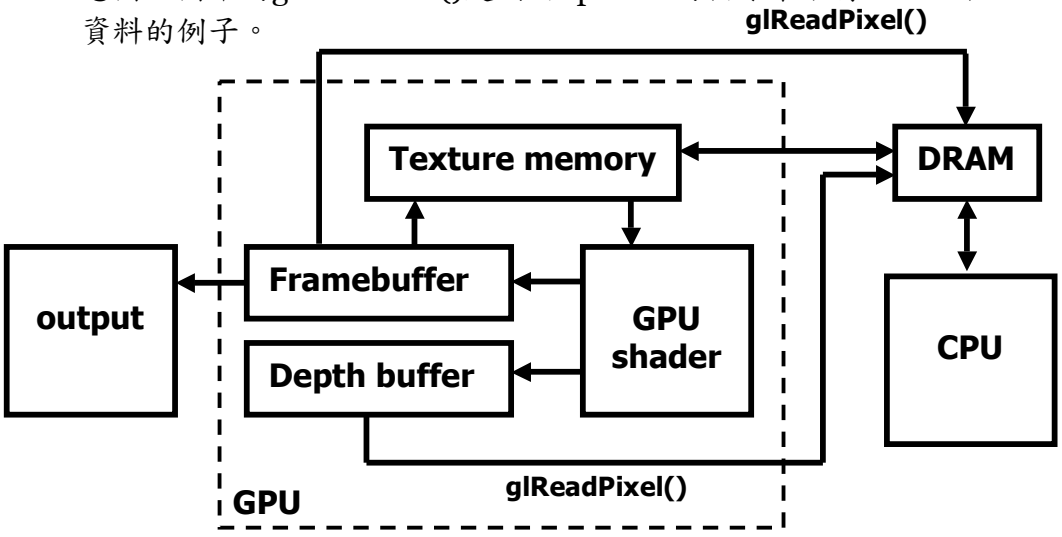
Evolution of GPU Programming Model
圖形處理器在運算架構上的演進

II. Evolution of GPU Programming Model

i. General purpose GPU

(1) 利用GPU良好平行運算能力，來進行一般性（不只是圖形）的運算，以分擔CPU的運算負擔。簡單來說，就是利用原先操作GPU的圖形運算指令，產生Thread在GPU裡面執行一般性的運算。一般的GPGPU使用video memory來做和CPU和DRAM類似的結構安排，而利用GPGPU的程式需要以圖形運算的程式語言來撰寫，例如openGL，cg等等。

(2) 程式運行架構：下圖即為GPU和CPU之間傳遞program訊息的示意圖。圖中的glReadPixel()是利用openGL的指令來取得DRAM裡面資料的例子。



3. GPGPU的缺點：

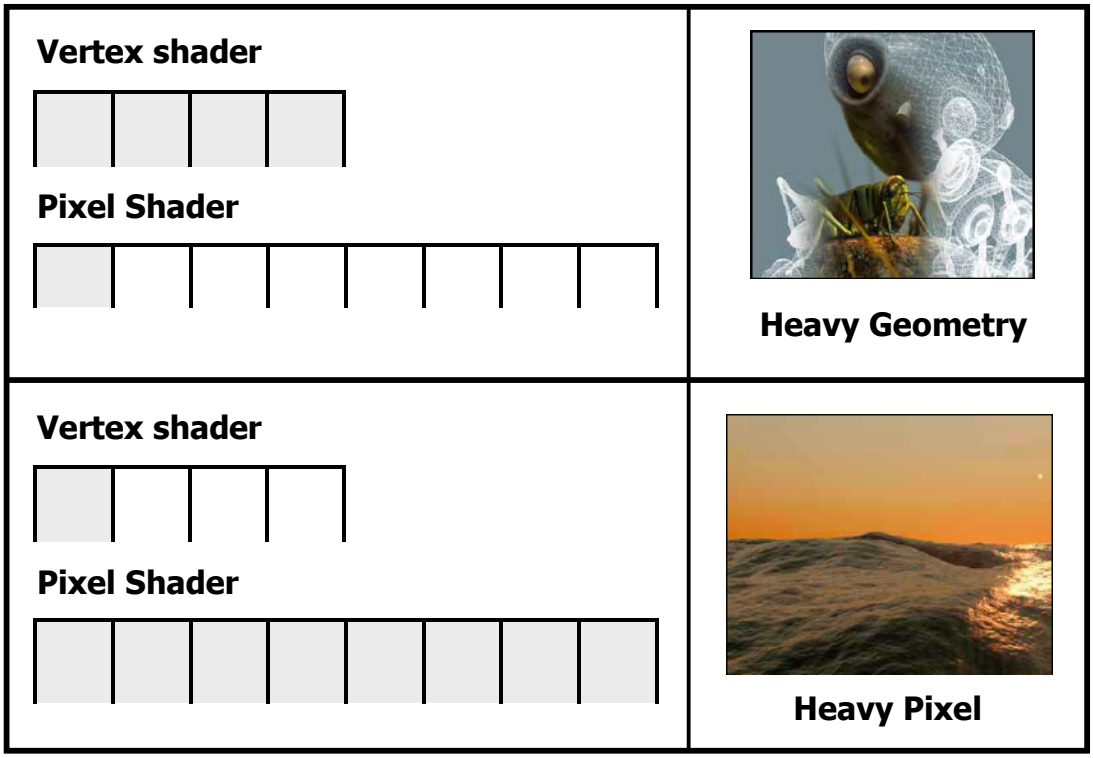
(1) 必須使用shader language來作操作，因為程式語言學習的成本造成了採用這套架構的進入障礙，使得非圖形運算領域的程式很難採用，也是GPGPU推行上最大的困難。

(2) 從架構圖上來看，資料的傳遞路徑很遠，而起必須重複在GPU和DRAM之間做資料的傳輸，但是DRAM是設計給CPU取得資料方便，因此跟GPU之間的距離一定比和CPU之間遠，造成效率低落。

因為這些缺點，使用GPU做備用運算單元能不普遍，因為仍然和CPU利用的方便性有很大的距離。但是隨著技術不斷的演進，第三代GPU的出現和GPU程式執行架構的改良，正式宣告了平行處理時代的來臨。

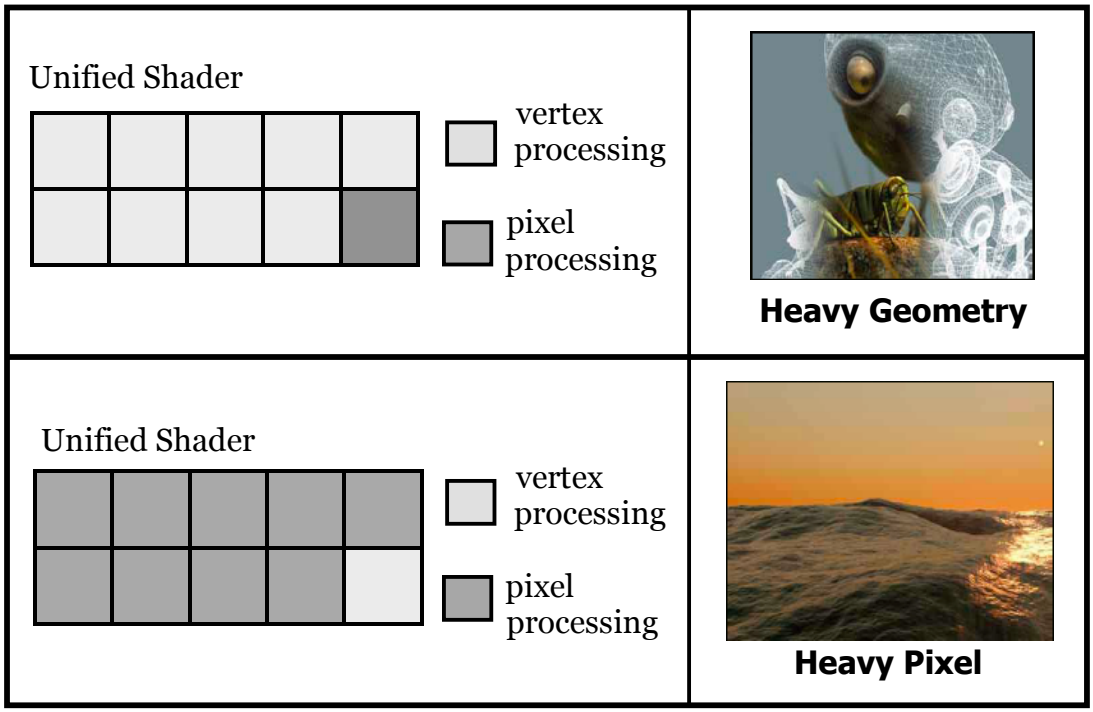
ii. Unified Shader GPU:

(1) Unified Shader GPU: 主要目的是再提升Programmable GPU的效率，由於圖形的處理流程採用垂直式的處理，必須要等到上一個階段處理完成才能進行下一個階段的工作，因此在效能上，可能會遇到這樣的情形：



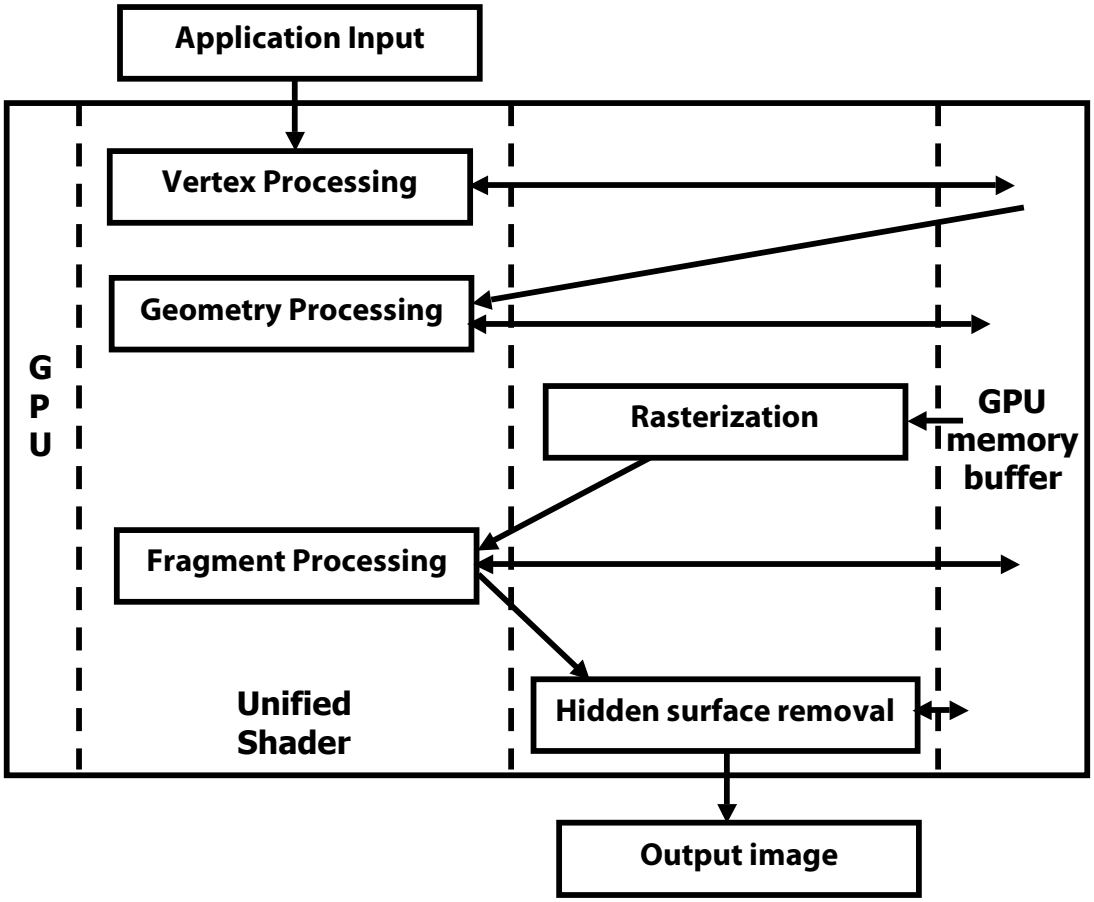
閒置中的shader capacity　使用中的shader capacity

在處理不同性質的輸出時，因為shader的功能被限制住了，因此即使仍然有多餘的工作能力，但是無法進行其他的工作只能閒置，造成資源的浪費。因此若能打破shader執行工作種類的限制，讓閒置中的單元可以分擔忙碌的單元，減少效能利用的不平均，就可以使得執行效率提升。再這樣的概念之下產生的，就是Unified Shader。拿掉了shader的功能定義，再一個shader不管是vertex processing或者是pixel processing，都可以進行運算，而改良後的結果，可以使上圖的執行結果變成：



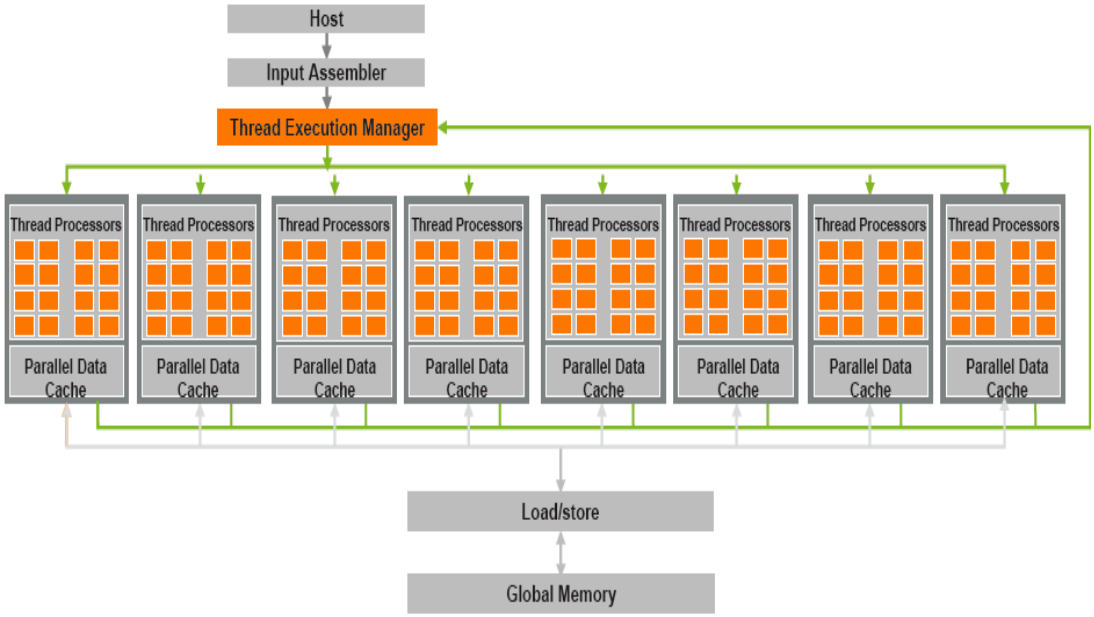
同時提升了兩種工作的效能，使得資源利用率大幅的提升。

(2) Graphics Pipeline of Unified Shader GPU



將shader的部份都使用unified shader取代，而unprogramable的部份則保留由原本的unit繼續執行。

(3) GPU實際的硬體實現



iii. CUDA: Compute Unified Device Architecture

(1) CUDA: 指的便是在unified shader的GPU architecture下，使用GPU作為一般性運算的處理單元。在CUDA的架構下，CPU稱為host為主要的運算單元，而GPU則是扮演著coprocessor的角色，稱為device。因此，在host上如果運算工作是有關平行的資料處理，或者需要大量的計算，host可以將這個工作分擔到device上去執行。

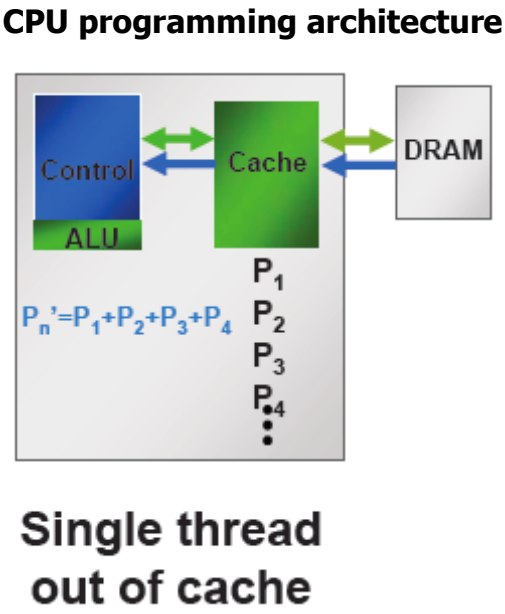
(2) 和前一代的GPGPU相比，CUDA有下面三個特點：

a) Shared memory: Host和device各自有獨立的memory，而device的memory則由device中的元件共享，這是受到了unified shader硬體設計的好處，因為在unified shader的架構下，每個shader比需要可以迅速的分享處理後的結果，以達到平行處理的作用。

b) Hierarchy of Thread Group: 將Thread分成個體thread, thread block 和grid of thread blocks，將每一個shader可以各自處理的城市段變多，充分發揮平行處理的優勢。

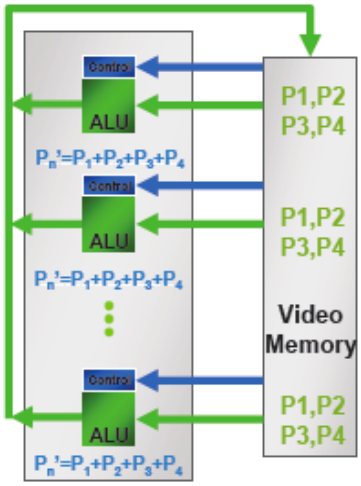
c) 為C的extension language: 之前提到GPGPU使用的是各種圖形操作的程式語言，造成了很大的使用困難，因此CUDA藉由runtime library的建立和支援，將整個架構建立在C的基礎之上，讓使用者的接受度提高，也讓使用的方便性提高。

iv. The Programming architecture of CPU, GPGPU, and CUDA



CPU的運作模式，使用單一個thread來執行，一次只跑一個指令。雖然需要從DRAM中讀出資料，但是在CPU本身有Cache的設置，可以利用refetch或者任何memory access的policy，預先將資料讀入Cache，以加快執行時的讀取速度。

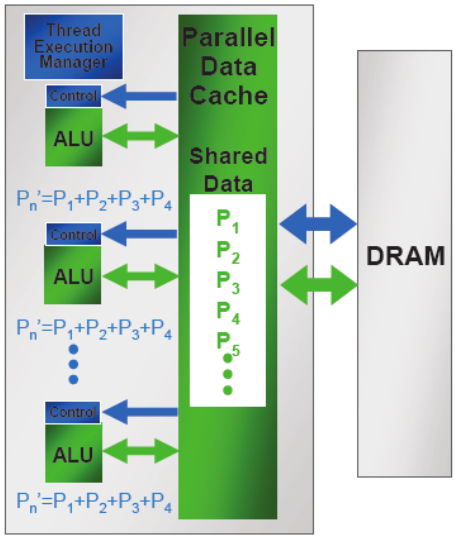
GPGPU programming architecture



Multiple passes through video memory

GPGPU的執行必須要不斷的從Video memory中讀出資料，再把運算完的資料寫回去，每次同步可以執行很多的instruction，每一個instruction有一個ALU component和control unit控制，等同於跑在一個有很多個CPU的機器上。但是缺點就是Video memory和實際執行的ALU是分屬在不同元件上，勢必有讀取效率和時間上的浪費。

CUDA programming architecture



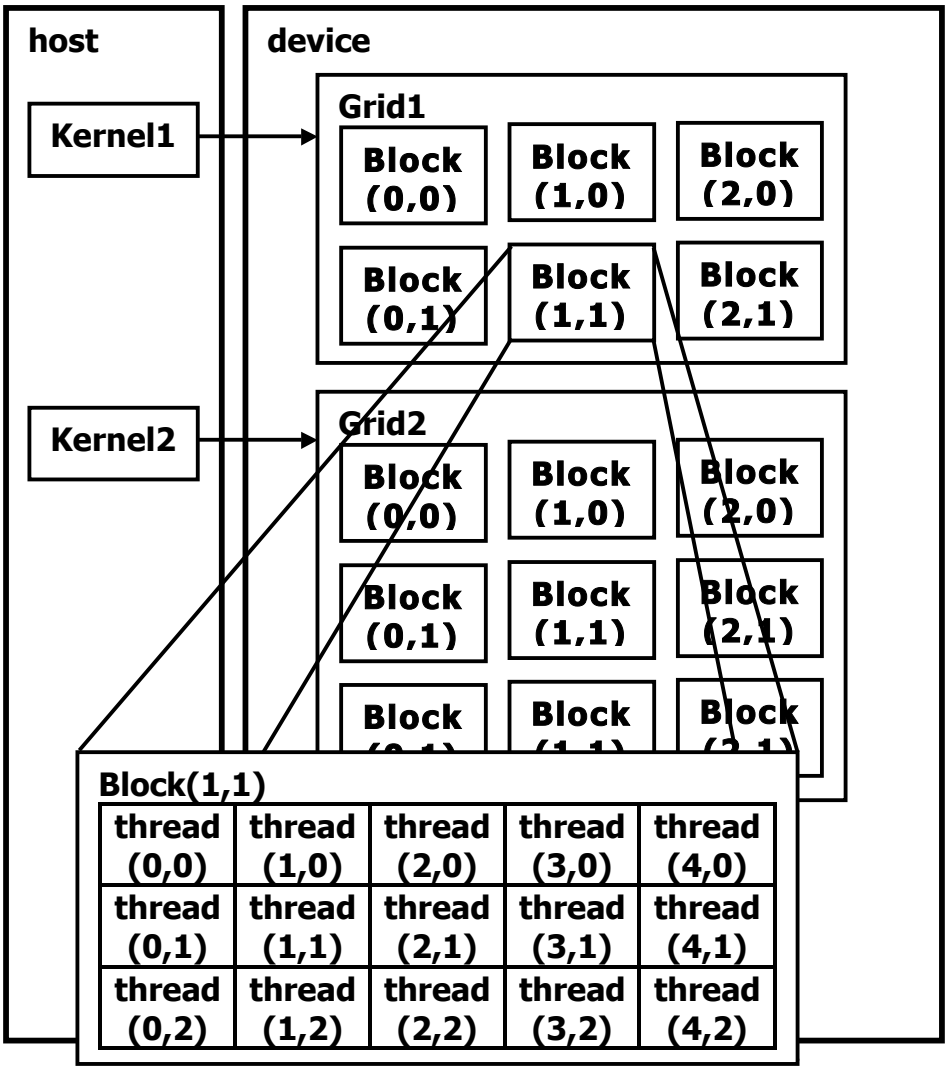
Parallel execution through cache

CUDA結合了GPGPU平行處理和CPU cache的優點，可以同時以很多的thread來執行不同的程式段，以增加運算的效率，還在GPU的本身建有Parallel Data Cache可以加快資料的讀取，再者因為不同thread因為執行同一程式的不同區塊，很有可能一筆資料要被很多的thread access，或者由thread執行完的結果，要很快的存回去給其他thread使用，因此建立的Cache必須要可以讓thread間很快速的分享資料。

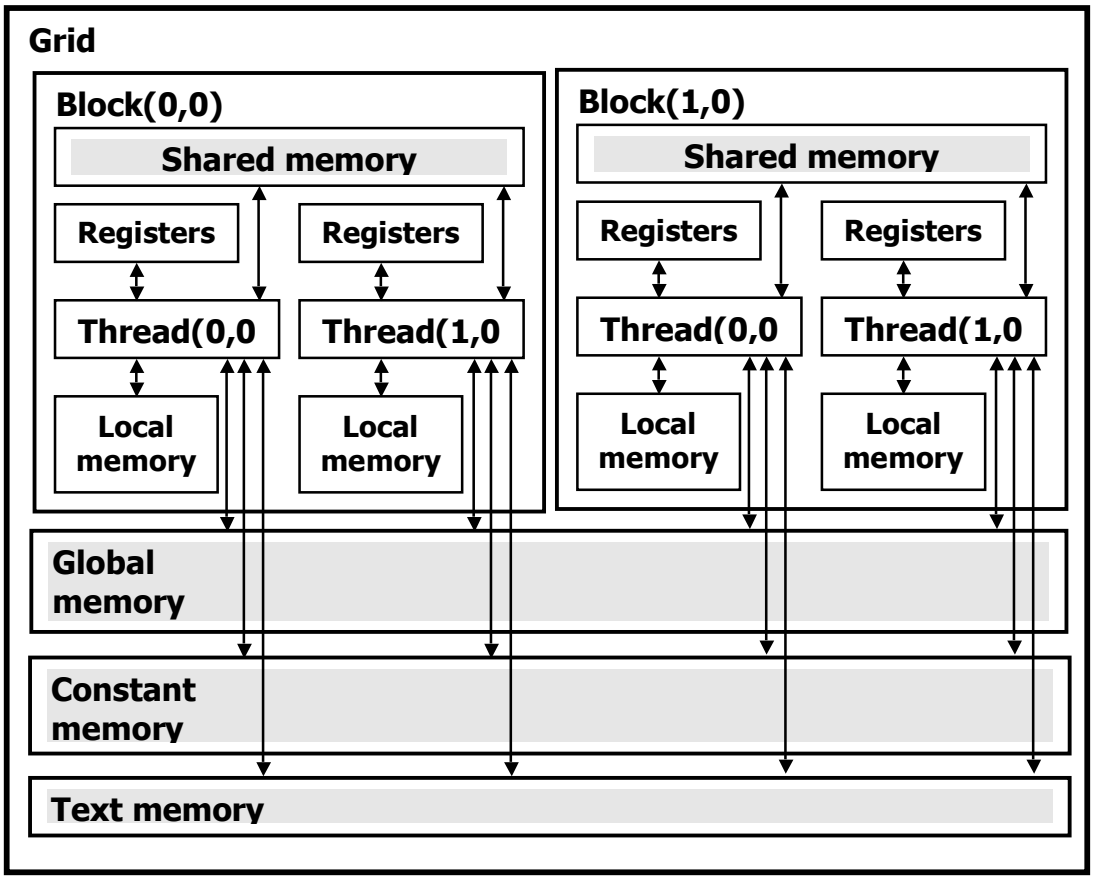
The Programming Model of CUDA
CUDA程式運行架構

III.The Programming Model of CUDA

i. CUDA的程式運作細節 (Detailed Architecture of CUDA Programming)
   a. Thread Batching
      i. Thread block:讓一群thread可以共同使用shared memory裡面的資料，分別執行access不同資料區段的同一程式，並且藉由同步化的執行可以保持資料的完整性。
      ii. Grid of thread blocks:一個block裡面包含的thread的數目有限，但在同一個kernel裡執行，而且有同樣維度和大小的thread blocks可以再組成grid of thread blocks，這樣一來，一個kernel可以使用的thread的數目就會提高。



b. Memory model

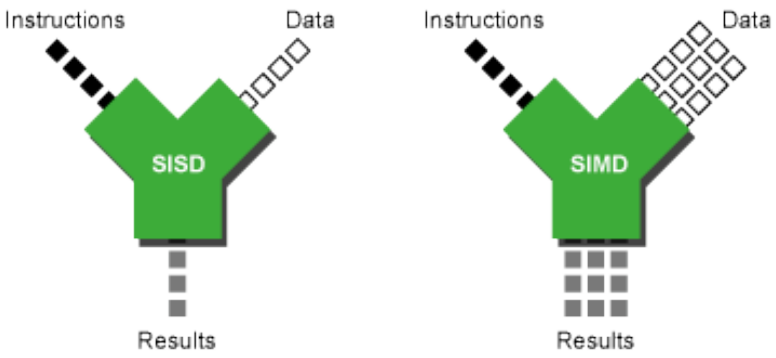


(1) Shared memory: 同一個block中的thread分享資料，外界不可讀取。

(2) Global memory:給不同的block但同一個grid中的thread分享資料用，host中的kernel可以access這裡的資料。

(3) Constant memory, Text memory: 給不同的block但同一個grid中的thread讀取資料，不能寫入， host中kernel可以access。

ii. 硬體實作細節(Hardware Implementation): 用一multiprocessor實現，每個multiprocessor具有SIMD(Single Instruction Multiple Data)架構：在給定的clock cycle，multiprocessor的個processor執行相同的指令，但操作在不同的資料上。



iii. 語法定義(Application Interface):以C語言為基礎，衍生出四種外加的形態定義。

a. Function type qualifiers: 用來指明以下的function是執行在host, device還是kernel，分別在程式的開頭宣告為_host_, _device_,或_global_ ，但是在device 和kernel的程式有很多限制，像是不支援recursion，不能宣告static variable，不支援動態配置的變數宣告等等。宣告的方式如：

```
_devise_ void fun(){
      程式內容；
}
```

b. Variable type qualifier: 指明某個device內的變數在記憶體中的位置，有_device_, _constant_, 和_shared_三種。宣告的方式如：

```
extern __shared__ float shared[];
```

表示宣告一個在shared memory中，形態為浮點數陣列的外部變數shared[]。

c. 宣告成_global_的function在執行時，需要以一個特殊形式的指令驅動，例如：
宣告為

```
__global__ void Func(float* parameter);
```

的kernel程式，必須以：

```
Func<<< Dg, Db, Ns >>>(parameter);
```

來執行。其中Dg代表grid of thread block的數量, Db代表thread block的數量, Ns表示每一個thread block在shared memory裡可以使用多少動態配置記憶體,

d. 四個build-in variable表示有grid dimension(gridDim), block index(blockIdx), block dimension(blockDim), thread index(threadIdx)等資訊。

iv. 程式範例(Example Program: Matrix Multiplication)

```
// 矩陣相乘：C = A * B
// hA 矩陣A的高度
// wA 矩陣A的寬度
// wB 矩陣B的寬度

void Mul(const float* A, const float* B, int hA,
int wA, int wB,
```

```
float* C)
{

  // 把Matrix A和B的資料load到device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
  // 在device中宣告一個空的space給C Matrix
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);
  // 計算block dimension和grid dimension
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
  // 呼叫device function
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
  // 把device算好的C的內容複製回host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
  // 清掉device使用的memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);

}

__global__ void Muld(float* A, float* B, int wA, int
wB, float* C)
{

  // Block index標示要執行這個動作的block的編號
     int bx = blockIdx.x;
     int by = blockIdx.y;
  // Thread index標示要執行這個動作的thread的編號
     int tx = threadIdx.x;
     int ty = threadIdx.y;
  // A第一個被執行的submatrix的block編號
    int aBegin = wA * BLOCK_SIZE * by;
  // A最後一個被block執行的submatrix的block編號
    int aEnd = aBegin + wA - 1;
  // A的submatrix執行的迴圈數
    int aStep = BLOCK_SIZE;
  // B第一個被執行的submatrix的block編號
    int bBegin = BLOCK_SIZE * bx;
  // B的submatrix執行的迴圈數 int bStep = BLOCK_SIZE * wB;
```

```
    // thread計算submatrix element的數目
      float Csub = 0;
    // 掃過所有的submatrix of A和B計算出結果
      for (int a=aBegin, b=bBegin; a<=aEnd; a+=aStep,
b+=bStep)
      {
          // 宣告一個給A的submatrix用的shared memory space
          __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

          // 宣告一個給B的submatrix用的shared memory space
            __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
          //從global memory load 矩陣的資料到shard memory去
          // 每一個thread load一個element
            As[ty][tx] = A[a + wA * ty + tx];
            Bs[ty][tx] = B[b + wB * ty + tx];
          // 同步讀取
             __syncthreads();
          // 兩個陣列相乘
          // 每個thread計算一個一個submatrix block的element
             for (int k = 0; k < BLOCK_SIZE; ++k)
             Csub += As[ty][k] * Bs[k][tx];
          //確保再thread讀入新的值前，上面的element值已經被算出來
          填入
             __syncthreads();
    }
    // 把block submatrix寫到global memory
    // 同樣一個thread負責寫一個element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

References
參考資料

IV.References

1. **How GPU works,** David Luebke, NVIDIA Research, Greg Humphreys, University of Virginia, IEEE computer, 2008
2. **GPU a Close Look,** Kayvon Fatahalian, Mike Houston, ACM Queue 2008
3. **NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.1**, NVIDIA Corporation, 2007
4. **Scalable Parallel Programming**, John Nickolls, Ian Buck, and Michael Garland, NVIDIA, Kevin Skadron, University of Virginia, ACM Queue 2008.
5. **A Survey of General-Purpose Computation on Graphics Hardware**, John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, Computer Graphics Forum, 26(1):80–113, March 2007
6. **GPU Gems2, Chapter 30 GeForce 6 Architecture**, Emmett Kilgariff, Randima Fernando, NIVIDIA Corporation, 2005
7. **GPU Computing with NVIDIA CUDA**, Ian Buck, NVIDIA Corporation, SIGGRAPH 2007 Lecture.
8. **GPU Architecture Overview,** John Owens, UC Davis, SIGGRAPH 2007 GPGPU lecture
9. **CUDA,** Wikipedia, http://en.wikipedia.org/wiki/CUDA