POLITECHNIKA WROCŁAWSKA

Wydział elektroniki

Organizacja i Architektura Komputerów

Automatyczna wektoryzacja kodu -

analiza ograniczeń na przykładzie kompilatora gcc

Dokumentacja projektu

Prowadzący: Dr. inż. Tadeusz Tomczak
Autorzy: Arkadiusz Carzyński
Oskar Rek
Termin: Czw TN 11.15
Data: 06.06.2019 r.

1. Wstęp

Wektoryzacja jest mechanizmem umożliwiającym wykonywanie działań na typach wektorowych. Oznacza to, że w tym samym czasie przeprowadzanie jest kilka operacji na raz, co może prowadzić do znacznej poprawy wydajności kodu. Ręczne stosowanie wektoryzacji wymaga jednak pisania skomplikowanego i złożonego kodu, dlatego powszechnie wykorzystywana jest automatyczna wektoryzacja. Twórcy kompilatora zapewnili możliwość optymalizacji kodu bez konieczności zagłębiania się w języki assemblera. Muszą jednak zostać spełnione pewne warunki, co wymaga to pisania kodu w szczególny sposób. Ponadto nie każdy algorytm może zostać napisany w sposób umożliwiający autowektoryzację.

2. Założenia projektu

Projekt ten ma na celu pokazanie problemów, na jakie może się natknąć programista próbujący wykorzystać mechanizm automatycznej wektoryzacji. Na podstawie przykładów omówimy oraz pokażemy, w których przypadkach mechanizm może zostać zastosowany a w których nie, a na podstawie wygenerowanych kodów assemblera omówimy dlaczego taka sytuacja zaistniała bądź nie. Na potrzeby projektu poczyniliśmy następujące założenia:

- wykorzystujemy kompilator GNU GCC w wersji 7.2.0
- kod pisany jest w językach C/C++ standardzie c++11 oraz kompilowany do języka asemblera w architekturze x86 i składni AT&T
- podstawową jednostką jest liczba zmiennoprzecinkowa pojedynczej precyzji (float)
- oprócz badania wektoryzacji przy nieznanej wielkości pętli (punkt 5.1), we wszystkich przykładach korzystamy ze stałej długości pętli, która wynosi 100
- do kompilacji i wyświetlania informacji na temat wektoryzacji korzystamy z następującego polecenia: *g++ -fopt-info-vec-optimized -O3 -ffast-math -march=bdver2 -m32 plik.cpp* oraz *g++ -fopt-info-vec-missed -O3 -ffast-math -march=bdver2 -m32 plik.cpp.*
- do otrzymania kodów assemblera korzystamy z polecenia *g++ -S 3.3_plik.cpp -O3 -march=bdver2 -ffast-math -m32 -fno-exceptions -fno-rtti -fno-asynchronous-unwind-tables*
- w niektórych przykładach zakładamy wyrównanie danych (__builtin_assume_aligned()), by analizowany kod assemblera był bardziej przejrzysty
- w kilku przykładach skorzystaliśmy ze standardowej biblioteki szablonów C++ - STL. Na strukturach tablicy dynamicznej oraz listy dwukierunkowej rozważaliśmy problem wektoryzacji najbardziej przydatnych algorytmów dla programisty.
- projekt został zamieszczony w repozytorium (link:https://github.com/GhulJr/Projekt_wektoryzacja)

3. Generalne zasady

3.1 Słowo kluczowe

Kompilator gcc jest ograniczony w kwestiach tablic. Im więcej informacji mu przekażemy, tym większa jest szansa na większą optymalizację. Możemy poinformować kompilator o niemożliwości nakładania się danych na siebie. Do tego celu , dla wskaźników używamy kwalifikatora restrict. Słowo kluczowe restrict mówi, że tylko wskaźnik albo wartość pochodząca bezpośrednio od wskaźnika (np. pointer + 1), będzie używana, żeby otrzymać dostęp do obiektu.
Przykładowo, dla parametrów funkcji test jego konstrukcja może wyglądać następująco:

```
void test(int* __restrict__ a, int* __restrict__ b).
```

Używając flagi -std=c99 możemy również zapisać:

```
void test(int *restrict a, int *restrict b).
```

3.2 Wyrównanie danych

Niezwykle ważnym pojęciem jest **wyrównanie** (ang. alignment). Chcąc uzyskać pożądaną przez nas optymalizację, musimy powiedzieć kompilatorowi gcc o wyrównaniu danych. Rozważmy wbudowaną funkcję:

```
void * __builtin_assume_aligned (const void *exp, size_t align, ...).
```

Zwraca ona swój pierwszy argument i pozwala kompilatorowi założyć, że zwrócony wskaźnik jest wyrównany przynajmniej do podanej ilości bajtów. Taka funkcja może mieć dwa lub trzy argumenty. Jeśli posiada ich trzy, trzeci powinien mieć typ całkowity, a jeśli jest niezerowy, oznacza przesunięcie przesunięcia (ang. misalignment offset). Przykładowo:

```
void *x = __builtin_assume_aligned (arg, 16);
```

Oznacza to, że kompilator może założyć (ustawiając na arg), że x będzie wyrównany do 16 bajtów.
Możemy również, uwzględniając misalignment offset, wyrównać x do 32 bajtów, zapisując:

```
void *x = __builtin_assume_aligned (arg, 32, 8);
```

3.3 Prosty przykład

Na początku rozpatrzmy prosty przykład zastosowania wektoryzacji. W dalszej części projektu będziemy wykorzystywać ten sam schemat omawiania problemu. Przykładowa funkcja otrzymuje dwie dynamicznie alokowane tablice, następnie dodaje elementy drugiej tablicy do pierwszej:

```
1: #define SIZE 100
2: void foo(float* __restrict__ a, float* __restrict__ b)
3: {
4:     float *x = (float*)__builtin_assume_aligned(a, 16);
5:     float *y = (float*)__builtin_assume_aligned(b, 16);
```

```
6:      int i;
7:      for(i = 0; i < SIZE; ++i)
8:      {
9:              x[i]+=y[i];
10:     }
11: }
```

W tym, tak jak w każdym kolejnym przykładzie, będziemy korzystać z wcześniej opisanych słowa kluczowego restrict oraz funkcji wyrównującej dane. Korzystając z polecenia -fopt-info-vec-optimized, odpowiedzialnego za zwrócenie informacji na temat optymalizacji wektoryzacji, otrzymujemy następujący komunikat:

Program.cpp:7:17: note: loop vectorized

Oznacza to, że pętla została zwektoryzowana. Gdyby nie udało nam się tego osiągnąć polecenie nie zwróciłoby żadnego komunikatu.
Przy analizie kodu warto upewnić się, czy mechanizm został faktycznie wykorzystany kompilując kod źródłowy z naszego przykładu do kodu assemblerowego:

```
1: foo:
2:      movl 4(%esp), %edx
3:      movl 8(%esp), %ecx
4:      xorl %eax, %eax
5: .L2:
6:      vmovaps (%edx,%eax),  %xmm1
7:      vaddps (%ecx,%eax), %xmm1, %xmm0
8:      vmovaps %xmm0, (%edx,%eax)
9:      addl $16, %eax
10:     cmpl $400, %eax
11:     jne .L2
12: ret
```

Możemy tutaj zauważyć, iż w linijce numer 6 są przechowane spakowane liczby typu float, należąca do pierwszej tablicy w rejestrze %xmm1, w następnej linijce dodawane są do siebie liczby znajdujące się w drugiej tablicy do liczb w rejestrze z przechowywanymi liczbami z pierwszej tabeli, a wynik zostaje przechowany w rejestrze %xmm0. Zauważamy również, że wartość w rejestrze %eax, pełniącego rolę offsetu przy uzyskiwaniu liczb, zostaje zwiększony, umożliwiając powtarzanie operacji na kolejnych liczbach.
W powyższym przykładzie została wykorzystana instrukcja działająca na zbiorze liczb, co oznacza, że mechanizm wektoryzacji został poprawnie zastosowany.

3.4 Funkcje systemowe

Wewnętrzne funkcje matematyczne np. sin(), log(), exp(), sqrt() są dozwolone, ponieważ biblioteka wykonawcza (ang. runtime library) posiada zwektoryzowane wersje tych funkcji. Poniższa tabela przedstawia przykłady takich funkcji.

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	

Przykład:

```
1: #include <math.h>
2: #define SIZE 100
3: void sqrt_test(float *__restrict__ a, float *__restrict__ b)
4: {
5:         int i;
6:         float *x = __builtin_assume_aligned(a, 16);
7:         float *y = __builtin_assume_aligned(b, 16);
8:         for(i=0; i<SIZE; i++)
9:         {
10:                 x[i] = sqrtf(y[i]);
11:        }
12: }
```

Dla powyższego kodu otrzymujemy następujący kod assemblera:

```
1: sqrt_test:
2:         movl 4(%esp), %ecx
3:         movl 8(%esp), %edx
4:         vxorps %xmm2, %xmm2, %xmm2
5:         vmovaps .LC0, %xmm4
6:         vmovaps .LC1, %xmm3
7:         xorl %eax, %eax
```

```
8: .L2:
9:      vcmpneqps (%edx,%eax), %xmm2, %xmm1
10:     vrsqrtps (%edx,%eax), %xmm0
11:     vandps %xmm1, %xmm0, %xmm0
12:     vmulps (%edx,%eax), %xmm0, %xmm1
13:     vmulps %xmm0, %xmm1, %xmm0
14:     vmulps %xmm3, %xmm1, %xmm1
15:     vaddps %xmm4, %xmm0, %xmm0
16:     vmulps %xmm1, %xmm0, %xmm0
17:     vmovaps %xmm0, (%ecx,%eax)
18:     addl $16, %eax
19:     cmpl $400, %eax
20:     jne .L2
21: ret
22: .LC0:
23:     .long 3225419776
24:     .long 3225419776
25:     .long 3225419776
26:     .long 3225419776
27: .LC1:
28:     .long 3204448256
29:     .long 3204448256
30:     .long 3204448256
31:     .long 3204448256
```

Dokładnie w linii numer 10 widzimy, że zastosowana została instrukcja wyliczająca pierwiastek kwadratowy z czterech spakowanych liczb typu float, znajdujących się w obszarze pamięci reprezentowanym przez adres, znajdujący się w rejestrach %edx oraz %eax, następnie przechowuje wynik w rejestrze %xmm0. Oznacza to, że dla naszego kodu został zastosowany mechanizm wektoryzacji.

Wektoryzacji nie możemy natomiast uzyskać korzystając z funkcji rand (generatora pseudolosowego), której deklaracja wygląda w następujący sposób: srand(time(NULL)); Kompilator w momencie startu programu nie ma wystarczających informacji o pochodzeniu danej funkcji oraz jej przeznaczeniu.

Dla tablic dwuwymiarowych możemy wykonywać operacje tak jak na jednowymiarowych. Jesteśmy więc w stanie przeprowadzać działania matematyczne takie jak np. dodawanie.

3.5 Zmienne lokalne

Należy mieć na uwadzę, czy napisany przez nas kod jest użyteczny z punktu widzenia programu. Rozważmy prosty przykład sumowania liczb z tablicy:

```
1: #define SIZE 100
2: void foo(float* __restrict__ a)
3: {
4:        float c = 0;
5:        int i;
6:        for(i = 0; i < SIZE; ++i)
7:        {
8:                  c+=a[i];
9:         }
10:}
```

Pętla nie zostaje zwektoryzowana. Warto spojrzeć na kod assemblera:

```
foo:
ret
```

Nie został wygenerowany żaden kod. Dzieje się tak, ponieważ zasięg życia tablicy c[] ogranicza się do ciała funkcji; kod wywoływany w funkcji nie ma żadnego wpływu na resztę programu, więc kompilator stwierdza, iż jest on zbędny, czego wynikiem jest brak wygenerowania kodu assemblera, a co za tym idzie, brak wektoryzacji. Z kolei jeżeli kod funkcji będzie miał zastosowanie, np. będzie zwracał liczbę c, wtedy mechanizm zadziała poprawnie:

```
1: #define SIZE 100
2: float foo(float* __restrict__ a)
3: {
4:        float c = 0;
5:        int i;
6:        for(i = 0; i < SIZE; ++i)
7:        {
8:                  c+=a[i];
9:        }
10:       return c;
11: }
```

Dla powyższej funkcji otrzymujemy następujący kod assemblera:

```
1: foo:
2:        subl $28, %esp
```

```
3:      vxorps %xmm0, %xmm0, %xmm0
4:      movl 32(%esp), %eax
5:      leal 400(%eax), %edx
6: .L2:
7:      vaddps (%eax), %xmm0, %xmm0
8:      addl $16, %eax
9:      cmpl %eax, %edx
10:     jne .L2
11:     vhaddps %xmm0, %xmm0, %xmm0
12:     vhaddps %xmm0, %xmm0, %xmm0
13:     vmovss %xmm0, 12(%esp)
14:     flds 12(%esp)
15:     addl $28, %esp
16: ret
```

Kod assemblera został poprawnie wygenerowany. W linii numer siedem została zastosowana wektorowa instrukcja dodawania zestawów liczb, co oznacza, że wektoryzacja została poprawnie wykonana. Warto jednak zaznaczyć, że gdybyśmy jedynie kopiowali tablicę (c[i]=a[i]) zamiast ją modyfikować, mimo zwracania tablicy, wektoryzacja nie nastąpi.

4. Instrukcje warunkowe

4.1 Warunkowe przerwanie pętli

Nie można zwektoryzować warunkowego wyjścia z pętli (zgodnie z zasadą Single-Entry-Single-Exit. Przykład:

```
1: #define SIZE 100
2: void foo(int* __restrict__ a, int* __restrict__ b)
3: {
4:      int i;
5:      for(i = 0; i < SIZE; ++i)
6:      {
7:              if(a[i] == b[i]) break; //Przerwanie pętli
8:              a[i] += b[i];
9:      }
10: }
```

Wygenerowany zostanie następujący kod assemblera:

```
1: foo:
2:      pushl %esi
3:      xorl %eax, %eax
4:      pushl %ebx
5:      movl 12(%esp), %ebx
6:      movl 16(%esp), %esi
7:      jmp .L3
8: .L8:
9:      addl %ecx, %edx
10:     movl %edx, (%ebx,%eax,4)
11:     incl %eax
12:     cmpl $100, %eax
13:     je .L5
14: .L3:
15:     movl (%ebx,%eax,4), %edx
16:     movl (%esi,%eax,4), %ecx
17:     cmpl %ecx, %edx
18:     jne .L8
19: .L5:
20:     popl %ebx
21:     popl %esi
22: ret
```

Od linii 15 do 17 sprawdzany jest warunek wyjścia z pętli, jednakże można zauważyć, iż w całym kodzie nie została zastosowana ani jedna instrukcja wektorowa.

4.2 Operacje warunkowe

Nie jest możliwe wykonywanie operacji za pomocą instrukcji warunkowej if. Przykład:

```
1: #define SIZE 100
2: void foo(int* __restrict__ a, int* __restrict__ b)
3: {
4:      int i;
5:      for(i = 0; i < SIZE; ++i)
6:      {
7:              if(a[i] < b[i]) a[i] += b[i];
8:      }
9: }
```

Otrzymujemy następujący kod assemblera:

```
1: foo:
2:      pushl %esi
3:      pushl %ebx
4:      xorl %eax, %eax
5:      movl 12(%esp), %ebx
6:      movl 16(%esp), %esi
7: .L3:
8:      movl (%ebx,%eax,4), %edx
9:      movl (%esi,%eax,4), %ecx
10:     cmpl %ecx, %edx
11:     jge .L2
12:     addl %ecx, %edx
13:     movl %edx, (%ebx,%eax,4)
14: .L2:
15:     incl %eax
16:     cmpl $100, %eax
17:     jne .L3
18:     popl %ebx
19:     popl %esi
20: ret
```

Widzimy, że wektoryzacja nie nastąpiła. Istnieje rozwiązanie. Wyrażenie warunkowe ?: może zostać zwektoryzowane. Dla tego samego przykładu:

```
1: #define SIZE 100
2: void foo(int* __restrict__ a, int* __restrict__ b)
3: {
4:      float *x = (float*)__builtin_assume_aligned(a, 16);
5:      float *y = (float*)__builtin_assume_aligned(b, 16);
6:
7:      int i;
8:      for(i = 0; i < SIZE; ++i)
9:      {
10:             x[i] += x[i] < y[i] ? y[i] : 0;
11:     }
12: }
```

Jeżeli spojrzymy na kod assemblera:

```
1: foo:
2:      movl 4(%esp), %edx
3:       movl 8(%esp), %ecx
4:      xorl %eax, %eax
5: .L2:
6:      vmovaps (%edx,%eax), %xmm2
7:      vmovaps (%ecx,%eax), %xmm3
8:      vaddps (%ecx,%eax), %xmm2, %xmm1
9:      vcmpleps %xmm2, %xmm3, %xmm0
10:     vpcmov %xmm0, %xmm1, %xmm2, %xmm0
11:     vmovaps %xmm0, (%edx,%eax)
12:     addl $16, %eax
13:     cmpl $400, %eax
14:     jne .L2
15: ret
```

Widzimy, że zastosowane zostały instrukcje wektorowe, a więc wektoryzacja przebiegła pomyślnie.

4.3 Instrukcja wyboru switch

Nie da się zwektoryzować instrukcji switch w żadnej jego postaci. Przykład:

```
1:  #define SIZE 100
2: void switch_test(float* __restrict__ a, float* __restrict__ b)
3: {
4:      int i, z;
5:
6:      for(i=0; i<SIZE; i++)
7:      {
8:              z = i % 2;
9:              switch (z)
10:             {
11:                     case 0: { a[i] -= b[i]; break; }
12:                     case 1: { a[i]+=b[i]; break; }
13:                     default: return;
14: }
15:             }
16: }
```

Otrzymamy następujący kod assemblera:

```
1: switch_test:
2:      movl 4(%esp), %edx
3:      movl 8(%esp), %ecx
4:      xorl %eax, %eax
5: .L5:
6:      vmovss (%edx,%eax,4), %xmm0
7:      vmovss (%ecx,%eax,4), %xmm1
8:      testb $1, %al
9: je .L8
10:     vaddss %xmm1, %xmm0, %xmm0
11:     vmovss %xmm0, (%edx,%eax,4)
12: .L4:
13:     incl %eax
14:     cmpl $100, %eax
15:     jne .L5
16: ret
17: .L8:
18:     vsubss %xmm1, %xmm0, %xmm0
19:     vmovss %xmm0, (%edx,%eax,4)
20:     jmp .L4
```

Widzimy, że mimo pojawiających się rejestrów %xmm wektoryzacja nie następuje. Możemy zauważyć, iż do działań na rejestrach %xmm wykorzystane zostały instrukcje, działające na pojedynczych wartościach (VADDSS) zamiast na spakowany co oznacza, iż wektoryzacja nie nastąpiła.

5. Pętle

5.1 Nieznana długość pętli

Mechanizm autowektoryzacji działa dla nieznanej ilości iteracji (np. podanej jako argument funkcji):

```
1: void foo(int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int* __restrict__
rozmiar)
2: {
3:      int i;
```

```
4:      float *x =(float*) __builtin_assume_aligned(a, 16);
5:      float *y =(float*)__builtin_assume_aligned(b, 16);
6:      for (i = 0; i < *rozmiar; i++)
7:      {
8:              x[i]+=y[i];
9:      }
10: }
```

Otrzymujemy następujący kod assemblera:

```
1: foo:
2:      pushl %ebp
3:      pushl %edi
4:      pushl %esi
5:      pushl %ebx
6:      movl 32(%esp), %eax
7:      movl 24(%esp), %ebp
8:      movl (%eax), %edi
9:      testl %edi, %edi
10:     jle .L12
11:     leal -1(%edi), %eax
12:     cmpl $2, %eax
13:     jbe .L8
14:     movl %edi, %esi
15:     shrl $2, %esi
16:     cmpl $4, %esi
17:     jbe .L9
18:     movl 20(%esp), %eax
19:     leal -5(%esi), %ebx
20:     movl %ebp, %edx
21:     andl $-4, %ebx
22:     xorl %ecx, %ecx
23:     addl $4, %ebx
24: .L5:
25:     vmovaps (%eax), %xmm2
26:     vmovaps 16(%eax), %xmm3
27:     prefetcht0 320(%edx)
28:     addl $4, %ecx
29:     vmovaps 32(%eax), %xmm4
```

```
30:     vmovaps 48(%eax), %xmm5
31:     prefetcht0 320(%eax)
32:     addl $64, %edx
33:     addl $64, %eax
34:     vaddps -64(%edx), %xmm2, %xmm0
35:     vmovaps %xmm0, -64(%eax)
36:     vaddps -48(%edx), %xmm3, %xmm0
37:     vmovaps %xmm0, -48(%eax)
38:     vaddps -32(%edx), %xmm4, %xmm0
39:     vmovaps %xmm0, -32(%eax)
40:     vaddps -16(%edx), %xmm5, %xmm0
41:     vmovaps %xmm0, -16(%eax)
42:     cmpl %ebx, %ecx
43:     jne .L5
44: .L4:
45:     xorl %ecx, %ecx
46: .L6:
47:     vmovaps (%eax,%ecx), %xmm1
48:     incl %ebx
49:     vaddps (%edx,%ecx), %xmm1, %xmm0
50:     vmovaps %xmm0, (%eax,%ecx)
51:     addl $16, %ecx
52:     cmpl %ebx, %esi
53:     ja .L6
54:     movl %edi, %eax
55:     andl $-4, %eax
56:     cmpl %eax, %edi
57:     je .L12
58: .L3:
59:     movl 20(%esp), %ecx
60:     leal 0(,%eax,4), %edx
61:     addl %edx, %ecx
62:     vmovss (%ecx), %xmm0
63:     vaddss 0(%ebp,%edx), %xmm0, %xmm0
64:     vmovss %xmm0, (%ecx)
65:     leal 1(%eax), %ecx
66:     cmpl %ecx, %edi
67:     jle .L12
```

```
68:     movl 20(%esp), %ecx
69:     leal 4(%edx), %ebx
70:     addl $2, %eax
71:     addl %ebx, %ecx
72:     vmovss (%ecx), %xmm0
73:     vaddss 4(%ebp,%edx), %xmm0, %xmm0
74:     vmovss %xmm0, (%ecx)
75:     cmpl %eax, %edi
76:     jle .L12
77:     movl 20(%esp), %eax
78:     addl $8, %edx
79:     addl %edx, %eax
80:     vmovss (%eax), %xmm0
81:     vaddss 0(%ebp,%edx), %xmm0, %xmm0
82:     vmovss %xmm0, (%eax)
83: .L12:
84:     popl %ebx
85:     popl %esi
86:     popl %edi
87:     popl %ebp
88: ret
89: .L8:
90:     xorl %eax, %eax
91:     jmp .L3
92: .L9:
93:     movl %ebp, %edx
94:     movl 20(%esp), %eax
95:     xorl %ebx, %ebx
96:     jmp .L4
```

Wygenerowano bardzo dużo kodu, jednakże można zauważyć, że większość kodu odnosi się do konstrukcji pętli, z naszego punktu widzenia najistotniejsze są linijki od 34 do 41, gdzie możemy zaobserwować wykonywanie kopiowania spakowanych liczb.

5.2 Zależność od pętli wstecznej

Pętla nie może wymagać wykonania drugiej instrukcji pierwszej iteracji przed pierwszą instrukcją drugiej iteracji.
Przykład wektoryzacji (a[i-1] jest zawsze obliczone, nim zostanie użyte):

```
1:  #define SIZE 100
2: void no_backward_loop_carried_dependencies(float* __restrict__ a, float* __restrict__ b,
float* __restrict__ c, float* __restrict__ d, int* __restrict__ e)
3: {
4:        int i;
5:        float *x = (float*)__builtin_assume_aligned(a, 16);
6:        float *y = (float*)__builtin_assume_aligned(b, 16);
7:        float *z = (float*)__builtin_assume_aligned(c, 16);
8:        float *m = (float*)__builtin_assume_aligned(d, 16);
9:        float *n = (float*)__builtin_assume_aligned(e, 16);
10:       for (i=1; i<SIZE; i++)
11:       {
12:               x[i] = y[i] + z[i];
13:               m[i] = n[i] - x[i-1];
14:       }
15: }
```

Otrzymujemy następujący kod assemblera:

```
1: no_backward_loop_carried_dependencies:
2:        pushl %edi
3:        pushl %esi
4:        movl $4, %eax
5:        pushl %ebx
6:        movl 16(%esp), %edx
7:        movl 20(%esp), %edi
8:        movl 24(%esp), %esi
9:        movl 28(%esp), %ecx
10:       movl 32(%esp), %ebx
11: .L2:
12:       vmovups (%edi,%eax), %xmm3
13:       vmovups (%ebx,%eax), %xmm4
14:       vaddps (%esi,%eax), %xmm3, %xmm0
15:       vmovups %xmm0, (%edx,%eax)
16:       vsubps -4(%edx,%eax), %xmm4, %xmm0
17:       vmovups %xmm0, (%ecx,%eax)
18:       addl $16, %eax
19:       cmpl $388, %eax
20:       jne .L2
```

```
21:     vmovss 388(%esi), %xmm1
22:     vmovss 388(%ebx), %xmm0
23:     vmovss 392(%ebx), %xmm2
24:     vaddss 388(%edi), %xmm1, %xmm1
25:     vsubss 384(%edx), %xmm0, %xmm0
26:     vmovss %xmm1, 388(%edx)
27:     vsubss %xmm1, %xmm2, %xmm1
28:     vmovss %xmm0, 388(%ecx)
29:     vmovss 392(%edi), %xmm0
30:     vaddss 392(%esi), %xmm0, %xmm0
31:     vmovss %xmm1, 392(%ecx)
32:     vmovss 396(%edi), %xmm1
33:     vaddss 396(%esi), %xmm1, %xmm1
34:     vmovss %xmm0, 392(%edx)
35:     vmovss %xmm1, 396(%edx)
36:     vmovss 396(%ebx), %xmm1
37:     vsubss %xmm0, %xmm1, %xmm0
38:     vmovss %xmm0, 396(%ecx)
39:     popl %ebx
40:     popl %esi
41:     popl %edi
42: ret
```

Można zauważyć, że zastosowane zostały instrukcje wektorowe, więc kod został poprawnie zwektoryzowany.
Przykład, gdzie nie wektoryzuje (a[i-1] może być potrzebne, zanim zostanie obliczone):

```
1:  #define SIZE 100
2: void no_backward_loop_carried_dependencies(float* __restrict__ a, float* __restrict__ b,
float* __restrict__ c, float* __restrict__ d, int* __restrict__ e)
3: {
4:      int i;
5:      float *x = (float*)__builtin_assume_aligned(a, 16);
6:      float *y = (float*)__builtin_assume_aligned(b, 16);
7:      float *z = (float*)__builtin_assume_aligned(c, 16);
8:      float *m = (float*)__builtin_assume_aligned(d, 16);
9:      float *n = (float*)__builtin_assume_aligned(e, 16);
10:     for (i=1; i<SIZE; i++)
```

```
11:     {
12:             m[i] = n[i] + x[i-1];
13:             x[i] = y[i] - z[i];
14:     }
15: }
```

Otrzymujemy następujący kod assemblera:

```
1: no_backward_loop_carried_dependencies:
2:      pushl %edi
3:      pushl %esi
4:      movl $1, %eax
5:      pushl %ebx
6:      movl 16(%esp), %edx
7:      movl 20(%esp), %edi
8:      movl 24(%esp), %esi
9:      movl 28(%esp), %ebx
10:     movl 32(%esp), %ecx
11:     vmovss (%edx), %xmm0
12: .L2:
13:     vaddss (%ecx,%eax,4), %xmm0, %xmm0
14:     vmovss %xmm0, (%ebx,%eax,4)
15:     vmovss (%edi,%eax,4), %xmm0
16:     vsubss (%esi,%eax,4), %xmm0, %xmm0
17:     vmovss %xmm0, (%edx,%eax,4)
18:     incl %eax
19:     cmpl $100, %eax
20:     jne .L2
21:     popl %ebx
22:     popl %esi
23:     popl %edi
24: ret
```

Widzimy, że mimo działania na rejestrach wektorowych, instrukcje działają na pojedynczych liczbach, oznacza to, że wektoryzacja nie zaszła.

5.3 Przechodzenie pętli od tyłu.

Przykład:

```
1: #define SIZE 100
2: void no_backward_loop_carried_dependencies(float* __restrict__ a, float* __restrict__ b)
3: {
4: float *x =(float*) __builtin_assume_aligned(a, 16);
5:      float *y =(float*)__builtin_assume_aligned(b, 16);
6:      int i;
7:      for (i = 0; i < SIZE; i++)
8:      {
9:              x[SIZE-i-1] += y[SIZE-i-1];
10:     }
11: }
```

Wygenerowany zostanie następujący kod assemblera:

```
1: no_backward_loop_carried_dependencies:
2:      movl 4(%esp), %edx
3:      movl 8(%esp), %ecx
4:       movl $384, %eax
5: .L2:
6:      vpermilps $27, (%edx,%eax), %xmm1
7:      vpermilps $27, (%ecx,%eax), %xmm0
8:      vaddps %xmm1, %xmm0, %xmm0
9:      vpermilps $27, %xmm0, %xmm0
10:     vmovaps %xmm0, (%edx,%eax)
11:     subl $16, %eax
12:     cmpl $-16, %eax
13:     jne .L2
14: ret
```

W tym przypadku wektoryzacja zachodzi. Możemy zauważyć, że instrukcja vpermilps, działającą w podobny sposób do instrukcji vshufps, która reorganizuje dane w tablicy przed wykonaniem działania.

6. Algorytmy

Sprawdzenie wektoryzacji dla kilku użytecznych algorytmów bez użycia standardowej biblioteki szablonów STL. Jedyną instrukcją, która nie została napisana ręcznie jest std::swap, którego składnia wygląda następująco:

```
#include <algorithm>
namespace std
{
 template < typename T >
 void swap( T & a, T & b );
}
```

Funkcja ta przypisuje wartość obiektu a obiektowi b i wartość obiektu b obiektowi a.

6.1 Najmniejszy element w tablicy

```
1: #include<iostream>
2: #include<cstdlib>
3:
4: #define SIZE 100
5: void min_element(float * __restrict__ a)
6: {
7:         float min;
8:         int i;
9:         float *tab = (float*)__builtin_assume_aligned(a, 16);
10:        min = tab[0]; //pierwszą liczbę przypisujemy do zmiennej min
11:        for(i=1;i<SIZE;i++)
12:                 if(min>tab[i])
13:                 min = tab[i];
14:        std::cout<<"Najmniejszą liczbą jest "<<min<<std::endl;
15:
16: }
17: int main()
18: {
19:        return 0;
20: }
```

Czy wektoryzuje: tak.
Komunikaty gcc:
loop at find_min.cpp:12: if (ivtmp_49 >= 24)
find_min.cpp:11:11: note: LOOP VECTORIZED

6.2 Największy element w tablicy

```
1: #include<iostream>
2: #include<cstdlib>
3:
4: #define SIZE 100
5: void max_element(float * __restrict__ a)
6: {
7:    float max;
8:    int i;
9:    float *tab = (float*)__builtin_assume_aligned(a, 16);
10:   max = tab[0]; //pierwszą liczbę przypisujemy do zmiennej max
11:   for(i=1;i<SIZE;i++) //przeszukanie pozostałych 9 liczb
12:      if(max<tab[i])
13:         max = tab[i];
14:   std::cout<<"Najwieksza wczytaną liczbą jest "<<max<<std::endl;
15:
16: }
17: int main()
18: {
19:       return 0;
20: }
```

Czy wektoryzuje: tak.
Komunikaty gcc:
loop at find_max.cpp:12: if (ivtmp_49 >= 24)
find_max.cpp:11:11: note: LOOP VECTORIZED

6.3 Sortowanie przez scalanie

```
1: #include <iostream>
2: #include <cstdlib>
3: #define SIZE 100
4: //scalenie posortowanych podtablic
5: void scal(float *tab, int lewy, int srodek, int prawy, float *pom)
6: {
7:        int i = lewy, j = srodek + 1;
8:        int k;
9:        //kopiujemy lewą i prawą część tablicy do tablicy pomocniczej
10:       for (i = lewy; i <= prawy; i++)
11:                pom[i] = tab[i];
12:
13:       //scalenie dwóch podtablic pomocniczych i zapisanie ich
14:       //we własciwej tablicy
```

```
15:     for (k = lewy; k <= prawy; k++)
16:             if (i <= srodek)
17:                     if (j <= prawy)
18:                             if (pom[j] < pom[i])
19:                             tab[k] = pom[j++];
20:                             else
21:                             tab[k] = pom[i++];
22:                     else
23:                     tab[k] = pom[i++];
24:             else
25:             tab[k] = pom[j++];
26: }
27:
28: void sortowanie_przez_scalanie(float *tab, int lewy, int prawy, float *pom)
29: {
30:     //gdy mamy jeden element, to jest on już posortowany
31:     if (prawy <= lewy)
32:     return;
33:
34:     //znajdujemy srodek podtablicy
35:     int srodek = (prawy + lewy) / 2;
36:
37:     //dzielimy tablice na część lewą i prawa
38:     sortowanie_przez_scalanie(tab, lewy, srodek, pom);
39:     sortowanie_przez_scalanie(tab, srodek + 1, prawy, pom);
40:
41:     //scalamy dwie już posortowane tablice
42:     scal(tab, lewy, srodek, prawy, pom);
43: }
44:
45: void sort_scalanie(float *__restrict__ a, float *__restrict__ b)
46: {
47:     int i;
48:     float *tab = (float *)__builtin_assume_aligned(a, 16);
49:     float *pom = (float *)__builtin_assume_aligned(b, 16);
50:
51:     sortowanie_przez_scalanie(tab, 0, SIZE - 1, pom);
52:
53:     //wypisanie wyników
54:     for (i = 0; i < SIZE; i++)
55:     std::cout << tab[i] << " ";
56:
57: int main()
58: {
59:     return 0;
60: }
```

Czy wektoryzuje: tak, zwektoryzowało jedną pętle.
Komunikaty gcc:
loop at sortowanie.cpp:11: if (ivtmp_227 >= bnd.156_209)
sortowanie.cpp:10:17: note: LOOP VECTORIZED
sortowanie.cpp:28:6: note: vectorized 1 loops in function.

6.4 Sortowanie przez wstawianie

```
1: // Sortowanie przez wstawianie
2: #include<cstdio>
3: #define SIZE 100
4:
5: void sort_wstawianie(float * __restrict__ a)
6: {
7:          float *tab = (float*)__builtin_assume_aligned(a, 16);
8:         float pom;
9:
10:        int i, j;
11:        for (i = 1; i < SIZE; i++)
12:        {
13:                  //wstawienie elementu w odpowiednie miejsce
14:                  pom = tab[i]; //ten element będzie wstawiony w odpowiednie miejsce
15:                  j = i - 1;
16:
17:                  //przesuwanie elementów większych od pom
18:                  while (j >= 0 && tab[j]>pom)
19:                  {
20:                            tab[j + 1] = tab[j]; //przesuwanie elementów
21:                            --j;
22:                  }
23:                  tab[j + 1] = pom; //wstawienie pom w odpowiednie miejsce
24:        }
25: }
26: int main()
27: {
28:        return 0;
29: }
```

Czy wektoryzuje: nie.
Komunikaty gcc:
Analyzing loop at sortowanie.cpp:11
sortowanie.cpp:11:16: note: ===== analyze_loop_nest =====
sortowanie.cpp:11:16: note: === vect_analyze_loop_form ===
sortowanie.cpp:11:16: note: not vectorized: control flow in loop.

sortowanie.cpp:11:16: note: bad loop form.

Analyzing loop at sortowanie.cpp:18
sortowanie.cpp:18:25: note: ===== analyze_loop_nest =====
sortowanie.cpp:18:25: note: === vect_analyze_loop_form ===
sortowanie.cpp:18:25: note: not vectorized: control flow in loop.
sortowanie.cpp:18:25: note: bad loop form.
sortowanie.cpp:5:5: note: vectorized 0 loops in function.

6.5 Sortowanie bąbelkowe

```
1: // Sortowanie babelkowe
2: #include <cstdlib>
3: #include <algorithm>
4: #define SIZE 100
5: void sort_babelkowe(float * __restrict__ a)
6: {
7:         int i,j;
8:         float *x =(float*) __builtin_assume_aligned(a, 16);
9:         for (i = 0; i < SIZE; i++)
10:                 for (j = 1; j < SIZE - i; j++) //pętla wewnętrzna
11:                 {
12:                         if (x[j - 1]>x[j])
13:                          std::swap(x[j-1], x[j]);
14:                 }
15: }
16: int main()
17: {
18:       return 0;
19: }
```

Czy wektoryzuje: nie.
Komunikaty:
Analyzing loop at sortowanie.cpp:9
sortowanie.cpp:9:16: note: ===== analyze_loop_nest =====
sortowanie.cpp:9:16: note: === vect_analyze_loop_form ===
sortowanie.cpp:9:16: note: not vectorized: multiple nested loops.
sortowanie.cpp:9:16: note: bad loop form.
…
sortowanie.cpp:10:17: note: === vect_analyze_scalar_cycles ===
sortowanie.cpp:10:17: note: Analyze phi: j_12 = PHI <j_18(7), 1(5)>
sortowanie.cpp:10:17: note: Access function of PHI: {1, +, 1}_2
sortowanie.cpp:10:17: note: step: 1, init: 1
sortowanie.cpp:10:17: note: Detected induction
...

sortowanie.cpp:10:17: note: dependence distance = 0.
sortowanie.cpp:10:17: note: dependence distance == 0 between *_5 and *_5
sortowanie.cpp:10:17: note: dependence distance = 1.
sortowanie.cpp:10:17: note: not vectorized, possible dependence between data-refs *_5 and *_8
sortowanie.cpp:10:17: note: bad data dependence.
sortowanie.cpp:5:5: note: vectorized 0 loops in function.

6.6 Sortowanie szybkie (ang. quicksort)

```
1: #include<iostream>
2: #define SIZE 100
3: void quick_sort(float *tab, int lewy, int prawy)
4: {
5:         if(prawy <= lewy) return;
6:         int i = lewy - 1, j = prawy + 1,
7:         pivot = tab[(lewy+prawy)/2]; //wybieramy punkt odniesienia
8:
9:         while(1)
10:     {
11:       //szukam elementu wiekszego lub rownego piwot stojacego
12:       //po prawej stronie wartosci pivot
13:       while(pivot>tab[++i]);
14:        //szukam elementu mniejszego lub rownego pivot stojacego
15:        //po lewej stronie wartosci pivot
16:                 while(pivot<tab[--j]);
17:                         if( i <= j)
18:                         std::swap(tab[i],tab[j]);
19:                         else
20:                         break;
21:     }
22:
23:       if(j > lewy)
24:       quick_sort(tab, lewy, j);
25:       if(i < prawy)
26:       quick_sort(tab, i, prawy);
27: }
28: void sort_quick_test(float * __restrict__ a)
29: {
30:     int i;
31:     float *tab = (float*)__builtin_assume_aligned(a, 16);
32:   quick_sort(tab,0, SIZE-1);
33:   //wypisanie posortowanych elementów
34:   for(int i=0;i<SIZE;i++)
35:   std::cout<<tab[i]<<" ";
```

```
36:   return 0;
37: }
38: int main()
39: {
40:       return 0;
41: }
```

Czy wektoryzuje: nie.
Komunikaty gcc:
sortowanie.cpp:16:14: note: ===== analyze_loop_nest =====
sortowanie.cpp:16:14: note: === vect_analyze_loop_form ===
sortowanie.cpp:16:14: note: === get_loop_niters ===
sortowanie.cpp:16:14: note: not vectorized: number of iterations cannot be computed.
sortowanie.cpp:16:14: note: bad loop form.

7. STL

Standardowa biblioteka szablonów (STL) jest biblioteką C++ zawierającą algorytmy, kontenery, iteratory oraz inne konstrukcje w formie szablonów, gotowe do użycia w programach. Poniższe przykłady sprawdzają możliwości wektoryzacji paru przydatnych algorytmów, operując na tablicy dynamicznej oraz liście dwukierunkowej.

7.1 Min_element

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: int main()
6: {
7:       std::vector<int> v{ 3, 1, -14, 143, 5, 9423 };
8:       std::vector<int>::iterator result;
9:       result = std::min_element(v.begin(), v.end());
10:      std::cout << "min element at: " << std::distance(v.begin(), result) << '\n';
11: }
```

Wektoryzacja: nieudana
Komunikaty gcc:

/usr/include/c++/7/bits/stl_heap.h:405:31: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_heap.h:405:31: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: not vectorized: multiple nested loops.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: not vectorized: control flow in loop.

/usr/include/c++/7/bits/stl_algo.h:1907:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: bad loop form.

Fragment pliku nagłówkowego stl_heap.h:

```
400:  template<typename _RandomAccessIterator, typename _Compare>
401:    void
402:    __sort_heap(_RandomAccessIterator __first, _RandomAccessIterator __last,
403:     _Compare& __comp)
404:    {
405:    while (__last - __first > 1)
406:   {
407:     --__last;
408:     std::__pop_heap(__first, __last, __last, __comp);
409:   }
410:    }
```

Kompilator gcc informuje o tym, że nie może obliczyć ilości iteracji pętli while (linijka 405).

Fragment pliku nagłówkowego stl_algo.h:

```
1893: /// This is a helper function...
1894:  template<typename _RandomAccessIterator, typename _Compare>
1895:   _RandomAccessIterator
1896:   __unguarded_partition(_RandomAccessIterator __first,
1897:                 _RandomAccessIterator __last,
1898:                 _RandomAccessIterator __pivot, _Compare __comp)
1899:   {
1900:   while (true)
1901:    {
1902:     while (__comp(__first, __pivot))
1903:           ++__first;
1904:     --__last;
1905:     while (__comp(__pivot, __last))
1906:           --__last;
1907:     if (!(__first < __last))
1908:           return __first;
1909:     std::iter_swap(__first, __last);
1910:     ++__first;
1911:    }
1912:   }
```

Kompilator gcc informuje o tym, że nie może obliczyć ilości iteracji pętli while (linijka 1905) orazwystępuje kontrola przepływu, wskazując na linijkę 1907.

7.2 Max_element

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: int main()
6: {
7:         std::vector<int> v{ 3, 1, -14, 143, 5, 9423 };
8:         std::vector<int>::iterator result;
9:         result = std::max_element(v.begin(), v.end());
10:        std::cout << "max element at: " << std::distance(v.begin(), result) << '\n';
11: }
```

Wektoryzacja: nieudana
Komunikaty gcc:

```
Analyzing loop at /usr/include/c++/7/bits/stl_algo.h:5652
/usr/include/c++/7/bits/stl_algo.h:5652:24: note: ===== analyze_loop_nest =====
/usr/include/c++/7/bits/stl_algo.h:5652:24: note: === vect_analyze_loop_form ===
/usr/include/c++/7/bits/stl_algo.h:5652:24: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_algo.h:5652:24: note: bad loop form.
stl.cpp:28:5: note: vectorized 0 loops in function.
```

Fragment pliku nagłówkowego stl_algo.h:

```
5644:   template<typename _ForwardIterator, typename _Compare>
5645:     _GLIBCXX14_CONSTEXPR
5646:     _ForwardIterator
5647:     __max_element(_ForwardIterator __first, _ForwardIterator __last,
5648:                   _Compare __comp)
5649:     {
5650:       if (__first == __last) return __first;
5651:       _ForwardIterator __result = __first;
5652:       while (++__first !=__last)
5653:     if (__comp(__result, __first))
5654:       __result = __first;
5655:       return __result;
5656:     }
```

GCC informuje o kontroli przepływu wskazując na linijkę 5652. Używany iterator korzysta z pętli while, która nie ma określonej liczby iteracji ale ma za to warunek kończący jej wykonywanie.

7.3 Sortowanie

Sortowanie z użyciem domyślnego operatora:

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: #include <array>
6: int main()
7: {
8: std::array<int, 10> s = { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
9:
10:     // sort using the default operator<
11:     std::sort(s.begin(), s.end());
12:     for (auto a : s) {
13:     std::cout << a << " ";
14:     }
15:     std::cout << '\n';
16: }
```

Wektoryzacja: nieudana

Komunikaty gcc:

/usr/include/c++/7/bits/stl_heap.h:405:31: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_heap.h:405:31: note: bad loop form.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: not vectorized: latch block not empty.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: not vectorized: multiple nested loops.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: bad loop form.

Fragmenty kodu pliku nagłówkowego stl_algo.h

```
1893:   /// This is a helper function...
1894:   template<typename _RandomAccessIterator, typename _Compare>
1895:    _RandomAccessIterator
1896:    __unguarded_partition(_RandomAccessIterator __first,
1897:                          _RandomAccessIterator __last,
1898:                          _RandomAccessIterator __pivot, _Compare __comp)
1899:   {
1900:   while (true)
1901:    {
1902:      while (__comp(__first, __pivot))
1903:              ++__first;
1904:      --__last;
1905:      while (__comp(__pivot, __last))
```

```
1906:             --__last;
1907:     if (!(__first < __last))
1908:             return __first;
1909:     std::iter_swap(__first, __last);
1910:     ++__first;
1911:   }
1912:   }
```

Kompilator gcc informuje, że nie może zostać obliczona ilość iteracji pętli (linijka 1905), co jest charakterystyczne dla pętli while. Wskazując na linijkę 1907, zaznacza kontrolę przepływu uniemożliwiając wektoryzację.

```
1937:   /// This is a helper function for the sort routine.
1938:   template<typename _RandomAccessIterator, typename _Size, typename _Compare>
1939:   void
1940:   __introsort_loop(_RandomAccessIterator __first,
1941:                    _RandomAccessIterator __last,
1942:                    _Size __depth_limit, _Compare __comp)
1943:   {
1944:   while (__last - __first > int(_S_threshold))
1945:    {
1946:     if (__depth_limit == 0)
1947:           {
1948:           std::__partial_sort(__first, __last, __last, __comp);
1949:           return;
1950:           }
1951:     --__depth_limit;
1952:     _RandomAccessIterator __cut =
1953:           std::__unguarded_partition_pivot(__first, __last, __comp);
1954:     std::__introsort_loop(__cut, __last, __depth_limit, __comp);
1955:     __last = __cut;
1956:    }
1957:   }
```

GCC informuje o wielokrotnym zagnieżdżeniu pętli wskazując na linijkę 1951.

Fragment kodu pliku nagłówkowego stl_heap.h:

```
324:   template<typename _RandomAccessIterator, typename _Compare>
325:     void
326:     __make_heap(_RandomAccessIterator __first, _RandomAccessIterator __last,
327:      _Compare& __comp)
328:     {
329:     typedef typename iterator_traits<_RandomAccessIterator>::value_type
330:      _ValueType;
331:     typedef typename iterator_traits<_RandomAccessIterator>::difference_type
```

```
332:      _DistanceType;
333:
334:     if (__last - __first < 2)
335:    return;
336:
337:     const _DistanceType __len = __last - __first;
338:     _DistanceType __parent = (__len - 2) / 2;
339:     while (true)
340:     {
341:      _ValueType __value = _GLIBCXX_MOVE(*(__first + __parent));
342:      std::__adjust_heap(__first, __parent, __len, _GLIBCXX_MOVE(__value),
343:                         __comp);
344:      if (__parent == 0)
345:     return;
346:      __parent--;
347:    }
348:     }
...
400:   template<typename _RandomAccessIterator, typename _Compare>
401:     void
402:     __sort_heap(_RandomAccessIterator __first, _RandomAccessIterator __last,
403:      _Compare& __comp)
404:     {
405:     while (__last - __first > 1)
406:    {
407:      --__last;
408:      std::__pop_heap(__first, __last, __last, __comp);
409:    }
410:     }
```

Kompilator gcc wskazuje na linijkę 344 informując, że blokada zatrzasku nie jest pusta (ang. latch block not empty).

Sortowanie tablicy przy użyciu obiektu funkcji porównywania:

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: #include <array>
6:
7: void test2()
8: {
9: std::array<int, 10> s = { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
10:
11: std::sort(s.begin(), s.end(), std::greater<int>());
12:       for (auto a : s) {
```

```
13:       std::cout << a << " ";
14:       }
15:       std::cout << '\n';
16: }
```

Wektoryzacja: nieudana

Komunikaty gcc:

/usr/include/c++/7/bits/stl_heap.h:135:14: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_heap.h:135:14: note: bad loop form.
/usr/include/c++/7/bits/stl_heap.h:219:28: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_heap.h:219:28: note: bad loop form.

Fragment kodu pliku nagłówkowego stl_heap.h:

```
122:   // Heap-manipulation functions: push_heap, pop_heap, make_heap, sort_heap,
123:   // + is_heap and is_heap_until in C++0x.
124:
125:   template<typename _RandomAccessIterator, typename _Distance, typename _Tp,
126:     typename _Compare>
127:     void
128:     __push_heap(_RandomAccessIterator __first,
129:       _Distance __holeIndex, _Distance __topIndex, _Tp __value,
130:       _Compare& __comp)
131:     {
132:     _Distance __parent = (__holeIndex - 1) / 2;
133:     while (__holeIndex > __topIndex && __comp(__first + __parent, __value))
134:     {
135:       *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first + __parent));
136:       __holeIndex = __parent;
137:       __parent = (__holeIndex - 1) / 2;
138:     }
139:     *(__first + __holeIndex) = _GLIBCXX_MOVE(__value);
140:     }
...
File: c:\Users\Arek\Documents\STUDIA IV SEMESTR\OIAK\WEKTORYZACJA -
projekt\stl_heap.h
211: template<typename _RandomAccessIterator, typename _Distance,
212:     typename _Tp, typename _Compare>
213:     void
214:     __adjust_heap(_RandomAccessIterator __first, _Distance __holeIndex,
215:       _Distance __len, _Tp __value, _Compare __comp)
216:     {
217:     const _Distance __topIndex = __holeIndex;
218:     _Distance __secondChild = __holeIndex;
219:     while (__secondChild < (__len - 1) / 2)
220:     {
221:       __secondChild = 2 * (__secondChild + 1);
```

```
222:      if (__comp(__first + __secondChild,
223:                 __first + (__secondChild - 1)))
224:      __secondChild--;
225:      *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first + __secondChild));
226:      __holeIndex = __secondChild;
227:    }
228:    if ((__len & 1) == 0 && __secondChild == (__len - 2) / 2)
229:    {
230:      __secondChild = 2 * (__secondChild + 1);
231:      *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first
232:                                                 + (__secondChild - 1)));
233:      __holeIndex = __secondChild - 1;
234:    }
235:    __decltype(__gnu_cxx::__ops::__iter_comp_val(_GLIBCXX_MOVE(__comp)))
236:    __cmp(_GLIBCXX_MOVE(__comp));
237:    std::__push_heap(__first, __holeIndex, __topIndex,
238:                     _GLIBCXX_MOVE(__value), __cmp);
239:    }
```

Kompilator gcc informuje o kontroli przepływu wskazując na linijki 135 i 219.

Sortowanie tablicy przy użyciu niestandardowego obiektu funkcji:

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: #include <array>
6:
7: int main()
8: {
9:       std::array<int, 10> s = { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
10:      struct {
11:      bool operator()(int a, int b) const
12:      {
13:              return a < b;
14:      }
15:      } customLess;
16:      std::sort(s.begin(), s.end(), customLess);
17:      for (auto a : s) {
18:      std::cout << a << " ";
19:      }
20:      std::cout << '\n';
21:
22: }
```

Wektoryzacja: nieudana
Komunikaty gcc:

/usr/include/c++/7/bits/stl_heap.h:405:31: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_heap.h:405:31: note: bad loop form.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: not vectorized: latch block not empty.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: not vectorized: multiple nested loops.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: bad loop form.

Kompilator wskazuje na analogiczne miejsca w plikach nagłówkowych jak w przykładzie z domyślnym operatorem.

Sortowanie tablicy przy użyciu operatora lambda:

```
1: #include <algorithm>
2: #include <iostream>
3: #include <vector>
4: #include <cmath>
5: #include <array>
6:
7: int main()
8: {
9:      std::array<int, 10> s = { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
10:     std::sort(s.begin(), s.end(), [](int a, int b) {
11:     return a > b;
12:     });
13:     for (auto a : s) {
14:     std::cout << a << " ";
15:     }
16:     std::cout << '\n';
17: }
```

Wektoryzacja: nieudana
Komunikaty gcc:

/usr/include/c++/7/bits/stl_heap.h:405:31: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_heap.h:405:31: note: bad loop form.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: not vectorized: latch block not empty.
/usr/include/c++/7/bits/stl_heap.h:344:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1951:4: note: not vectorized: multiple nested loops.

/usr/include/c++/7/bits/stl_algo.h:1951:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_algo.h:1907:4: note: bad loop form.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: not vectorized: number of iterations cannot be computed.
/usr/include/c++/7/bits/stl_algo.h:1905:17: note: bad loop form.

Kompilator wskazuje na analogiczne miejsca w plikach nagłówkowych jak w przykładzie z domyślnym operatorem.

7.4 Operacje na liście dwukierunkowej

Lista dwukierunkowa nie jest uporządkowana w pamięci tak jak tablica. Nie ma zatem możliwości wektoryzacji. Przykład realizacji kilku operacji na liście dwukierunkowej:

```
1: #include <algorithm>
2: #include <iostream>
3: #include <list>
4: int main()
5: {
6:      // Tworzy listę zawierającą liczby całkowite
7:      std::list<int> l = { 7, 5, 16, 8 };
8:
9:      // Dodaje element na początku listy
10:     l.push_front(25);
11:     // Dodaje element na końcu listy
12:     l.push_back(13);
13:
14:     // Wstawia element przed 16-ką, poprzez wyszukanie jej
15:     auto it = std::find(l.begin(), l.end(), 16);
16:     if (it != l.end()) {
17:             l.insert(it, 42);
18:     }
19:     // Iteruje po wartościach w liście i wypisuje je
20:     for (int n : l) {
21:             std::cout << n << '\n';
22:     }
23: }
```

Wektoryzacja: nieudana
Komunikaty gcc:

Analyzing loop at /usr/include/c++/7/bits/list.tcc:70
/usr/include/c++/7/bits/list.tcc:70:20: note: ===== analyze_loop_nest =====
/usr/include/c++/7/bits/list.tcc:70:20: note: === vect_analyze_loop_form ===
/usr/include/c++/7/bits/list.tcc:70:20: note: === get_loop_niters ===
/usr/include/c++/7/bits/list.tcc:70:20: note: not vectorized: number of iterations cannot be computed.

```
/usr/include/c++/7/bits/list.tcc:70:20: note: bad loop form.

Analyzing loop at sortowanie.cpp:22
list_test.cpp:22:18: note: ===== analyze_loop_nest =====
list_test.cpp:22:18: note: === vect_analyze_loop_form ===
list_test.cpp:22:18: note: not vectorized: control flow in loop.
list_test.cpp:22:18: note: bad loop form.

Analyzing loop at /usr/include/c++/7/bits/stl_list.h:162
/usr/include/c++/7/bits/stl_list.h:162:21: note: ===== analyze_loop_nest =====
/usr/include/c++/7/bits/stl_list.h:162:21: note: === vect_analyze_loop_form ===
/usr/include/c++/7/bits/stl_list.h:162:21: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_list.h:162:21: note: bad loop form.

Analyzing loop at /usr/include/c++/7/bits/stl_list.h:143
/usr/include/c++/7/bits/stl_list.h:143:7: note: ===== analyze_loop_nest =====
/usr/include/c++/7/bits/stl_list.h:143:7: note: === vect_analyze_loop_form ===
/usr/include/c++/7/bits/stl_list.h:143:7: note: not vectorized: control flow in loop.
/usr/include/c++/7/bits/stl_list.h:143:7: note: bad loop form.
list_test.cpp:5:5: note: vectorized 0 loops in function.
```

8. Wnioski

Zadanie projektowe umożliwiło nam poznanie jednego z podstawowych sposobów optymalizacji kodu. Automatyczna wektoryzacja powinna być stosowane przede wszystkim w przypadkach, gdy wielokrotnie wykonywane są proste operacje. W projekcie nie wykonywaliśmy pomiarów czasu, więc nie możemy porównać jak dokładnie wpływa ona na czas wykonania programu. W czasie projektu nauczyliśmy się kompilować programy w sposób umożliwiający jego dalszą analizę.
Naszymi głównymi źródłami informacji były komunikaty na temat wektoryzacji otrzymywane poleceniami *-fopt-info-vec-optimized* oraz *-fopt-info-vec-missed* oraz listingi kodu asemblera. W pierwszym przypadku byliśmy informowani, czy wektoryzacja faktycznie miała miejsce, oraz o powodach braku jej wystąpienia. Natomiast analiza kodu assemblera pokazała nam w jaki sposób kompilator przeprowadza wektoryzację, jak zarządza pamięcią oraz jakie akcje wykonuje. Niejednokrotnie generowany kod był bardzo obszerny, a samo wykorzystanie instrukcji wektorowych potrafiło ograniczać się do kilku linijek. W celu poprawy przejrzystości zakładaliśmy, że dane są wyrównane w pamięci, co niejednokrotnie omijało konieczność generowania niepotrzebnego z punktu widzenia projektu kodu. Problem ten mógł być spowodowany starszą wersją kompilatora (gcc 7.2), który został wykorzystany

na potrzeby projektu. W najnowszej wersji (obecnie 9.1) pewne mechanizmy zostały usprawnione, co przekłada się na jakość generowanego kodu.
Ważnym aspektem projektu było poznanie ograniczeń kompilatora w zakresie automatycznej wektoryzacji. Komunikaty kompilacji najczęściej traktowały o zbyt złożonych warunkach bądź złej formie pętli. Aby wektoryzacja przebiegła pomyślnie kod musi być pisany w możliwie jak najprostszej postaci. Mimo tego nie jest to gwarancją pomyślnej wektoryzacji, gdyż jak udało nam się ustalić, są przypadki w których kod nie może zostać zwektoryzowany, jak warunkowe przerwanie pętli. Również w przypadku gotowych algorytmów oraz struktur danych wektoryzacja wielokrotnie kończyła się fiaskiem. Problemem bezapelacyjnie jest ich poziom skomplikowania. Są one za bardzo złożone, by kompilator mógł sobie z nimi poradzić. Pokazuje to, że w przypadku chęci pisania dającego się zwektoryzować kodu trzeba rozpocząć implementację w odpowiedni sposób od samego początku, inaczej na dalszym etapie pracy niezbędne będzie przepisywanie kodu od nowa, co może przysporzyć wiele innych problemów i być kompletnie nieopłacalne z punktu widzenia twórcy oprogramowania.

Spis treści