Architektury systemów komputerowych

Pracownia 3: „Optymalizacja kodu”

24 maja 2021

Wprowadzenie

Celem tej pracowni jest zdobycie praktycznej wiedzy na temat optymalizacji programów pod kątem użycia pamięci podręcznych i wykorzystania właściwości mikroarchitektury procesora, tj. superskalarnego przetwarzania instrukcji i predykcji skoków warunkowych. Oddawanie rozwiązań będzie zrealizowane przy użyciu systemu GitHub Classroom. Rozwiązanie każdego z zadań składa się z kilku etapów, które opisano poniżej.

Najpierw należy uzupełnić procedury w wyznaczonym pliku źródłowym. Rozwiązanie musi spełniać kryteria podane w pliku «`GNUmakefile`», np. maksymalny procent liczby chybień w pamięć podręczną lub liczby źle przewidzianych skoków. Rozwiązanie będzie testowane automatycznie w symulowanym środowisku przy pomocy narzędzia profilującego `callgrind`[1] będącego częścią programu programu `valgrind`[2]. Przeprowadzenie symulacji jest możliwe na komputerze studenta – wystarczy wykonać polecenie «`make sim`».

Następnie należy wyznaczyć parametry systemu, na którym student rozwiązuje zadanie, co najmniej nazwę kodową mikroarchitektury (np. Skylake) i konfigurację podsystemu pamięci podręcznej. Część z tych parametrów można odczytać przy pomocy narzędzia «`x86info`». Możliwe, że przyda się również wiedza na temat opóźnień przetwarzania instrukcji lub koszt źle przewidzianego skoku. Szczegółowe informacje na temat danej realizacji mikroarchitektury procesora można znaleźć w dokumencie: „The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”[3].

Po zebraniu informacji na temat systemu należy przeprowadzić szereg eksperymentów. Polegają one na uruchomieniu optymalizowanego program na swoim komputerze i zebraniu wyników z liczników sprzętowych[4] wbudowanych w procesor. Należy zauważyć, że monitor maszyn wirtualnych (np. `VirtualBox`) nie daje dostępu do tychże liczników. Innymi słowy, eksperymenty trzeba wykonywać pod kontrolą systemu `Linux` zainstalowanego na fizycznej maszynie. Program testowy, skonsolidowany z biblioteką `PAPI`[5], przyjmuje opcję «`-p`», przy pomocy której można wybrać do odczytu określony zestaw liczników sprzętowych. Następnie należy próbować obniżyć czas uruchomienia programu poprzez (w nawiasie podano jak odczytać liczniki):

- zmniejszenie liczby chybień w pamięć podręczną i `TLB` (`-p memory`),
- zmniejszenie liczby źle przewidzianych skoków (`-p branch`),
- zwiększenie liczby instrukcji przetworzonych w ciągu jednego cyklu zegarowego procesora (`-p ipc`).

UWAGA! Pamiętaj, że właściwy sposób mierzenia czasu wykonania programu polega na wielokrotnym jego uruchomieniu, odrzuceniu skrajnych pomiarów i uśrednieniu reszty wyników. Należy zadbać o niską wariancję wyników pomiaru czasu wykonywania programu na procesorze. Najłatwiej osiągnąć to minimalizując obciążenie systemu, tj. ograniczając liczbę procesów.

Po zakończeniu eksperymentów należy napisać sprawozdanie z wykonania zadania. Do każdego zadania dołączono listę pytań, na które należy odpowiedzieć. Przedstawiona w raporcie teza musi być solidnie podparta danymi zebranymi z eksperymentów. Raporty, które będą zawierać mętne wyjaśnienia niepodparte danymi i wiedzą na temat architektury systemów komputerowych, nie zostaną dobrze ocenione. Nie ulegaj pokusie naginania wyników – oceniający na pewno to wychwyci.

Ciężar skonstruowania przekonującej argumentacji spoczywa na Was!

[1] `https://valgrind.org/docs/manual/cl-manual.html`
[2] `https://valgrind.org/docs/manual`
[3] `https://www.agner.org/optimize/microarchitecture.pdf`
[4] `https://en.wikipedia.org/wiki/Hardware_performance_counter`
[5] `https://icl.utk.edu/papi/`

Rozwiązania

Dla każdego zadania prowadzący dostarczy odnośnik do serwisu GitHub Classroom służący do wygenerowania prywatnego repozytorium, w którym przed upływem terminu należy umieścić kod rozwiązania zadania i raport. Wszystkie elementy rozwiązania należy umieścić w gałęzi «`master`». Można modyfikować wyłącznie te pliki, które zostały wskazane w treści zadania. Należy zadbać o to by programy kompilowały się bez ostrzeżeń pod systemem Debian 10 dla architektury `x86-64`. Nie wolno modyfikować zawartości gałęzi «`feedback`»!

Raport należy zawrzeć w pliku «`raport.md`» w formacie `markdown`. Rezultaty uruchomień programu zbierz do pliku tekstowego i utwórz z nich wykres. Tworzenie wykresów z danych numerycznych przy pomocy narzędzia `gnuplot`[6] przystępnie wyjaśniono na stronie gnuplot: not so Frequently Asked Questions[7]. W repozytorium można umieszczać wyłącznie pliki tekstowe.

Prowadzący będzie oceniał zmiany plików w prośbie o połączenie (ang. *pull request*) o nazwie «`Feedback`». Inne prośby będą przez sprawdzającego ignorowane. W zakładce „Files changed” mają być widać **wyłącznie** zmiany, które są częścią rozwiązania i zostały przygotowane przez studenta. Rozwiązania z nadmiarowymi plikami lub zbędnym kodem będą kierowane do poprawki.

[6] `http://www.gnuplot.info/`
[7] `http://lowrank.net/gnuplot/datafile2-e.html`

Zadania

Punktacja: W nawiasach przy zadaniu podano dwie liczby, które należy rozumieć następująco: pierwsza to punkty do zdobycia za poprawne zakończenie symulacji w automatycznej ocenie zadania, druga to punkty do uzyskania za sprawozdanie. Sprawozdanie napisane wyłącznie na podstawie wyników z symulatora nie uzyska pełnej liczby punktów. Za wyjątkowo dobrze napisane sprawozdanie, w którym dodatkowo użyto danych zebranych z liczników sprzętowych, można otrzymać jeden punkt bonusowy.

Zadanie 1 (1+2). Na slajdach do wykładu pt. „Cache Memories" zaprezentowano różne podejścia do implementacji mnożenia dwóch macierzy. Na slajdzie 47[8] podano trzy rozwiązania o różnej kolejności przeglądania elementów tablicy. Na slajdzie 53 widnieje rozwiązanie wykorzystujące technikę kafelkowania. Należy uzupełnić ciało procedur «matmult0» ... «matmult3» w pliku źródłowym «matmult.c». Powtórz eksperyment ze slajdu 48 dla rosnących wartości n rozmiaru boku macierzy.

W pliku źródłowym «matmult.h» 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.

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

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 «matmult3»? Dla jakich wartości n obserwujesz znaczny spadek wydajności? Czy rozmiar kafelka ma znaczenie? Czy inny wybór wartości domyślnych «*_OFFSET» daje poprawę wydajności?

Zadanie 2 (1+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 transpose0(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 «transpose1» w pliku źródłowym «transpose.c».

Sprawozdanie: Jaki wpływ na wydajność «transpose1» ma rozmiar kafelka? Czy czas wykonania programu z różnymi rozmiarami macierzy identyfikuje rozmiary poszczególnych poziomów pamięci podręcznej?

Zadanie 3 (1+2). 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[9]) 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 }
```

[8] Chodzi o numer w prawym dolnym rogu slajdu.
[9] https://www.7-cpu.com/cpu/Skylake.html

Podglądając kod wynikowy z kompilatora poleceniem «objdump» zamień instrukcje warunkowe z linii 17...29 na obliczenia z użyciem zmiennych logicznych. Skorzystaj z faktu, że kompilator tłumaczy wyrażenia obliczające wartość porównania dwóch liczb z użyciem instrukcji «SETcc». Należy uzupełnić ciało procedury «randwalk1» w pliku źródłowym «randwalk.c». 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? Jak optymalizacja wpłynęła na IPC? Jak na IPC wpływa zmiana kolejności instrukcji w ciele pętli? Czy rozmiar tablicy ma duży wpływ na działanie programu?

Zadanie 4 (2+2). Posortowaną dużą tablicę liczb całkowitych będziemy wielokrotnie przeszukiwać używając metody wyszukiwania binarnego. Niestety podany niżej algorytm wykazuje niską lokalność przestrzenną. Możemy jednak zmienić organizację danych w tablicy, żeby zwiększyć lokalność odwołań. Na przykład możemy ułożyć w pamięci liniowo kolejne poziomy drzewa poszukiwań binarnych i w ten sposób uzyskać znaczące przyspieszenie. Dla uproszczenia przyjmujemy, że w tablicy jest $2^n - 1$ elementów, tj. zajmujemy się tylko pełnymi drzewami binarnymi.

```
1  bool binsearch0(T *arr, long size, T x) {
2    do {
3      size >>= 1;
4      T 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 }
```

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

W pliku źródłowym «bsearch.c» należy uzupełnić ciało procedury «linearize» i «binsearch1». Pierwsza procedura ma zmienić ułożenie elementów tablicy w taki sposób, żeby wszystkie elementy z kolejnych poziomów drzewa wyszukiwania binarnego były ułożone w tablicy sekwencyjnie.

Sprawozdanie: Czemu zmiana organizacji danych spowodowała przyspieszenie algorytmu wyszukiwania? Jak zmieniła się lokalność czasowa elementów przechowywanych w pamięci podręcznej? W jaki sposób zmiana kolejności instrukcji w ciele pętli zmienia IPC? Czy można to wytłumaczyć posługując się programem llvm-mca? Czy użycie instrukcji wbudowanych kompilatora «__builtin_expect» i «__builtin_prefetch» przynosi jakieś zyski? Jeśli ich użyjesz należy wytłumaczyć co robią.

Dla odważnych: Dodatkowo można zamienić reprezentację pełnego drzewa binarnego na pełne drzewo n-arne, gdzie jeden wierzchołek mieści się w całości w bloku pamięci podręcznej.

Zadanie 5 (1+2). Mamy dwuwymiarową tablicę wartości typu «uint8_t», którą będziemy 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 wyznaczenia punktów na linii używa algorytmu Bresenham'a[10].

Twoim zadaniem jest poprawić lokalność odwołań do pamięci. Jedyne co możesz zrobić to zmienić porządek przechowywania pikseli w teksturze. W tym celu w pliku «texture.c» należy uzupełnić ciało procedury «index_1», która wyznacza pozycję piksela o współrzędnych (x, y) w teksturze. 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. Można też rozważyć użycie krzywej Mortona[11].

Sprawozdanie: Czemu zmiana organizacji danych spowodowała przyspieszenie? Czy translacja adresów ma zauważalny wpływ na wydajność przeglądania tekstury? Jaki jest optymalny rozmiar kafla? Czy zoptymalizowana wersja wykonuje więcej instrukcji na jeden element? Jak zmieniło się IPC?

[10] https://pl.wikipedia.org/wiki/Algorytm_Bresenhama
[11] https://en.wikipedia.org/wiki/Z-order_curve