Lista 12 - raport

Mateusz Basiak
nr indeksu: 300487

Konfiguracja

Informacje o systemie:
- Dystrybucja: Linux Mint 18.3 Cinnamon
- Jądro systemu: GNU/Linux 4.10.0
- Kompilator: GCC 6.3.0
- Procesor: Intel® Core™ i5-4200U CPU @ 1.60GHz
- Liczba rdzeni: 4

Pamięć podręczna:
* L1: 64 KiB, 4-drożny (per rdzeń), rozmiar linii 64B
* L2: 256 KiB, 4-drożny (per rdzeń), rozmiar linii 64B
* L3: 3MiB, 12-drożny (współdzielony), rozmiar linii 64B

Pamięć TLB:
* L1: 4KiB strony, 4-drożny, 64 wpisy
* L2: 4KiB strony, 6-drożny, 1536 wpisów

Zadanie 1.

W pliku wykres1.pdf przedstawiony został performance poszczególnych funkcji w zależności od n, czyli rozmiaru boku tablicy. Każda funkcja była odpalana dla n = 128 * k, k = 1,2,3,...,8. Dla każdego n test był powtarzany wielokrotnie a wynik został uśredniony. Jak widać funkcje matmult1 i matmult3 osiągają najlepsze rezultaty, co zgadza się z zasadą lokalności i pokrywa się z wykresem z prezentacji z wykładu.

Funkcje matmult1 i matmult3 zachowują lokalność przestrzenną, przez co procesor może przez spore fragmenty operować na tej samej pamięci i dzięki temu oszczędza czas. Funkcje matmult0 i matmult2 wymagają częstszych odwołań do pamięci, co skutkuje znacznym pogorszeniem czasu działania.

Optymalnym rozmiarem kafelka jest 8. Zmniejszając rozmiar kafelka sprawiam, że jednorazowo pobieram mniej interesujących mnie informacji - marnuję część pobieranych danych. Przez to muszę pobierać je częściej i czas się wydłuża. Podobnie kiedy zwiększam rozmiar kafelka, nie jestem w stanie pobrać całego kafelka do L1 i przez to muszę pobierać cyklicznie jego części, by wykonywać operacje, co również wydłuża czas działania programu. Stąd wybór wielkości kafelka jest bardzo ważny dla czasu działania matmult3.

Zadanie 2.

Wykres 2 przedstawia różnice, między czasem wykonywania programu dla domyślnych offsetów i czasem dla offsetów wyzerowanych w zależności od wielkości tablicy (przy wielkości kafelka 8). Jak widać im większa tablica tym większe różnice czasowe. Wykres 3 przedstawia natomiast podobną zależność, tyle że tym razem parametrem jest wielkość kafelka przy tablicy 1024x1024. Jak widać niezależnie od wielkości kafelka, różnica czasowa prezentuje się podobnie.

Z moich obserwacji przy zmienianiu wartości zmiennych offset wynika, że o ile są parami różne modulo 4096 (rozmiar strony, 4KiB) to program wywołany z argumentami

./matmult -n 1024 -v 3

daje podobne wyniki, rzędu 1.165035 sekundy. Jeśli dwa spośród nich są równe modulo 4096 to czasy są gorsze, rzędu 1.311182 sekundy, natomiast kiedy wszystkie są równe to średni czas wynosi około 1.853761 sekundy.

Obserwowany efekt prawdopodobnie wynika ze zbyt małych rozmiarów zbiorów cache L1 (u mnie czterodrożny). Jeśli wszystkie trzy tablice odwołują się do tego samego zbioru, to jest on przeładowany, ponieważ nie ma w nim wystarczająco dużo miejsca. Stąd musi on często wymieniać linie, które w nim są, co ma odbicie w czasie działania. Jeśli natomiast offsety są różne, to tablice odwołują się do różnych zbiorów, przez co każdy ze zbiorów nie jest tak bardzo przeciążony i rzadziej musi wymieniać linie, co skutkuje krótszym czasem działania programu.

Zadanie 3.

Analogicznie jak w zadniu pierwszym rozmiar kafelka ma kluczowe znaczenie dla wydajności programu. W pliku wykres4.pdf znajduje się wykres średniego czasu działania programu wywołanego z argumentami

./transpose -n 15360 -v 1

gdzie $15360 = 2^{10} * 5 * 3$. Jak widać optymalnym rozmiarem kafelka ponownie jest około 8, mniejsze rozmiary powodują drobne różnice ze względu na mniej pobieranych jednorazowo ważnych danych. Zwiększanie wielkości kafelka powoduje natomiast, że nie można zmieścić całego kafelka na poszczególnych poziomach cache, co sprawia, że konieczne jest odwołanie się głębiej i strata czasu.

Widać wyraźnie, że wykres ma postać schodkową co pozwala wysnuć wniosek, że poszczególne wyraźne różnice czasowe są związane z odwołaniami do coraz to niższych poziomów cache. Innymi słowy kafelek nie mieści się na danym poziomie cache i trzeba poszczególne jego części odzyskiwać z cache niżej. Z wykresu można wyczytać, że L1 mieści kafelek o wielkości co najwyżej 8x8, L2 około 500x500, a L3 około 1100x1100. Do dokładniejszych pomiarów trzeba by testować funkcję na wielu rozmiarach tabeli i aproksymować wyniki.

Zadanie 4.

Moje rozwiązanie opiera się na rozbiciu funkcji na cztery przypadki w zależności od d i dla każdego z nich wyliczeniu operacjami bitowymi, czy d jest odpowiednie modulo 4 i czy indeks mieści się w tablicy. Praktycznym dowodem poprawności algorytmu jest zgodność wyników z funkcją randwalk1.

W funkcji ranwalk0 ciało pętli ma 56 linijek z czego 6 to skoki warunkowe. W mojej funkcji randwalk1 ciało funkcji ma 53 linikji, z których tylko 2 to instrukcje warunkowe.

Rozmiar tablicy nie ma wielkiego znaczenia dla działania programu, ponieważ w kazdej iteracji pętli możemy (choć nawet nie musimy!) przesunąć się tylko do komórki bezpośrednio obok komórki, w której obecnie jesteśmy, więc nie potrzeba zbyt wielu odwołań do pamięci i lokalność przestrzenna jest duża.

Funkcja randwalk0 dla danych

./randwalk -S 0xea3495cc76b34acc -n 7 -s 16 -t 14

działa średnio 9.251800 sekundy, podczas gdy ranwalk1 działa średnio 8.366841 sekundy, co daje około 10% poprawy wydajności czasowej na tym teście.

Zadanie 5.

Wcześniej synowie każdego wierzchołka położeni byli daleko po obu jego stronach, przez co zwłaszcza na najwyższych, najczęściej odwiedzanych poziomach kopca cały wiersz był często ładowany do cache aby odczytać z niego tylko jedną komórkę. Dodatkowo procesor musiał czekać z decyzją, którą linię załadować do cache, bo nie wiedział, którego syna będzie potrzebował. Po reorganizacji kilka pierwszych stopni drzewa mieści się w jednej linii cache, przez co można je bardzo szybko obsłużyć. Co więcej synowie każdego wierzchołka są bezpośrednio obok siebie w tablicy, przez co zazwyczaj są w tej samej linii i procesor z góry wie, którą linię trzeba ściągnąć do cache, co pozwala poprosić o nią wcześniej i przez to oczekiwanie na nią jest krótsze. Wszystko to w sumie daje około trzykrotną optymalizację czasową algorytmu (z średnio 12,298898 sekundy na 4,206384 sekundy).

Oczywiście ułożenie instrukcji w ciele funkcji heap search również może przyspieszyć działanie programu. W szczególności przy dużej liczbie zapytań kolejność if'ów ma znaczenie, ponieważ jeśli pierwszy jest if przerywający, to w wypadku, kiedy jego warunek zostanie spełniony, drugi if w ogóle się nie wykona, co daje pewną oszczędność czasową.

Zadanie 6.1.

W pierwszym podpunkcie miałem sprawdzić długość linii cache. W tym celu tworzyłem permutacje, w których w i-tym elemencie tablicy była liczba i+j dla coraz większych j. Spoglądając na wyniki dla kolejnych j miałem nadzieję zauważyć, kiedy j stanie się większe od długości linii cache. Wówczas pomiędzy każdymi dwoma zapytaniami program musiałby zmieniać linię cache, na którą aktualnie patrzy (co przy mniejszych j zdarza się tylko przy niektórych zapytaniach). Zadbawszy uprzednio, aby tablica była większa niż wielkość cache L1 (ale mniejsza niż cache L2 aby nie wprowadzać zaburzeń związanych z odwoływaniem się raz do L2 a innym razem do L3), który znam z konfiguracji (dostępna na górze raportu) powoduję, że w takich przypadkach procesor musi pobierać nową linię z L2, co generuje koszt czasowy. Innymi słowy gdy j przekroczy długość linii cache, liczba odwołań do L2 powinna znacząco wzrosnąć, co powinno być widać w czasie działania programu. Przeprowadziłem więc eksperyment uruchamiając program dla kolejnych j z argumentami:

./cache -n 15 -s 16 -t 4000

Wyniki uśredniłem i przedstawiłem w pliku wykres5, w postaci wykresu czasu od zmiennej j. Rzeczywiście widać na nim, że pomiędzy 15 a 16 następuje znaczący wzrost czasu działania. Pozwala to przypuszczać, że w i+15 mieści się w linii cache, podczas gdy i+16 już nie, czyli linia ta ma 16 elementów.

Zadanie 6.2.

W punkcie drugim mieliśmy zbadać wielkość L1 dla danych, L2 i L3. Znając wielkość linii cache stworzyłem permutację cykliczną, w której w każdym elemencie tablicy (oprócz ostatniego, w którym była liczba 1) był numer o 32 od niego większy. To pozwala uniknąć zaburzeń przy większych tablicach. Przy każdej iteracji pętli program odwołuje się dzięki temu do innej linii cache, przez co musi jej szukać od nowa. Teraz wystarczy iterować się po kolejnych wielkościach tablicy. Na początku cała będzie się mieściła w L1 i program nie będzie się musiał odwoływać do cache L2. Następnie przy rosnącym n tablica nie będzie się mieściła na kolejnych poziomach i program będzie musiał szukać potrzebnych mu linii, co będzie generowało coraz większe koszty czasowe. Jedyne o co muszę zadbać to odpowiednio duża liczba kroków, która wymusi na programie przejście przez całą tablicę co najmniej raz.

Przeprowadziłem eksperyment uruchamiając program z argumentami -s 30 -t 1 i dla kolejnych n od 1 do 27. Wyniki uśredniłem i przedstawiłem w pilku wykres6 w postaci wykresu czasu od n. Widać wyraźnie, że wykres ma postać schodkową, gdzie kolejne schodki oznaczają, że tablica nie mieści się na danym poziomie pamięci i program musiał zacząć odwoływać się do poziomu niżej, co kosztowało go znaczną ilość czasu. I tak można z tego wywnioskować, że rozmiar dostępnego dla programu:
- L1 dla danych to około $2^{13} * \text{sizeof(int)} = 32768$ B = 32 KiB
- L2 to około $2^{16} * \text{sizeof(int)} = 262144$ B = 256 KiB
- L3 to około $2^{19} * \text{sizeof(int)} = 2097152$ B = 2 MiB

Co mniej więcej pokrywa się z danymi z konfiguracji.