Architektury systemów komputerowych

Pracownia 3: „Optymalizacja kodu”

Termin oddawania 12 czerwca 2019

Wprowadzenie

Rozwiązanie zadań polega na zmodyfikowaniu plików źródłowych dostępnych na stronie przedmiotu oraz napisaniu sprawozdania. Raport ma być pojedynczym plikiem tekstowym w formacie markdown[1] lub LaTeX, do którego można dołączyć wykresy i diagramy. Do każdego zadania dołączono listę pytań, na które należy odpowiedzieć w sprawozdaniu. Zarówno odpowiedź pozytywną jak i negatywną należy właściwie uzasadnić demonstrując posiadaną wiedzę o strukturze pamięci podręcznych. Sprawozdanie musi zawierać również informacje o środowisku, w którym przeprowadzono eksperymenty – patrz przykładowy plik «`raport.md`».

Wymogi

Programy dostarczone przez prowadzącego należy kompilować i uruchamiać pod kontrolą systemu Linux dla architektury x86-64. Należy zadbać o niską wariancję wyników uruchomienia polecenia z danym zestawem parametrów. Najłatwiej osiągnąć to minimalizując obciążenie systemu, np. ograniczając liczbę procesów.

UWAGA! Uruchamianie eksperymentów pod systemem zainstalowanym w maszynie wirtualnej może poważnie zaburzać wyniki!

Do każdego z rozwiązanych zadań należy dostarczyć wszystkie pliki niezbędne do powtórzenia eksperymentu na komputerze osoby sprawdzającej zadanie. Wyniki umieszczone w raporcie muszą jednoznacznie wspierać prezentowaną tezę – sprawdzający nie ulegnie pokusie naginania rzeczywistości w optymistycznym tonie.

UWAGA! Pamiętaj, że właściwy sposób mierzenia czasu wykonania programu polega na wielokrotnym jego uruchomieniu, odrzuceniu skrajnych pomiarów i uśrednieniu reszty wyników.

Żeby uzasadnić wyniki pomiarów należy posłużyć się narzędziem `perf`[2] przy pomocy którego można odczytać **liczniki monitorowania sprzętu** (ang. *hardware performance monitoring counters*). Na poniższym wydruku widnieje zwartość wybranych liczników po uruchomieniu rozwiązania trzeciego wariantu zadania 1:

```
# sudo ./perf.sh ./matmult -n 1024 -v 3
...
Performance counter stats for './matmult -n 1024 -v 3':

 2429096754      cycles:u
   14860857      bus-cycles:u
 7096743620      instructions:u            #    2,92  insn per cycle
 1134126229      branches:u
    4656407      branch-misses:u           #    0,41% of all branches
  202036076      cache-references:u
   24061707      cache-misses:u            #   11,910 % of all cache refs
 2147087260      L1-dcache-loads
  544549683      L1-dcache-load-misses     #   25,36% of all L1-dcache hits
   71097288      L1-dcache-stores
    9998663      LLC-loads
     788933      LLC-load-misses           #    7,89% of all LL-cache hits
      20423      LLC-stores
       1244      LLC-store-misses
 2292483602      dTLB-loads
     418953      dTLB-load-misses          #    0,02% of all dTLB cache hits
   70424948      dTLB-stores
        447      dTLB-store-misses

0,644310769 seconds time elapsed
```

[1] https://daringfireball.net/projects/markdown/syntax
[2] https://perf.wiki.kernel.org/index.php/Main_Page

Na wydruku widać:

- «cycles:u» liczbę wykorzystanych cykli procesora w przestrzeni użytkownika,
- «instructions:u» liczbę wykonanych instrukcji,
- «branches:u» liczbę wykonanych instrukcji skoków warunkowych,
- «branch-misses:u» liczbę błędnie przewidzianych skoków,
- «L1-dcache-*» liczbę odczytów, zapisów i chybień w pamięć podręczną danych pierwszego poziomu,
- «LLC-*» j.w. dla pamięci podręcznej niższego poziomu,
- «dTLB» j.w dla pamięci podręcznej translacji adresów.

Można według potrzeb zmodyfikować zawartość pliku «perf.sh», aby dodać inne liczniki sprzętowe lub programowe, które można wydrukować poleceniem «sudo perf list».

Rozwiązanie

Wyniki swojej pracy należy wysłać w archiwum «tgz» o nazwie «indeks_imie_nazwisko.tgz» z użyciem systemu SKOS. Rozpakowanie plików poleceniem «tar» ma dać następującą strukturę katalogów:

```
999999_jan_nowak/
  Makefile
  raport.md
  bsearch.c
  cache.c
  matmult.c
  randwalk.c
  transpose.c
  ...
```

Oceniający zadania używa komputera z zainstalowanym systemem Debian GNU/Linux 9 dla architektury x86-64. Ściąga z systemu SKOS archiwum dostarczone przez studenta, po czym:

- sprawdza poprawność struktury katalogów,
- wykołuje polecenie «make» by zbudować pliki wykonywalne (w tym dokument «pdf» z pliku «tex»),
- czyta raport i sprawdza dostępność plików niezbędnych do powtórzenia eksperymentów,
- czyta treść rozwiązań celem znalezienia usterek i plagiatów,
- powtarza wybrane eksperymenty zgodnie z instrukcjami w raporcie,
- wywołuje polecenie «make clean», by usunąć wszystkie pliki otrzymane w procesie budowania.

Zadanie 1 (2). Na slajdach do wykładu pt. „Cache Memories" zaprezentowano różne podejścia do implementacji mnożenia dwóch macierzy. Na slajdzie 47[3] podano trzy rozwiązania o różnej kolejności przeglądania elementów tablicy. Na slajdzie 53 widnieje rozwiązanie wykorzystujące technikę kafelkowania.

Należy uzupełnić ciało procedur «`multiply0`» ... «`multiply3`» w pliku źródłowym «`matmult.c`». Po dodaniu ich implementacji na komputerze testowym uzyskano następujące wyniki:

```
$ ./matmult -n 1024 -v 0
Time elapsed: 3.052755 seconds.
$ ./matmult -n 1024 -v 1
Time elapsed: 0.746337 seconds.
$ ./matmult -n 1024 -v 2
Time elapsed: 9.882309 seconds.
$ ./matmult -n 1024 -v 3
Time elapsed: 0.698795 seconds.
```

Powtórz eksperyment ze slajdu 48 dla rosnących wartości n rozmiaru boku macierzy. Zbierz rezultaty uruchomień do pliku tekstowego i utwórz z nich wykres. Tworzenie wykresów z danych numerycznych przy pomocy narzędzia gnuplot[4] przystępnie wyjaśniono na stronie gnuplot: not so Frequently Asked Questions[5].

Sprawozdanie: Czy uzyskane wyniki różnią się od tych uzyskanych na slajdzie? Z czego wynika rozbieżność między wynikami dla poszczególnych wersji mnożenia macierzy? Jaki wpływ ma rozmiar kafelka na wydajność «`multiply3`»?

Zadanie 2 (bonus). W pliku źródłowym «`matmult.c`» do poprzedniego zadania zdefiniowano wartości «`A_OFFSET`», «`B_OFFSET`», «`C_OFFSET`». Dobrane wartości wymuszają, aby macierze nie zaczynały się pod takimi samymi adresami wirtualnymi modulo rozmiar strony. Jeśli po ustawieniu definicji tych wartości na 0 obserwujesz spadek wydajności w kafelkowanej wersji mnożenia macierzy postaraj się wyjaśnić ten fenomen.

Sprawozdanie: Dla jakich wartości n obserwujesz znaczny spadek wydajności? Czy rozmiar kafelka ma znaczenie? Czy inny wybór wartości domyślnych «`OFFSET`» daje poprawę wydajności?

Wskazówka: Obserwowany efekt najprawdopodobniej wynika z generowania konfliktów w obrębie zbiorów.

Zadanie 3 (2). Poniżej podano funkcję transponującą macierz kwadratową o rozmiarze n. Niestety jej kod charakteryzuje się niską lokalnością przestrzenną dla tablicy «`dst`». Używając metody kafelkowania zoptymalizuj poniższą funkcję pod kątem lepszego wykorzystania pamięci podręcznej.

```
1 void transpose(int *dst, int *src, int n) {
2   for (int i = 0; i < n; i++)
3     for (int j = 0; j < n; j++)
4       dst[j * n + i] = src[i * n + j];
5 }
```

Należy uzupełnić ciało procedury «`transpose2`» w pliku źródłowym «`transpose.c`». Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./transpose -n 4096 -v 0
Time elapsed: 21.528841 seconds.
$ ./transpose -n 4096 -v 1
Time elapsed: 5.251710 seconds.
```

Sprawozdanie: Jaki wpływ na wydajność «`transpose2`» ma rozmiar kafelka? Czy czas wykonania programu z różnymi rozmiarami macierzy identyfikuje rozmiary poszczególnych poziomów pamięci podręcznej?

[3] Chodzi o numer w prawym dolnym rogu slajdu.
[4] `http://www.gnuplot.info/`
[5] `http://lowrank.net/gnuplot/datafile2-e.html`

Zadanie 4 (3). Poniższy kod realizuje losowe błądzenie po tablicy. Intuicyjnie źródłem problemów z wydajnością powinny być dostępy do pamięci. Zauważ, że instrukcje warunkowe w liniach 17, 20 i 23 zależą od losowych wartości. W związku z tym procesorowi będzie trudno przewidzieć czy dany skok się wykona. Kara za błędną decyzję predyktora wynosi we współczesnych procesorach (np. i7-6700[6]) około 20 cykli.

```
1  int randwalk(uint8_t *arr, int n, int len) {
2    int sum = 0, k = 0;
3    uint64_t dir = 0;
4    int i = n / 2;
5    int j = n / 2;
6
7    do {
8      k -= 2;
9      if (k < 0) {
10       k = 62;
11       dir = fast_random();
12     }
13
14     int d = (dir >> k) & 3;
15
16     sum += arr[i * n + j];
17     if (d == 0) {
18       if (i > 0)
19         i--;
20     } else if (d == 1) {
21       if (i < n - 1)
22         i++;
23     } else if (d == 2) {
24       if (j > 0)
25         j--;
26     } else {
27       if (j < n - 1)
28         j++;
29     }
30   } while (--len);
31
32   return sum;
33 }
```

Podglądając kod wynikowy z kompilatora poleceniem «objdump» zamień instrukcje warunkowe z linii 17...29 na obliczenia bez użycia instrukcji skoków warunkowych. Skorzystaj z faktu, że kompilator tłumaczy wyrażenia obliczające wartość porównania dwóch liczb z użyciem instrukcji «SETcc».

Należy uzupełnić ciało procedury «randwalk2» w pliku źródłowym «randwalk.c». Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./randwalk -S 0xea3495cc76b34acc -n 7 -s 16 -t 14 -v 0
Time elapsed: 5.943448 seconds.
$ ./randwalk -S 0xea3495cc76b34acc -n 7 -s 16 -t 14 -v 1
Time elapsed: 3.066678 seconds.
```

Opcja «-S» służy do podawania ziarna generatora liczb pseudolosowych. Bez tej opcji każde uruchomienie programu będzie generowało inną tablicę, a zatem i inne wyniki.

Sprawozdanie: Ile instrukcji maszynowych ma ciało pętli przed i po optymalizacji? Ile spośród nich to instrukcje warunkowe? Czy rozmiar tablicy ma duży wpływ na działanie programu?

Zadanie 5 (3). Posortowaną dużą tablicę liczb całkowitych będziemy wielokrotnie przeszukiwać używając metody wyszukiwania binarnego. Niestety podany niżej algorytm wykazuje niską lokalność przestrzenną. Dzięki zbudowaniu kopca binarnego z elementów tablicy (tj. układamy w pamięci liniowo kolejne poziomy drzewa poszukiwań binarnych) można uzyskać znaczące przyspieszenie — w trakcie prezentacji zadania podaj uzasadnienie. Dla uproszczenia przyjmujemy, że w tablicy jest $2^n - 1$ elementów, tj. zajmujemy się tylko pełnymi drzewami binarnymi.

```
1  bool binary_search(int *arr, int size, int x) {
2    do {
3      size >>= 1;
4      int y = arr[size];
5      if (y == x)
6        return true;
7      if (y < x)
8        arr += size + 1;
9    } while (size > 0);
10   return false;
11 }
```

[6] https://www.7-cpu.com/cpu/Skylake.html

Wskazówka: Rozważ prawdopodobieństwo wykorzystania elementów ściągniętych do pamięci podręcznej w kolejnych przebiegach procedury «binary_search». Zależy nam by w cache przechowywać dane o wysokim prawdopodobieństwie ponownego użycia.

W pliku źródłowym «bsearch.c» należy uzupełnić ciało procedury «heapify», która zmienia ułożenie elementów tablicy na strukturę kopcową, a także procedurę «heap_search». Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./bsearch -S 0x5bab3de5da7882ff -n 23 -t 24 -v 0
Time elapsed: 7.616777 seconds.
$ ./bsearch -S 0x5bab3de5da7882ff -n 23 -t 24 -v 1
Time elapsed: 2.884369 seconds.
```

Sprawozdanie: Czemu zmiana organizacji danych spowodowała przyspieszenie algorytmu wyszukiwania? Czy odpowiednie ułożenie instrukcji w ciele «heap_search» poprawia wydajność wyszukań?

Uwaga! W tym zadaniu mogą wychodzić bardzo nieintuicyjne wyniki, należy bacznie przyglądać się wynikom z programu perf!

Zadanie 6 (bonus). Tablica T przechowuje n elementów typu «int». Zaczyna się pod adresem podzielnym przez rozmiar strony i ma długość wielokrotności rozmiaru strony. S to zbiór wszystkich indeksów tej tablicy. Należy wygenerować pewne szczególne permutacje zbioru $U \subseteq S \setminus \{0\}$, tj. ciągi niepowtarzających się indeksów $i_1, i_2, \ldots, i_l$, gdzie $l \leq n$. Będziemy reprezentować je w tablicy T następująco: $T[0] := i_1$, $T[i_k] := i_{k+1}$, $T[i_l] \in \{0, -1\}$. Procedura «array_walk» w pliku «cache.c» przechodzi kolejno po elementach tablicy T. Działanie zakończy po osiągnięciu ostatniego elementu ciągu lub po wykonaniu k kroków.

Uważny wybór permutacji pozwala kontrolować liczbę chybień towarzyszących przeglądaniu T. Dodatkowo należy zminimalizować wariancję stosunku kosztu chybienia do kosztu trafienia. Przy tak ostrożnie zaprojektowanych eksperymentach chcemy ustalić następujące parametry podsystemu pamięci:

- **(2 pkt.)** długość linii pamięci podręcznej,
- **(2 pkt.)** rozmiar w bajtach pamięci podręcznej $\mathrm{L1}$ dla danych, $\mathrm{L2}$ i $\mathrm{L3}$,
- **(2 pkt.)** rozmiar zbioru sekcyjno-skojarzeniowej pamięci podręcznej $\mathrm{L1}$ dla danych, $\mathrm{L2}$ i $\mathrm{L3}$,
- **(2 pkt.)** liczbę wpisów w TLB pierwszego poziomu dla danych i TLB drugiego poziomu.

Oczekuje się, że student w trakcie prezentacji rozwiązania będzie w stanie sprawnie wytłumaczyć w jaki sposób zbadał organizację pamięci podręcznej i na jakiej podstawie wyznaczył poszczególne parametry. Zebrane wyniki i tok rozumowania muszą wystarczyć do przekonania prowadzącego.

Jeśli jest taka potrzeba można zmodyfikować listę parametrów linii poleceń przyjmowanych przez program.

UWAGA! Twoim zadaniem jest napisać przekonujący raport! Ocenie będzie podlegać tylko jego tekst.

Sprawozdanie: Jak zaprojektowano eksperyment? Czy eksperyment mierzy to co powinien? Jak uzyskane dane popierają twierdzenie? Jakie jest zadanie wygenerowanej permutacji? Czy koszt chybienia jest stały? Jak poradzono sobie z wyeliminowaniem czynników zakłócających pomiary?

Wskazówka: Mając na uwadze strukturę pamięci DRAM postaraj się zmaksymalizować koszt chybienia w pamięć podręczną.

Zadanie 7 (3, bonus). Tablicę o rozmiarze n liczb typu «long» skanujemy m razy wyliczając sumę wszystkich elementów. Wiemy, że dla n równego 4096 cała tablica «arr» mieści się w pamięci podręcznej pierwszego poziomu. Dodatkowo widać, że skanujemy tablicę sekwencyjne. Zatem wydajność w poniższej procedury nie jest ograniczona przepustowością pamięci.

```
1  #define IDENT 0
2  #define OP +
3  typedef long T;
4
5  T combine(T *arr, size_t n, size_t m) {
6    T r = IDENT;
7    for (size_t j = 0; j < m; j++)
8      for (size_t i = 0; i < n; i++)
9        r = r OP arr[i];
10   return r;
11 }
```

Żeby zoptymalizować poniższy kod spróbujemy wykorzystać fakt, że procesor dysponuje wieloma jednostkami funkcyjnymi i potrafi wykonywać kod poza porządkiem programu (ang. *out-of-order execution*). Procesor potrafi zlecić do wykonania w jednym cyklu zegarowym wiele instrukcji, które są one od siebie niezależne. W przypadku powyższego programu widać, że obliczenie wartości `r` w linii 9 nie może się rozpocząć póki procesor nie obliczy wartości `r` w poprzedniej iteracji pętli.

Na poniższym wydruku zaprezentowano wynik działania kolejnych prób prostej optymalizacji pętli. Przy czym w ostatnim przypadku zdano się na automatyczą wektoryzację obliczeń przez kompilator.

```
$ ./combine -n 4096 -t 2097152 -v 0
Time elapsed: 2.874563 seconds.
$ ./combine -n 4096 -t 2097152 -v 1
Time elapsed: 2.244405 seconds.
$ ./combine -n 4096 -t 2097152 -v 2
Time elapsed: 1.880398 seconds.
$ ./combine -n 4096 -t 2097152 -v 3
Time elapsed: 1.507615 seconds.
$ ./combine -n 4096 -t 2097152 -v 4
Time elapsed: 0.999397 seconds.
```

Twoim zadaniem jest znalezienie najbardziej optymalnej organizacji pętli, która pozwoli procesorowi wykonać jak najwięcej instrukcji. Możesz założyć, że adres początku tablicy jest podzielny przez rozmiar strony. Liczba elementów w tablicy jest potęgą dwójki nie mniejszą niż 6. Wynik swoich optymalizacji porównaj z czasem działania automatycznie zwektoryzowanego kodu.

Sprawozdanie: Czy jesteś w stanie wyznaczyć liczbę jednostek funkcyjnych potrafiących wykonywać dodawanie? Jak wygląda graf przepływu danych dla zoptymalizowanej pętli? Kiedy zwiększanie liczby niezależnych operacji przestaje się opłacać? Co się dzieje, jeśli tablica przestaje się mieścić w pamięci podręcznej pierwszego poziomu?

Zadanie 8 (3, bonus). Mamy dwuwymiarową tablicę wartości typu «`uint8_t`» – będziemy ją nazywać teksturą. Program w pętli wykonuje n razy następującą procedurę: losuje dwa punkty (x_1, y_1) i (x_2, y_2), po czym sumuje wartość tekstury we wszystkich punktach leżących na linii między wybranymi punktami, do czego używa algorytmu Bresenham'a[7].

Twoim zadaniem jest poprawić lokalność odwołań do pamięci. Jedyne co możesz zrobić to zmienić sposób przechowywania danych w teksturze. W tym celu w pliku «`texture.c`» należy uzupełnić ciało procedur «`get_2`» i «`put_2`», które odpowiednio ustawiają i pobierają wartość tekstury w punkcie (x, y). Na komputerze testowym uzyskano następujące wyniki przed i po optymalizacji:

```
$ ./texture -t 65536 -v 0
Time elapsed: 1.700562 seconds.
$ ./texture -t 65536 -v 1
Time elapsed: 1.031514 seconds.
```

Sugerowane rozwiązanie polega na podzieleniu tekstury na kafelki o boku długości 2^k. Ze starszych bitów współrzędnych punktu najpierw należałoby wyliczyć adres początku kafla w pamięci, a z dolnych bitów zaadresować wartość wewnątrz kafla.

Sprawozdanie: Czemu zmiana organizacji danych spowodowała przyspieszenie? Czy translacja adresów ma w tym przypadku istotny wpływ na wydajność? Jaki jest optymalny rozmiar kafla? Czy zoptymalizowana wersja wykonuje więcej instrukcji?

[7] `https://pl.wikipedia.org/wiki/Algorytm_Bresenhama`