以通用計算圖形處理器(GPU)加速回歸測試套件精簡
Accelerating Test Suite Reduction Using General-purpose Computing on Graphics Processing Units

張洛嘉[1]、陳玟瑗[2]、林楚迪[3]
國立嘉義大學資訊工程學系[1, 2, 3]
Department of Computer Science and Information Engineering, National Chiayi University
Email: {s1013041[1], s1014679[2], chutilin[3]}@ncyu.edu.tw

摘　要

測試自動化可大幅降低軟體開發與維護成本，且有效提升回歸測試的準確性，因而近年來漸趨普及。然而，測試個案會隨著軟體的功能新增與改版而持續增加，即使測試以自動化方式執行，仍需耗費過多時間，導致開發進度的延遲。因此，測試團隊必須從大規模測試套件中挑選並執行適當的測試個案，以期在維持測試效益的前提下縮短回歸測試所需時間。此研究議題稱為測試套件精簡(Test Suite Reduction)，是多年來軟體工程領域極被看重的研究議題。值得注意的是，測試套件精簡雖可降低回歸測試時間，但精簡動作本身亦會耗時。所幸近年來平行運算已成為處理大量資料的主流做法，其中通用計算圖形處理器(General-purpose Computing on Graphics Processing Units，簡稱 GPU)擁有多個計算單元，若適當搭配 CUDA 語言則可充分發揮 GPU 優異的運算能力，可用於加速程式的運行或處理大量資料。因此，本研究計畫擬重新架構現有軟體測試套件精簡演算法，藉由 GPU 加速其精簡過程，以縮短軟體回歸測試所耗費的時間，進而提升軟體開發速度。

關鍵字：軟體測試、回歸測試、測試套件精簡、通用計算圖形處理器

一、前言

如同多數產品在出售前會接受嚴格的品質管控，軟體系統在交付給客戶或上線運作前也必須經過嚴格的把關。在眾多軟體品質提升與控制的手段中，軟體測試是相當常見且受到矚目的一種方法。軟體測試團隊會針對軟體的功能需求或程式碼設計測試個案(包含可能的輸入與操作過程，以及軟體執行後的預期結果)，並根據測試個案執行軟體系統，以比對其執行結果是否符合預期。軟體在開發過程中不斷修改，原本可通過測試的功能在經過修改後亦可能不正常運作。為避免此問題，每當軟體修改後，測試人員皆須重新執行已曾執行過的測試個案，以確保該功能仍可正確運行，此概念稱為回歸測試(Regression Testing)[1, 2]。

然而，隨著硬體能力的提升，軟體系統的規模也日趨龐大，為確保眾多功能的正確性，測試套件的規模也會相當驚人，且測試個案亦可藉由工具自動化地大量產生，若每次回歸測試皆由測試人員手動執行，除了耗費大量人力資源，測試耗費時間也相當可觀。事實上，許多生產工作均已採用自動化執行，軟體測試亦不例外。許多測試團隊已逐漸從以往的純人工測試提升至自動化測試，並藉由持續性整合系統(Continuous Integration，簡稱 CI)[3]的架設，讓回歸測試在夜晚下班時間自動執行，待隔日上班時即可檢視測試結果，軟體開發時程將可明顯縮短。

雖然測試自動化已經可以縮短大量開發時間與人力，但測試個案會隨著軟體的功能新增與改版而持續增加，若測試套件的規模成長到自動化回歸測試無法在規劃時間內完成，將導致軟體開發的效率降低。因此，測試團隊被迫從大規模測試套件中挑選並執行適當的測試個案，以期在維持測試效益的前提下縮短回歸測試所需時間。此研究議題稱為測試套件精簡(Test Suite Reduction)，是多年來軟體工程領域極被看重的研究議題。其定義如下[4]：

給定條件：

- 測試需求集合 $R = \{ r_1, r_2, r_3, \ldots, r_x \}$，其中 r_i $(i = 1, 2, \ldots, x)$表示各個相異的測試需求，x 表示測試需求的總量，且 R 中所有的測試需求必須全部被滿足(即測試執行過程中全部測試需求皆曾被執行過)；
- 測試套件 $T = \{ t_1, t_2, t_3, \ldots, t_y \}$，其中 t_j $(j = 1, 2, \ldots, y)$表示測試套件中的各個相異的測試個案，而 y 表示測試套件精簡前內含的測試個案數量；
- T 的子集合 $T_1, T_2, T_3, \ldots, T_x$，其中 T_i 表示所有可以滿足 r_i 的測試個案所形成之集合，此系列的子集合用於描述測試個案與測試需求間的關係。

目標：

- 在可滿足 R 中全部需求的前提下，從 T 中找出一個最小的代表集合 RS，做為精簡後的測試套件。

以表 1 中的測試套件為例，原始測試套件為 $T=\{t_1, t_2, t_3, t_4\}$，該套件共可滿足 R 中的 5 個測試需求(即 r_1, r_2, r_3, r_4, r_5)。然而，我們可發現，T 的子集合 $T'=\{t_1, t_3\}$即足以滿足所有測試需求，應該執行的測試個案數目為原本的 50%，測試套件精簡的重要性可見一斑。

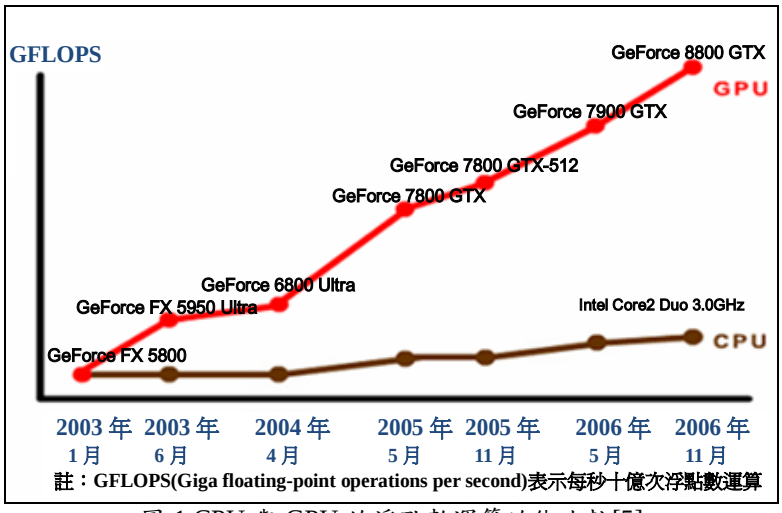

圖 1 CPU 與 GPU 的浮點數運算效能比較[5]

值得注意的是，測試套件精簡雖可降低回歸測試時間，但精簡動作本身亦會耗費時間。所幸近年來平行運算已成為處理大量資料的主流做法，其中通用計算圖形處理器(General-purpose Computing on Graphics Processing Units，簡稱 GPU)擁有多個計算單元，因而可用於加速程式的運行或平行處理大量資料，是近年受到矚目的應用。University of Alaska Fairbanks 的研究團隊[5]曾針對特定程式比較多個 GPU(包含 GeForce 8800 GTX、GeForce 7900 GTX、GeForce 7800 GTX、GeForce 6800 Ultra、GeForce FX 5950 Ultra 和 GeForce FX 5800 等)與著名 CPU(即 Intel Core2 Duo 3.0GHz)在進行浮點數運算時之效能，結果如圖 1 所示。雖未深究其實驗採用的程式，但由圖可以得知，此實驗中多數 GPU 的浮點數運算能力明顯優於 CPU，且其差距隨著 GPU 技術的提升而更加顯著。因此，本論文擬重新架構現有軟體測試套件精簡演算法，藉由 GPU 加速其精簡過程，以縮短軟體回歸測試耗費時間，進而提升軟體開發速度。

表 1 測試套件精簡範例

測試個案	r_1	r_2	r_3	r_4	r_5
t_1	•	•		•	
t_2	•				•
t_3		•	•		•
t_4			•		•

註：”•”表示該測試個案可滿足所對應的測試需求。

二、文獻回顧與探討

2.1. 貪婪演算法(Greedy Approach)

由於測試套件精簡問題類似最小集合覆蓋問題(Minimal Set-covering Problem)，故可採用貪婪演算法(Greedy Algorithm，後續簡稱 Greedy)搜尋區域最佳解(Local Optimal Solutions)，以期得到測試套件精簡問題的近似最佳解[6]。若以 *coverage*(*t*) 代表測試個案 *t* 所覆蓋的測試需求數目，此方法在精簡過程持續從測試套件 *T* 中挑選 *coverage*(*t*)最大的測試個案，並將其移至另一集合 *RS* 中，直到 *RS* 中的測試個案可滿足所有測試需求為止，最後得到的 *RS* 即是 *T* 的一個代表集合。執行回歸測試時若僅執行 *RS* 中的測試個案，將可縮短回歸測試所需時間，並維持一定測試成效。Greedy 演算法的虛擬碼請參見圖 2，此演算法的計算時間複雜度為 $O(x \times y \times \min(x,y))$[7]。事實上，許多現有的測試套件精簡演算法皆是依據 Greedy 的精神所設計，這些演算法皆是根據不同標準持續地挑選當下最適當的測試個案，以期能由區域最佳解得到近似最佳解。因此，若 GPU 平行優勢可有效加速 Greedy 的執行，則將這些設計概念類似的演算法皆可望蒙受其利。

```
1  algorithm Greedy
2  input   T: the set of test cases for the program
3          R: the set of test requirements in the program
4          S: the relation S={(t, r)| t satisfies r, t∈T, and r∈R}
5  output RS: a representative set of T
6  begin
7          RS = ϕ;
8          while (R ≠ ϕ)
9              t = the test case with the maximum coverage(t) in T;
10             RS = RS ∪ {t};
11             T = T – {t};
12             R = R – {r| (t, r)∈S};
13         return RS;
14 end
```

圖 2 Greedy 演算法之虛擬碼

2.2.比例度量精簡方法(Ratio Approach)

章節 2.1 所介紹的 Greedy 是以降低代表性集合中測試個案數目為目的，但每個測試個案的執行時間不同，測試個案少的代表性集合不代表會耗費較少的測試執行時間。因此，Smith 和 Kapfhammer[8]提出以比例度量(Ratio Metric)評估測試個案價值的測試套件精簡方法(後續簡稱 Ratio)，目的即是降低執行代表性集合中測試個案

所耗時間。比例度量的定義如下

$$ratio(t) = coverage(t) \,/\, cost(t)，\quad (1)$$

其中 t 代表受評估的測試個案，$coverage(t)$代表此測試個案所滿足的測試需求之數量，而 $cost(t)$表示執行此測試個案所耗費的時間成本。因此，$ratio(t)$代表每單位測試成本可滿足的測試需求數量，值越高表示將測試個案 t 選入代表集合所獲得的價值越高；相反地，較低的 $Ratio(t)$值代表 t 是價值較低的測試個案。相較於 Greedy 在精簡過程中持續地從候選測試個案中選出 $coverage(t)$值最大的測試個案，Smith 和 Kapfhammer[8]等人提出的 Ratio 演算法將比例度量套用於 Greedy，精簡過程持續地從候選測試個案中選出 $ratio(t)$值最大的測試個案，直到代表集合可滿足所有測試需求為止。若以 $ratio(t)$取代將圖 2 中的 $coverage(t)$，即可得到 Ratio 演算法，而其計算時間複雜度與 Greedy 同為 $O(x \times y \times \min(x,y))$[1]。事實上，除了套用於 Greedy，比例度量也可套用於任何奠基於 Greedy 的測試套件精簡演算法上，以提升該演算法對於測試套件執行成本的精簡能力[1]。

2.3.效率度量精簡方法(Efficiency Approach)

章節 2.1 和 2.2 介紹的兩個演算法以降低代表性集合中測試個案數目為目的，或將測試個案執行的成本納入考慮，但未考慮每個測試個案的不可取代性[1]。Jones 和 Harrold[9]曾提出測試個案之貢獻度(Contribution)的概念，由於每個測試個案所滿足的測試需求不同，當一個測試需求被許多測試個案滿足時，則可考慮挑選其中成本較低者。基於貢獻度的概念，Lin 等人[1]提出評估測試個案之不可替代性的公式如下：

$$irreplaceable(t) = \begin{cases} 0, & 若\ t 無法覆蓋任何需求， \\ \sum_{r_i \in R_t} \frac{1}{CovNum(r_i)}, & 若\ t 可覆蓋任一需求， \end{cases} \quad (2)$$

其中 t 代表受評估的測試個案，$irreplaceable(t)$表示此測試個案之不可替代程度；R_t 代表可被測試個案 t 滿足的需求之集合，其中每個獨立的 $r_i \in R_t$ 分別代表可被 t 滿足的測試個案，而 $CovNum(r_i)$表示原始測試套件 T 中可滿足 r_i 的測試個案數目。換句話說，$CovNum(r_i)$可以視為 t 針對 r_i 的可替代程度，而其倒數即為不可替代度。在此，針對 R_t 中每個獨立的 r_i 的不可替代度數值進行加總，計算出來的值越大表示其測試個案 t 的不可替代程度越高。

若基於不可替代的概念，進一步考量測試個案的執行耗費成本，測試個案的挑選優先序可參考

$$efficiency(t) = irreplaceable(t) \,/\, cost(t)，\quad (3)$$

其中 t 代表受評估的測試個案，$cost(t)$表示執行此測試個案所耗費的時間成本。因此，$Efficiency(t)$數值越大表示該測試個案每單位執行成本的不可取代性越高。在 Lin 等人[1]提出的效率度量精簡方法(後續簡稱 Efficiency)中，每個測試個案的不可取代性將會被納入考量，選出每單位執行成本中不可取代性較高的測試個案，以減少軟體回歸測試時成本消耗，進而使精簡效果得到提升。其精簡過程持續地從候選測試個案中選出 $efficiency(t)$值最大的測試個案，直到代表集合可滿足所有測試需求為止。若以 $efficiency(t)$取代將圖 2 中的 $coverage(t)$，即可得 Efficiency 演算法。由於 $efficiency(t)$的計算較耗時，因而 Efficiency 的計算時間複雜度與 Greedy 和 Ratio 不同，高達 $O(x \times y \times \min(x,y) \times z)$，在此 z 表示單一測試個案可滿足的測試需求最大數目[1]。如同 $ratio(t)$，$efficiency(t)$亦可套用於任何奠基於 Greedy 的精簡演算法[1]。

2.4.文獻綜合評述

雖然文獻[1]顯示，本章節所介紹的測試套件精簡方法皆能有效降低測試成本，但其精簡過程所耗費時間亦相當可觀。從其演算法的時間複雜度即可發現，當測試套件規模龐大時，此精簡過程也會是回歸測試的顯著負擔之一。若能進一步善用 GPU 的平行化計算優勢來加速此精簡過程，回歸測試的耗時將可更進一步縮短。

三、研究方法

3.1.GPU 加速應用簡介

隨著數位科技的發展，數位資料的規模急劇增加，電腦工程相關領域的研究人員必須找尋新方法來處理這些龐大的資料。近年來，平行運算已成為處理大量資料的主流做法。其中 GPU 擁有多個計算單元，適用於加速程式的運行或平行處理大量資料。根據文獻，不少名列 TOP 500 和 Green 500[10]的超級電腦(例如：臺灣國網中心的 Formosa 5)也採用 GPU 以提升其效能[11]。GPU 與 CPU 協作示意圖請參見圖 3，由此圖可發現，CPU 發出運算命令給 GPU，並將運算資料傳送至 GPU 內建的記憶體空間，經 GPU 運算完後再將運算結果傳回主記憶體 (即 CPU 可存取的記憶體空間)。

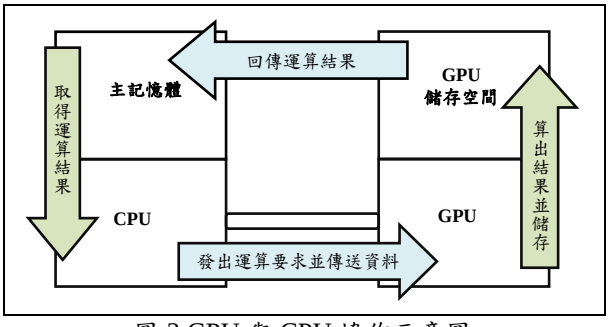

圖 3 GPU 與 CPU 協作示意圖

圖 4 所示的 GPU 架構圖則說明 GPU 中的每一個執行序區塊 (Block) 內可啟動多個執行序 (Threads)，在 GPU 中執行序為基本的運算單元。雖然資料在主記憶體和 GPU 內建記憶體之間搬運也會耗時，但當執行序的數量增多，平行運算所省下的時間可補償此搬運時間，GPU 平行運算的效

益即可浮現。此外，如果同時開啟多個執行序區塊，則執行效益將可以倍數成長。

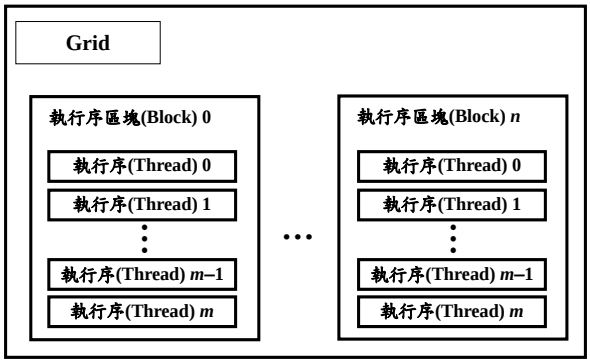

圖 4 通用計算圖形處理器架構圖

我們曾分別以 CPU(Core i7，處理器時脈約為 2500MHz)和 GPU(Nvidia GT 630M，處理器時脈約為 1500MHz)執行用於計算平方和的簡易程式，若採 Core i7 執行共耗費 29,887,816 個 CPU 執行時脈；若改採 Nvidia GT 630M，執行約花費 14,841,296 個 GPU 執行單元的時脈。換算後發現，採用 GPU 的執行速度為純 CPU 執行速度的 1.21 倍。另外，一個執行序區塊中可啟動多個執行序，若同時啟動 GPU 中的多個執行序區塊，平行運算的效能應能有更為顯著的提升。我們將此概念套用於平方和運算程式後，所花費的處理器時脈可大幅降至 396,146 個，亦即效能約為原先之 37 倍(從 14,841,296 個處理器時脈降至 396,146 個處理器時脈)。因此，與純 CPU 執行速度相較，約有 45 倍的效能提升。

上述簡易的實驗讓我們對 GPU 加速的成效相當期待，這也驅使我們討論將其應用於測試套件精簡的可能性。在此研究中，我們擬利用 GPU 平行運算的優勢改良現有的測試套件精簡方法，以期降低測試套件精簡時所需消耗的時間。

3.2.以 GPU 加速測試套件精簡

針對章節 2.1-2.3 所介紹的 Greedy、Ratio 和 Efficiency 等三個演算法，我們皆會先說明以 CPU 執行的主要步驟，再指出可藉 GPU 加速的部分，最後則說明這些部份如何以 GPU 進行加速。

3.2.1. Greedy 的實作與比較

若單以 CPU 執行 Greedy，程式執行步驟如下所示，執行流程圖請參見圖 5：

Step 1. 初始化： 將已被滿足之測試需求的數量設為 0；

Step 2. *coverage*(*t*)分析：計算各個尚未被選取之測試個案的 *coverage*(*t*)(計算 *coverage*(*t*)時必須排除已經被滿足之測試需求)；

Step 3. 挑選測試個案：從尚未被挑選的測試個案中挑選出 *coverage*(*t*)最高的測試個案，並將此測試個案的狀態設為已被挑選；

Step 4. 測試需求記錄更新：將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 5. 判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 2 繼續執行；否則，即進行 Step6；

Step 6. 回傳測試套件：回傳所挑選的測試個案之集合(即精簡後之測試套件)，並結束運算。

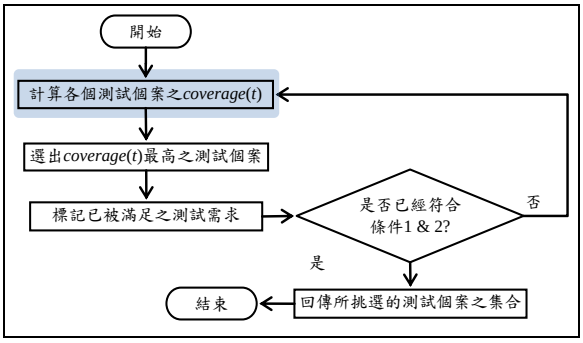

圖 5 以 CPU 實作 Greedy 之程式執行流程

圖 6 展示以 GPU 為主要運算元件所實作的 Greedy 演算法之程式執行流程，相較於以 CPU 為主要運算元件的做法，圖中以橘色區塊標示的為

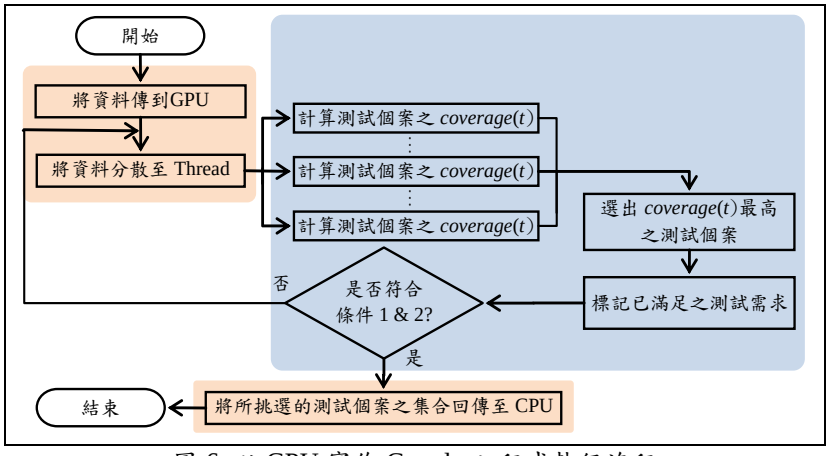

圖 6 以 GPU 實作 Greedy 之程式執行流程

CPU 與 GPU 之間所需的資料來回傳遞，是額外增加的成本，而以藍色區塊標示的是可藉 GPU 平行化加速的運算，是時間成本減少的主要來源。若以 GPU 為主要運算元件實作 Greedy，執行步驟如下所示：

Step 1. 資料傳輸：將所有測試個案的資料(包含測試個案與測試需求的對應關係)複製至 GPU 儲存空間；

Step 2. 初始化：將已被滿足之測試需求的數量設為 0；

Step 3. 平行化 *coverage*(*t*)分析：將測試個案分群，派送至多個執行序平行化計算每個測試個案的 *coverage*(*t*)(計算 *coverage*(*t*)時必須排除已經被滿足之測試需求)；

Step 4. 平行化挑選測試個案：平行化選出 *coverage*(*t*)最高的測試個案(每個 Thread 分別從各分群中挑選 *coverage*(*t*)最高的測試個案，並將這些測試個案再次分群並派送至多個 Threads 平行挑選。重複此動作，直到挑選出 *coverage*(*t*)最高的測試個案)；

Step 5. 平行化測試需求記錄更新：平行化將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 6. 平行化判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 3 繼續執行；否則，即進行 Step7；

Step 7. 回傳測試套件：回傳所挑選的測試個案之集合(即精簡後之測試套件)，將資料複製至 CPU，並結束運算。

3.2.2. Ratio 的實作與比較

若單以 CPU 執行 Ratio，程式執行步驟如下所示，執行流程圖請參見圖 7：

Step 1. 初始化：將已被滿足之測試需求的數量設為 0；

Step 2. *ratio*(*t*)分析：計算各個尚未被選取之測試個案的 *ratio*(*t*)(計算 *ratio*(*t*)時必須排除已經被滿足之測試需求)；

Step 3. 挑選測試個案：從尚未被挑選的測試個案中挑選出 *ratio*(*t*)最高的測試個案，並將此測試個案的狀態設為已被挑選；

Step 4. 測試需求記錄更新： 將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 5. 判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 2 繼續執行；否則，即回傳所挑選的測試個案之集合，結束運算。

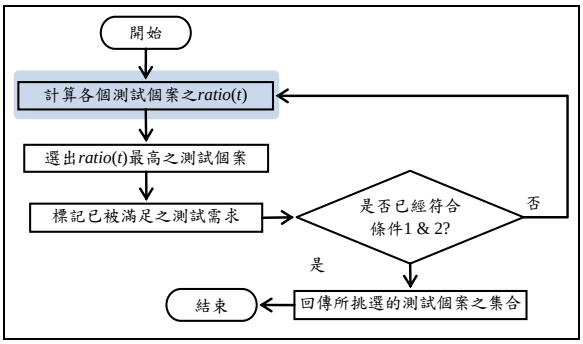

圖 7 以 CPU 實作 Ratio 之程式執行流程

以 GPU 為主要運算元件所實作的 Ratio 演算法如圖 8 所示，與圖 6 相似，橘色區塊和藍色區塊標示的分別是額外增加的傳輸成本與減少的運算。其詳細執行步驟如下所示：

Step 1. 資料傳輸：將所有測試個案的資料(包含測試個案與測試需求的對應關係)複製至 GPU 儲存空間；

Step 2. 初始化：將已被滿足之測試需求的數量設為 0；

Step 3. 平行化 *ratio*(*t*)分析：將測試個案分群，派送至多個執行序平行化計算每個測試個案的 *ratio*(*t*)(計算 *ratio*(*t*)時必須排除已經被滿足之測試需求)；

Step 4. 平行化挑選測試個案：平行化選出 *ratio*(*t*)最高的測試個案(每個 Thread 分別從各分

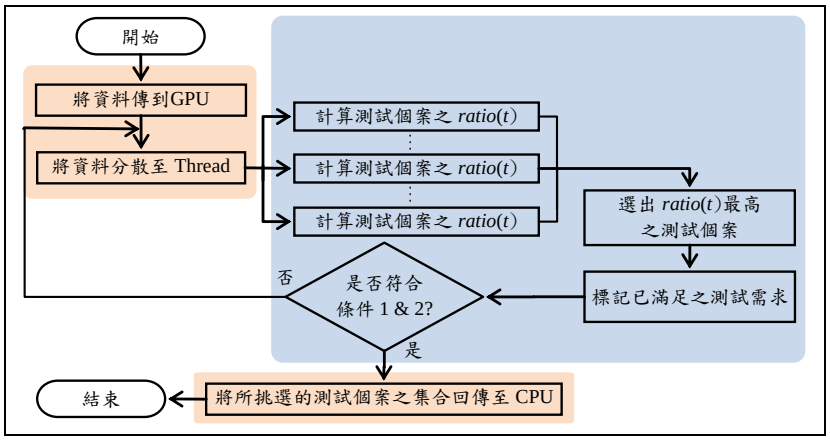

圖 8 以 GPU 實作 Ratio 之程式執行流程

群中挑選 *ratio*(*t*)最高的測試個案，並將這些測試個案再次分群並派送至多個 Threads 平行挑選。重複此動作，直到挑選出 *ratio*(*t*)最高的測試個案)；

Step 5. 平行化測試需求記錄更新：平行化將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 6. 平行化判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 3 繼續執行；否則，即進行 Step7；

Step 7. 回傳測試套件：回傳所挑選的測試個案之集合(即精簡後之測試套件)，將資料複製至 CPU，並結束運算。

3.2.3. Efficiency 的實作與比較

若單以 CPU 執行 Efficiency，程式執行步驟如下所示，執行流程圖請參見圖 9：

Step 1. 初始化：將已被滿足之測試需求的數量設為 0；

Step 2. *efficiency*(*t*)分析：計算各個尚未被選取之測試個案的 *efficiency*(*t*)(計算 *efficiency*(*t*)時必須排除已經被滿足之測試需求)；

Step 3. 挑選測試個案：從尚未被挑選的測試個案中挑選出 *efficiency*(*t*)最高的測試個案，並將此測試個案的狀態設為已被挑選；

Step 4. 測試需求記錄更新： 將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 5. 判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 2 繼續執行；否則，即回傳所挑選的測試個案之集合，結束運算。

圖 10 展示以 GPU 為主要運算元件實作 Efficiency 演算法的之程式執行流程，與圖 6 和圖 8 相似，橘色和藍色區塊標示的分別是額外增加或減少的成本。若以 GPU 為主要運算元件實作 Efficiency，執行步驟如下所示：

Step 1. 資料傳輸：將所有測試個案的資料(包含測試個案與測試需求的對應關係)複製至 GPU 儲存空間；

Step 2. 初始化：將已被滿足之測試需求的數量設為 0；

Step 3. 平行化 *efficiency*(*t*)分析：將測試個案分群，派送至多個執行序平行化計算每個測試個案的 *efficiency*(*t*)(計算 *efficiency*(*t*)時必須排除已經被滿足之測試需求)；

Step 4. 平行化挑選測試個案：平行化選出 *efficiency*(*t*)最高的測試個案(每個 Thread 分別從各分群中挑選 *efficiency*(*t*)最高的測試個案，並將這些測試個案再次分群並派送至多個 Threads 平行挑選。重複此動作，直到挑選出 *efficiency*(*t*)最高的測試個案)；

Step 5. 平行化測試需求記錄更新：平行化將被挑選之測試個案所滿足的測試需求之狀態更新為已被滿足；

Step 6. 平行化判斷是否已完成運算：確認(條件 1)是否尚有測試需求未被滿足；(條件 2)是否剩餘測試個案可滿足任一尚未被滿足之測試需求。若上述至少一個條件成立，則回到 Step 3 繼續執行；否則，即進行 Step7；

Step 7. 回傳測試套件：回傳所挑選的測試個案之集合(即精簡後之測試套件)，將資料複製至 CPU，並結束運算。

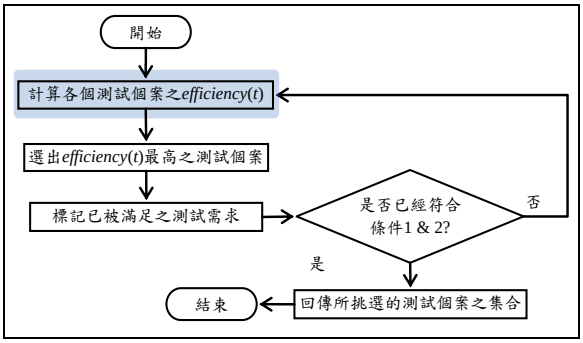

圖 9 以 CPU 實作 Efficiency 之程式執行流程

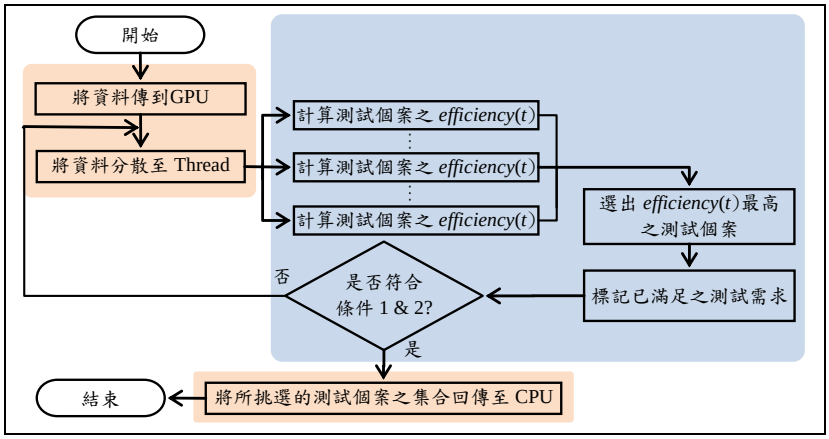

圖 10 以 GPU 實作 Efficiency 之程式執行流程

表 2 SIR 受測程式與其測試套件介紹

受測程式	版本數量	行碼數	測試需求*	測試個案
space	1	6218	1067	13,585
flex	5	12422-14245	2011-2680	525
gzip	5	6577-7997	1496-1923	214

* 此實驗以程式碼中的決策分支作為測試需求。

四、實驗結果與討論

4.1. 實驗資料簡介

本實驗以取自 SIR 網站[12]上的 space、flex 和 gzip 等三支程式檢驗所提出方法的成效，這些程式皆是相關研究經常採用的受測程式，其規模、測試個案數量和測試需求數量請參閱表 2。表 3-5 分別為 Greedy、Ratio 和 Efficiency 等三個精簡演算法採用 CPU 與 GPU 作為主要計算元件的實驗結果比較。這些表格的第一欄條列實驗所採用的受測程式，第二欄為各個測試套件的資料量規模(此規模為測試個案與測試需求數目相乘所得的數字)，而末兩欄則分別為以 CPU 和 GPU 實作的執行時間。

4.2. 實驗數據分析與討論

在表 3 中，相較於以 CPU 實作的作法而言，以 GPU 實作 Greedy 演算法的效果明顯不佳。就 space 程式而言，GPU 實作版本所花費時間約為 CPU 版本的 2.57 倍；就 flex 程式的 5 個版本而言，GPU 實作版本所花費時間約為 CPU 版本的 6 至 10 倍；就 gzip 程式的 5 個版本而言，GPU 實作版本所花費時間約為 CPU 版本的 19 至 34 倍。將此結果對照資料量規模後可發現，若資料量規模越大，GPU 實作版本的表現就越接近 CPU 實作版本。事實上，GPU 確實可加速測試套件精簡過程中 *coverage*(*t*)的計算和比較與測試需求標記的更新。造成此現象的原因在於，若使用 GPU 進行運算，必須先將資料傳遞至 GPU 可存取的記憶體空間，經 GPU 運算完後再將運算結果傳回主記憶體，以供 CPU 存取，此來回傳輸耗費的時間會使得測試套件精簡的總體耗時拉長。因此，在對較大規模測試套件(即受測程式 space)進行精簡時，GPU 較易發揮其功效，使得兩者的耗時差距倍數縮小。然而，以 GPU 實作 Greedy 演算法確實導致總體耗時表現不佳。

就 Ratio 演算法而言，相較於以 CPU 實作的作法，以 GPU 實作的效果依舊明顯不佳。就 space 程式而言，GPU 實作版本所花費時間約為 CPU 版本的 1.86 倍；就 flex 程式的 5 個版本而言，GPU 實作版本所花費時間約為 CPU 版本的 7 至 9 倍；就 gzip 程式的 5 個版本而言，GPU 實作版本所花費時間約為 CPU 版本的 15 至 21 倍。與表 3 結果相似，若資料量規模越大，GPU 實作版本的表現就越接近 CPU 實作版本。然而，值得注意的是，與表 3 中 Greedy 的實驗數據相較，CPU 和 GPU 實作版本的耗時差距已有縮小的趨勢。其原因在於，雖然 Ratio 演算法和 Greedy 演算法的計算時間複雜度相同，但 Ratio 演算法額外將測試個案執行時間納入考量，導致運算量較 Greedy 演算法更大，因而 GPU 的平行加速運算更有發揮空間。然而，以 GPU 實作 Ratio 演算法的成效仍舊遠遜於以 CPU 實作的版本。

表 3 實驗結果

Greedy 演算法之實作比較			
受測程式	資料量規模	CPU 耗時(s)	GPU 耗時(s)
space	13585 × 1067	5.61362(s)	14.434092(s)
flexV1	525 × 2011	0.10499(s)	1.064519(s)
flexV2	525 × 2656	0.15136(s)	1.493514(s)
flexV3	525 × 2666	0.19528(s)	1.250487(s)
flexV4	526 × 2678	0.14644(s)	1.233677(s)
flexV5	525 × 2680	0.13422(s)	1.227929(s)
gzipV1	214 × 1496	0.02096(s)	0.724101(s)
gzipV2	214 × 1758	0.04063(s)	0.773665(s)
gzipV3	214 × 1752	0.03066(s)	0.760578(s)
gzipV4	214 × 1805	0.03947(s)	0.775610(s)
gzipV5	214 × 1923	0.03585(s)	0.785297(s)
Ratio 演算法之實作比較			
受測程式	資料量規模	CPU 耗時(s)	GPU 耗時(s)
space	13585 × 1067	8.71531(s)	16.212168(s)
flexV1	525 × 2011	0.11118(s)	1.064515(s)
flexV2	525 × 2656	0.17288(s)	1.251445(s)
flexV3	525 × 2666	0.14916(s)	1.254299(s)
flexV4	526 × 2678	0.15119(s)	1.245157(s)
flexV5	525 × 2680	0.14626(s)	1.224439(s)
gzipV1	214 × 1496	0.03398(s)	0.724475(s)
gzipV2	214 × 1758	0.03997(s)	0.785600(s)
gzipV3	214 × 1752	0.03891(s)	0.792117(s)
gzipV4	214 × 1805	0.05167(s)	0.789944(s)
gzipV5	214 × 1923	0.04651(s)	0.801100(s)
Efficiency 演算法之實作比較			
受測程式	資料量規模	CPU 耗時(s)	GPU 耗時(s)
space	13585 × 1067	31.20679(s)	19.661204(s)
flexV1	525 × 2011	0.05165(s)	0.704838(s)
flexV2	525 × 2656	0.08342(s)	0.768733(s)
flexV3	525 × 2666	0.07622(s)	0.761854(s)
flexV4	526 × 2678	0.07061(s)	0.769036(s)
flexV5	525 × 2680	0.07221(s)	0.757335(s)
gzipV1	214 × 1496	0.05541(s)	0.713313(s)
gzipV2	214 × 1758	0.07811(s)	0.743866(s)
gzipV3	214 × 1752	0.07263(s)	0.739568(s)
gzipV4	214 × 1805	0.06320(s)	0.738395(s)
gzipV5	214 × 1923	0.06516(s)	0.750981(s)

就 flex 和 gzip 受測程式而言，以 GPU 實作 Efficiency 演算法的成效仍舊比不上以 CPU 實作的版本，差距皆約有 9-13 倍。然而，對資料量規模較大的 space 程式而言，以 GPU 實作 Efficiency 演算法的成效可勝過以 CPU 實作的版本。探究其原因可發現，Efficiency 演算法執行過程必須評估測試個案之不可替代度，此部分運算的計算時間複雜度較高，造成整體資料計算量較大，相較於其他兩種演算法，更能發揮 GPU 之運算優勢。

根據上述討論可發現，就資料量規模較小的實驗資料或計算時間複雜度較低的演算法而言，GPU 較無空間發揮其加速的功效，因而 GPU 耗時明顯遜於 CPU 耗時。然而，當資料量的規模和所採用演算法的計算複雜度皆較大時，平行運算加速所省下的時間可補償 CPU 和 GPU 之間記憶體來回傳遞所花費的時間，此時 GPU 平行運算的效益即可浮現。由此可見，GPU 適用於大規模的受測程式與測試套件。

4.3. 實驗效度的威脅分析

在此探討實驗效度的潛在威脅(Threats to Validity)，將分別從外部威脅(External validity)和內部威脅(Internal validity)進行討論。

主要的外部威脅在於實驗僅採用 3 個受測程式(共 11 個版本)，所呈現的實驗結果是否具有代表性。有鑑於此，我們挑選相關研究經常採用的 SIR 程式與測試套件，以期降低此威脅。相較於其他受測程式與測試套件，在論文長度和研究進度限制下挑選這些資料所呈現的實驗結果應較具有說服力。

此外，就我們所知，並無文獻公開相關演算法的實作程式碼，且以 GPU 實作這些演算法亦是此研究的主要貢獻。因此，實驗效度的內部威脅主要來自於演算法實作的正確性。為了降低此威脅，我們針對所實作的全部程式碼進行同儕審查(Peer-review)，且設計多筆測資，並撰寫成單元測試程式碼(Unit-test scripts)進行自動化測試，以保障程式碼的品質。

五、結論與未來研究方向

此研究實作三種著名測試套件精簡演算法的平行化加速運算版本，其中只有 Efficiency 演算法處理大規模資料的加速成果較為明顯，對其他搭配組合的加速效果皆不佳。深究此現象，主要原因是實驗採用的資料規模不夠龐大，導致平行化運算的效益不明顯。目前我們正嘗試針對 SIR 網站中的大型程式生成測試套件，以期更客觀地評估以 GPU 加速測試套件精簡的可行性。此外，我們未來亦將實作其他著名的測試套件精簡技術，包含以基因演算法(Genetic Algorithm-based Approach)[13]和整數線性編程(Integer Linear Programming)執行測試套件精簡的技術[14]，並進行不同演算法間的效能比對，以更進一步檢驗 GPU 加速測試套件精簡的成效。

誌謝

本研究受國科會計畫編號 MOST 104-2628-E-415-001-MY3 補助。

參考文獻

[1] C. T. Lin, K. W. Tang, and G. M. Kapfhammer, “Test Suite Reduction Methods that Decrease Regression Testing Costs by Identifying Irreplaceable Tests,” *Information and Software Technology*, Vol. 56, No. 10, pp. 1322-1344, October 2014.

[2] D. Binkley, “Semantics Guided Regression Test Cost Reduction,” *IEEE Trans. on Software Engineering*, Vol. 23, No. 8, pp. 498-516, August 1997.

[3] “Continuous Integration (CI),” Available at: http://www.dotblogs.com.tw/hatelove/archive/2011/12/25/introducing-continuous-integration.aspx, Access date: 2015/01/22.

[4] M. J. Harrold, R. Gupta, and M. L. Soffa, “A Methodology for Controlling the Size of a Test Suite,” *ACM Trans. on Software Engineering and Methodology*, Vol. 2, No. 3, pp. 270-285, July 1993.

[5] “CPU vs GPU,” Available at: https://www.cs.uaf.edu/2007/fall/cs441/proj1notes/favier/, Access date: 2016/01/14.

[6] T. Y. Chen and M. F. Lau, “A New Heuristic for Test Suite Reduction,” *Information and Software Technology*, Vol. 40, No. 5-6, pp. 347-354, July 1998.

[7] T. Mücke and M. Huhn, “Minimizing Test Execution Time during Test Generation,” *International Federation for Information Processing*, Vol. 227, pp. 223-235, 2007.

[8] A. M. Smith and G. M. Kapfhammer, “An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization,” *Proceedings of the 24th ACM SIGAPP Symposium on Applied Computing, Software Engineering Track*, pp. 461-467, Honolulu, USA, March 2009.

[9] J. A. Jones and M. J. Harrold, “Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage,” *IEEE Trans. on Software Engineering*, Vol. 29 No. 3, pp. 195-209, March 2003.

[10] “The Green500 List News And Submitted Items”, Available at: http://www.green500.org/, Access date: 2014/12/05.

[11] S. M. Teng, *Auto-Tuning for GPGPU Applications Using Performance and Energy Model*, Master Thesis, National Chung Cheng University, June 2013.

[12] “Software-artifact Infrastructure Repository (SIR),” Available at: http://sir.unl.edu/portal, Access date: 2016/03/28.

[13] N. Mansour and K. El-Fakih, “Simulated annealing and genetic algorithms for optimal regression testing,” *Journal of Software Maintenance*, Vol. 11, No. 1, pp. 19-34, January 1999.

[14] J. Black, E. Melachrinoudis, and D. Kaeli, “Bi-criteria models for all-uses test suite reduction,” *Proceedings of the 26th International Conference on Software Engineering*, pp. 106-115, Washington, DC, USA, May 2004.