Wrocław 31.05.2023

Jakub Gonczarek 263999

„Automatyczna wektoryzacja kodu - analiza ograniczeń na podstawie kompilatora gcc / clang”

Spis treści:

1. Wprowadzenie do metodyki przeprowadzania badań - założenia

Zadanie polega na przetestowaniu mechanizmu automatycznej wektoryzacji w kompilatorze gcc/clang. Hasło analizy ograniczeń skłania się ku interpretacji podstawowych ograniczeń kompilatora jak i powodów wyniknięcia takowych, stąd testy zostały dostosowane do wypełnienia założenia o zbadaniu krytycznych ograniczeń uniemożliwiających przeprowadzenie skutecznej wektoryzacji kodu skalarnego. Podczas testów zdecydowałem się użyć obydwu kompilatorów - gcc (g++) oraz clang (clang++). Środowisko programistyczne, na którym testy zostały przeprowadzone to Visual Studio Code na maszynie o systemie macOS, wyposażonej w procesor Intel i7 , zgodnie z notą producenta dostosowany do rozszerzeń wektorowych rozkazów z bibliotek MMX, SSE, AVX. Procedura testów została uczyniona następująco: wyszukanie informacji o danym zagadnieniu / możliwości wektoryzacji, następnie wykonane zostały testy przy zmiennych trybach pracy kompilatora (flagi w podp. 2.3) , oraz porównanie wyników działania było stwierdzane poprzez przegląd listy rozkazów kodu asemblerowego w poszukiwaniu wykorzystania porządnych mnemoników. Dodatkowo poszczególne konfiguracje pozwalają na wypisywanie komunikatów o procesie kompilacji, gdzie dochodzi do np. Loop-unrollingu, Super-word-parellelismu, skutecznym zwektoryzowaniu pętli bądź informacjach o przekroczeniu limitów lub zbyteczności (nieefektywności) wektoryzacji. Kod wejściowy jest napisany w języku C, na potrzeby zbadania niektórych zachowań została wykorzystana rozszerzona wersja tego języka - C++. Do przeprowadzania procesu budowania linkowania i kompilacji został użyty program Makefile, pozwalający na skrótowe wywoływanie szeregu komend wiersza poleceń Terminala.

2. Konfiguracje kompilatora

2.1 Proces kompilacji

W badanym kompilatorze clang / clang++ występują dwie jednostki odpowiedzialne za wektoryzację kodu: “Loop-Vectorizer” i “SLP-Vectorizer”. Pierwsza z nich według swojej nazwy poszukuje możliwości wektoryzacji wszelakich pętli, które omówione są w podrozdziale 4.1. Istotny w procesie badań okazała się model kosztu wbudowany w tą jednostkę, ponieważ wiele wyników spełniających wymagania wektoryzacji było odrzucane lub pomijane ze względu na swoją nieefektywność (omówienie w podp. 6). W podpunkcie 2.3 omówiona została możliwość modyfikacji tego zachowania, jednakże przypadek wektoryzacji “na siłę” nie jest przedmiotem badań, stąd nie zostały przeprowadzone wyczerpujące testy tego zachowania. Druga jednostka poszukuje optymalizacji wektorowej w “Superword level parallelismie” co oznacza złożenie niezależnych instrukcji względem siebie w instrukcje wektorowe. Mowa tutaj o sięgnięciach do pamięci, operacjach arytmetycznych, porównaniach i innych. Sam kod skalarny już ma pewną optymalizacje w postaci sprzętowej do wykonywania

niezależnych rozkazów, jednakże każdy taki rozkaz jest osobą pozycją w potoku. Wektoryzacja zapewnia wykorzystanie jednego rozkazu do wykonania zadań paru i tym samym przyspiesza proces obliczeń.

2.2 Flagi kompilatora

Istotna w drodze testów była modyfikacja zachowań kompilatora. Poniższe flagi zostały użyte w różnych przypadkach do osiągnięcia rezultatów. Do każdej wypisanej flagi dodany jest opis, gdzie różnica zachowania została zauważona w ciągu testów.

- Flaga wyłączająca działanie wektoryzacji:

"-fno-vectorize"

Flaga ta podana bądź nie powodowała brak występowania mnemoników wskazujących na użycie instrukcji wektorowych niezależnie od modyfikacji flag "-O". Bazowe działanie kompilatora nie wskazywało na domyślne użycie modułu wektoryzacji w procesie kompilacji.

- Flagi optymalizacji:

"-O1"

Flaga indykująca najniższy stopień optymalizacji. Przy użyciu nie dała żadnych rezultatów w kwestii wektoryzacji mimo podania flag wskazujących na użycie rozszerzeń.

"-O2"

Flaga większej optymalizacji kodu kompilowanego, podobnie co "-O1" nie dawała kodu zwektoryzowanego bez instrukcji wskazujących na użycie rozszerzeń. Większość testów została wykonana z użyciem tej flagi, gdyż rezultaty były generowane automatycznie bez żadnych ostrzeżeń i wyłącznie dla przypadków ściśle dozwolonych do wektoryzacji

"-O3"

Flaga największej (agresywnej) optymalizacji kodu kompilowanego, podobnie co pozostałe flagi bez dodania stosownych flag rozszerzeń sama nie powodowała otrzymania kodu zwektoryzowanego. Przy użyciu wraz z odpowiednimi flagami dostarczała różnych rezultatów w postaci zwektoryzowanego kodu z pewnymi uchybieniami, o których mowa w podp. 6.

- Flagi wskazujące na użycie określonego rozszerzenia:

"-avx" "-avx2" "-avx512"

Flagi odpowiadające za użycie rozszerzeń wektorowych dla bibliotek AVX zoptymalizowanych pod zmiennoprzecinkowe reprezentacje liczb. Do testów została użyta flaga "-avx2".

"-msse" " -msse2" "-msse3" i inne

Flagi odpowiadające za użycie rozszerzeń wektorowych dla bibliotek SSE zoptymalizowanych pod zmiennoprzecinkowe reprezentacje liczb, będącymi poprzednimi wersjami rodziny AVX. Nie zostały uwzględnione w testach.

"-ftree-vectorize"

Flaga kompilatora GCC do aktywacji automatycznej wektoryzacji. Została użyta parę razy do porównania wyników z kompilatorem clang / clang++.

- Flagi realizujące wypisywania informacji o procesie wektoryzacji kodu podczas kompilacji:

"-Rpass=loop-vectorize"

Flaga identyfikująca czy dana pętla została zwektoryzowana z powodzeniem. Po zakończeniu kompilacji daje informacje w postaci testowej wskazując na numer linii rozpoczęcia pętli oraz niektóre informacje o charakterze zastosowania podziału w typie *packed* użytym do wykonania instrukcji.

"Rpass-missed=loop-vectorize"

Flaga identyfikująca czy dana pętla nie została zwektoryzowana z powodzeniem.

"-Rpass-analysis=loop-vectorize"

Flaga wypisująca przyczynę (dodatek informacyjny do flagi poprzedniej) o braku możliwości, bądź nie opłacalności wektoryzacji.

- Flagi kontrolujące zachowanie wektoryzacji:

"-mllvm"

Flaga ukazująca na użycie dodatkowego kompilatora, a wraz z nim dodatkowych mechanizmów optymalizacji. Użycie tej flagi znikomo wpływało na uzyskiwane wyniki, stąd została ona pominięta w przykładach.

“-loop-vectorize”

Flaga aktywująca moduł odpowiedziały na optymalizacje pętli i ich wektoryzację, została wykorzystana wielokrotnie podczas testów zarówno w kompilatorze gcc jak i clang.

“-force-vector-width=[0-n]”

Flaga wymuszają stosowanie określonej szerokości wektora, stosowana jest do kompilatora LLVM. Wymusza manualne zastąpienie systemowo wybranych ułożeń danych. Przy testowaniu przykłady automatycznie były układane w sposób optymalny, stąd użycie tej flagi było zbędne. Dodatkowo użycie mogło się wiązać z ryzykiem błędów kompilacji.

“-force-vector-interleave=[0-n]”

Flaga stosowana do kontroli rozwijania pętli. Pozwala w niektórych przypadkach na zapobiegnięcie odrzucenia wektoryzacji poprzez nieefektywność sugerowaną poprzez model kosztu. Zastosowana dawała podobne jak nie identyczne wyniki co automatyczna kompilacja, zatem stosowana była do prób ominięcia modelu kosztu.

“-fno-slp-vectorize”

Flaga włączająca moduł SLP do działania kompilatora. Stosowana dawała różne wyniki wektoryzacji omówione w rozdziale 5.

3. Opis badanego problemu

3.1 Zarys mechanizmu wektoryzacji kodu

Mechanizm wektoryzacji to metoda zrównoleglania obliczeń. Skrót rodziny wykorzystującej założenie równoległego zastosowania operacji do więcej niż jednego źródła danych to SIMD (“single instruction multiple data”). Główne zastosowania rozkazów równoległego przetwarzania stosuje się w grafice i multimediach jako remedium na powtarzalne obliczenia o dużych rozmiarach. Sama metoda polega na wykorzystaniu szerszych rejestrów np. xmm (128 bit-wide), ymm (254 bit-wide), zmm (512 bit-wide) do równoległego zapisu paru / parunastu liczb i wykonaniu tego samego działania na każdej z tych liczb. Do zarządzania upakowaniem różnego rodzaju liczb stosowany jest specjalny typ danych - “packed”, który zawiera podstawowe informacje o ułożeniu liczb w wektorze. Rozszerzenia rodziny AVX stosują w mnemonikach sufix -PD oznaczający “packed double” jako, że natywnym typem adresowanym przez bibliotekę AVX jest typ zmiennoprzecinkowy double. Dla rozszerzenia MMX natywnym typem są liczby stałoprzecinkowe zapisywane na mniejszej ilości bitów, stąd

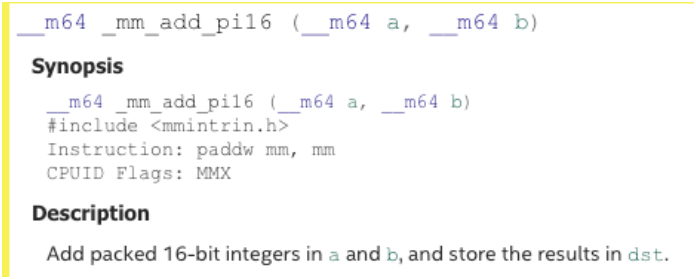
podstawowymi rejestrami są rejestry “mmx[0-7]”. Poniższa tabela pokazuje jakie rozmiary danych są używane w rozszerzeniach wektorowych.

Type	Instruction suffix	Form
Byte	“-b”	8x8 bits
Word	“-w”	4x16 bits
(double) dWord	“-d”	2x32 bits
(quad) qWord	“-q”	1x64 bits

Tabela 1

3.2 Charakterystyczne mnemoniki bibliotek XMM, AVX, AVX2

Dla testów z użyciem MMX mnemoniki charakterystyczne dla tego rozszerzenia są zapisane w dokumentacji rozkazów firmy Intel pod linkiem (1) zawartym w bibliografii. Strona zawiera opisy używalnych z poziomu języka C “Intrinsics” - specjalnych rozkazów wbudowanych do języka, których implementacją zajmuje się kompilator. Pozwalają na wykorzystanie konkretnej instrukcji assemblera z użyciem dwóch typów *packed* - *__m64*, które oznaczają “spakowane” reprezentacje liczb typu *Integer* z podziałem na te bezznakowe i te ze znakiem. Przykładem takiej *inline* funkcji kompilatora jest:



Pozwala spakować po 64 / 16 = 4, 4 reprezentacje liczb w formacie Integer i dodać je do drugiego spakowanego typu, a wynik zapisać do wskazanego miejsca.
Przykład takiego wykorzystania znajduje się w rozdziale 6 - Dodatkowe spostrzeżenia. Mnemoniki rozszerzenia MMX wyglądają następująco: “*PADDB, PADDW, PADDD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PACKUSWB, PACKSSWB, PACKSSDW*”. Prefix P- oznacza *packed,* następnie zapisane są charakterystyczne mnemoniki tradycyjnych rozkazów *ADD, CMP, SUB, MUL*, rozkaz kończy sufix informujący o charakterze rozkazu, gdzie:

-U oznacza reprezentacje bezznakową

-S oznacza reprezentacje ze znakiem

-UNPCK oznacza rozpakowanie wektora

-PACK oznacza spakowanie wektora

Dodatkowo jest zawarta informacja o rozmiarze pojedynczej liczby w wektorze zgodna z < Tabela 1 >. Dla rozszerzenia AVX przykładem mnemoników są: *VFMADDSUBPD, VFMADDSD, VFMADDPD, VFMSUBSS.* Charakterystycznymi sufiksami dla tego rozszerzenia są -PD -PS oznaczające instrukcje dostosowaną dla wektora zapakowanego liczbami zmiennoprzecinkowymi, natomiast sufiksy -SS -SD aplikują się do instrukcji procesujących operacja na liczbach pojedynczej precyzji (-SS) i podwójnej (-SD). Przykładowy zestaw instrukcji znajdują się w <Wstawka 1>. Dodatkowo instrukcje z tego rozszerzenia zawierają prefiks V- oznaczający “Vector”.

```
vcvtpd2ps   %ymm6, %xmm6
vcvtpd2ps   %ymm3, %xmm3
vinsertf128 $1, %xmm3, %ymm6, %ymm3
vmaskmovps  %ymm3, %ymm7, (%r9,%rax,4)
vxorps  (%rdx,%rax,4), %ymm2, %ymm3
vsubps  %ymm5, %ymm3, %ymm3
vcvtps2pd   %xmm3, %ymm5
vcmpnleps   %ymm4, %ymm1, %ymm4
vextractf128    $1, %ymm3, %xmm3
vcvtps2pd   %xmm3, %ymm3
vcvtps2pd   16(%rsi,%rax,4), %ymm6
vcvtps2pd   (%rsi,%rax,4), %ymm7
vaddpd  %ymm7, %ymm7, %ymm7
vdivpd  %ymm7, %ymm5, %ymm5
vaddpd  %ymm6, %ymm6, %ymm6
vdivpd  %ymm6, %ymm3, %ymm3
```

Wstawka 1

Podstawowe ograniczenia

4.1 Niezmienniki pętli

Wektoryzacja z swojego charakteru jest operacją złożoną. Zapętlenie instrukcji jest metodą seryjnego wykonania określonej liczby instrukcji, która pozwala na zrównoleglanie w określonych sytuacjach. Wszystkie rodzaje tradycyjnych pętli tzn. “For, while, do-while, pętle z skokiem (goto)” pozwoliły na uzyskanie wyniku kodu zwektoryzowanego. Modyfikacje zmiennych pętlowych są kluczowe, gdyż jawności (znana liczba podczas kompilacji) ilości instrukcji / zapętleń decyduje o możliwości zastosowania zrównoleglenia. Poniższe obserwacje określają kiedy auto-wektoryzacja przebiega pomyślnie.

- Pętla policzalna: Oznacza to znaną liczbę iteracji pętli w momencie kompilacji, przykładem takiej pętli jest : *“for (int i = 88; i < 1088; i++)”*. W tym przykładzie

liczba iteracji nie jest zależna od zmiennej zewnętrznej czy globalnej, która mogła by ulec zmianie w trakcie wykonywania ciała pętli. Przykładowo jeśli pętla byłaby zależna od takiej zmiennej to mógłby wystąpić przypadek kiedy nie starczy danych do stworzenia kompletnego typu packed, a zatem instrukcja będzie operowała na niepełnych danych o nieznanym formacie przypisania do danego miejsca w pamięci co łatwo może prowadzić do zapisu w niewłaściwych miejscach a finalnie do błędnego bądź nie bezpiecznego operowania na pamięci. Istnieje możliwość przypisania wartości sterujących pętlą do zmiennych z tym zastrzeżeniem, że programista musi zagwarantować ich niezmienność podczas wykonania pętli. *"const int wartosc = 999; for (int i = wartosc; i < 10*wartosc; i++)"* jest przykładem implementacji takiej pętli.

- Pętla policzalna z zmiennym charakterem przyrostu wartości: Mowa tu o pętlach, których inkrementacja zmiennej pętlowej jest różna od 1 (i++). Realizacja takich pętli jest dozwolona o ile warunek policzalności nie został naruszony. Skoki zmiennej o 1,2,5,n oraz o wartości ujemne w pętli zostaną z powodzeniem zwektoryzowane. Zmienna pętlowa nie musi też być całkowita, typy double, float oraz ich podtypy także są możliwe do użycia.
- Pętle z kontrolą zamieszczoną wewnątrz ciała funkcji (while, do-while): Pętle te w swoim zapisanie nie przewidują pełnej kontroli w swojej definicji, niemiej wektoryzacja ich jest również możliwa. Przykład: *"int i = 100; while (i< 1000) { ... i++;}"*. Operacje arytmetyczne na zmiennej wewnątrz ciała są dozwolone, również na warunku stopu pętli.

Każda pętla, która naruszy warunek policzalności automatycznie zostanie pominięta z komunikatem: "remark: loop not vectorized: value that could not be identified as reduction is used outside the loop".

Przykłady nie właściwie zadeklarowanych pętli:

*"while (i < 1.2*i)"*

"while (zmienna_petli < zmienna_zewnetrzna) "

" do {...} while (true)"

*"For (int zmienna_petli = 10*I; I < I/2; I+=5)"*

4.2 Kontrola Przepływu

Każda operacja wewnątrz pętli musi zachować ramy ciągłości by dało się dopasować instrukcję operującą na zbiorczych danych. Dużym utrudnieniem dla wektoryzacji jest nieciągłość kodu lub zwyczajne skoki podczas występowania różnych warunków. Kontrol przepływu jest największym ograniczeniem dla procesu zrównoleglania kodu, poprzez znaczące komplikowanie kolejności instrukcji. Warunkiem powodzenia wektoryzacji pętli jest jej nierozgałęzianie (*"Branching"*). Każda zagnieżdżona instrukcja wykonująca skok do innego miejsca niż w pętli jest potencjalnie powodem uniemożliwiającym określenie ciągłości pętli - nagłe przerwanie działania schematycznych obliczeń na rzecz kompletnie innego działania. Kontrola przepływu obejmuje takie instrukcję jak if, goto, switch, inline if, continue, break, return, quit, guard, try catch czy wywołanie innej funkcji (rozbudowana wersja skoku). Występują wyjątki, które zostaną dokładniej omówione w dalszej części pozwalające na użycie instrukcji warunkowych if, lecz ich zakres jest znacząco ograniczony. Dozwolony jest jedynie system przerwań i obsługa wyjątków, które są realizowane poza właściwym programem i niektóre bloki sterujące są uznawane za kompilator za "bezpieczne" do włączenia pod proces automatycznej wektoryzacji. Powodem, przez który kontrola przepływu jest utrudnieniem zrównoleglania jest nie przewidywalność kolejności wykonania instrukcji. Przykładowo próba skompletowania 8 reprezentacji 16 bitowych liczb stałoprzecinkowych do rejestru XMM (instrukcja z rozszerzenia AVX) może zostać przerwana w trakcie poprzez zmianę obecnej ramki stosu, co wprowadza wiele niepewności czy przerwana instrukcja będzie kontynuowana czy zostanie unieważniona.

<u>Wyjątki umożliwiające zastosowanie kontroli przepływu:</u>

- Instrukcje warunkowe if mogące zostać przetworzone do postaci *"masked assigment"*. Kompilator może rozpatrzyć niektóre sprawdzenia jako złożenie paru innych instrukcji, przykładowo czy dane elementy w wektorze są zerowe czy nie zerowe. Działanie takie implikuje wytworzenie paru instrukcji sprawdzających zawartość badanego wektora z przygotowaną *"maską"* , co w rezultacie rozwiązuje problem rozgałęzienia, gdyż nie jest zastosowana instrukcja skoku, lecz ciąg instrukcji realizujących warunek ciągłości. Wszelakie instrukcję sprawdzające jakieś cechy elementów wektora są przetwarzane do postaci operacji na maskach, włączając w to operacje logiczne i arytmetyczne.
- Instrukcje rozgałęziające się w pętlach zagnieżdżonych. W tym przypadku warunkiem powodzenia zrealizowania takiego rozgałęzienia jest rozwinięcie pętli wewnętrznej bądź wewnętrznej i zewnętrznej, gdzie skok nie będzie implikował zmiany kontekstu danych.
- Kiedy możliwe jest zastosowanie *"loop interchangmentu",* co oznacza że funkcje są z sobą powiązane w taki sposób, że możliwa jest zamiana ich zmiennych sterujących. Zmiana kontroli przepływu wtedy nie warunkuje utracenia ciągłości instrukcji, zatem wektoryzacja przebiegnie pomyślnie. Przykładem złożenia pętli realizujących

tą zależność są tradycyjne pętle przechodzące po zawartości macierzy: *“for (i = 0 ; io < n; i++) for (j = 0; i < n; j++)”.*

- *K*iedy pętle mogą zostać poddane zastosowaniu *“loop-unrolingu”*. Mechanizm ten oznacza zamianę pętli w szereg repetetywnych sekwencji wykonujących niezależne zadania, zatem poszczególne zmiany kolejności (zmiana instrukcji po skoku) nie wpływają na wynik działania.
- Wywoływanie możliwych do uproszczenia funkcji, bądź zoptymalizowanych pod wektoryzację wewnątrz pętli. Rozszerzenia AVX zapewniają pewną pulę funkcji wspierających użycie w kontekście zwektoryzowanego kodu. Przykłady takich funkcji zawiera poniższa <Tabela 1>. Również funkcje typu *lambda* są dozwolone do użycia wewnątrz pętli z zastrzeżeniami o zachowaniu ciągłości.

acos	ceil	fabs	round
acosh	Cos	floor	sin
asin	Cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	Erfc	log	tan
atan2	Erfinv	log10	tanh
atanh	Exp	log2	trunc
cbrt	exp2	pow	

Tabela 1

4.3 Zagnieżdżanie pętli

Zagnieżdżanie jest operacją stosowaną przy operacjach na strukturach wielowymiarowych, zatem w większości o znanych parametrach rozmiaru i warunku stopu. Wektoryzacja wspiera szeroką gamę użycia pętli zagnieżdżonych, gdzie ograniczenia sprowadzają się do warunków opisanych w podpunktach 4.1 i 4.2. Dodatkowych warunkiem przy optymalizacji zagnieżdżeń jest konieczność modyfikacji od najbardziej zagnieżdżonej pętli do pętli zewnętrznej. Próby wielokrotnych zagnieżdżeń (5 - 10 zagnieżdżeń) o różnych charakterach spotykały się z odpowiedzią od modelu kosztu o nie efektywności stosowania wektoryzacji, natomiast wielokrotne zagnieżdżenia realizujące podobny charakter obliczeń z powodzeniem były optymalizowane. Przypadki zawarte w podpunkcie <4.5 Przykłady> obrazują zamysł. Przy niskim poziomie zaawansowania operacji w ramach pętli oraz jej stosunkowego niedługiego czasu życia (mała liczba iteracji przy przetwarzaniu wektorowym) wynikowy kod zawierał rozwinięcia pętli do szeregu instrukcji realizujących działanie pętli bez skoku (np. 20 instrukcji VADDCPD, VMULPD pod rząd). Przy realizacji zagnieżdżeń, gdzie zmienne sterujące pętli wewnętrznej były zależne od pętli zewnętrznej wektoryzacja się nie powodziła. Komunikat diagnostyczny był następujący:

“loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]”

Oznacza to naruszenie wymogu o policzalności pętli, gdzie zmienne sterujące miały wiele odniesień do nie stałych parametrów podczas wykonywania któregoś z zagnieżdżenia. Wynikowy kod był skalarny, co było oczekiwanym wynikiem. Istnieje ręczna metoda, która pozwala na bezpośrednie przekazanie do kompilatora wymogu rozwinięcia pętli - za pomocą dyrektywy *“#pragma unroll”* stawanej przed inicjowaniem pętli. Wynik kompilacji przy użyciu tej dyrektywy przedstawi ciąg instrukcji bez skoku realizujący ciało pętli. Użycie tej dyrektywy rozwija także pętle, które nie zostały by automatycznie rozwinięte przez kompilator (ciągi instrukcji stanowiące ok. 50 + instrukcji), lecz jest to dodatkowa opcja determinowana działaniem programisty.

4.4 Funkcje

Wywoływanie funkcji jest akceptowalnym działaniem w module realizującym procesowanie SLP. Tam moduł szuka niezależnych względem siebie instrukcji pasujących formatem do przetworzenia równoległego. W pętlach wywoływanie funkcji w znaczącej większości przypadków wiąże się z naruszeniem wymogu o ciągłości. Wywołanie funkcji spoza puli przedstawionej w <Tabela 1> jest operacją trudną, gdyż na etapie kompilacji musi ona zostać złączona z potokiem operacji wykonywanym w ramach ciała pętli. Niemiej tworzenie własnych funkcji z możliwością ich wektoryzacji jest dostępne. Przykładowo funkcja:

```c
int params_func( float a, float b, float c, float d) {
  float wynik;
  wynik = sqrt((a-c)*10+(b-d)*20);
  wynik = (int)floor(wynik);
  return wynik;
}
```

```
__Z11params_funcffff:                   ## @_Z11params_funcffff
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    vsubss  %xmm2, %xmm0, %xmm0
    vmulss  LCPI1_0(%rip), %xmm0, %xmm0
    vsubss  %xmm3, %xmm1, %xmm1
    vmulss  LCPI1_1(%rip), %xmm1, %xmm1
    vaddss  %xmm1, %xmm0, %xmm0
    vsqrtss %xmm0, %xmm0, %xmm0
    vroundss    $9, %xmm0, %xmm0, %xmm0
    vcvttss2si  %xmm0, %eax
    popq    %rbp
    retq
    .cfi_endproc
                                        ## -- End function
```

realizująca operacje na argumentach z powodzeniem zostanie zwektoryzowana. Wywołanie jej w ciele funkcji z ręcznie podanymi parametrami realizującymi ciągłość dostępu do pamięci zostanie wykonane, a wektoryzacja pętli się powiedzie. W procesie testowania następujące obserwacje zostały określone:

- Niezależne funkcje mogą być wektoryzowane i wywoływane w innych funkcjach / funkcji głównej.
- Niezależne funkcje mogą być wywoływane w pętlach, jeśli są dostosowane do operacji zwektoryzowanych to próba wektoryzacji pętli się powiedzie, jeśli są niedostosowane zostanie wypisany komunikat: *"Disabling scalable vectorization, because target does not support scalable vectors."*.
- Zwektoryzowana funkcja wywołująca następną zwektoryzowaną funkcję produkuje w pełni zwektoryzowany kod. W przypadku kiedy wektoryzowalna funkcja będzie operowała na typach nie dostosowanych do używanego rozszerzenia, wektoryzacja nie zajdzie (argumenty int dla funkcji z rozszerzeniem AVX nie produkują kodu wykorzystującego instrukcje równoległe i na odwrót - double / float z MMX).

4.5 Przykłady

```c
void f_vect() {
    i = 10;
    do
    {
      f1:
      out[i] = sqrt(out[i] * 2);

      if (out[i] > 50) {
        goto f1;
      }

    } while ( i++ < 1000 );
    return;
}
```

```
__Z6f_vectv:                            ## @_Z6f_vectv
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    $10, %eax
    leaq    _out(%rip), %rcx
    vmovsd  LCPI0_0(%rip), %xmm0            ## xmm0 = mem[0],zero
    .p2align    4, 0x90
```

Brak wektoryzacji poprzez wystąpienie rozgałęzienia wewnątrz pętli.

```
int foo(int *A) {
  unsigned n = 50;
  unsigned sum = 0;
  for (int i = 0; i < n; ++i
      sum += A[i];
  return sum;
}
```

```
__Z3fooPi:                              ## @_Z3fooPi
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    vmovdqu 32(%rdi), %ymm0
    vpaddd  (%rdi), %ymm0, %ymm0
    vpaddd  64(%rdi), %ymm0, %ymm0
    vpaddd  96(%rdi), %ymm0, %ymm0
    vpaddd  128(%rdi), %ymm0, %ymm0
    vpaddd  160(%rdi), %ymm0, %ymm0
    vextracti128    $1, %ymm0, %xmm1
    vpaddd  %xmm1, %xmm0, %xmm0
    vpshufd $238, %xmm0, %xmm1              ## xmm1 = xmm0[2,3,2,3]
    vpaddd  %xmm1, %xmm0, %xmm0
    vpshufd $85, %xmm0, %xmm1               ## xmm1 = xmm0[1,1,1,1]
    vpaddd  %xmm1, %xmm0, %xmm0
    vmovd   %xmm0, %eax
    addl    192(%rdi), %eax
    addl    196(%rdi), %eax
    popq    %rbp
    vzeroupper
    retq
    .cfi_endproc
                                        ## -- End function
    .globl  _in                             ## @in
```

Nastąpiło rozwinięcie pętli i wektoryzacja jest pomyślnie zastosowana.

```
122  void f_vect() {
123    int i = 10;
124    while ( i++ < 300 ) {
125      out[i] = in[i] > 10? floor(in[i]) : ceil(in[i]);
126    }
127  }
```

```
    movl    $23, %eax
    leaq    _in(%rip), %rcx
    vbroadcastsd    LCPI2_0(%rip), %ymm0    ## ymm0 = [1.0E+1,1.0E+1,1.0E+1,1.0E+1]
    leaq    _out(%rip), %rdx
    .p2align    4, 0x90
LBB2_1:                                 ## =>This Inner Loop Header: Depth=1
    vmovupd -96(%rcx,%rax,8), %ymm1
    vmovupd -64(%rcx,%rax,8), %ymm2
    vmovupd -32(%rcx,%rax,8), %ymm3
    vmovupd (%rcx,%rax,8), %ymm4
    vcmpltpd    %ymm1, %ymm0, %ymm5
    vcmpltpd    %ymm2, %ymm0, %ymm6
    vcmpltpd    %ymm3, %ymm0, %ymm7
    vcmpltpd    %ymm4, %ymm0, %ymm8
    vroundpd    $9, %ymm1, %ymm9
    vroundpd    $9, %ymm2, %ymm10
    vroundpd    $9, %ymm3, %ymm11
    vroundpd    $10, %ymm1, %ymm1
    vblendvpd   %ymm5, %ymm9, %ymm1, %ymm1
    vroundpd    $9, %ymm4, %ymm5
    vroundpd    $10, %ymm2, %ymm2
    vblendvpd   %ymm6, %ymm10, %ymm2, %ymm2
    vroundpd    $10, %ymm3, %ymm3
    vblendvpd   %ymm7, %ymm11, %ymm3, %ymm3
    vroundpd    $10, %ymm4, %ymm4
    vblendvpd   %ymm8, %ymm5, %ymm4, %ymm4
    vmovupd %ymm1, -96(%rdx,%rax,8)
    vmovupd %ymm2, -64(%rdx,%rax,8)
    vmovupd %ymm3, -32(%rdx,%rax,8)
    vmovupd %ymm4, (%rdx,%rax,8)
    addq    $16, %rax
    cmpq    $311, %rax                  ## imm = 0x137
    jne LBB2_1
```

in.cpp:124:3: remark: vectorized loop (vectorization width: 4, interleaved count: 4) [-Rpass=loop-vectorize]

Przykład użycia pętli z instrukcja warunkową przekształconą do postaci “masked assignment”

Złożone ograniczenia

5.1 Nieciągłość instrukcji sięgania do pamięci

Instrukcje równoległe są najbardziej efektywne kiedy dane można załadować serią, co znaczy że dane muszą być ulokowane “blisko siebie” w pamięci. Kompilator użyty z flagą “-O3” w większości przypadków będzie wektoryzował wszelakie permutacje pobierania z pamięci, lecz po kompilacji będzie widoczna informacja o możliwej nieefektywności działania. Warunkiem powodzenia wektoryzacji z użyciem flagi “-O3” jest schemat pobrania danych z pamięci. Przykładowo dla przetwarzania dużych tablic znacznie lepiej wypadnie następująca pętla:

“ For (long unsigned int loop_variable = 0; loop_variable < 10000000; loop_variable++)”

Niż pętla realizującą swoje działanie z odstępem integracyjnym:

“ For (long unsigned int loop_variable = 0; loop_variable < 10000000; loop_variable+=10)”

Trudniejszym przypadkiem dla wektoryzacji są pętle z niestałą inkrementacją zmiennej pętlowej, przykładowo:

*“ For (long unsigned int i = 0; i < 10000000; I+=**i**)”*

Podobnie trudnym przypadkiem do wektoryzacji jest następująca pętla zawierająca agregację wektora do zmiennej sumującej, gdzie pętla nie została rozwinięta:

```
14  int func1() {
15    int i = 0;
16    double sum;
17    while ( i++ < sizeof(in)/sizeof(double) )
18      sum += in[i];
19    return (int)sum;
20  }
```

```
__Z5func1v:
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    popq    %rbp
    retq
    .cfi_endproc

    .globl  _main
    .p2align    4, 0x90
```

in.cpp:17:3: remark: loop not vectorized [-Rpass-missed=loop-vectorize]

Jeśli ciało funkcji zawierałoby jakiekolwiek odniesienie do pamięci to nieciągłość nie pozwala na efektywne ładowanie danych, zatem pętla nawet jeśli byłaby dostosowana do działania równoległego na danych nie zostanie zwektoryzowana. Podobnie dla funkcji, gdzie ładowanie argumentów niezależnych będzie znacznie szybsze jeśli są umiejscowione w jednym bloku pamięci, niż gdyby pobranie pojedynczego wiązało by się z dostępem do pamięci z oddalonego bloku (w takich sytuacjach w zależności od powiązania argumentów kod może ale nie musi zostać zwektoryzowany). Zależność w

instrukcji *“tab[a] = tab[a-1]”* jest trudna to zrównoleglania, szczególnie w przypadku zastosowania w pętlach, dla instrukcji w ciele funkcji o ile wystąpień takich instrukcji będzie parę, wektoryzacja ma szansę na powodzenie (większa odpowiedzialność leży w rękach programisty o zadbanie o charakter dostępu do danych by uzyskać kod równoległy). Kod, gdzie dostęp do pamięci nie wykazuje schematycznego działania nie będzie wektoryzowany ze względu na nieefektywność, z wyjątkiem “wektoryzacji skalarnej”, która może być uznana za błąd co opisuje podrozdział 6.

5.2 Współzależności danych

Charakter przetwarzania równoległego wiąże się z założeniem, że seryjnie pobrane i przetworzone dane zapisywane są w tej samej kolejności do miejsca docelowego. Zmiana w tym schemacie powoduje konieczność rozpakowania wyniku do postaci skalarnej i poszczególnego dopasowania wyników do ich pożądanej destynacji. Dodatkowy nakład na realizacje tego aspektu rzutuje na skrajną nieefektywność działania instrukcji równoległych. Innym utrudnieniem dla działania zwektoryzowanego jest zazębianie się przetworzonych danych z nowymi danymi pobieranymi do kolejnej instrukcji. Taki wypadek powoduje zależność nowego operandu od wyniku poprzedniej instrukcji, co terminuje efektywność działania równoległego poprzez przestoje, bądź niepoprawność wyników. Zależności pomiędzy danymi można podzielić na 3 sekcje:

- Zależności “Read after write”: Przykładowo operacja na 8 reprezentacjach liczb typu *int* zazębia się z następną w następujący sposób: *“ tab_1[4:12] += tab_1[0:8] + 10 ”,* gdzie [0:n] oznacza zakres indeksów użytych do realizacji instrukcji. Widoczna jest zatem zależność indeksów od 4 do 12 od wyniku działania dodawania 10 do indeksów od 0 do 8. Instrukcje pobierają “naraz” po 8 reprezentacji *int* z pamięci, zatem instrukcja dodania dla [4:12] nie będzie miała właściwych danych. Próby wektoryzacji z taką zależnością spotykają się z komunikatem:

“loop not vectorized: cannot prove it is safe to reorder floating-point operations; allow reordering by specifying '#pragma clang loop vectorize(enable)' before the loop or by providing the compiler option '-ffast-math'. [-Rpass-analysis=loop-vectorize]”

Przykład:

```
int func1() {
  int i = 100;
    for ( int i = 1; i < 100; i++ )
      out[i] = out[i-1] + 20;
  return i;
}
```

```
__Z5func1v:                             ## @_Z5func1v
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    $3, %eax
    vmovsd  _out(%rip), %xmm1               ## xmm1 = mem[0],zero
    vmovsd  LCPI0_0(%rip), %xmm0            ## xmm0 = mem[0],zero
    leaq    _out(%rip), %rcx
    .p2align    4, 0x90
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
    vaddsd  %xmm0, %xmm1, %xmm1
    vmovsd  %xmm1, -16(%rcx,%rax,8)
    vaddsd  %xmm0, %xmm1, %xmm1
    vmovsd  %xmm1, -8(%rcx,%rax,8)
    vaddsd  %xmm0, %xmm1, %xmm1
    vmovsd  %xmm1, (%rcx,%rax,8)
    addq    $3, %rax
    cmpq    $102, %rax
    jne LBB0_1
```

Pomimo wystąpienia instrukcji wektorowych, kod nie charakteryzuje się zrównolegleniem, o czym mówi ułożenie danych w wektorach xmm1 i xmm0.

- Zależności “Write after read” jest odwrotnością poprzedniej zależności *“read after write”*. Chodzi o zależność kiedy druga iteracja przykładowej pętli pisze do danych wcześniej odczytanych przez poprzednią iteracje. Kod skalarny nie wykaże żadnego niebezpieczeństwa, jednakże seryjne korzystanie z pamięci z paru miejsc może powodować sytuacje przeczytania już nieaktualnych danych, co obrazuje poniższy przykład:

```
void func1() {
    for ( int i = 1; i < 100; i++ ) {
      out[i-1] = out[i] *22;
      in[i] = out[i]*5;
    }
}
```

Dwie wektorowe instrukcje zostaną zapełnione danymi, gdzie tablica *out[]* z każdą iteracją zmienia swoje dane, gdyż operacja *out[i-1] = out[i]*22* wykazuje się zależnością od następnych iteracji, zatem operacja przypisania *in[i] = out[i]*5* nie jest bezpieczna do przeprowadzenia.

- Zależności “Write after write”, gdzie zazębione dane z dwóch różnych przetwarzań piszą w to samo miejsce w pamięci co może powodować utracenie części wyników mogących być danymi dla następnych instrukcji. Wykrycie takiej zależności przez kompilator skutkuje wypisaniem ostrzeżenia o nie bezpiecznym przetwarzaniu równoległym i możliwej utracie wartości.

5.3 “Aliasowanie” wskaźników

Przypadek przetwarzania wskaźników jest potencjalnie niebezpiecznym zadaniem, szczególnie jeśli występują powiązania pomiędzy danymi. Kompilator instrukcje typu: *int* b = (a+k)* , gdzie *“a”* jest wskaźnikiem na tablicę traktuje w przetwarzaniu równoległym jako niebezpieczne i nie stosuje wektoryzacji. Przykład:

```
void function1() {
  for ( int i = 0; i < 100; i+=5 )
    *pout++ = *pin++;
}
```

```
__Z9function1v:
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movq    _pin(%rip), %rax
    movq    _pout(%rip), %rcx
    movl    (%rax), %edx
    movl    %edx, (%rcx)
    movl    4(%rax), %edx
    movl    %edx, 4(%rcx)
    movl    8(%rax), %edx
    movl    %edx, 8(%rcx)
    movl    12(%rax), %edx
    movl    %edx, 12(%rcx)
```

5.4 Przykłady

```
void arg_func(int length, float *a, float *b, float *c, float *x1, float *x2) {
    for (int i=0; i<length; i++) {
      float s = b[i]*b[i] - 4*a[i]*c[i];
      if ( s >= 0 ) {
        s = sqrt(s) ;
        x2[i] = (-b[i]+s)/(2.*a[i]);
        x1[i] = (-b[i]-s)/(2.*a[i]);
      } else {
        x2[i] = 0.;
        x1[i] = 0.;
      }
    }
}
```

```
## %bb.10:
    movl    %r10d, %edi
    andl    $-8, %edi
    xorl    %eax, %eax
    vbroadcastss    LCPI1_0(%rip), %ymm0
    vxorps  %xmm1, %xmm1, %xmm1
    vbroadcastss    LCPI1_1(%rip), %ymm2
    .p2align    4, 0x90
LBB1_11:                                ##
    vmovups (%rdx,%rax,4), %ymm3
    vmulps  %ymm3, %ymm3, %ymm4
    vmulps  (%rsi,%rax,4), %ymm0, %ymm5
    vmulps  (%rcx,%rax,4), %ymm5, %ymm5
    vaddps  %ymm5, %ymm4, %ymm4
    vsqrtps %ymm4, %ymm5
    vsubps  %ymm3, %ymm5, %ymm3
    vcvtps2pd   %xmm3, %ymm6
    vcmpleps    %ymm4, %ymm1, %ymm7
    vextractf128    $1, %ymm3, %xmm3
    vcvtps2pd   %xmm3, %ymm3
```

```
void func(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
  A[4] = a1*(a1 + b1);
  A[5] = a2*(a2 + b2);
  A[6] = a1*(a1 + b1);
  A[7] = a2*(a2 + b2);
  A[8] = a1*(a1 + b1);
  A[9] = a2*(a2 + b2);
  A[10] = a1*(a1 + b1);
  A[11] = a2*(a2 + b2);
  A[12] = a1*(a1 + b1);
  A[13] = a2*(a2 + b2);
  A[14] = a1*(a1 + b1);
  A[15] = a2*(a2 + b2);
}
```

```
__Z4funciiiiPi:                         ## @_Z4funciiiiPi
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    vmovd   %edx, %xmm0
    vpinsrd $1, %ecx, %xmm0, %xmm0
    vmovd   %edi, %xmm1
    vpinsrd $1, %esi, %xmm1, %xmm1
    vpaddd  %xmm1, %xmm0, %xmm0
    vpmulld %xmm1, %xmm0, %xmm0
    vpbroadcastq    %xmm0, %ymm0
    vmovdqu %ymm0, (%r8)
    vmovdqu %ymm0, 32(%r8)
    popq    %rbp
    vzeroupper
    retq
    .cfi_endproc
                                        ## -- End function
```

Dodatkowe spostrzeżenia

- Wykorzystanie funkcji, które pozwalają ręcznie ustawiać ułożenie danych w wektorze umożliwia ominięcie odrzucenia funkcji / pętli poprzez kompilator. W poniższym przykładzie użyta jest funkcja *_builtin_assume_aligned(),* która pozwala na sterowanie ułożeniem danych w wektorze. Dodatkowo przykład zawiera instrukcje warunkową, która zostaje przyjęta i przekształcona w postać dostosowaną do wektorów.

```
78   void func_test_8(double * a, double * b)
79   {
80     size_t i;
81     double *x = (double*)__builtin_assume_aligned(a, 16);
82     double *y = (double*)__builtin_assume_aligned(b, 16);
83     for (i = 0; i < SIZE; i++) {
84       if (y[i] > x[i]) x[i] = y[i];
85     }
86   }
```

in.cpp:83:2: remark: vectorized loop (vectorization width: 4, interleaved count: 4) [-Rpass=loop-vectorize]

- Podczas testów wielokrotnie udało się napotkać informacje od modelu kosztu o nieefektywności pętli, gdzie działanie jego było uzależnione od stopnia komplikacji sięgania do pamięci i zapisu od pamięci. Użycie funkcji czy aplikowanych instrukcji warunkowych nie powodowało występowania komunikatu.
- Nie występuje limit zagnieżdżania funkcji wektoryzowalnych, tzn: *“int w = floor(ceil(sin(tan(sqrt(b)))));”*

Podsumowanie i wnioski

Założenia projektu zostały spełnione, podstawowe i zaawansowane ograniczenia automatycznej wektoryzacji bez użycia dedykowanych funkcji języka C do realizacji instrukcji na kompilatorze zostały zidentyfikowane i przetestowane. Narzędzie wektoryzacji jest rozbudowaną drogą zrównoleglania spójnych i schematycznych obliczeń, co pozwala na znaczne przyspieszenie zdań w zakresie przetwarzania obrazów, dźwięku, grafiki komputerowej, modelowania trójwymiarowego, symulacji, itp. Największym ograniczeniem dla wektoryzacji są współzależności danych od siebie, gdzie próba zrównoleglania jest niedostosowanym podejściem do takich problemów i wiąże się z wieloma ryzykownymi zabiegami w pamięci prowadzących do niespójnych bądź błędnych rozwiązań. Narzędzie wektoryzacji posiada opcje dostosowywania pracy kompilatora do określonego podejścia programistycznego nastawionego na pisanie kodu pod operacje zrównoleglania, lecz wtedy automatyczna wektoryzacja przestaje być głównym tematem na rzecz programowania równoległego. Z powodzeniem zostało przeprowadzonych szereg testów wskazujących na opisane w sprawozdaniu ograniczenia i ryzyka

Bibliografia:

1. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=31,250,143,152&techs=MMX
2. https://en.wikipedia.org/wiki/Loop_interchange
3. https://en.wikipedia.org/wiki/Loop_unrolling
4. https://llvm.org/docs/Vectorizers.html#slp-vectorizer
5. https://gcc.gnu.org/projects/tree-ssa/vectorization.html
6. https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf
7. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)