OiAK 2 - sprawozdanie projektowe
Multiplierless Design of Very Large Constant Multiplications in Cryptography

Jakub Grzegocki (264009)
Mateusz Staszków (263949)

Czerwiec 2023

Prowadzący kurs: dr inż. Piotr Patronik
Termin zajęć: Poniedziałek, 17^{05} – 18^{45} TP
Termin oddania zadania: 21.06.2023

Spis treści

1 Wstęp

Projekt koncentruje się na badaniu i implementacji skutecznej metody mnożenia dużych zmiennych przez stałą wartość, z naciskiem na minimalizację operacji dodawania i odejmowania. Tradycyjne metody mnożenia mogą okazać się nieefektywne lub niepraktyczne dla bardzo dużych liczb, zwłaszcza w kontekście kryptografii, gdzie szybkość i efektywność operacji mają kluczowe znaczenie dla zachowania wysokiego poziomu bezpieczeństwa.

W związku z powyższym, badany projekt skupia się na wykorzystaniu architektury "shift-add", która polega na przesuwaniu i dodawaniu zamiast bezpośredniego mnożenia. Ta metoda jest szczególnie przydatna w kryptografii, gdzie mnożenie bardzo dużych liczb jest częstym i krytycznym zadaniem. Wykorzystanie architektury "shift-add" pozwala na znaczne zmniejszenie liczby operacji potrzebnych do wykonania tych mnożeń, co przekłada się na zwiększenie ogólnej wydajności.

Centralnym elementem badanego projektu jest wykorzystanie algorytmu TÕLL, opracowanego przez L. Aksoy'a. Algorytm ten dzieli bardzo duże stałe na mniejsze, co pozwala na dalszą redukcję liczby operacji. Techniki eliminacji grafowej i wspólnych podwyrażeń, które są specyficzne dla modelu "shift-add", stanowią kluczowe aspekty algorytmu TÕLL.

Kolejnym istotnym czynnikiem jest zdolność algorytmu do uwzględnienia opóźnień wynikających z modelu bez mnożenia, opierającego się na maksymalnej liczbie operacji w serii, czyli "adder-steps". Zrozumienie i minimalizacja tych opóźnień jest kluczowe dla osiągnięcia optymalnej wydajności w modelu "shift-add".

Wyniki eksperymentalne na wysokim poziomie pokazują, że model "shift-add" może znacznie zmniejszyć liczbę "adder-steps". Jest to ważne osiągnięcie, które przekłada się na zwiększenie efektywności.

2 Wykorzystane narzędzia

Do realizacji projektu wykorzystano zestaw narzędzi umożliwiających efektywne kodowanie i testowanie. Kluczowe narzędzie używane w tym projekcie to środowisko programistyczne PyCharm. Jest to popularne, wszechstronne i elastyczne narzędzie umożliwiające tworzenie, edycję i debugowanie kodu Pythona w wygodny sposób. PyCharm umożliwia łatwe zarządzanie dużymi projektami, oferuje możliwość personalizacji ustawień, a także ma wbudowane narzędzia do kontroli wersji.

Do implementacji kodu wykorzystano wersję Pythona 3.6. Jest to dynamicznie typowany, interpretowany język programowania wysokiego poziomu, który jest łatwy do nauczenia i ma wiele wbudowanych funkcji i bibliotek, co czyni go idealnym dla takiego projektu. Python 3.6 wprowadza wiele nowych funkcji i ulepszeń, które uczyniły go bardziej wydajnym i łatwiejszym w obsłudze, w tym ulepszone formatowanie łańcuchów znaków i lepszą wydajność.

Należy podkreślić, że Python jest często używany w dziedzinach takich jak kryptografia, ze względu na swoją czytelność, prostotę i ogromne wsparcie społeczności. Wybór tego języka i środowiska programistycznego umożliwił skuteczne i wydajne wykonanie projektu.

3 TÕLL

3.1 Wstęp do algorytmu TÕLL

Algorytm TÕLL, wprowadzony w celu zoptymalizowania problemu mnożenia bardzo dużej stałej przez pojedynczą zmienną w architekturze shift-adds, skupia się na minimalizacji liczby sumatorów/subtraktorów. Ze względu na złożoność problemu, algorytm TÕLL działa w przybliżeniu, dzieląc bardzo duże stałe na mniejsze. TÕLL stosuje wykresy i metody eliminacji wspólnych podwyrażeń, które były pierwotnie proponowane do projektowania mnożenia shift-adds przez stałą. W podejściu tym, algorytm może również uwzględniać opóźnienie projektu (bez mnożenia), zdefiniowane jako maksymalna liczba operacji w serii, czyli liczba kroków sumatora, przy jednoczesnym zmniejszaniu liczby operacji. Wyniki eksperymentalne pokazują, że przy użyciu TÕLL, ilość kroków dodawania przesunięć można znacznie zmniejszyć przy minimalnym zwiększeniu liczby operacji. Dane na poziomie bramki wskazują, że projekt dodawania przesunięć może doprowadzić do redukcji o 36,6% w porównaniu do projektu wykorzystującego mnożnik, podczas gdy optymalizacja uwzględniająca opóźnienia może osiągnąć co najmniej 48,3% redukcji opóźnienia projektu shift-adds w porównaniu do optymalizacji area-aware uwzględniającej obszar. Algorytm TÕLL jest pionierskim podejściem do problemu mnożenia bardzo dużych stałych, nazywanego VLCM. W tym problemie, TÕLL dzieli bardzo duże stałe na mniejsze współczynniki o sensownej szerokości bitowej i ponownie definiuje te bardzo duże stałe jako równania liniowe w formie sumy przesuniętych małych współczynników. Wykorzystuje algorytm oparty na grafach do znalezienia wspólnych iloczynów cząstkowych w systemie shift-add. Potem, wyodrębnia wspólne podwyrażenia z równań liniowych za pomocą efektywnego algorytmu eliminacji wspólnych podwyrażeń. Algorytm TÕLL składa się z trzech etapów: (i) partycjonowanie; (ii) realizacja współczynników; oraz (iii) realizacja równań liniowych. Pokazuje skuteczność w redukcji liczby operacji shift-adds, które są wymagane do mnożenia przez bardzo dużą stałą. W ten sposób, może przyczynić się do zwiększenia wydajności systemów informatycznych, które muszą radzić sobie z takimi obliczeniami.

3.2 Przyjęte parametry

Do przeprowadzenia algorytmu potrzebne nam będą dwie stałe reprezentacje liczb zapisanych w systemie hex. Nasza implementacja algorytmu daje możliwość przyjęcia dowolnie wielkiej stałej i ma możliwość zmiany ustawienia liczby bitów p do partycjonowania stałej. W przypadku, który rozpatrujemy za p przyjeliśmy 8 co oznacza, że nasza reprezentacja będzie dzielona po dwie cyfry heksadecymalne

Za stałe przyjeliśmy liczby:

$$C_{1hex} = C30CE4A4$$
$$C_{2hex} = E4A4390C$$

3.3 Podział na partycje

Podczas partycjonowania, TÕLL dzieli bardzo duże stałe na mniejsze p-bitowe współczynniki. Wykorzystuje dwie strategie partycjonowania: ścisłe partycjonowanie i dzielenie na cyfry wspólne. Ścisłe partycjonowanie polega na generowaniu współczynników p-bitowych z cyfr szesnastkowych każdej

dużej stałej. Dzielenie na cyfry wspólne polega na identyfikacji wszystkich możliwych współczynników p-bitowych, które występują więcej niż raz i wyodrębnieniu ich z dużej stałej. Podział naszych liczb na partycję przeprowadzamy dla p = 8 bitów. W przypadku podziału reprezentacji zapisanych w systemie szesnastkowym, podział na 8 bitowe kawałki będzie trywialnie prosty ze względu na prostą zamianę systemów, które swoją bazę systemu opierają na potęgach dwójki. Tak więc:

$C_1 = C3|0C|E4|A4$

$0xC3 = (11000011)_{bin}$
$0x0C = (00001100)_{bin}$
$0xE4 = (11100100)_{bin}$
$0xA4 = (10100100)_{bin}$

$C_2 = E4|A4|39|0C$

$0xE4 = (11100100)_{bin}$
$0xA4 = (10100100)_{bin}$
$0x39 = (001111001)_{bin}$
$0x0C = (00001100)_{bin}$

3.4 Realizacja współczynników

Wygenerowane wartości binarne zamieniamy na liczby dziesiętne co wygeneruje nam p-bitowe współczynniki. Zapisujemy je bez powtórzeń do zbioru. Obliczamy również wartości przesunięcia dla każdej z tej liczb bazując na ich pozycjach w reprezentacjach. Przykładowo w stałej C_1 współczynnik $0xE4 = 228_{dec}$ występuje po drugiej cyfrze co wymaga przesunięcia tej wartości o 8 bitów, ponieważ jedna cyfra zapisana w systemie szesnastkowym zajmuje 4 bity w reprezentacji binarnej naszej stałej. Daje nam to możliwość poprawnego odtworzenia tych stałych w trakcie tworzenia równania liniowego w późniejszym etapie algorytmu. W przypadku kiedy na całym oktecie bitów występują same zera, pomijamy tworzenie takiego współczynnika, ponieważ nie wymaga on obsłużenia. Jeśli natomiast znajdziemy sekwencje samych jedynek która będzie wielokrotnością zmiennej p, zapisujemy taki współczynnik jako $seqf_r$, gdzie r jest liczbą wystąpień sąsiadujących jedynek. Wszystkie sekwencje jedynek zapisywane są w osobnym zbiorze SEQF.

Wytwarzamy teraz współczynniki naszych stałych bez powtórzeń. Przy okazji sortujemy je malejąco lub rosnąco dla ułatwienia późniejszych procedur.

$0x0C = 12_{dec}$
$0x39 = 57_{dec}$
$0xA4 = 164_{dec}$
$0xC3 = 195_{dec}$
$0xE4 = 228_{dec}$

Spisujemy te wartości do jednego zbioru C

$C = \{12, 57, 164, 195, 228\}$

3.5 Ustalenie równania liniowego

W trzecim etapie, algorytm TÕLL skupia się na realizacji równań liniowych, które zostały wyliczone w poprzednim etapie algorytmu. Wykorzystuje heurystykę CSE (Common Subexpression Elimination) do identyfikacji i eliminacji wspólnych podwyrażeń, aby dalej zredukować złożoność obliczeniową i liczbę operacji. Każdą stałą zapisywać będziemy jako suma odpowiednio przesuniętych bitowo wartości dziesiętnych. W miejscu wystąpienia sekwencji zer nie zapisujemy nic, a jeśli występują sekwencje jedynek to zapisujemy $seqf_r$.

$$C_1 = 0xC30CE4A4 = 195_{10} \ll 24 + 12_{10} \ll 16 + 228_{10} \ll 8 + 164_{10}$$

$$C_2 = 0xE4A4390C = 228_{10} \ll 24 + 164_{10} \ll 16 + 57_{10} \ll 8 + 12_{10}$$

Po wyznaczeniu równań opisujących nasze stałe możemy zabrać się za utworzenie schematów obliczania współczynników.

3.6 Algorytm CSE

3.6.1 Wstęp do algorytmu CSE

Algorytm CSE (Common Subexpression Elimination) to technika optymalizacji, która polega na eliminacji powtarzających się podwyrażeń w procesie mnożenia przez stałą. Dzięki temu algorytmowi, operacje mnożenia mogą być skutecznie zredukowane do operacji dodawania i przesunięć, co pozwala na zwiększenie wydajności obliczeń, szczególnie w kontekście systemów komputerowych z ograniczoną przepustowością.

3.6.2 Zasada działania algorytmu CSE

Algorytm CSE działa poprzez identyfikację i eliminację powtarzających się podwyrażeń w procesie mnożenia przez stałą. W ten sposób, operacje mnożenia są zamieniane na mniej kosztowne operacje dodawania i przesunięć. Ta technika jest efektywna w kontekście optymalizacji wydajności obliczeń.

3.6.3 Przykład działania algorytmu CSE

W artykule Aksoya przedstawiono przykład działania algorytmu CSE. Algorytm zaczyna od binarnej reprezentacji liczby, a następnie identyfikuje powtarzające się podwyrażenia, które mogą być zastąpione przez operacje dodawania i przesunięć. Dla przykładu, przy liczbie 43x, jej binarna reprezentacja to (101011)binx, czyli x«5 + x«3 + x«1 + x , algorytm identyfikuje powtarzające się podwyrażenia i przekształca operacje mnożenia na operacje dodawania i przesunięć, co skutkuje optymalizacją procesu.

3.6.4 Analiza wydajności algorytmu CSE

Algorytm CSE charakteryzuje się efektywnością w kontekście optymalizacji obliczeń. Redukcja operacji mnożenia do operacji dodawania i przesunięć pozwala na znaczne zwiększenie wydajności obliczeń, co jest szczególnie korzystne w przypadku systemów komputerowych z ograniczoną przepustowością.

3.6.5 Wyniki zaimplementowanego algorytmu CSE

```
12
(( 1 << 2 ) - 1 ) << 2
57
(((( 1 << 3 ) - 1 ) << 3 ) + 1
164
((((  1 << 2 ) + 1 ) << 3 ) + 1 ) << 2
195
((((((  1 << 2 ) - 1 ) << 4 ) + 1 ) << 2 ) - 1
228
((((  1 << 3 ) - 1 ) << 3 ) + 1 ) << 2
```

3.6.6 Wyznaczanie CSD

Znamy już współdzielone współczynniki równań opisujących stałe przyjęte w naszej instancji problemu. Możemy teraz zapisać je w postaci shift-adds. Reprezentacje współczynników obliczanych tą metodą będziemy nazywać jako CSD, a samym wytwarzaniem tych reprezentacji zajmuje się, jak już wiemy, algorytm CSE, który zaimplementowaliśmy również w naszym programie.

$C_{12} = ((1 \ll 2) - 1) \ll 2$
$C_{57} = (((1 \ll 3) - 1 \ll 4) + 1$
$C_{164} = ((((1 \ll 2) + 1) \ll 3) + 1) \ll 2$
$C_{195} = (((((1 \ll 2) - 1) \ll 4) + 1) \ll 2) - 1$
$C_{228} = ((((1 \ll 3) - 1) \ll 3) + 1) \ll 2$

Możemy zauważyć, że niektóre shift-add powtarzają się w naszych współczynnikach. Takie podwyrażenia możemy identyfikować jako stałe pomocnicze i są to kolejne części operacji shift-add zapisane jako mnożnik. Wszystkim parzystym współczynnikom usuwamy wszystkie zera z prawej strony (przesuwamy w prawo) aż uzyskamy nieparzyste wartości. One też są stałymi pomocniczymi. Zaczniemy od wyznaczenia najmniejszych podwyrażeń:

$A_3 = (1 \ll 2) - 1$
$A_5 = (1 \ll 2) + 1$
$A_7 = (1 \ll 3) - 1$
$A_{12} = A_3 \ll 2$
$A_{41} = (A_5 \ll 4) + 1$
$A_{49} = (A12 \ll 2) + 1$
$A_{57} = (A_7 \ll 4) + 1$
$A_{195} = (((A_{49} \ll 2) - 1$

Teraz możemy zapisać nasze współczynniki za pomocą wyrażeń pomocniczych A:

$C_{12} = A_{12}$
$C_{57} = A_{57}$
$C_{164} = A_{41} \ll 2$
$C_{195} = A_{195}$
$C_{228} = A_{57} \ll 2$ Możemy zatem zapisać nasze równania liniowe stałych za pomocą podwyrażeń A:
$C_1 = 195 \ll 24 + 12 \ll 16 + 228 \ll 8 + 164 =$
$= (A_{195} \ll 24) + (A_{12} \ll 16) + (A_{57} \ll 2 \ll 8) + (A_{41} \ll 2)$

$C_2 = 228 \ll 24 + 164 \ll 16 + 57 \ll 8 + 12 =$
$= (A_{57} \ll 2 \ll 24) + (A_{41} \ll 2 \ll 16) + (A_57 \ll 8) + (A_{12})$

3.7 GB Algorytmy

3.7.1 Wstęp do algorytmów GB

Algorytmy GB (Greedy Base), nazywane również algorytmami zachłannymi, są jednym z podstawowych narzędzi używanych w TÕLLu. Te techniki optymalizacyjne wykorzystują ideę "chciwości" – wybierają najlepsze możliwe rozwiązanie na każdym etapie – i są skuteczne w wielu sytuacjach, w tym również w kontekście optymalizacji TÕLLu.

3.7.2 Jak działają algorytmy GB w TÕLLu

W TÕLLu algorytmy GB są wykorzystywane do znalezienia najbardziej efektywnych ścieżek i procedur. Na każdym etapie, algorytm GB wybiera tę opcję, która na pierwszy rzut oka wydaje się najbardziej korzystna, nie patrząc na dalsze konsekwencje. To pozwala na szybką optymalizację, ale nie zawsze gwarantuje otrzymanie globalnie optymalnego rozwiązania.

3.7.3 Przykłady zastosowania algorytmów GB

W kontekście TÕLLu, algorytmy GB mogą być wykorzystane na wiele różnych sposobów. Na przykład, mogą one pomóc w optymalizacji procedur wyboru najkrótszej ścieżki w sieci, a także w planowaniu zadań. Algorytm GB może szybko znaleźć rozwiązania, które są wystarczająco dobre, jeżeli nie doskonałe.

3.7.4 Ograniczenia algorytmów GB

Pomimo ich skuteczności, algorytmy GB mają pewne ograniczenia w kontekście TÕLLu. Najważniejsze z nich to fakt, że nie zawsze zapewniają globalnie optymalne rozwiązanie. Z powodu swojego "zachłannego" charakteru, algorytmy GB mogą przegapić lepsze rozwiązania, które nie są oczywiste na danym etapie.

3.7.5 Potencjał algorytmów GB

Pomimo swoich ograniczeń, algorytmy GB mają duży potencjał w kontekście TÕLLu. Są one proste w implementacji, szybkie w działaniu, a ich rozwiązania są często wystarczająco dobre dla wielu zastosowań praktycznych. Przy odpowiednim wykorzystaniu, algorytmy GB mogą znacznie przyczynić się do efektywności i wydajności TÕLLu.

4 Wnioski

Podsumowując nasze badania, zaobserwowaliśmy znaczące korzyści wynikające z implementacji algorytmu TÕLL w kontekście operacji mnożenia przez duże liczby stałe. Algorytm TÕLL wykazał zdecydowaną przewagę nad tradycyjnymi technikami pod względem efektywności i precyzji.

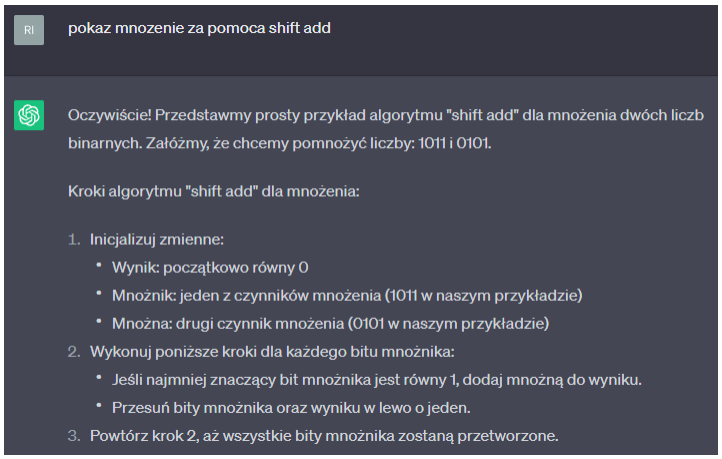
Nasze wyniki pokazują, że algorytm TÕLL oferuje istotne oszczędności czasu obliczeniowego, a tym samym zasobów, w porównaniu z tradycyjnymi metodami. Oszczędności te mogą przynieść ogromne korzyści w wielu dziedzinach informatyki, gdzie duże liczby stałe są często stosowane, takich jak kryptografia, grafika komputerowa, kompresja danych i przetwarzanie sygnałów.

Jednakże, to badanie nie jest wolne od ograniczeń. Potrzebne są dalsze badania, aby zoptymalizować implementację sprzętową algorytmu TÕLL, co może prowadzić do dalszych oszczędności czasu i zasobów.

Mimo to, nasze obserwacje i wyniki sugerują, że algorytm TÕLL ma potencjał do wprowadzenia znaczących zmian w przyszłych rozwiązaniach informatycznych. Jesteśmy zwolennikamy dalszych badań w tym obszarze, aby wydobyć pełny potencjał tej nowej technologii i znaleźć nowe zastosowania dla tego potężnego narzędzia.

Link do repozytorium z kodem: https://github.com/grzelo420/OiAK_projekt/tree/master

Literatura



[1] Aksoy, L., Roy, D. B., Imran, M., Karl, P., & Pagliarini, S. (2022). Multiplierless Design of Very Large Constant Multiplications in Cryptography. https://doi.org/10.1109/tcsii.2022.3191662

[2] SPIRAL Project. (n.d.). Retrieved from https://spiral.ece.cmu.edu/mcm/generate2.php

[3] Voronenko, Y., & Püschel, M. (2007). Multiplierless multiple constant multiplication. *ACM Trans. Algorithms*, 3(2), 11.

[4] Aksoy, L., Güneş, E. O., & Flores, P. (2010). Search algorithms for the multiple constant multiplications problem: Exact and approximate. *Microprocess. Microsyst.*, 34(5), 151–162.

[5] Kumm, M. (2018). Optimal constant multiplication using integer linear programming. *IEEE Trans. Circuits Syst. II, Exp. Briefs*, 65(5), 567–571.

[6] Hartley, R. I. (1996). Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Trans. Circuits Syst. II, Exp. Briefs*, 43(10), 677–688.

[7] Hosangadi, A., Fallah, F., & Kastner, R. (2005). Reducing hardware complexity of linear DSP systems by iteratively eliminating two-term common subexpressions. In *Proc. ASP-DAC* (pp. 523–528).

[8] Ercegovac, M., & Lang, T. (2003). *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann.

[9] Aksoy, L., da Costa, E., Flores, P., & Monteiro, J. (2008). Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 27(6), 1013–1026.

[10] Ho, Y.-H., Lei, C.-U., Kwan, H.-K., & Wong, N. (2008). Global optimization of common subexpressions for multiplierless synthesis of multiple constant multiplications. In *Proc. ASP-DAC* (pp. 119–124).

[11] Lefevre, V. (2001). Multiplication by an integer constant. Dept. Laboratoirede l'informatique du parallélisme, Institut National de Recherche en Informatique et en Automatique, Le Chesnay-Rocquencourt, France, Rep. LIP RR-1999-06.

[12] Park, I.-C., & Kang, H.-J. (2001). Digital filter synthesis based on minimal signed digit representation. In *Proc. DAC* (pp. 468–473).

[13] Potkonjak, M., Srivastava, M. B., & Chandrakasan, A. P. (1996). Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 15(2), 151–165.

[14] Dempster, A. G., & Macleod, M. D. (1994). Constant integer multiplication using minimum adders. *IEE Proc. Circuits Devices Syst.*, 141(5), 407–413.

[15] Dempster, A. G., & Macleod, M. D. (1995). Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, 42(9), 569–577.

[16] Gustafsson, O. (2007). A difference based adder graph heuristic for multiple constant multiplication problems. In *Proc. ISCAS* (pp. 1097–1100).

[17] Kang, H.-J., Kim, H., & Park, I.-C. (2000). FIR filter synthesis algorithms for minimizing the delay and the number of adders. In *Proc. ICCAD* (pp. 51–54).

[18] Aksoy, L., Costa, E., Flores, P., & Monteiro, J. (2010). Optimization of area and delay at gate-level in multiple constant multiplications. In *Proc. DSD* (pp. 3–10).

[19] Aksoy, L., & Gunes, E. (2008). Area optimization algorithms in high-speed digital FIR filter synthesis. In *Proc. SBCCI* (pp. 64–69).