Analiza i modelowanie wydajności obliczeń

Lab 3. Wydajność przetwarzania operacji zmiennoprzecinkowych – opóźnienie i przepustowość

Wstęp.

- Celem laboratorium jest próba eksperymentalnego uzyskania **maksymalnej wydajności przetwarzania operacji zmiennoprzecinkowych** przez potoki przetwarzania pojedynczego rdzenia mikroprocesora.
- W ramach obecnego laboratorium badane będą rozmaite sposoby zwiększenia wydajności przetwarzania, tak aby w pełni wykorzystać możliwości rdzenia (zmaksymalizować liczbę operacji wykonywanych w jednostce czasu – a więc liczbę rozkazów kończonych w pojedynczym takcie)
- Wydajność będzie silnie zależała od wybranego kompilatora oraz opcji kompilacji. Jako podstawowy można w trakcie laboratorium wybrać kompilator *gcc*, jako najbardziej popularny. Bardziej optymalne (choć nie zawsze) wyniki mogą dawać kompilatory firmy Intel (serwer Honorata jest wyposażony w procesory Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz) *icc* i *icx* .
- Po wykonaniu każdego z punktów przy użyciu wybranego podstawowego kompilatora ***bardzo pouczające*** jest powtórzenie eksperymentów dla pozostałych kompilatorów

[na serwerze Honorata w celu korzystania z kompilatorów icc i icx należy wykonać polecenie:
source /opt/intel/oneapi/setvars.sh intel64
(Na potrzeby kolejnych laboratoriów najlepiej umieścić powyższą linijkę w pliku ~/.bashrc)]

Kroki realizowane w ramach laboratorium:

1. Pobierz ze strony przedmiotu paczkę *latency_throughput_flops.tgz*, rozpakuj w nowym katalogu *lab_03*, wejdź do podkatalogu *latency_throughput_flops*
2. Wybierz kompilator *gcc i s*prawdź inne definicje w pliku *Makefile* (konsolidator, opcje kompilacji, położenie odpowiednich katalogów z plikami nagłówkowymi i bibliotekami – najbezpieczniejszymi są zawsze ścieżki absolutne plików i katalogów)
    1. sprawdź poprawność dokonanych wyborów poprzez kompilację bibliotek wykorzystywanych w kodzie testującym wydajność przetwarzania (*make lib*)
3. Przeanalizuj plik *latency_throughput_scalar_flops.c*, szczególnie podstawową pętlę obliczeniową zawierającą operacje arytmetyczne na zmiennej a (`a = 1.000002*a+0.000002;`) oraz wydruki (razem z obliczeniami wypisywanych liczb) zawierające wynik pomiaru czasu, liczbę operacji w pętli (za pomocą zmiennej *nr_oper_local*) oraz wydajność przetwarzania zmiennoprzecinkowego w GFLOP/s
    1. przypomnij sobie, w odniesieniu do podstawowej pętli obliczeniowej, wnioski z wykładu (i ewentualnie z drugiej części poprzedniego laboratorium) dotyczące zależności pomiędzy instrukcjami, w szczególności pomiędzy instrukcjami w kolejnych iteracjach pętli
    - *Punktem wyjścia badań może być kod z drugiej części poprzedniego laboratorium (pp. 11-17), w którym realizowana była pętla operująca na pojedynczej zmiennej. W każdej iteracji zmienna była modyfikowana (przez wykonanie pojedynczego mnożenia i oddawania).*
    - *Założeniem badań wydajności, w tym i wielu innych kolejnych programach, założeniem wynikającym ze znajomości sposobu funkcjonowania rdzeni procesora, jest to, że **czas wykonania programu zależy tylko od czasu wykonania operacji arytmetycznych**, pozostałe operacje są wykonywane w tle (przez odpowiednie potoki przetwarzania)*
    - *W badanym kodzie operacje zmiennoprzecinkowe wykonywane są w taki sposób, że pomiędzy kolejnymi operacjami (na jednej tylko zmiennej) **występują zawsze zależności**, uniemożliwiając rdzeniowi wykorzystanie potokowości i innych form współbieżności.*
    - *Odbiciem zależności w kodzie źródłowym jest odpowiadający badanej pętli fragment w pliku asemblera produkowanym przez kompilatory, w którym rozkazy zmiennoprzecinkowe w pętli pracują na tym samym rejestrze.*
    - *Opisana powyżej sytuacja, w której każde dwie kolejne operacje są od siebie zależne, jest charakterystyczna dla zjawiska **opóźnienia (latency)**, czyli sytuacji kiedy rdzeń pracuje korzystając tylko z jednego potoku, i to w dużym stopniu sekwencyjnie – aby wykonać kolejną operacje na zmiennej (czyli kolejny rozkaz na rejestrze) musi czekać na skończenie poprzedniej operacji (poprzedniego rozkazu)*
4. Dokonaj kompilacji kodu (*make latency_throughput_scalar_flops*), uruchom go w terminalu i zanotuj czas wykonania pętli obliczeniowej, **podaną liczbę wykonanych operacji zmiennoprzecinkowych** (Gflops) oraz obliczoną wydajność w GFLOP/s
   Każdorazowo dokonując pomiaru wydajności należy powtórzyć obliczenia kilka razy. Jako wynik pomiaru można przyjąć średnią z wyników po odrzuceniu wyników znacznie odbiegających od

innych (co najprawdopodobniej spowodowane jest narzutami zewnętrznymi w stosunku do wykonania programu) lub *wybrać wynik odpowiadający najkrótszemu czasowi wykonania.* Szczególnie wiarygodne są wyniki, kiedy najkrótszy czas wykonania (z drobnymi, co najwyżej kilka procent) odchyleniami powtarza się wielokrotnie.

1. sprawdź wyniki wydajnościowe w przypadku rezygnacji z wstępnego rozgrzewania procesora (pętla po komentarzu // warm up)
    1. dla wielu przypadków rozgrzewka jest konieczna, żeby procesor osiągnął swoją maksymalną dostępną wydajność: podniósł częstotliwość pracy, przygotował obsługę stron pamięci wirtualnej, pamięci podręcznych itp.

5. Dla kompilatora *gcc* przeprowadź serię eksperymentów (z przywróconą rozgrzewką!) zwiększając liczbę zmiennych modyfikowanych w każdej iteracji pętli (**każdorazowo zwiększając liczbę zmiennych poprzez odkomentowanie odpowiedniej linii kodu należy także zmodyfikować wartość zmiennej *nr_oper_local - pierwszy czynnik w iloczynie to liczba odkomentowanych zmiennych!***)
    - *Pierwszym krokiem zwiększania wydajności przetwarzania rdzenia będzie wykorzystanie wielu potoków, z jednoczesnym lepszym użyciem każdego z nich. Osiągnięte zostanie to poprzez umieszczenie w tej samej pętli podobnych operacji arytmetycznych, dla coraz większej liczby zmiennych. Operacje na różnych zmiennych nie są połączone zależnościami, a więc rdzeń może przekazywać je do wykonania różnym potokom przetwarzania i wysyłać do pojedynczego potoku kolejny rozkaz, przed zakończeniem wykonania poprzedniego rozkazu na innej zmiennej.*
    - *Kod napisany jest tak, że wewnątrz pętli (dla małej liczby zmiennych) nie ma potrzeby uzyskiwania dostępu do pamięci – wartość zmiennej jest pobierana do rejestru przed pętlą i zapisywana z powrotem do pamięci po zakończeniu pętli*
    - *Zwiększanie liczby zmiennych, a więc także rejestrów użytych w pętli pozwala lepiej wykorzystywać rdzeniowi możliwości pracy współbieżnej (i związanych z nią technik ukrywania opóźnienia), a przez to zwiększać wydajność przetwarzania.*
    - *Granicą tego wzrostu może być z jednej strony wykorzystanie wszystkich możliwości sprzętowych przetwarzania przez rdzeń (wszystkich jego potoków i ich optymalnej pracy), ewentualnie wykorzystanie wszystkich dostępnych rejestrów – co powoduje, że w pojedynczej iteracji pętli, co najmniej jedna lub więcej zmiennych musi zostać zapisana do pamięci, żeby zwolnić rejestr dla innej zmiennej odczytywanej z pamięci (tzw. zjawisko ciśnienia na rejestry, register pressure).*
    - *W przypadku pojawienia się przy wykonywaniu pętli odniesień do pamięci (można to sprawdzić badając kod asemblera, patrz następny punkt) – wydajność przetwarzania przestaje rosnąć (bardzo często maleje, czasem znacząco).*

    1. wyniki pomiarów wydajności zapisz w tabelce o poniższym formacie **(tylko pierwsze 4 wiersze tabeli, dwa ostatnie są wypełniane w p. 10, po realizacji p. 9)**

        (uwaga: dla rosnącej liczby zmiennych rośnie liczba wykonanych operacji zmiennoprzecinkowych (tylko takie nas interesują - zakładamy, że wszystkie inne są wykonywane w tle i nie wpływają na czas wykonania!), należy to uwzględnić w kodzie poprzez **modyfikację obliczania wartości *nr_oper_local***, poprawne obliczenie wydajności wymaga poprawnej liczby wykonanych operacji)

        [dla poprawnej wartości *nr_oper_local* liczba wykonanych w pętli operacji zmiennoprzecinkowych (flops) jest podawana przez program]

        [eksperymenty prowadź do momentu kiedy wydajność zacznie spadać]

	Liczba zmiennych w pętli									
Liczba zmiennych modyfikowanych w pętli	1	2	4	8	10	11	12	...	...	...
Liczba wykonanych operacji * 10^9 (Gflop)										
gcc – czas realizacji obliczeń (sek.)										
gcc – wydajność Gflop/s										
gcc – liczba taktów CPU w trakcie pętli										
*gcc - IPC dla rozkazów **mul** i **add***										
gcc – wydajność flop/takt										

- **wnioski dotyczące współbieżnego (lub ewentualnie równoległego dzięki superskalarności – wielu potokom przetwarzania) wykonywania rozkazów umieść w sprawozdaniu – wyjaśnij**

szczegółowo jak zwiększając liczbę zmiennych w pętli uzyskuje się wzrost wydajności (jak wykorzystywane są potoki przetwarzania dla jednej zmiennej i dla wielu zmiennych)

---------- 3.0 ----------

2. ***Przeanalizuj kod asemblera dla wybranych przypadków (np. 1 zmienna, pierwsza wartość liczby zmiennych dla wydajności zbliżonej do maksymalnej i liczba wszystkich zmiennych w petli (17))***

[przypomnienie: kompilacja - gcc -O3 -S latency_throughput_scalar_flops.c -I../utd_time_unix otrzymujemy w wyniku plik asemblera latency_throughput_scalar_flops.s]

   - *Analiza powinna przebiegać zgodnie z punktami z poprzedniego laboratorium (znajdź pętlę odpowiadającą pętli obliczeniowej z kodu źródłowego - powinna zaczynać się od etykiety, np. .L3: , zawierać tylko kilka rozkazów i kończyć skokiem do etykiety, np. jne .L3)*
     [kompilatory Intela umieszczają informacje o numerze linii z pliku z kodem źródłowym po prawej stronie rozkazów asemblera]
   - *W kodzie asemblera zwiększanie liczby zmiennych w kodzie źródłowym będzie do pewnego momentu odzwierciedlone poprzez użycie w pętli rosnącej liczby rejestrów (każda zmienna w innym rejestrze). Kiedy dostępne rejestry zostaną wyczerpane, kompilator będzie zmuszony do zastosowania dostępów do pamięci wewnątrz pętli*
   - *Zaobserwuj ewentualne użycie rejestrów wektorowych - w badanym kodzie, jeśli w ogóle są użyte, to najprawdopodobniej w sposób nieoptymalny: do rejestru wektorowego jest pakowana tylko jedna zmienna, więc operacja na rejestrze odpowiada jednej operacji skalarnej*

- **umieść kod badanych przypadków asemblera w sprawozdaniu (oczywiście tylko kilka linijek podstawowej pętli obliczeniowej) razem z wnioskami dotyczącymi związku wykorzystania rejestrów i ewentualnych odniesień do pamięci z wydajnością obliczeń**

---------- 3.5 ----------

6. W celu zwiększenia wydajności uzyskiwanej eksperymentalnie zastosuj przetwarzanie wektorowe – plik *latency_throughput_vector_flops.c*
   - *Wadą rozważanego powyżej rozwiązania (zwiększania wydajności obliczeń poprzez zwiększanie liczby zmiennych przetwarzanych w pętli), jest praktyczne uniemożliwienie użycia przez rdzeń przetwarzania wektorowego na potrzeby programu (wprawdzie kompilatory mogą stosować rozkazy wektorowe, ale rejestry wektorowe najczęściej będą wypełnione wartością tylko jednej zmiennej z kodu – co w praktyce oznacza działanie skalarne)*
   - *Aby umożliwić rdzeniowi przetwarzanie wektorowe najprostszym rozwiązaniem jest umieszczenie przetwarzanych wartości nie w pojedynczych zmiennych, a w tablicy.*
   - *W kolejnej części laboratorium badany jest właśnie taki kod – z przetwarzanymi liczbami w pojedynczej tablicy o zmiennej długości*
   - *W przypadku odpowiedniej długości tablicy, kompilator użyje rozkazów wektorowych do pobrania danych z pamięci do rejestrów wektorowych dla kilku kolejnych wyrazów tablicy, a następnie rozkazów wektorowych do przetwarzania danych w rejestrach.*
   - *Kluczem do osiągnięcia wysokiej wydajności, podobnie jak poprzednio, jest użycie wielu rejestrów przy realizacji obliczeń w pętli (dzięki czemu jest wiele niezależnych rozkazów wysyłanych do wykonania przez potoki). Tym razem istotne jest także jakie rejestry są użyte (64-, 128-, 256-bitowe) oraz, podobnie jak poprzednio, czy są w pełni obsadzone danymi z kodu.*
   - *Zwiększanie rozmiaru tablicy o kolejne wielokrotności rozmiaru rejestrów wektorowych powinno powodować wykorzystanie do obliczeń w pętli coraz większej liczby rejestrów, a co za tym idzie coraz lepsze wykorzystanie wszystkich wektorowych potoków przetwarzania w rdzeniu*
     - ***w ramach optymalizacji kompilator stosuje rozwinięcie pętli - zamiast dokonywać obliczeń w krótkiej pętli ze względu na zmienną k, zamienia ją na sekwencję indywidualnych rozkazów - w kodzie pozostaje tylko zewnętrzna pętla po zmiennej i***
   - *Granicą wzrostu wydajności jest, również tak samo jak w poprzednim wypadku, osiągnięcie maksymalnych możliwości sprzętu (czyli potoków przetwarzania) lub pojawienie się wewnątrz pętli odniesień do pamięci, kiedy o wydajności zaczną decydować nie tylko możliwości przetwarzania rozkazów, ale także możliwości pobierania danych z pamięci (te zazwyczaj spowalniają wykonanie programu)*

1. **Założeniem eksperymentu jest wykorzystanie możliwości automatycznej wektoryzacji kodu przez kompilatory w ramach odpowiednich opcji optymalizacji (oznacza to zazwyczaj nie tylko wskazanie wysokiego poziomu optymalizacji *-O3*, ale także jawne wskazanie architektury procesora (rdzenia) determinującej jakiego typu rozkazy wektorowe można użyć, np *-march=core-avx2*)**

[*uwaga: zmiana opcji kompilacji w pliku Makefile nie powoduje automatycznej rekompilacji kodu, konieczne jest ręczne sterowanie, np. użycie sekwencji:* make clean; make]

2. **Kompilator powinien dokonać wektoryzacji** w przypadku wykonywania takiej samej operacji dla kilku zmiennych przechowywanych w sąsiednich komórkach pamięci
   1. dlatego w kodzie znajduje się pętla wykonująca takie same operacje jak w programie obliczeń skalarnych, ale tym razem na kolejnych wyrazach krótkiej, statycznie zaalokowanej tablicy

```
for(k=0; k<tab_l; k++){
      a_tab[k] = 1.00000001*a_tab[k]+0.000001;
   }
```

7. W pierwszej fazie eksperymentuj z rozmiarem tablicy a_tab i liczbą iteracji tab_l (tab_l jest równe rozmiarowi tablicy LOCAL_SIZE - to jedyny parametr, którym należy manipulować w tej części ćwiczeń). Jeśli procesor wykonujący obliczenia posiada rejestry 256-bitowe (*ymmxx*), rozmiar tablicy powinien być wielokrotnością 4 (4x64 bity dla liczb podwójnej precyzji = 256 bitów). Dokonaj kolejno kompilacji kodu dla rozmiaru 4, 24, 48, 52, itd.
8. Przeprowadź eksperymenty obliczeniowe i zapisz wyniki pomiarów wydajności w tabeli o poniższym formacie **(tylko pierwsze 4 wiersze tabeli, dwa ostatnie są wypełniane w p. 10, po realizacji p. 9)** (przerwij wykonywanie eksperymentów kiedy zaobserwujesz spadek wydajności procesora)

	Rozmiar tablicy – liczba iteracji wewnętrznej pętli				
Liczba zmiennych w tablicy	4	24	48	52	...
Liczba wykonanych operacji * 10^9 (Gflop)					
gcc – czas realizacji obliczeń					
gcc – wydajność Gflop/s					
gcc – liczba taktów CPU w trakcie pętli					
gcc - IPC dla rozkazów wektorowych ***fma***					
gcc – wydajność flop/takt					

-> wnioski dotyczące współbieżnego (lub ewentualnie równoległego dzięki superskalarności – wielu potokom przetwarzania) wykonywania rozkazów wektorowych umieść w sprawozdaniu – wyjaśnij szczegółowo jak zwiększając liczbę zmiennych w tablicy uzyskuje się wzrost wydajności (jak wykorzystywane są potoki przetwarzania dla zmieniającej się liczby zmiennych w tablicy)

---------- 4.0 ----------

9. ***Przeanalizuj kod asemblera dla wybranych przypadków obliczeń wektorowych (np. rozmiar tablic 4, rozmiar dla wydajności maksymalnej i rozmiar o 4 większy - dla ostatniego wydajność powinna być znacznie niższa)***
   [przypomnienie: kompilacja -
   gcc -O3 -march=core-avx2 -S latency_throughput_vector_flops.c -I../utd_time_unix
   otrzymujemy w wyniku plik asemblera latency_throughput_vector_flops.s]
   1. *Podobnie jak dla przypadku kodu skalarnego znajdź w kodzie asemblera rozkazy realizujące pętlę obliczeniową. W przypadku poprawnej kompilacji powinny to być odpowiednie rozkazy wektorowe wykorzystujące w pełni odpowiednie rejestry wektorowe.* ***Dla procesorów realizujących zestaw rozkazów wektorowych AVX rejestry 256-bitowe oznaczane są jako ymm0-ymm15. Kluczową kwestią dla wydajności jest to czy w pętli obliczeniowej znajdują się poza operacjami na rejestrach, także odniesienia do pamięci, czy nie. Kod bez odniesień do pamięci, pracujący tylko na rejestrach, będzie na pewno wydajniejszy.***
      [uwaga: kompilatory często korzystają z rozkazów wektorowych dla wykonywania operacji na nie w pełni obsadzonych rejestrach (np. dla badanego w poprzednich krokach kodu skalarnego częstym jest korzystanie z rejestru wektorowego z jedną tylko pozycją zajmowaną przez zmienną z kodu źródłowego). Sprawdzeniem, czy możliwości przetwarzania wektorowego są w pełni wykorzystywane, jest ustalenie czy rejestry wektorowe są w całości wypełniane danymi z kodu. Można to sprawdzić odszukując rozkazy pobierania danych z pamięci do rejestrów (np. ***vmovupd****). Dla kolejnych rejestrów wykorzystywanych przy realizacji pętli obliczeniowej, rozkazy powinny*

dotyczyć lokalizacji pamięci odległych o rozmiar rejestru wektorowego (np. dla serwera Honorata 32 bajty (256 bitów):

```
vmovupd   1248(%rsp), %ymm2
vmovupd   1280(%rsp), %ymm3
```

Gwarantuje to, że całą zawartością rejestru są dane z kolejnych komórek w pamięci, a więc kolejne wyrazy przechowywanej w pamięci tablicy

2. ***Zaobserwuj (na podstawie analizy asemblera) sposób w jaki kompilator wykorzystuje rejestry wektorowe przy zwiększaniu rozmiaru tablicy.***
   - ***Jeśli kompilator nie wykorzystał rozkazów AVX (na w pełni wykorzystanych rejestrach ymm), przeanalizuj jakie rozkazy użył w zastępstwie. Na serwerze Honorata jest możliwe (a więc konieczne do poprawnego wykonania ćwiczenia) uzyskanie kodu stosującego rejestry 256-bitowe – odpowiednie fragmenty pliku asemblera powinny znaleźć się w sprawozdaniu)***

---------- 4.5 ----------

10. Posługując się narzędziami *perf* lub *PAPI* (podejście dokładniejsze i preferowane) uzyskaj liczbę taktów procesora w trakcie wykonywania obliczeń **dla przypadków najniższej i najwyższej wydajności**, a następnie **oblicz parametry CPI i IPC dla wykorzystanych rozkazów zmiennoprzecinkowych z asemblera**
    [wersje ze zliczaniem zdarzeń za pomocą PAPI uzyskuje się przez wywołanie ***make papi*** – kompilator tworzy odpowiednie pliki binarne, dodając na końcu nazwy **_papi**]
    [wynik *perf* dla całego programu jest tylko przybliżeniem wyniku dla samej pętli obliczeniowej - należy to uwzględnić przy wykonywaniu obliczeń parametrów]
    [**korzystając z perf należy usunąć rozgrzewkę z programu, co spowoduje, że liczba taktów całego programu będzie bliższa liczbie taktów samej pętli obliczeniowej - fakt niższej częstotliwości pracy zmniejszy wydajność w Gflop/s natomiast nie powinien mieć wpływu na IPC i CPI]**
    [dla przypadku skalarnego należy **użyć odpowiedniej wartości *nr_oper_local***, zgodnie z liczbą zmiennych]
    - *W ramach badania wydajności pojedynczego rdzenia obliczane są uśrednione parametry IPC i CPI (CPI zawsze będzie traktowana jako odwrotność IPC)*
      - *uśredniona miara IPC (instructions per cycle, liczba rozkazów kończonych w pojedynczym takcie) jest obliczana dzieląc całkowitą liczbę wykonanych rozkazów przez liczbę taktów CPU w trakcie wykonywania rozkazów*
      - *liczbę taktów podczas wykonywania pętli obliczeniowych można pobrać korzystając z interfejsu PAPI, odczytując wartości odpowiedniego licznika sprzętowego przed i po pętli*
      - *można także użyć narzędzia perf, zdając sobie sprawę z niedokładności pomiaru, ponieważ perf uwzględnia nie tylko czas wykonania pętli, ale także czas innych operacji w kodzie (np. "rozgrzewki" rdzenia)*
    - *parametr IPC można obliczać w kilku wariantach*
      - *IPC dla wszystkich rozkazów procesora – najprostsza, zwracana przez perf miara, uwzględniająca wszystkie rozkazy procesora, a więc także te nieistotne z punktu widzenia wydajności przetwarzania zmiennoprzecinkowego (operacje logiczne, skoki, operacje na liczbach całkowitych)*
      - *bardziej interesująca jest miara, w której uwzględnia się tylko rozkazy zmiennoprzecinkowe, zakładając, że pozostałe wykonywane są w tle (tzn. na innych potokach przetwarzania rdzenia, bez wpływu na wydajność przetwarzania potoków zmiennoprzecinkowych) – wtedy liczbę rozkazów należy obliczyć na podstawie kodu asemblera: pomnożyć liczbę rozkazów w pojedynczej iteracji przez liczbę iteracji w pętli*
        [uwaga: liczba iteracji pętli z asemblera może być inna niż w kodzie źródłowym, za względu na optymalizację rozwinięcia pętli dokonaną przez kompilator – jako liczbę iteracji należy wtedy wziąć wartość licznika liczby skoków w programie (dla wersji PAPI to zdarzenie 3: PAPI_BR_INS]
      - *najbardziej z punktu widzenia użytkownika interesująca jest miara dla prostych operacji arytmetycznych na liczbach zmiennoprzecinkowych w kodzie źródłowym, niezależnie od tego za pomocą jakich rozkazów procesora są one realizowane – liczbę tych operacji należy obliczyć z kodu źródłowego mnożąc liczbę operacji w iteracji przez liczbę iteracji w pętli*
        (liczbę operacji można także obliczyć z kodu asemblera, odpowiednio przeliczając wykonane rozkazy asemblera na wykonane elementarne operacje arytmetyczne na pojedynczych liczbach - jednemu rozkazowi asemblera może odpowiadać do 8 operacji dodawania i mnożenia)

1. **obliczenia współczynników CPI i IPC dokonaj dla rozkazów operacji arytmetycznych w kodzie asemblera (zweryfikuj dane uwzględniając liczbę wykonanych w pętli operacji zmiennoprzecinkowych (flops) podawaną przez program)**
    1. **wyniki odpowiadają wartości opóźnienia (dla najniższej wydajności) i przepustowości (dla najwyższej wydajności) dla badanych rozkazów**
    2. **porównaj tak uzyskane dane z wartościami opóźnienia i przepustowości teoretycznej z danych producenta – czym różni się pomiar przepustowości wykonywania rozkazów od pomiaru opóźnienia wykonywania operacji wykonywanego w poprzednich punktach?**
        1. dane producenta procesora serwera Honorata: "Intel® 64 and IA-32 Architectures Optimization Reference Manual", Appendix C (mikroarchitektura Broadwell) (https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual) (uwaga: w materiałach zdarzają się błędy i nieścisłości)
        2. lepsze (zgodnie z opinią twórców) dane, eksperymentalne, znajdują się na stronie: https://www.uops.info/table.html
2. **zwróć uwagę czy rozkazy dotyczą pojedynczej operacji zmienoprzecinkowej (mul, add), czy też są zoptymalizowanymi rozkazami umożliwiającymi kończenie w pojedynczym takcie dwóch operacji**
   [w przypadku rozkazów wektorowych na rejestrach 256-bitowych (np. dla rozkazów AVX2 `vfmadd213pd`) jedna operacja FMA (fused multiply-add) odpowiada 8-miu standardowym operacjom arytmetycznym dodawania i mnożenia liczb podwójnej precyzji – CPI=1 dla `vfmadd213pd` oznacza wydajność 8 razy większą niż CPI=1 dla skalarnych rozkazów mnożenia i dodawania].
    1. **Dokonaj przeliczenia jak parametry CPI (i IPC) dla rozkazów wektorowych należy przeliczać na liczbę operacji arytmetycznych (dodawań i mnożeń pojedynczych liczb) w pojedynczym takcie.**

11. Uzyskane dane dla najwyższej i najniższej wydajności umieść w tabelkach (dla operacji skalarnych i wektorowych) - wiersze z szarym tłem
    1. w celu lepszego zilustrowania wzrostu wartości IPC w przypadku skalarnym i wektorowym przeprowadź eksperymenty dla wszystkich przypadków z tabel
    2. zaobserwuj jak początkowo przyrost wydajności jest powiązany z regularnym wzrostem wartości IPC dla odpowiednich rozkazów

[wskazówki: przykładowymi danymi z dokumentacji procesora mogą być:
- IPC dla rozkazów zmiennoprzecinkowych **`mulsd`** i **`addsd`** - 2.5 (przeciętnie 2.5 tych rozkazów zmiennoprzecinkowych kończonych w każdym takcie procesora)
- IPC dla wektorowych rozkazów zmiennoprzecinkowych **`vfmadd213pd`** - 1.5 (przeciętnie 1.5 takich rozkazów kończonych w każdym takcie procesora)
- flop/takt dla operacji zmiennoprzecinkowych dodawania i mnożenia – 12 (przeciętnie 12 operacji zmiennoprzecinkowych kończonych w jednym takcie) **[właśnie ta miara jest najbardziej przydatna do obliczania wydajności w GFLOP/s – wystarczy miarę flop/takt pomnożyć przez częstotliwość pracy rdzenia – użycie flop/takt zamiast GFLOP/s uniezależnia od zmiennej częstotliwości pracy rdzenia]**

Dalsze kroki nadobowiązkowe:

1. Zbadaj wpływ zwiększania liczby tablic o rozmiarze 16 w pętli obliczeniowej na efektywne wykorzystania rejestrów przez kompilator, a w konsekwencji na wydajność obliczeń (można użyć kodu z *latency_throughput_vector_flops.c*, ustawiając rozmiar tablicy na 16 i odkomentowując kod dla tablic **`b_tab, c_tab,`** itd.). Zaobserwuj pojawiające się w kodzie asemblera, wraz z rosnącą liczbą tablic, zjawisko "ciśnienia na rejestry" – wymagania coraz większej liczby rejestrów do efektywnej realizacji obliczeń w pętli, aż do momentu kiedy z powodu braku dostępnych rejestrów kompilator zmuszony jest do użycia dostępów do pamięci przy realizacji wewnętrznej pętli obliczeniowej
    1. przeprowadź odpowiednie eksperymenty numeryczne i zanotuj osiągnięte wydajności (uwaga: do obliczenia wydajności w GFLOP/s należy wziąć liczbę wykonanych operacji, która zmienia się wraz z liczbą uwzględnianych tablic)
    2. spróbuj dokonać optymalizacji rozdzielenia pętli (*loop fission*) dla przypadku 6 tablic – zaobserwuj czas realizacji obliczeń i wydajność przetwarzania przed i po optymalizacji

2. Zbadaj efekt tzw. rozpychania (rozciągania?,wypełniania?, watowania?) tablic (*array padding*).
    1. Wykorzystaj najbardziej optymalną wersję przetwarzania wektorowego z wieloma tablicami (na serwerze Honorata - 3 tablice o rozmiarze 16)
    2. Załóż, że aplikacja wymaga zastosowania tablic mających tylko 15 elementów i wykonania tylko 15 iteracji w pętli wewnętrznej).
    3. Zmodyfikuj odpowiednio kod – rozmiar alokowanych tablic i liczbę iteracji w pętli wewnętrznej
    4. Skompiluj i uruchom program – zaobserwuj uzyskaną wydajność
        1. Przeanalizuj kod asemblera – zaobserwuj zmiany w stosunku do wersji poprzedniej: co powoduje zmniejszenie wydajności?
    5. Zastosuj rozpychanie tablic – zaalokuj większe tablice (rozmiar 16) i przeprowadź więcej obliczeń (16 iteracji wewnętrznej pętli), powracając do pierwotnego kodu i czasu wykonania
    6. Zwróć uwagę, że na potrzeby aplikacje są teraz wykonywane operacje tylko na 15 elementach tablic, dodatkowe operacje są bezużyteczne – służą tylko optymalizacji, dzięki nim czas wykonania programu skraca się w stosunku do wykonania z naturalnym z punktu widzenia aplikacji rozmiarem 15 (mimo większej liczby wykonanych operacji przez procesor).
    7. Zmodyfikuj sposób obliczania liczby operacji, tak żeby odpowiadał efektywnej liczbie operacji na potrzeby aplikacji, a nie liczbie operacji wykonanych przez procesor
    8. W sprawozdaniu umieść czasy wykonania i wydajności dla wersji bez rozciągania i z rozciąganiem tablic oraz odpowiednie fragmenty kodu asemblera z komentarzem dotyczącym wydajności

Sprawozdanie:

1. Zrealizowane kroki
2. Tabelki z parametrami wydajnościowymi wykonania kodu
3. Wnioski, fragmenty kodu dodanego do pobranych procedur lub samodzielnie zmodyfikowanego, fragmenty kodu asemblera odpowiadające analizowanym pętlom obliczeniowym, zrzuty ekranu z czasami wykonania i wydajnościami, itp. itd. zgodnie z regulaminem laboratoriów

Zawartość sprawozdania (zależnie od zrealizowanych kroków, najlepiej wplecione w opis wykonanych kroków):

1. Zadanie 1 – *latency_throughput_scalar_flops*
    1. kod pętli podstawowej (dla jednego wybranego przypadku z kilkoma np. 3,4 zmiennymi) – wydruk, analiza zależności
    2. sposób obliczania wydajności – wzór, skąd biorą się składowe wzoru
    3. tabelka wyników wydajności
    4. wnioski na podstawie analizy kodu oraz pomierzonych i obliczonych wartości: jak użycie wielu zmiennych zmienia sposób wykorzystania potoków przetwarzania w przypadku zależności dla pojedynczej zmiennej?, jak wpływa to na wydajność?, jakie są ograniczenia dla takiej wydajności?
    ---
    5. *analiza kodu asemblera dla wybranych wariantów pętli podstawowej - wnioski*
    ---
    6. *analiza CPI i IPC dla rozkazów zmiennoprzecinkowych z asemblera - wnioski*

2. Zadanie 2 - *latency_throughput_vector_flops*
    1. kod pętli podstawowej – wydruk, analiza zależności (lub ich braku)
    2. sposób obliczania wydajności – wzór, skąd biorą się składowe wzoru
    3. tabelka wyników wydajności
    4. wnioski na podstawie analizy kodu oraz pomierzonych i obliczonych wartości: jak zastosowanie rosnącego rozmiaru tablicy zmienia sposób wykorzystania rejestrów wektorowych i wektorowych potoków przetwarzania?, jak wpływa to na wydajność?, jakie są ograniczenia dla takiej wydajności?
    ---
    5. *analiza kodu asemblera dla wybranych wariantów pętli podstawowej - wnioski*
    ---
    6. *analiza CPI i IPC dla wektorowych rozkazów zmiennoprzecinkowych z asemblera - wnioski*

3. *Dalsze kroki nadobowiązkowe – format dowolny*