Analiza i modelowanie wydajności obliczeń

Lab 09. Mnożenie macierz-macierz

Cel: Analiza wydajności i optymalizacja algorytmu mnożenia macierz-macierz.

Kroki:

1. Rozpakuj paczkę *matrix_multiplication.tgz* w nowym katalogu np. lab_09.
2. Uruchom kod w każdym z katalogów *matrix_multiplication_gcc* i *matrix_multiplication_icc* przez wykonanie polecenia ***make***
   [standardowo dla skorzystania z kompilatora Intela icc należy wywołać w terminalu:
   source /opt/intel/oneapi/compiler/latest/env/vars.sh intel64
   a na potrzeby lab 09 dodatkowo także
   source /opt/intel/oneapi/mkl/latest/env/vars.sh intel64
   w celu użycia biblioteki Intela MKL (Math Kernel Library)]
   [w celu uniezależnienia się od zmiennej częstotliwości pracy procesora zakłócającej porównania wyników wydajnościowych, kod oprócz wyników w GFlop/s podaje także wyniki w flop/takt - wymaga to wywołania odpowiednich funkcji z interfejsu PAPI (odpowiednie pliki w katalogach matrix_multiplication_... *tworzone są za pomocą standardowego* **`make recreate_papi_lib_mth`** *)]*
3. W każdym z katalogów wykonywane są te same wersje algorytmu mnożenia macierz-wektor skompilowane przez różne kompilatory. Algorytmy zawarte są w trzech funkcjach:
   1. w pliku *mat_mul_par.c* znajduje się wersja "zdroworozsądkowa" realizująca wzór matematyczny, z korektą zmieniającą kolejność pętli, tak żeby wyeliminować dostęp do pamięci ze skokiem *n* w najbardziej wewnętrznej pętli algorytmu
   2. w pliku *mat_mul_avx_1.c* zawarta jest wersja z ręcznym rozwinięciem wszystkich trzech pętli o czynnik 4, tak żeby umożliwić wykorzystanie rejestrów i operacji wektorowych. Operacje wykonywane są przez wywołanie tzw. *intrinsics* , funkcji dających się wywoływać ze standardowego kodu w C, realizujących konkretne rozkazy procesora (możliwości realizacji funkcji *intrinsics* zależne są od kompilatora, ich nagłówki znajdują się w plikach *immintrin.h* i *emmintrin.h*)
   3. plik *mat_mul_avx_2* zawiera jeszcze jeden wariant rozwinięcia pętli i wykorzystania rejestrów oraz rozkazów wektorowych – tym razem o czynniki 4, 12 i 4 (dla kolejnych pętli po *i, j* i k)

   Wersje dla różnych kompilatorów realizują te same funkcje w plikach *mat_mul_....c* (różnią się opcjami kompilacji i szczegółami alokacji tablic) mogą jednak różnić się wydajnością – częściową odpowiedź na pytanie dlaczego, może dać analiza wyprodukowanego przez kompilatory asemblera.
   [Uwaga: jako że wersje algorytmu mnożenia macierz-macierz dla różnych kompilatorów są identyczne, optymalizacji można dokonywać dla jednego kompilatora, a następnie kopiować plik do właściwego katalogu i dokonywać pomiarów dla drugiego kompilatora]
4. Kod zarządzający wykonaniem całości programu znajduje się w pliku *mat_mul_driver.c*. Funkcja *main* dokonuje alokacji tablic biorących udział w obliczeniach (z wyrównaniem na granicy będącej wielokrotnością rozmiaru linii pamięci podręcznej – w kodzie założono wartość ALIGNMENT równą 64). Rozmiary tablic są tak dobrane, żeby były wielokrotnościami rozmiarów bloków używanych przy optymalizacji *cache blocking* i *register blocking.*
   Funkcja *main* wywołuje kolejno funkcje *mat_mul_par, mat_mul_avx_1* i *mat_mul_avx_2*, mierząc ich czas wykonania i sprawdzając poprawność. Na początku funkcja *main* woła także funkcję BLAS poziomu 3 *dgemm* , która zostanie wykonana przez **implementację w dołączonej do kodu bibliotece** (w plikach *Makefile* założone dołączenie implementacji w bibliotece MKL firmy Intel).
   - wywołanie wysoko zoptymalizowanej funkcji bibliotecznej ma na celu z jednej strony pokazanie jak daleko od możliwego do uzyskania optimum znajdują się badane implementacje, a z drugiej do sprawdzenia, czy wykonanie nie jest zaburzane przez zewnętrzne, nieuwzględniane czynniki – wydajność wersji *dgemm* w bibliotece MKL dla stałego rozmiaru zadania powinna być taka sama przy każdym uruchomieniu (i oczywiście niezależna od optymalizacji dokonywanych w innych funkcjach programu)

- *[dla serwera Honorata wydajnośc funkcji bibliotecznej na pojedynczym rdzeniu powinna przekraczać 13 flop/takt (co dla częstotliwości pracy np. 2.8 GHz daje wydajności powyżej 36 Gflop/s)]*
- w celu optymalizacji użycia zasobów serwera, uruchomienie każdej z wersji w funkcjach *mat_mul_par, mat_mul_avx_1* i *mat_mul_avx_2* wymaga zdefiniowania odpowiedniego symbolu (linie 17-20 w pliku *mat_mul_driver.c*). Testując wybraną wersję implementacji należy uruchamiać tylko tę wersję, pozostawiając pozostałe symbole niezdefiniowane (wersja MKL powinna być uruchomiona zawsze, w celu weryfikacji poprawności działania pozostałych implementacji)

5. Obliczenia domyślnie są realizowane w wersji jednowątkowej (dyrektywy zrównoleglające, które będą wykorzystywane w kolejnych laboratoriach, są wykomentowane).
   *[w celu jawnego przeprowadzenia obliczeń na pojedynczym, konkretnym rdzeniu mikroprocesora (co wyklucza migrację na inny rdzeń spowalniającą obliczenia oraz rywalizację o rdzenie) obliczenia przeprowadzane są pod kontrolą narzędzia **numactl** (w pliku **Makefile** istnieje symbol **MY_CORE** , który należy zdefiniować jako równy numerowi indywidualnie przydzielonego rdzenia serwera, oraz polecenie wykonania kodu (w ramach **make run**) uruchamiające kod na konkretnym rdzeniu:*
   `numactl -C $(MY_CORE) ./mat_mul_driver_papi.exe` *]*
6. Zapisz wyniki wydajnościowe dla oryginalnych wersji procedur *mat_mul_par, mat_mul_avx_1* i *mat_mul_avx_2*
   [*Uwaga: plik* sizes.h *zawiera definicje stałych determinujących zachowanie algorytmu:*
   1. *SCALAR – określająca typ zmiennych w tablicach, ustawiona na double, zmiana kodu na mierzący wydajność dla typu float następuje przez zmianę tej wartości (dotyczy tylko funkcji mat_mul_par nie korzystającej z 256-bitowych funkcji intrinsics, dla której porównanie ze zmienioną wartością dokładności można przeprowadzić z biblioteczną funkcją sgemm) ,*
   2. ***BLOCK_SIZE_S i BLOCK_SIZE_L, które należy użyć do określania wielkości bloków zastosowanych w cache blocking, (BLOCK_SIZE_L powinien być wielokrotnością BLOCK_SIZE_S, a z kolei rozmiar tablic jest w programie dostosowywany tak aby stanowić wielokrotność BLOCK_SIZE_L)***
   3. *ALIGNMENT, określająca wyrównanie zadane kompilatorowi – można tu użyć rozmaitych wielokrotności 4*
      W przesłanej paczce wielkości BLOCK_SIZE_S i BLOCK_SIZE_L są sobie równe, a rozmiar tablic tak dobrany, żeby dać rozsądne czasy obliczeń. Manipulując parametrami można zwiększać rozmiar tablic – im większe tablice, tym niektóre z algorytmów mogą uzyskiwać lepsze wydajności (na pewno dotyczy to implementacji dgemm w MKL).]
7. Dokonaj optymalizacji *cache blocking* dla funkcji *mat_mul_par* (należy potrójną pętlę po *i,k,j* zamienić na dwie potrójne pętle, gdzie w zewnętrznych pętlach dokonuje się iteracji po blokach macierzy, a w wewnętrznych pętlach obliczeń dla pojedynczego bloku – można posłużyć się wzorcem podanym na wykładzie).
   1. **rozmiar bloku BLS należy przyjąć równy parametrowi BLOCK_SIZE_S z pliku *sizes.h* i wszelkich zmian dokonywać wyłącznie w pliku *sizes.h***

   [*Uwaga: za względu na postać funkcji mat_mul_avx_2 rozmiar bloku powinien być wielokrotnością 12.*]
   [*Uwaga: wersje niezoptymalizowane zachowaj w tym samym pliku (zakomentowane) lub w innych plikach, tak żeby można było wrócić do nich w celu przeprowadzenia dalszych badań.*]
8. Przeprowadź obliczenia dla różnych rozmiarów bloku (wybierz co najmniej 3 rozmiary: bloki w całości w L1, w L2 i w L1+L2), znajdź rozmiar (BLS_OPT) dający najlepszą wydajność dla używanej maszyny. Spróbuj uzasadnić jego optymalność.
   - powtórz obliczenia dla BLS_OPT-12 i BLS_OPT+12 - porównaj wyniki wydajnościowe

   [Uwaga: Należy pamiętać, że dobierając rozmiar bloku BLS do konkretnej pamięci podręcznej powinno się założyć, że pamięć ma pomieścić 3xBLSxBLSxsizeof(SCALAR) bajtów (co oznacza np. dla pamięci L1 mającej 32kB, wartości mniejsze od 37). Dodatkowo należy uwzględnić, czy pamięć podręczna w trakcie obliczeń będzie współdzielona przez

różne wątki, czy będzie wyłączna dla wątku – liczbę wątków i rozmiar bloku należy dopasować do siebie.]
[Uwaga: w jednowątkowej wersji kodu przyjęty mały rozmiar macierzy (1080) może oznaczać, że w całości mieści się w pamięci podręcznej L3 - nie należy przeprowadzać wtedy optymalizacji cache blocking *dla L3]*

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

9. Korzystając z materiałów z wykładu oraz wzorca wektoryzacji w pliku *mat_mul_avx_1.c* przeprowadź wektoryzację w pliku *mat_mul_par.c* – dla jednej operacji wektorowej ***fma***
    1. *pierwszym krokiem może być ręczne rozwinięcie najbardziej wewnętrznej pętli po **j** o czynnik **4** – obliczenia będą przeprowadzane dla 4 kolejnych wartości tablicy **c** i 4 kolejnych wartości **b** oraz jednej wartości tablicy **a***
    2. *następnie należy spakować 4 wyrazy **c** do jednego rejestru wektorowego, 4 wyrazy **b** do jednego rejestru wektorowego oraz "rozgłosić" wyraz **a** do jednego rejestru wektorowego*
    3. *na tak otrzymanych 3 rejestrach należy wykonać jedną operacje **fma***
    4. *otrzymaną wersję należy zweryfikować pod katem wektoryzacji przeglądając wyprodukowany dla niej asembler, a następnie sprawdzić jej wydajność w testach dla różnych rozmiarów bloków*
    5. *ewentualne dalsze kroki mogą być związane z wprowadzaniem nowych rejestrów (zgodnie z materiałami z wykładu) i przeprowadzanie powyższych operacji (pakowanie, rozgłaszanie, fma) na kolejnych rejestrach, najlepiej według wzorca* register blocking

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

10. Korzystając z wzorca optymalizacji *cache blocking* w funkcji *mat_mul_par* , przeprowadź optymalizację *cache blocking* w funkcji *mat_mul_avx_1*.
11. Dokonaj kolejnych obliczeń sprawdzając osiągniętą wydajność dla różnych rozmiarów bloków - podobnie jak w p.8

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

12. Ostatnie zadanie polega na próbie dokonania optymalizacji *cache blocking* w funkcji *mat_mul_avx_2*. Kod funkcji (podobnie jak kod w *mat_mul_avx_1*) jest tak napisany, żeby stanowić wskazówkę, jak dokonać optymalizacji. Optymalizacja w *mat_mul_avx_2* może dotyczyć dwóch rozmiarów bloków, dla różnych poziomów pamięci *cache.*
    1. *przeprowadzając testy implementacji należy pamiętać o zasadach z punktu 7, dodatkowo uwzględniając fakt, że rozmiar dużego bloku musi być wielokrotnością małego bloku – jako początek można przyjąć rozmiar małego bloku dopasowany do pamięci L1 (np. 36 jako wielokrotność 12), a rozmiar dużego bloku jako 2,3,4 razy większy od małego bloku*
13. Dokonaj obliczeń dla ostatecznie zoptymalizowanej wersji kodu, sprawdzając osiągniętą wydajność dla różnych rozmiarów bloków. Wyniki jak zwykle umieść w sprawozdaniu z odpowiednimi komentarzami i wnioskami.

Dalsze kroki (6.0):

1. Korzystając z rozmaitych informacji w sieci (m.in. artykuł na stronie przedmiotu) spróbuj dokonać dalszych optymalizacji algorytmu mnożenia macierz-macierz.

Sprawozdanie:

1. Zrealizowane kroki, najważniejsze fragmenty modyfikowanego kodu (a także ewentualnie uzyskiwanego asemblera), spostrzeżenia z analizy kodu (i ewentualnie odpowiadającego kodu asemblera), tabele, wykresy, zrzuty ekranu, opisy, wnioski - zgodnie z regulaminem laboratorium