專題報告

使用向量化技術優化與實作雙方安全運算C/C++函式庫

指導教授: 陳鵬升教授

專題學生: 邱晨恩、許峻豪

(一)摘要

多方安全運算(MPC)的應用程式中，其函式庫的使用在運算時需要進行加密，其中包含明文和密文間的轉換與操作，這使得雙方安全運算之應用程式開發非常困難。而我們將透過Intel Vtune進行MPC程式的效能分析出耗時佔比前三名的函式，並針對此函式使用向量化的技術優化。與此同時，我們使用ABY來實作支援支援雙方安全運算函式庫的的功能。ABY是一個高效混合協議的雙方安全計算框架框架，提供了高效的加密操作和安全計算，可支援明文與密文之間的轉換。我們在ABY框架下撰寫支援MPC程式的函式庫，解決MPC應用程式開發困難以及應用上的效能瓶頸。

關鍵字 : 多方安全運算(Secure multi-party computation)、向量化(Vectorization)

(二)研究動機與研究問題

現今有許多系統服務可能會使用到敏感資料，須在隱私保護的情況下進行計算，多方安全運算（MPC）被廣泛運用於此。但是，其中明文和密文間的轉換與操作，使得編寫MPC程式十分困難。此外，由於在執行MPC應用程式時需要處理大量的加密計算，這使得開發MPC應用程式變得更加複雜，未來程式碼維護工作也可能帶來額外的挑戰。

基於上述的困難，針對MPC的應用程式，降低開發的複雜度，又同時兼顧資料隱私性就顯得十分重要。然而，兼顧隱私性必然會拖累程式效能，但是系統服務流暢又是不可或缺的，因此，我們必須盡可能取得兩者間的平衡。在這重重困難中找到解決方案，成為了我們研究的主要目標。

(三)文獻回顧與探討

1. Practical SIMD Vectorization Techniques for Intel® Xeon Phi™ Coprocessors

Xinmin Tian、Hideki Saito 等人曾提出幾種SIMD向量化技術[1]，在Intel coprocessor上運行高效能的應用程式，最高可達到12.5倍的效能提升。其中一種方式為使用Masking Techniques對不完全向量迴圈優化，目的是可以有效地利用處理器的架構，只針對有效元素進行計算，而不需處理補位(padding)的元素，從而提高程式效能。另一種方式為針對特定的資料對齊優化，目的是當資料對齊時，SIMD指令可以更有效地進行操作，提高指令的並行性和向量化程度，從而加速程式的執行速度。同時，行資料對齊優化可以確保所有內存訪問都需要元素進行對齊，從而避免因對齊錯誤而引起的故障。這個資料對齊的方法也有實作在我們的專題當中，詳細的實作程式碼會放在相關資料中[2]。

2. Simple and efficient GPU accelerated topology optimisation: Codes and applications

Erik Träff、Anton Rydahl等人曾在2023年的《Computer Methods in Applied Mechanics and Engineering》中提出，利用GPU向量化技術進行並行處理，加速拓撲優化技術解決大規模線性彈性最小相容性問題（linear elastic compliance minimization)[3]。作者說明GPU能充分利用圖形處理中的並行性，從而在CPU上實現顯著性能提升。一般情形使用的的GPU框架（如CUDA或OpenCL）通常是基於kernels的概念，以確保程式碼結構正確且高效。然而，將CPU程式轉移到GPU上具有挑戰性，需重新定義為kernels，因此，作者提到使用high level language（如Futhark）能更快速進行程式開發，並通過Futhark編譯器的預先優化生成高效計算kernels，從而節省開發時間與工作。在我們的專題中，目前也正在嘗試利用Futhark，進一步優化程式效能。

(四)實作方法

實驗環境

cmake version	3.16.3
gcc version	9.4.0
g++ version	9.4.0
Thread model	posix
Target	x86_64-linux-gnu

Architecture	x86_64
Model name	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
CPU(s)	8
On-line CPU(s) list	0~7
Thread(s) per core	2

第一部分: 程式效能分析+向量化技術優化

1-1. 利用Intel Vtune對雙方安全運算程式碼進行效能分析，找出耗時函式。

Intel VTune是一個效能分析工具[4]，用於探索和最佳化軟體效能。它提供豐富的資訊如CPU使用率、記憶體存取等。然而，儘管VTune具有全面的profiling能力，程式碼執行的即時資訊無法取得，特別是在硬體相關的即時效能問題方面，如cache miss。雖然如此，但是Vtune仍然有助於程式設計師了解程式執行中的效能和資源消耗，並提供實時建議和優化方法，以改善程式效能，降低能源成本。我們針對7個不同的MPC應用程式進行10次量測，取平均值後歸納出ABY library耗時函式前3名。

耗時函式排名	占比
FixedKeyHashing	53.63%
EklundhBitTranspose	12.24%
XORBytesReverse	5.3%

1-2. 利用向量化技術優化耗時函式: Intel® Intrinsics Guide

在Intel Vtune列舉出耗能的函式後，我們利用Intel處理器支援的intrinsic function，使用向量化技術優化程式碼。 Intel® Intrinsics Guide是為程式開發者提供特定CPU指令的低階函式，用於直接指定硬體操作，以提升撰寫程式碼效率和執行效能[5]。透過intrinsics function，編譯器能直接生成對應的硬體指令，使程式的效能有更好的表現[6]。

(四)實作方法

1-2. 利用向量化技術優化耗時函式: Intel® Intrinsics Guide

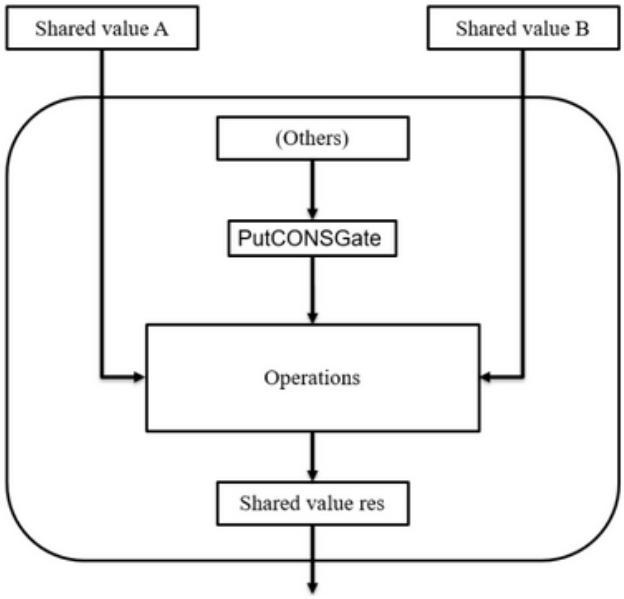
這些intrinsics function往往直接對應到machine code指令，有助於充分發揮硬體能力。我們目前使用的是SSE2和SSE4.1架構的指令集，能夠支援64位元雙精度浮點數[7]。我們使用Cmale[8]讓ABY架構與intrinsic function能夠共同編譯，並且於編譯時加入"CMAKE_C_FLAGS:STRING=-msse -msse2"，詳細的編譯環境於研究結果中說明。下方表格的資訊是我們使用到的函式名稱。

name	SIMD ISA	function
_mm_set_epi64x	SSE2	設定一個 128 bits 的變數
_mm_xor_si128	SSE2	將兩個 128 bits 做 XOR
_mm_extract_epi64	SSE4.1	將 128 bits 拆開成兩個 64 bits
_mm_insert_epi64	SSE4.1	將兩個 64 bits 合為一個 128 bits

第二部分: 實作支援多方安全運算之函式庫

2-1. 實作支援雙方安全運算的函式庫: <math.h>

實作可支援C標準函示庫的<math.h>，用於多方安全運算的應用程式。實作的方式如下方的流程圖，首先input是密文(shared value)的處理，接著會針對不同operation進行加密運算，最後函式會回傳計算出來的答案，也是以密文(shared value)以當作回傳值。

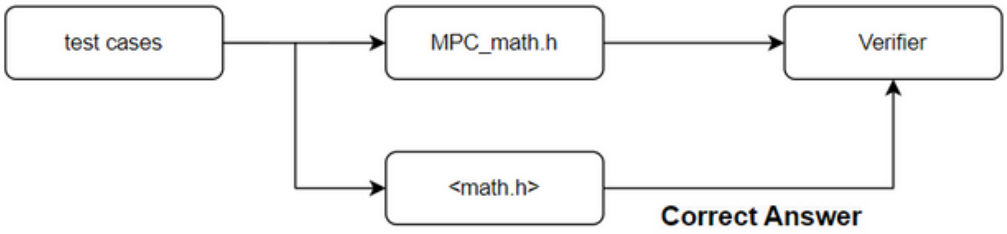


以sin operation為例，我們利用泰勒展開式，開一個迴圈計算每一項的次方和階乘，將每一項加總後去求得double浮點數精確度的近似值。此外，我們將這些函式寫成一個.h檔案，當使用者需要使用到MPC版本的數學計算函式，即可直接include這個檔案進行程式開發。 開發之MPC_math.h的github連結有您可於附錄中查詢[9]。

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots \qquad \forall x$$

2-2. 進行實作函式庫的正確性驗證

目前我們實作的function是支援C標準函式庫中的math.h，讓需要保護隱私性的資料能夠進行運算。然而，除了要維持函式中資料的安全性，同時也要維持正確性。因此，我們驗證的方式如下方的流程圖，首先以亂數產生10000筆測試資料(test cases)，接將我們自己開發的MPC_math.h以及C標準函式庫中的math.h算出來的答案輸出到檔案上，最後針對這兩個檔案進行比對，完成函式庫的正確性驗證。



(五)研究結果

第一部分: 程式效能分析+向量化技術優化

我們針對耗時最多的函式FixedKeyHashing優化後，經由Intel Vtune測量7種應用程式的執行時間。每種應用程式測量20次後取平均，和未優化前的執行時間計算speedup，最高可達到6.6倍的效能提升。而其他應用程式的效能優化結果如下方表格，我們發現在迴圈較少的應用程式，優化的幅度比較小。這些用於測量效能的應用程式的程式碼有附上github的連結[10]。

program name	speedup
millionaire_prob	6.6
SHA1	2.15
min-euclidean-dist	1.81
euclidean_distance	1.79
innerproduct	1.01
threshold_euclidean	1.01
abyfloat	0.99

第二部分: 實作支援多方安全運算之函式庫

原本C標準函式庫中的math.h，可依照功能拆成三部份。分別是三角函數、取模運算以及基本的運算函式。首先，我們將實作的函式庫與使用向量化技術優化後的成果一起進行編譯。編譯完成後，我們將以隨機方式產生測試資料，進行10次測試，然後取平均值呈現測量函式的執行時間。最後，將實驗數據與未經優化的原版進行比較，並整理成下表。

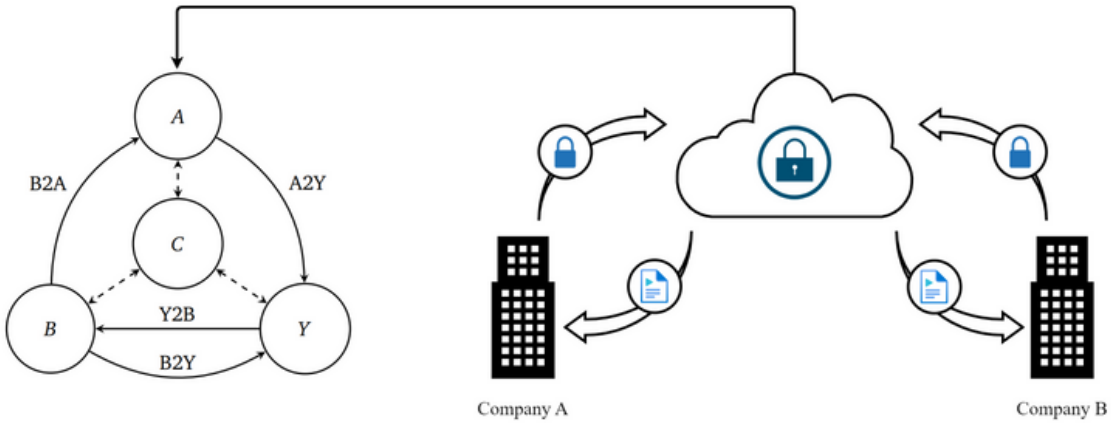
name	execution time	update execution time
floor	1.68(s)	1.62(s)
ceil	1.87(s)	1.58(s)
fabs	1.66(s)	1.43(s)
power	2.63(s)	2.48(s)
exp	2.78(s)	2.61(s)
sqrt	1.54(s)	1.36(s)
sinh	4.03(s)	3.92(s)
cosh	3.97(s)	3.68(s)
tanh	5.81(s)	5.26(s)

name	execution time	update execution time
sin	7.31(s)	5.82(s)
cos	8.17(s)	6.71(s)
tan	8.63(s)	6.78(s)
asin	532.14(s)	527.63(s)
acos	501.79(s)	488.23(s)
atan	549.72(s)	552.39(s)
sinh	4.03(s)	3.92(s)
cosh	3.97(s)	3.68(s)
tanh	5.81(s)	5.26(s)

name	execution time	update execution time
fmod	2.07s	1.75s
modf	2.93s	2.76s

從表格中可以觀察到，有些function在和優化函式一起編譯後，更新後執行時間會比較慢。根據Intel Vtune的數據觀察，優化函式的時間降低，但是因為優化函式在程式中所佔比例不大，以及其他function所花的時間變長，才會有此現象。此外，在與三角函數相關的函式中，由於使用泰勒展開式進行實作，其效能相對較低。因此，對於目前已開發並使用泰勒展開式逼近double精確度的函式，我們在函式輸入參數中提供了選項，以增加使用的彈性。需要注意的是，由於在計算過程中需要保護資料的隱私性，因此相較於C標準函式庫中的函式，我們的函式會消耗較多的時間。

第三部分: 實際案例說明



我們實際應用了MPC_math.h庫，撰寫了一個MPC應用程式。該程式的主要功能是在確保合作商業公司的隱私數據受到保護的前提下，允許銀行進行數據分析，以評估風險，從而決定是否合作是否可行。我們評估風險的方法，權重分配如下的數學式，它是基於個人的資產收入和過去在系統內部的信用評分進行計算的。我們的假設是，在沒有負債的情況下，信用評分為600分，只有計算出的jointCreditScore大於700時，合作才可行。

$$personCreditScore = personIncome^{0.6} \cdot \frac{|personCreditHistory - 600|}{100.0}$$

$$jointCreditScore = \frac{\lfloor person1CreditScore \rfloor + \lceil person2CreditScore \rceil}{\left|\sin\left(\frac{person1CreditScore + person2CreditScore}{2.0}\right)\right|}$$

我們將這個應用程式與沒有使用自定函式庫的版本進行了比較，舊新執行時間比例為23.41(s) / 17.68(s)，行數比為 270 / 140。 這些數據顯示，我們的解決方案在處理效能瓶頸和開發困難的問題方面是合理且有效的。詳細程式碼與實驗數據，您可以在附錄中查閱[11]。

(六)未來展望

我們目前正在研究比Intel intrinsic function更好的向量化技術，我們觀察到目前的方法在牽涉多個迴圈的應用程式中才能明顯提升性能。為了進一步優化MPC程式的效能，我們正在將Erik Träff等人論文中提到的Futhark技術納入我們的程式碼中，希望能在每種應用程式有5倍以上的speedup。同時，我們也正在增加支援安全運算的相關字串處理版本，希望能將開發MPC應用程式的時間成本再降低百分之八十。

(七)參考文獻與附錄

[1] Xinmin Tian and Hideki Saito and S. Preis and Eric N. Garcia and Sergey Kozhukhov and Matt Masten and Aleksei G. Cherkasov and Nikolay Panchenko, "Practical SIMD Vectorization Techniques for Intel® Xeon Phi™ Coprocessors," in IEEE IPDPS 2013 PhD Forum.

[2] ABY library程式碼連結: https://github.com/encryptogroup/ABY

[3] E. Träff and Anton Rydahl and Sven Karlsson and O. Sigmund and N. Aage, "Simple and efficient GPU accelerated topology optimisation: Codes and applications," in Computer Methods in Applied Mechanics and Engineering 2023.

[4] Intel Vtune網址: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/introduction.html

[5] SIMD指令集Intel® Intrinsics Guide: www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[6] SIMD Introduction網址: https://hackmd.io/@yenWu/BkRs9_I1g?type=view

[7] SSE指令集架構參考資料「AVX vs. SSE: expect to see a larger speedup」: https://stackoverflow.com/questions/47115510/avx-vs-sse-expect-to-see-a-larger-speedup

[8] Cmake tutorial: https://cmake.org/documentation/

[9] https://github.com/howcat/ABY/blob/main/MPC_math/MPC_math.h

[10] https://github.com/encryptogroup/ABY/tree/public/src/examples

[11] https://github.com/howcat/ABY/tree/main/Application

[12] 針對例外處理輸出nan方法參考 : https://stackoverflow.com/questions/16691207/c-c-nan-constant-literal

[13] futhark language tutorial :https://futhark-lang.org/

[14] GCC and Make Compiling, Linking and Building C/C++ Applications : https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html