AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

PROJEKT DYPLOMOWY

Mnożenie macierz-wektor dla macierzy rzadkich na procesorach z jednostkami wektorowymi
Sparse matrix-vector product on CPUs with vector units

Autor: Malwina Paulina Cieśla
Kierunek studiów: Inżynieria Obliczeniowa
Opiekun pracy: Dr hab. inż. Krzysztof Banaś

Kraków, rok 2022

Spis treści

1. Wstęp

W świecie owładniętym technologią duże znaczenie ma szybkość rozwiązywania problemów oraz wydajność obliczeń komputerowych. Wielokrotnie w ciągu dnia ludzie korzystają z niezawodnych algorytmów obliczeniowych, np. dostępnych w wyszukiwarce Google, nie zdając sobie sprawy jak implementacja użytych algorytmów może mieć znaczenie w prędkości uzyskiwania wyników.

Motywacją do realizacji tematu pracy była potrzeba szybkiego uzyskiwania wyników mnożenia macierz-wektor dla macierzy rzadkich, poprzez wydajną implementację algorytmu i użycie mikroprocesorów z jednostkami wektorowymi. Niniejsza praca głównie skupiać się będzie na architekturze mikroprocesorów, macierzach rzadkich oraz na algorytmie mnożenia macierz-wektor. W pracy pojawi się również tematyka obliczeń równoległych oraz wektoryzacji kodów. Poniżej przedstawiony został plan niniejszej pracy:

W drugim rozdziale skupiono się na wprowadzeniu trzech podstawowych grup informacji:

- zdefiniowanie algorytmu mnożenia macierz-wektor,
- przedstawienie formatów macierzy rzadkich wraz z zaprezentowaniem występujących między nimi różnic,
- opisanie architektur komputerów równoległych, obliczeń równoległych na maszynach z pamięcią wspólną i rozproszoną oraz związanego z tymi pierwszymi standardu programowania OpenMP.

W trzecim rozdziale zawarto implementację kodu funkcji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy rzadkich, wraz z implementacjami konwersji dla używanych w programie formatów macierzy. W kolejnym rozdziale zaprezentowano testowanie kodu dla jednej macierzy rzadkiej. W ostatnim rozdziale przedstawione zostały wnioski dotyczące uzyskanych wyników oraz podsumowanie pracy.

2. Mnożenie macierz-wektor dla macierzy rzadkich i obliczenia równoległe

2.1 Mnożenie macierz-wektor

W celu realizacji mnożenia należy zastosować wektor o rozmiarze N, gdzie N – liczba wierszy wektora oraz macierz A o wymiarach MxN, gdzie M przedstawia liczbę wierszy, a N – liczbę kolumn (rozmiar wektora równy jest liczbie kolumn używanej w mnożeniu macierzy). Mnożenie macierz-wektor można przedstawić w sposób matematyczny jako Y=AX. Po wymnożeniu macierzy A i wektora X otrzymany zostanie wektor Y o wymiarze M. Mnożenie to można również zapisać jako sposób realizacji odwzorowania z przestrzeni wektorowej N elementowej do przestrzeni wektorowej M elementowej. Podstawowy algorytm mnożenia macierz-wektor zapisany w reprezentacji tablicowej w języku C przedstawiony został poniżej:

```
for (int i = 0; i < liczba_wierszy; i++){
        for ( int j = 0; j < liczba_kolumn; j++){
                y[i]+=A[i][j]*x[j];
        }
}
```

Jeżeli zewnętrzną pętlą jest pętla po kolumnach, a wewnętrzną po wierszach, wtedy jedynie zakres zmiennych „i” oraz „j” ulegnie zmianie.

2.2 Macierze rzadkie

Macierze wykorzystywane w zdefiniowanej wyżej operacji mnożenia podzielić można na macierze: standardowe, inaczej zwane gęstymi oraz rzadkie. Macierz uważana jest za gęstą, gdy w występujących wartościach znajduje się niewielka liczba elementów zerowych. Natomiast macierz rzadka jest to rodzaj macierzy, w którym większość elementów stanowią zera. Nie istnieje żadna granica dzieląca macierz na rzadką lub gęstą, a zastosowanie nazewnictwa i algorytmów właściwych dla odpowiednich rodzajów macierzy jest stosowane w zależności od zadania oraz programisty, dlatego też rozmiary i ilość niezerowych elementów będących w macierzy jest tak różna.

2.3 Formaty przechowywania macierzy rzadkich

Z uwagi na fakt, że większość występujących w macierzy rzadkiej elementów to zera, przechowywanie wszystkich wartości zwiększałoby w sposób drastyczny miejsce zajmowane przez macierz w pamięci i czas wykonywanych na niej operacji arytmetycznych. W celu zminimalizowania tych czynników stworzono dodatkowe formaty przechowywania macierzy. Główne formaty wykorzystywane w zadaniach to:

- COO (ang. *coordinate format* – format współrzędnych) jest to format, w którym do zapisu macierzy używa się trzech tablic: ***row***, ***col***, ***data***. Tablice te odpowiednio przechowują indeksy wierszy, kolumn i dane niezerowych elementów macierzy. Ten format jest ogólnym formatem przechowywania macierzy rzadkich, ponieważ wymagana pamięć do przechowywania danych wraz z indeksami wierszy i kolumn w tym formacie jest zawsze proporcjonalna do liczby niezerowych elementów znajdujących się w macierzy. Dodatkowo w odróżnieniu od innych formatów COO przechowuje zarówno indeks wiersza jak i kolumny.

- CSR (ang. *compressed sparse row format* – skompresowany wierszowo format macierzy rzadkich) jest to najbardziej popularny format zapisu macierzy rzadkiej. Ten format jawnie przechowuje indeksy kolumn i wartości niezerowe w tablicach: **col_ind** i **dane**. Dodatkowo używa trzeciej tablicy **ptr**, która przechowuje początkowy indeks w tablicach **coli_ind** i **dane** dla każdego wiersza w macierzy rzadkiej (wskaźnik wierszy). Dla macierzy MxN, **ptr** ma rozmiar M+1 i zapisuje indeks początkowego elementu i-tego wiersza w **ptr[i]**. Dzięki temu ostatnia wartość w tablicy **ptr** przedstawia całkowitą liczbę niezerowych elementów. Format CSR uważany jest za naturalne rozszerzenie COO utworzone przy użyciu tablic, gdzie jedna występuje w zmniejszonym rozmiarze. W ten sposób CSR może zminimalizować zapotrzebowanie na pamięć tak operacyjną w trakcie obliczeń, jak i masową do trwałego przechowywania danych. Na dodatek wprowadzona tablica **ptr** ułatwia szybkie odczytywanie wartości macierzy wraz z innymi ważnymi wartościami, takimi jak liczba niezerowych elementów w określonym rzędzie.

- CSC (ang. *compressed sparse column format* – skompresowany kolumnowo format macierzy rzadkich) jest to mniej popularny format niż CSR. W tym formacie wartości macierzy przechowywane są kolumnami. W sposób jawny przechowywane są indeksy wierszy i wartości niezerowe w tablicach: **row_ind** i **dane**, a w tablicy **col_ptr** przechowywany jest początkowy indeks każdej kolumny w macierzy rzadkiej (wskaźnik kolumn). Rozmiar **col_ptr** jest wyliczany tak samo jak w formacie CSR, czyli dla macierzy

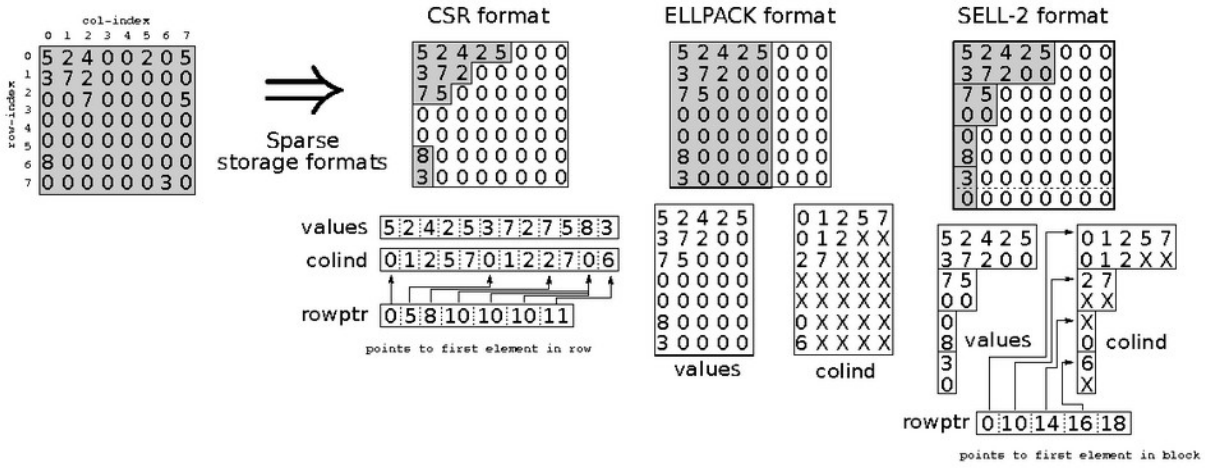
MxN przyjmuje wartość M+1. Dzięki temu ostatnia wartość tej tablicy przechowuje wartość liczby niezerowych elementów. Format ten jest wydajny w przypadku niektórych operacji arytmetycznych na macierzach.

- ELL (ELLPACK) jest to format stworzony, by zwiększyć wydajność mnożenia macierz-wektor dla obliczeń na procesorach graficznych GPGPU (ang. *General Purpose Graphics Processing Units*). W sytuacji, gdy dostępna jest macierz o rozmiarze MxN z maksymalnie P wartościami niezerowymi na wiersz, format ELL przechowuje macierz rzadką danych w tablicy o wymiarach MxP, w której wiersze mające mniej niż P elementów są dopełniane do rozmiaru P zerami. Inne indeksy opisanej struktury danych przechowują indeksy kolumn i również są uzupełniane zerami w taki sam sposób, jak dane. Ponieważ format ELLPACK dopełnia do rozmiaru P wszystkie wiersze zerami, może to powodować znaczne zwiększenie ilości zajmowanego miejsca do przechowywania danych. Jako rozwiązanie zaproponowano wariant o nazwie Sliced ELPACK, który znacznie zmniejsza ten parametr.
- SELL (Sliced ELLPACK) jest rozszerzeniem formatu ELL stworzonym poprzez podzielenie macierzy wejściowej na bloki wierszy (paski ang. *slices*) o wysokości C. Każdy pasek jest przechowywany w formacie ELL (kolumnowo), a liczba wartości niezerowych przechowywanych w każdym wierszu może się różnić pomiędzy blokami. Dodatkowo, aby dopasować długość pozostałych wierszy do najdłuższego wiersza w pasku, są one wypełnione zerami, dzięki temu znacznie zostaje zmniejszony rozmiar w porównaniu do formatu ELLPACK. Jeżeli wszystkie rzędy w pasku nie mają jednakowej długości, nadal może wystąpić znaczne zwiększenie objętości danych w stosunku do np. formatu CSR. W formacie SELL wykorzystywane są trzy tablice: **val**, **colind** oraz **rowptr** (również znany pod nazwą **slice_start**). Wartości występujące w tablicy **colind** dla miejsc, które zostały dopełnione zerami nie przedstawiają żadnej wartości. Tablica **slice_start** działa na tej samej zasadzie co tablica **ptr** w formacie CSR, jednakże tu również sumowane są zera, które występują jako dopełnienie do długości wiersza. W ten sposób ostatni element tej tablicy nie przedstawia całkowitej liczby niezerowych elementów lecz całkowitą liczbę elementów występujących w tablicy **val**.

W celu rozszerzenia formatu ELLPACK jak również w formacie Sliced ELLPACK zmodyfikowano format SELL, dzięki czemu powstał format nazywany „SELL-C-σ”. Format ten sparametryzowany jest poprzez rozmiar kawałka C i „zakres sortowania” σ. Oznacza to, że wybrane zostaje sortowanie wierszy macierzy w σ kolejnych wierszach. Prowadzi to do stosowania niezależnego od sprzętu formatu macierzy rzadkiej, który może zapewniać wysoką wydajność dla

rozmaitych wykorzystywanych macierzy na wszystkich platformach sprzętowych. Zazwyczaj zakres sortowania σ jest wybierany jako wielokrotność C. Ponadto liczba wierszy macierzy N musi być dopełniona do wielokrotności C (również tu stosowane jest dopełnienie zerami). Przy wyborze parametrów należy również pamiętać, że sortowanie wierszy według liczby niezerowych wpisów w ograniczonym zakresie sortowania σ wierszy zmniejsza rozmiar schematu i poprawia wydajność na wszystkich architekturach tylko jeśli parametr σ nie jest zbyt duży[1].

Porównanie wymienionych powyżej formatów CSR, ELL, SELL dla C=2 zostało przedstawione na macierzy 8x8 posiadającej 12 niezerowych elementów. Porównanie to zobrazowane zostało na ilustracji nr 1:



Ilustracja 1: Porównanie formatów przechowywania macierzy rzadkiej

Źródło: [2]

2.4 Mnożenie macierz-wektor dla macierzy rzadkich

Mnożenie macierz-wektor dla macierzy rzadkich (ang. *Sparse matrix-vector multiplication, spMVM*) uznawane jest za najbardziej czasochłonną pocedurę w wielu algorytmach numerycznych i było już szeroko badane na wszystkich nowoczesnych architekturach procesorów i akceleratorów. Możenie to uznawane jest za podstawową operację wykorzystywaną przy iteracyjnym rozwiązywaniu układów równań liniowych. Rozwiązywanie układów równań liniowych pojawia się również w programach z aproksymacją równań różniczkowych metodą elementów skończonych symulacji. Występuje tam wielokrotne rozwiązywanie układów równań liniowych, dzięki czemu problem mnożenia macierz-wektor dla macierzy rzadkich jest tak ważny.

2.5 Architektury komputerów równoległych i równoległe mnożenie macierz-wektor

Równoległe systemy komputerowe można sklasyfikować posługując się taksonomią Flynna. W praktyce pojawiają się trzy typy architektur:

- SISD (ang. *Single Instruction Single Data*) – rodzaj ten przedstawia klasyczną architekturę Von Neumanna – komputer w tej architekturze składa się z czterech głównych komponentów: pamięci komputerowej służącej do przechowywania danych programu i instrukcji programu, jednostki sterującej, która nadzoruje pobieranie danych i instrukcji z pamięci, jednostki arytmetyczno-logicznej, która odpowiada za wykonanie podstawowych operacji arytmetycznych oraz z urządzeń wejścia/wyjścia, które służą do interakcji z operatorem. Jednostka sterująca wraz z jednostką arytmetyczno-logiczną tworzą procesor. Architektura ta dodatkowo posiada wspólną przestrzeń adresową dla pamięci kodu i danych oraz wspólne magistrale, a instrukcje jak i dane są kodowane w postaci liczb,
- *SIMD (ang. Single Instruction Multiple Data)* – architektura, gdzie dostępny jest jeden ciąg instrukcji i wiele ciągów danych (pojedyncza instrukcja dotyczy wielu egzemplarzy danych). Współcześnie określenie to odpowiada m.in. wydajnemu przetwarzaniu równoległemu wykorzystującemu jednostki wektorowe procesorów, w których do jednego rejestru wektorowego jawnie pakuje się kilka liczb. Następnie wykonywana jest jedna operacja na rejestrze. Celem projektowania potoków przetwarzania wektorowego w rdzeniach procesorów jest osiągnięcie sytuacji, gdzie czas wykonania operacji w sposób wektorowy na całym rejestrze jest taki sam jak na pojedynczej liczbie (może to pozwalać na kilkukrotne przyspieszenie wykonania w stosunku do kodu, który nie korzysta z rejestrów wektorowych).
- MIMD (ang. *Multiple Instruction Multiple Data*) – w architekturze tej istnieje wiele ciągów instrukcji i danych, wiele jednostek może wykonywać operacje asynchronicznie i niezależnie, a procesory (rdzenie) wykorzystują pamięć wspólną lub model rozproszony[3].

W pracy zostały wykorzystane architektury SIMD oraz MIMD. Architekturą MIMD z pamięcią wspólną jest serwer z mikroprocesorami wielordzeniowymi, dla którego w pracy przeprowadzone zostały testy wydajności.. Do architektury SIMD należą wykorzystywane w kodzie rejestry wektorowe i potoki przetwarzania wektorowego.

Dla architektury MIMD kod napisany jest w modelu SPMD (ang. *Single Program Multiple Data*) – wiele wątków wykonuje ten sam plik binarny.

2.6 Obliczenia równoległe

Obliczenia równoległe są formą wykonywania obliczeń, w której wiele instrukcji wykonywanych jest jednocześnie w celu rozwiązania pojedynczego zadania. W celu zaprojektowania algorytmu rozwiązującego dany problem w sposób równoległy należy określić jakie operacje mogą być wykonywane na danych, sposób gromadzenia, przechowywania oraz przetwarzania danych i wyników. W programowaniu równoległym mogą pojawić się również problemy ze współbieżnością. Głównym problemem występującym w programach jest wyścig (ang. *race condition)* występujący, gdy wątki współbieżnie korzystają z zasobu współdzielonego. Wyścig powoduje brak deterministycznego wykonania programu. W celu przeciwdziałania wyścigowi wprowadza się sekcje krytyczne i wzajemne wykluczanie. Sekcja krytyczna przedstawia fragment kodu, gdzie wykorzystywany jest zasób dzielony – zasób ten powinien być wykorzystywany w danej w chwili tylko przez jeden wątek. Do bezpiecznego wykonania sekcji krytycznej wykorzystać można wzajemne wykluczanie realizowane przez procesy lub wątki, w postaci protokołów wejścia do i wyjścia z sekcji krytycznej. Inną z technik synchronizacji eliminującą wyścig jest zastosowanie tzw. operacji atomowych (niepodzielnych). Należy jednak pamiętać, że synchronizacja może prowadzić do problemów takich jak zakleszczenie (w tej sytuacji wątki czekają na siebie wzajemnie przez co żaden nie może kontynuować działania) lub zagłodzenie wątku (pojedynczy wątek nie ma możliwości zakończenia działania, choć inne pracują)[4].

2.7 Programowanie systemów z pamięcią wspólną – specyfikacja OpenMP

W niniejszej pracy wykorzystana została specyfikacja OpenMP, która obsługuje wieloplatformowe programowanie równoległe z pamięcią współdzieloną. OpenMP definiuje przenośny, skalowalny model z prostym i elastycznym interfejsem do tworzenia aplikacji równoległych na używanych architekturach procesora[5]. Podstawowym elementem w modelu programowania OpenMP są dyrektywy kompilatora, standard definiuje także szereg funkcji wspomagających realizacje równoległą obliczeń.

Specyfikacja ta używa formatu „**#pragma omp nazwa_dyrektywy lista_klauzul**”, po którym występuje znak nowej linii. Główną dyrektywą jest „**#pragma omp parallel**”, która jawnie

nakazuje zrównoleglenie wybranego bloku kodu i oznacza początek wykonania równoległego kodu. Najważniejszymi z dyrektyw są dyrektywy podziału pracy (ang. *work sharing constructs*), które występują w obszarze równoległym i są stosowane do rozdzielenia poleceń realizowanych przez poszczególne procesory. Oprócz opisanej powyżej dyrektywy „**#pragma omp parallel**" w pracy występują również dyrektywy:

- „**#pragma omp for**" - dyrektywa ta nakazuje kompilatorowi podzielenie między wątki iteracji pętli for, która w ten sposób staje się pętlą równoległa. Dyrektywa musi być napisana tuż przed implementacją pętli for.
- „**#pragma omp critical**" - dyrektywa ta nakazuje kompilatorowi realizację sekcji krytycznej, poprzez zagwarantowania wykonania danej części kodu przez tylko jeden wątek naraz.

Dodatkowo każda z dyrektyw może posiadać własny zestaw dopuszczalnych klauzul, a najważniejsze z nich określają sposób traktowania zmiennych poprzez wątki w obszarze równoległym. W pracy używane jest pięć klauzul:

- „**num_threads**" dyrektywy **"parallel** - umożliwia jawne określenie liczby wątków wykonujących kod w obszarze obowiązywania dyrektywy,
- „**schedule(sposób_rozdziału_iteracji, *wartość*)**" - klauzula **schedule** dyrektywy **for** opisuje sposób rozdziału iteracji petli równoległej pomiędzy wątki oraz liczbę iteracji (wartość rozmiaru porcji iteracji) przydzielanych jednorazowo pojedynczemu wątkowi (podział iteracji zawsze odbywa się w sposób cykliczny, tzw. *round robin*). Przy opisie sposobu rozdziału można wyróżnić trzy najczęściej używane: **static** (iteracje podzielone są na mniejsze zbiory o długości ***wartość***, a następnie kolejno przydzielane dostępnym wątkom), **dynamic** (każdy wątek posiada przypisaną liczbę iteracji określoną przez parametr ***wartość***, a po wykonaniu obliczeń otrzymuje kolejną porcję iteracji do wykonania) oraz **runtime** (w tym przypadku jedna z powyższych opcji oraz parametr **wartość** jest ustalana poprzez nadanie odpowiedniej wartości specjalnej zmiennej środowiskowej),
- „**default(none)**"- deklaruje, że domyślnie żadna zmienna nie jest ani wspólna ani prywatna, a typ zmiennej określa sam programista,
- „**private**" - klauzula współdzielenia zmiennych, która tworzy zmienną lokalna wątków,
- „**firstprivate**" - klauzula współdzielenia zmiennych, która tworzy zmienną lokalną wątków z kopiowaną wartością początkową z obszaru kodu przed dyrektywą[6].

2.8 Programowanie procesorów z rejestrami wektorowymi, wykorzystanie rozkazów wektorowych w kodzie źródłowym

Wektoryzacja to proces konwersji algorytmu z implementacji skalarnej, w której wykonuje operacje na argumentach skalarnych, do procesu wektorowego, w którym pojedyncza instrukcja może odnosić się do wektora ze spakowanymi wartościami skalarnymi. Instrukcje SIMD mogą działać na wielu danych w jednej instrukcji i wykorzystywać 128-bitowe rejestry zmiennoprzecinkowe. Dzięki temu modyfikowany kod po zastosowaniu wektoryzacji może stać się krótszy i bardziej czytelny, przez co w programie może nastąpić znaczne zmniejszenie złożoności obliczeniowej oraz czasu obliczeń, ponieważ spakowane instrukcje działają na więcej niż jednym elemencie danych na raz[7].

W pracy umieszczanie rozkazów procesora (ang. *intrinsics*) w języku C realizowane jest poprzez zastosowanie specyfikacji AVX2 (ang. *Advanced Vector Extensions 2*), która jest rozszerzeniem architektury x86-64. Użyte przez to zostały 256 bitowe wektory, przechowujące cztery 64-bitowe wartości zmiennoprzecinkowe podwójnej precyzji, zapisywane w przy użyciu typu __m256d. Funkcje, które wykorzystywane są w procesie wektoryzacji w programie to[8]:

- **_mm256_load_pd(double const *a)**, która wprowadza(pobiera?) zmiennoprzecinkowe wartości podwójnej precyzji z lokalizacji w pamięci „**a”** do wektora docelowego,
- **_mm256_insertf128_pd(__m256d a, __m128d b, int offset)**, która wstawia 128 bitów spakowanych wartości float 64 w wektorze „**b**” do docelowego wektora „**a**” z bitowym przesunięciem określonym przez „**offset**”,
- **_mm256_store_pd(double *a, __m256d b)**, która zapisuje spakowane wartości z wektora „**b**” do wyrównanej lokalizacji w pamięci wskazanej przez „**a**”
- **_mm_loadh_pd(__m128d a, *const f64 b)** oraz **_mm_loadl_pd(__m128d a, *const f64 b)**, funkcje te ładują wartości podwójnej precyzji z „**b**” do bitów wyższego (**_mm_loadh_pd**) lub niższego (**_mm_loadl_pd**) rzędu 128-bitowego wektora „**a**” ,
- **_mm256_add_pd(__m256d m1, __m256d m2)**, która sumuje cztery spakowane elementy zmiennoprzecinkowe podwójnej precyzji pierwszego wektora źródłowego „**m1**” z czterema elementami float 64 drugiego wektora źródłowego „**m2**”,
- **_mm256_mul_pd(__m256d m1, __m256d m2)**, która mnoży cztery spakowane elementy pierwszego wektora „**m1**” z czterema elementami drugiego wektora „**m2**”.

2.9 Wydajność obliczeń równoległych

Wydajność obliczeń arytmetycznych można wyrazić przy użyciu liczby operacji zmiennoprzecinkowych na sekundę (GFLOPS - 10^9 FLOPS ang. *floating point operations per second*). Jednakże częściej wykorzystywana jest jednostka GFLOPS. OObliczanie miary w GFLOPS'ach dla mnożenia macierz wektor zależy od liczby wyrazów niezerowych używanej macierzy – w macierzy gęstej używa się liczby wszystkich N^2 elementów, a w macierzy rzadkiej jedynie liczby niezerowych elementów, NONZ. W pracy jednostka GFLOPS liczona jest w sposób wynikający bezpośrednio z algorytmu:

$$\frac{2NONZ}{t}$$

gdzie:

t – czas obliczeń algorytmu mnożenia macierz-wektor w nanosekundach [ns]

W celu wyrażenia wydajności obliczeń równoległych utworzono także miary względne, w których porównuje się czas wykonywania programu na jednym oraz wielu procesorach/rdzeniach. Do miar względnych zalicza się[9]:

- Przyspieszenie obliczeń: S(p) = Ts / Tp(p) (lub Tp(1) / Tp(p)),
- Efektywność zrównoleglenia: E(p) = S(p) / p,

gdzie:

p – liczba procesorów,

Ts – czas wykonania algorytmu sekwencyjnego,

Tp(p) - czas wykonania algorytmu równoległego przy użyciu p procesorów.

Wraz ze wzrostem przyspieszenia obliczeń zmierzającym do nieskończoności program dąży do idealnej skalowalności. Wydajność uznajemy za dobrą, gdy czas otrzymany dla wersji równoległej jest mniejszy niż czas otrzymany dla wersji sekwencyjnej i maleje wraz z liczbą użytych wątków.

3. Implementacja

Rozdział ten poświęcony został prezentacji implementacji funkcji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy rzadkiej. Składa się z dwóch podrozdziałów – pierwszy przedstawia zaimplementowane funkcje zamiany formatu COO na CSR, CSC oraz SELL. W drugim podrozdziale analizie zostały poddane utworzone funkcje mnożenia macierz-wektor dla zaimplementowanych formatów.

3.1 Funkcje zamiany formatów przechowywania

Do zapisu formatów CSR, CSC oraz SELL utworzone zostały struktury, w których znajdują się tablice opisujące dany format. Struktury te przedstawione zostały we fragmencie kodu nr 1:

```
struct Sell{
    int nr_slice;     //wysokość plastra
    double* val;      //główna tablica wartości dla slice_C
    int* slice_col;  //tablica kolumn
    int* sliceStart; //tablica slice_start
    int* cl;        //pomocnicza tablica długości plastrów
};

struct CSR{
    double* a_csr;  //główna tablica wartości
    int* row_ptr;   //tablica row_ptr
    int* col_ind;  //tablica kolumn
    int* row_ind; //tablica potrzebna do ustawienia Sell-C
};

struct CSC{
    double* a_csc;  //główna tablica wartości
    int* col_ptr;   //tablica col_ptr
    int* col_ind;   //tablica koumn
    int* row_ind;   //tablica wierszy
};
```

Fragment kodu 1: Utworzone dla formatów struktury wraz z elementami je opisującymi

Źródło: opracowanie własne

W funkcjach zamiany formatu COO na formaty CSR oraz CSC używana jest dodatkowa struktura „wpisy” utworzona specjalnie do odczytywania z pliku numeru wiersza, kolumny i wartości danych. Numery wierszy i kolumn zapisywane są do struktur CSR i CSC jako pomniejszone o jeden wartości ze struktury „wpisy”. Dzieje się tak, ponieważ w pliku, z którego odczytywane są wartości niezerowych elementów iteracja kolumn i wierszy zaczyna się od liczby jeden, a w tabelach używanych w programowaniu iterowanie zaczyna się od zera. W formacie CSR

zapisywana jest w ten sposób tablica kolumn. Tabela **row_ptr** natomiast utworzona została poprzez zsumowanie wystąpień danej wartości wiersza w czytanym pliku (format COO), a następnie dodanie do niej wartości **row_ptr** poprzedniego wiersza. Wartości tablicy **value** dla formatu CSR zostały przepisane ze struktury „wpisy". Kod funkcji cooToCsr(), w którym występuje zamiana z formatu COO do formatu CSR został przedstawiony we fragmencie kodu nr 2:

```
struct CSR cooToCsr(struct CSR CSR, struct Wpis *wpisy){
  CSR.a_csr = (double*)malloc(NONZ * sizeof(double));
  CSR.row_ptr = (int*)malloc((WYMIAR + 1) * sizeof(int));
  CSR.col_ind = (int*)malloc(NONZ * sizeof(int));
  CSR.row_ind = (int*)malloc(NONZ * sizeof(int));
  int i,j, ilosc=0,k ,row_sum=1;
  int pomptr[2];

  for(i=0; i<WYMIAR+1; i++){
    CSR.row_ptr[i]=0;
  }
  for(i=0;i<NONZ;i++){
    CSR.col_ind[i]=0;
    CSR.row_ind[i]=0;
    CSR.a_csr[i]=0.0;
  }
    //obliczanie i zapis do ptr
    for (ilosc = 0; ilosc < NONZ; ilosc++) {
      int wiersz = wpisy[ilosc].row;
      CSR.row_ptr[wiersz] ++;
    }

    for (i = 1; i <= WYMIAR; i++) {
      CSR.row_ptr[i] += CSR.row_ptr[i-1];
    }

    k = 0;
    for(i = 0; i <= WYMIAR; i++){
      for (j = 0; j < NONZ; j++) {
        if(wpisy[j].row == i){
          CSR.row_ind[k]=i-1;
          CSR.col_ind[k]=wpisy[j].col-1;
          CSR.a_csr[k]=wpisy[j].value;
          k++;
        }
      }
    }
  return CSR;
}
```

Fragment kodu 2: Kod funkcji cooToCsr()

Źródło: opracowanie własne

Zapis danych z formatu COO do CSC został utworzony w sposób analogiczny do zamiany formatu COO do CSR. W tym przypadku wartość wiersza jest odczytywana z pomniejszonej o jeden

wartości wiersza ze struktury „wpisy”. Tabela **col_ptr** utworzona została poprzez zsumowanie występowań danej wartości kolumny w formacie COO, a następnie dodanie wartości **col_ptr** z poprzedniego wiersza. Natomiast wartości tablicy **value** dla formatu CSC są przepisywane ze struktury „wpisy”. Kod funkcji cooToCsc(), w którym zobrazowana została zamiana z formatu COO do formatu CSC została przedstawiona we fragmencie kodu nr 3:

```
struct CSC cooToCsc(struct CSC CSC,struct Wpis *wpisy){
  CSC.a_csc = (double*)malloc(NONZ * sizeof(double));
  CSC.col_ptr = (int*)malloc((WYMIAR + 1) * sizeof(int));
  CSC.col_ind = (int*)malloc(NONZ * sizeof(int));
  CSC.row_ind = (int*)malloc(NONZ * sizeof(int));
  int i,ilosc=0,k=0,row_sum=1;
  int pomptr[2];

  for(i=0; i<WYMIAR+1; i++){
    CSC.col_ptr[i]=0;
  }
  for(i=0;i<NONZ;i++){
    CSC.col_ind[i]=0;
    CSC.row_ind[i]=0;
    CSC.a_csc[i]=0.0;
  }

    //obliczanie i zapis do ptr
    for (ilosc = 0; ilosc < NONZ; ilosc++) {
      int wiersz = wpisy[ilosc].col;
      CSC.col_ptr[wiersz] ++;
    }

    for (i = 1; i <= WYMIAR; i++) {
      CSC.col_ptr[i] += CSC.col_ptr[i-1];
    }

    for(i=1;i<=WYMIAR;i++){
      for (ilosc = 0; ilosc < NONZ; ilosc++) {
        if(wpisy[ilosc].col==i){
          CSC.row_ind[k]=wpisy[ilosc].row -1;
          CSC.col_ind[k]=i-1;
          CSC.a_csc[k]=wpisy[ilosc].value;
          k++;
        }
      }
    }
  return CSC;
}
```

Fragment kodu 3: Kod funkcji cooToCsc()

Źródło: opracowanie własne

W celu realizacji zamiany formatu COO na format SELL należało posłużyć się formatem CSR, z którego wykorzystano wartości **dane** i **coli_ind** do zapisu tablic **val** i **colind**, a tablicę **row_ptr** do

zapisu pomocniczej tablicy **cl** reprezentującej długość plastrów. Proces zapisu do SELL został podzielony na cztery różne przypadki:

- Przypadek, gdy numer wiersza równy jest zero – wartości **dane** i **col_ind** z formatu CSR zapisywane są do pomocniczych tablic, dodatkowo zliczana jest liczba występujących niezerowych elementów,
- Przypadek dla każdego kolejnego wiersza, który nie jest wielokrotnością C – również tu wartości **dane** i **col_ind** z formatu CSR zapisywane są do pomocniczych tablic oraz zliczana jest liczba niezerowych elementów. Działania te wykonywane są w pętli, której długość zależna jest od rozmiaru C,
- Przypadek, gdy wiersz nie jest równy zero, dodatkowo jest wielokrotnością C (znajduje się w ostatnim pasku) – w tym przypadku przed zapisem wartości z formatu CSR do pomocniczych tablic należało obliczyć największą długość wiersza, a następnie zapisać w odpowiedniej kolejności wartości z tablic pomocniczych do głównych tablic **val** i **colind**, Realizowane jest to przy użyciu dodatkowych zmiennych A i B, które przedstawiają zliczanie kolejnych wpisywanych wartości do głównych tablic.
- Przypadek, gdy analizowany jest ostatni slice – **w** tym przypadku wartości z tablic pomocniczych są przepisywane w odpowiedniej kolejności do głównych tablic **val** i **colind**. W tym celu należało poprzednio obliczyć długość najdłuższego miejsca.

Trzy pierwsze przypadki znajdują się w pętli, która przechodzi po wszystkich elementach niezerowych, a ostatni przypadek wykonywany jest tylko raz. Poniżej przedstawione zostały ilustracje obrazujące kod funkcji csrToSell() i opisujące zamianę formatów przypadki:

```
struct Sell csrToSell(struct Sell Sell, struct CSR CSR) {
  int l_r = WYMIAR;  // liczba wierszy WYMIAR
  int l_s = ceil((float)l_r / C); // liczba plastrów (wysokość plastra C)
  Sell.nr_slice = l_s;
  Sell.cl = malloc(l_s * sizeof(int)); // pomocnicza tablica długości plastrów
  int i, j;
  // obliczanie długości wiersza w bloku wierszy i zapis w pomocniczej tablicy cl
  for (i = 0; i < l_s; i++) {
    Sell.cl[i] = 0;
    for (j = 0; j < C && i * C + j < l_r; j++) {
      int i_temp = CSR.row_ptr[i * C + j + 1] - CSR.row_ptr[i * C + j];
      if (i_temp > Sell.cl[i]) Sell.cl[i] = i_temp;
    }
  }

  int nr_entries = 0;
  for (i = 0; i < l_s; i++) {
    nr_entries += C * Sell.cl[i];
  }
  Sell.val = (double*)malloc(nr_entries * sizeof(double));
  Sell.sliceStart = (int*)malloc((l_s + 1) * sizeof(int));
  Sell.slice_col = (int*)malloc(nr_entries * sizeof(int));

  for (i = 0; i < l_s + 1; i++)
    Sell.sliceStart[i] = 0;
  for (j = 0; j < nr_entries; j++)
    Sell.val[j] = 0.0;
  for (j = 0; j < nr_entries; j++)
    Sell.slice_col[i] = 0;
```

Fragment kodu 4: Fragment kodu csrToSell.c, w którym przedstawiona została inicjalizacja głównych tablic i obliczenie wartości dla pomocniczej tablicy cl

Źródło: opracowanie własne

```
int wymiar = l_s;
double p[C][WYMIAR]; //tablica pomocniczych rzędów i tablica slice start
int col[C][WYMIAR], m[C]; //tablica pomocniczych kolumn
//i tablica zmiennych określających liczbę elementów w tablicy p
int pomptr[2];
int A = 0, B = 0, D, m1, zmM, pomM, slice_start = 0, sliceVal = 0, ilosc;
pomptr[0] = 0;

//inicjowanie tablic
for(i = 0; i < C; i++) {
  m[i] = 0;
  for(j = 0; j < WYMIAR; j++) {
    p[i][j] = 0.0;
    col[i][j] = 0;
  }
}
```

Fragment kodu 5: Fragment kodu csrToSell.c, w którym inicjowane są tablice pomocnicze

Źródło: opracowanie własne

```
for(ilosc = 0; ilosc < NONZ; ilosc++) {
  int j = CSR.col_ind[ilosc];
  int i = CSR.row_ind[ilosc];

  if(i == 0) { //pierwszy przypadek gdy rząd 0
    zmM = m[0]; //zmienna pomocnicza jest brana z tablicy
    p[0][zmM] = CSR.a_csr[ilosc]; //pomocniczy zerowy rząd wyełniany wartościami
    col[0][zmM] = CSR.col_ind[ilosc];
    m[0]++;
  }
  for (pomM = 1; pomM < C; pomM++) { //dla każdego kolejnego rzędu
    if ((i % C) == pomM) {//modulo C
      zmM = m[pomM];          //wpisywane wartości
      p[pomM][zmM] = CSR.a_csr[ilosc];//do pomocniczej tablicy rzędów
      col[pomM][zmM] = CSR.col_ind[ilosc];
      m[pomM]++;
      A = 0;                    //wyzerowanie zmiennych A i B potrzebnych
      B = 0;                    //do wpisywania do końcowej tablicy val i slice_col
    }
  }
  if (i != 0 && (i % C) == 0) {
      if(A==0){
        sliceVal++; //zapisać do tablic val i slice_col
        m1 = max(m); //wyliczanie maksymalnej długości rzędu implementacja funkcji na dole
        for (D = 0; D < m1; D++) { //wyliczanie val i colind dla sell-c
          for (pomM = 0; pomM < C; pomM++) { //dla wszystkich C rzędów
            Sell.val[slice_start + A] = p[pomM][B]; //wpisywane wartości do val
            Sell.slice_col[slice_start + A] = col[pomM][B]; //wpisywane wartości do slice_col
            A++; //po kazdej iteracji zwiększamy A dla następnego rzędu
          }
          B++; //po kazdej iteracji zwiększamy B dla kolejnego elementu
        }      //w pomocniczej tablicy rzędów
        slice_start += A;  //dodanie wartości do sliceStart
        Sell.sliceStart[sliceVal] = slice_start;
        for (i = 0; i < C; i++) {
          m[i] = 0;
          for (j = 0; j < WYMIAR; j++) {
            p[i][j] = 0.0;
            col[i][j] = 0;
          }
        }
      }
    zmM = m[0]; //wpisanie wartości do zerowego pomocniczego rzędu
    p[0][zmM] = CSR.a_csr[ilosc];
    col[0][zmM] = CSR.col_ind[ilosc];
    m[0]++;
  }
}
```

*Fragment kodu 6: Fragment kodu, w którym widoczna jest pętla po elementach niezerowych i opisane w niej trzy przypadki zapisu wartości do głównych tablic **val** i **colind***

Źródło: opracowanie własne

```
    //ostatni slice C
    A = 0;                      //wyzerowanie zmiennych A i B potrzebnych
    B = 0;                      //do wpisywania do końcowej tablicy val i slice_col
    sliceVal++;
    m1 = max(m);
    for (D = 0; D < m1; D++) { //wyliczanie val i colind dla sell-c
      for (pomM = 0; pomM < C; pomM++) {
        Sell.val[slice_start + A] = p[pomM][B];
        Sell.slice_col[slice_start + A] = col[pomM][B];
        A++;
      }
      B++;
    }
    Sell.sliceStart[sliceVal] = slice_start;
    return Sell;
}
```

Fragment kodu 7: Fragment kodu, w którym przedstawiony jest ostatni przypadek i funkcja odnajdująca najdłuższą długość wiersza

Źródło: opracowanie własne

Do przedstawienia działania utworzonych funkcji zapisu formatów użyta została przykładowa macierz rozmiaru 6x6, w której znajduje się 13 niezerowych elementów. Macierz ta zobrazowana została na ilustracji nr 2:

$$\begin{pmatrix} 0 & 1 & 0 & 3 & 0 & 0 \\ 2 & 0 & 8 & 0 & 7 & 0 \\ 0 & 0 & 4 & 5 & 0 & 0 \\ 0 & 9 & 0 & 11 & 0 & 0 \\ 23 & 0 & 0 & 43 & 0 & 0 \\ 0 & 0 & 18 & 0 & 0 & 20 \end{pmatrix}$$

Ilustracja 2: Macierz testowa

Źródło: opracowanie własne

Wyniki uzyskane po skompilowaniu i uruchomieniu funkcji zamiany formatów dla przykładowej macierzy przedstawione zostały na ilustracjach 3, 4, 6 i 8:

Format CSR:

```
row_ptr=[  0    2    5    7    9    11    13  ]
col_ind=[  1    3    0    2    4    2    3    1    3    0    3    2    5  ]
val=[  1.0    3.0    2.0    8.0    7.0    4.0    5.0    9.0    11.0    23.0    43.0    18.0    20.0  ]
```

Ilustracja 3: Uzyskane wartości po skompilowaniu i uruchomieniu kodu funkcji cooToCsr.c

Źródło: opracowanie własne

Format CSC:

```
col_ptr=[  0    2    4    7    11    12    13  ]
row_ind=[  1    4    0    3    1    2    5    0    2    3    4    1    5  ]
val=[  2.0    23.0    1.0    9.0    8.0    4.0    18.0    3.0    5.0    11.0    43.0    7.0    20.0  ]
```

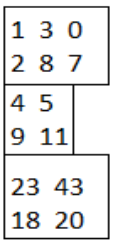
Ilustracja 4: Uzyskane wartości po skompilowaniu i uruchomieniu kodu funkcji cooToCsc.c

Źródło: opracowanie własne

Format SELL-C:

- Dla ***C=2***

VALUES:



Ilustracja 5: Rozpisane wartości macierzy dla C=2

Źródło: opracowanie własne

```
Val=[ 1.0,  2.0,  3.0,  8.0,  0.0,  7.0,  4.0,  9.0,  5.0,  11.0,  23.0,  18.0,  43.0,  20.0, ]
colind=[ 1,  0,  3,  2,  0,  4,  2,  1,  3,  3,  0,  2,  3,  5, ]
sliceStart=[ 0,  6,  10,  14, ]
```

Ilustracja 6: Uzyskane wartości po skompilowaniu i uruchomieniu kodu funkcji csrToSell.c dla C=2

Źródło: opracowanie własne

- Dla ***C=4***

VALUES:

1	3	0
2	8	7
4	5	0
9	11	0

23	43
18	20
0	0
0	0

Ilustracja 7: Rozpisane wartości macierzy dla C=4

Źródło: opracowanie własne

```
Val=[ 1.0,  2.0,  4.0,  9.0,  3.0,  8.0,  5.0,  11.0,  0.0,  7.0,  0.0,  0.0,  23.0,  18.0,  0.0,  0.0,  43.0,  20.0,  0.0,  0.0, ]

colind=[ 1,  0,  2,  1,  3,  2,  3,  3,  0,  4,  0,  0,  0,  2,  0,  0,  3,  5,  0,  0, ]

sliceStart=[ 0,  12,  20, ]
```

Ilustracja 8: Uzyskane wartości po skompilowaniu i uruchomieniu kodu funkcji csrToSell.cdla C=4

Źródło: opracowanie własne

3.2 Funkcje mnożenia macierz-wektor

Funkcja mnożenia macierz-wektor dla formatów CSR i CSC została zrealizowana w sposób sekwencyjny standardowy oraz w sposób zrównoleglony, a dla formatu SELL w sposób sekwencyjny standardowy, w sposób zrównoleglony oraz wektorowy sekwencyjny i w sposób wektorowy zrównoleglony. Mnożenie macierz-wektor dla formatów CSR i CSC może być wykonane następująco:

```
for i = 1, n
    y(i)  = 0
    for j = row_ptr(i), row_ptr(i+1) - 1
        y(i) = y(i) + val(j) * x(col_ind(j))
    end;
end;
```

Ilustracja 9: Mnożenie macierz-wektor przedstawione dla formatu CSR

Źródło: [10]

Na podstawie przedstawionego powyżej sposobu zrealizowane zostały implementacje funkcji mnożenia macierz-wektor dla formatów CSR i CSC.

Do utworzenia funkcji obliczającej algorytm w sposób równoległy należało funkcję zapisaną w sposób sekwencyjny przekształcić przy pomocy interfejsu OpenMP. W formatach CSR i CSC na początku funkcji należało wprowadzić dyrektywę „**#pragma omp parallel**”, a następnie dla zewnętrznej pętli dyrektywę „**#pragma omp for**”. W formacie CSR oraz SELL pętla po wierszach przedstawiona jest jako pierwsza pętla, a w formacie CSC najpierw funkcja przechodzi po kolumnach, a następnie po wierszach. Dodatkowo w celu realizacji równoległości w formacie CSC należało również użyć dyrektywy „**#pragma omp critical**”, do sumowania lokalnych wektorów **y** obliczonych przez pojedyncze wątki. Funkcje mnożenia macierz-wektor zrealizowane w sposób standardowy równoległy dla formatów CSR, CSC i SELL przedstawione zostały w dalszej części. Dzięki specyfice standardu OpenMP funkcje sekwencyjne dla każdego z formatów odpowiadają wersji równoległej z zakomentowanymi liniami kodu zawierającymi dyrektywy OpenMP.

Funkcja dla formatu CSR:

W funkcji wykorzystywany jest przedstawiony wyżej schemat algorytmu mnożenia macierz-wektor, w którym liczba wierszy uznawana jest jako wartość WYMIAR, a liczba kolumn obliczana jest na podstawie tablicy **row_ptr**, dzięki czemu uzyskujemy liczbę kolumn w danym wierszu. Zmienna **x0,** odczytywana jako numer analizowanej w danym momencie kolumny, służy jako indeks wartości wektora **x**, która używana jest do obliczeń iloczynu. Następnie do kolejnych wartości tablicy **y** dodawany jest obliczony iloczyn.

```
void mat_vec_crs(double* x, double* y,struct CSR csr,long int nn,long int nt){
  register long int i;
  #pragma omp parallel default(none) firstprivate(x,y,csr,nn,i) num_threads(nt)
  {
   register long int n=nn;
   register long int j;
   register long int x0;
   #pragma omp for
   for(i=0;i<n;i++){
      long int edge=csr.row_ptr[i+1]-1;
      for(j=csr.row_ptr[i];j<=edge;j++){
        x0=csr.col_ind[j];
        y[i]+=csr.a_csr[j] * x[x0];
      }
    }
  }
}
```

Fragment kodu 8: Równoległa implementacja mnożenia macierz-wektor dla formatu CSR

Źródło: opracowanie własne

Funkcja dla formatu CSC:

W funkcji wykorzystywany jest przedstawiony wyżej schemat algorytmu mnożenia macierz-wektor, w którym liczba kolumn uznawana jest jako wartość WYMIAR, liczba wierszy obliczana jest na podstawie tablicy **col_ptr**, dzięki czemu uzyskujemy liczbę kolumn w danym wierszu oraz występuje zmienna **x0** obliczana jako wartość analizowanego w danym momencie wiersza. Następnie w tablicy **ylocal** zsumowywane są wartości mnożenia. Wartość **x0** wykorzystywana jest tu jako wartość tablicy **ylocal**. Dopiero poza pętlami **for** w sekcji krytycznej wykonywane jest sumowanie wartości tablicy **ylocal** do tablicy **y**. Takie rozwiązanie zostało wykorzystane, aby czas wykonania funkcji był porównywalny z czasem wykonania funkcji dla formatu CSR (zrównoleglenie pętli po wierszach, unikające wyścigu przy zapisie do wektora **y,** jest dla formatu CSC zdecydowanie mniej wydajne).

```
void mat_vec_csc(double* x, double* y,struct CSC csc,long int nn,long int nt){
  register long int i;
  #pragma omp parallel default(none) firstprivate(x,y,csc,nn,i) num_threads(nt)
  {
   double* ylocal = malloc(nn * sizeof(double));
   register long int pom;
   register long int n=nn;
   register long int j;
   register long int x0;
   #pragma omp for
   for(i=0;i<n;i++){
      long int edge=csc.col_ptr[i+1]-1;
      for(j=csc.col_ptr[i];j<=edge;j++){
        x0=csc.row_ind[j];
        ylocal[x0]+=csc.a_csc[j] * x[i];
      }
    }
    #pragma omp critical(y)
    for(i=0;i<n;i++)
      y[i]+=ylocal[i];
  }
}
```

Fragment kodu 9: Funkcja macierz-wektor dla formatu CSC zrównoleglona w standardzie OpenMP

Źródło: opracowanie własne

Funkcja dla formatu SELL:

W funkcji tej, podobnie jak w formacie CSR, zewnętrzną pętlą jest pętla po wierszach, a dokładniej pętla po blokach wierszy (plastrach). Pętla po kolumnach w i-tym bloku wierszy swój zakres kończy na wartości tablicy długości plastrów dla i-tego wiersza. W tym formacie występuje również trzecia pętla przechodząca po wartościach do rozmiaru wysokości bloku wierszy C.

Następnie wyznaczana jest pomocnicza wartość przedstawiająca proces wyboru elementu z tablicy wartości. W tym formacie, tak jak w CSR występuje zmienna **x0**, która obliczana jest jako wartość analizowanej w danym momencie kolumny. Do użycia odpowiedniego elementu również wykorzystywana jest zmienna **pom**, a zmienna **x0** jest indeksem wartości wektora **x**, która używana jest do obliczeń iloczynu macierz-wektor. Na koniec do wartości odpowiedniego elementu w tablicy **y,** wyliczanego w oparciu o rozmiar C, dodawana jest wartość wyliczona na podstawie mnożenia.

```
void mat_vec_sell(double* x, double* y_global, struct Sell sell, long int nn, long int nt) {
  register long int i;
  double* y = malloc(C * sell.nr_slice * sizeof(double));
  for (i = 0; i < C * sell.nr_slice; i++) y[i] = 0.0;

  #pragma omp parallel default(none) firstprivate(x,y,y_global,sell,nn,i) num_threads(nt)
  {
    register long int n = nn;
    register long int j;
    register long int pom;
    register long int x0;
    int k;

    int wyn = sell.nr_slice;

    // pętla po blokach wierszy
    #pragma omp for
    for (i = 0; i < wyn; i++) {

      // pętla po kolumnach w i-tym bloku wierszy
      for (j = 0; j < sell.cl[i]; j++) {
        for(k=0;k<C;k++){
        pom = sell.sliceStart[i] + j * C + k;
        x0 = sell.slice_col[pom];
        y[i * C + k] += sell.val[pom] * x[x0];
        }
      }
    }
  }

  for (i = 0; i < nn; i++)
    y_global[i] = y[i];

  for (i = 0; i < nn; i++)
    printf("y[%ld] = %lf \n", i, y_global[i]);
}
```

Fragment kodu 10: Równoległa funkcja macierz-wektor dla formatu SELL

Źródło: opracowanie własne

Kod dla SELL z realizacją wektorową

W celu zwiększenia wydajności przetwarzania wykorzystana została wektoryzacja obliczeń, gdzie uzyskiwane są lepsze efekty, gdy elementy macierzy występujące w kolejnych operacjach algorytmu przechowywane są w następujących po sobie lokalizacjach w pamięci. Sposób ten realizowany jest poprzez format SELL i jego algorytm mnożenia, dlatego też w programie wykorzystano wektoryzację dla tego formatu.

W zastosowanej funkcji zewnętrzną pętlą jest pętla przechodząca po plastrach, a wewnętrzną pętla przechodząca po kolumnach plastra. W pętlach do zmiennych pomocniczych pakowane są cztery wartości macierzy rzadkiej (zmienna **val**) i dwie wartości wektora (zmienna **rhstmp**). Następnie 128 bitowa zmienna rhstmp wstawiana jest do 256 bitowej zmiennej **rhs**. Proces ten jest powtarzany również dla dwóch kolejnych elementów wektora **x**. Na koniec wykonywane jest mnożenie zmiennych **val** i **rhs**, a wynik dodawany do zmiennej **tmp**, która zapisywana jest do wektora **y**.

```
int mat_vec_vector(double *x,double* y,struct Sell sell,long int nt){

  int c;
  #pragma omp parallel for default(none) firstprivate(y,x,sell)
  for (c=0; c < sell.nr_slice; c ++){ //pętla po plastrach
    int j , offs ;
    __m256d tmp , val , rhs ;
    __m128d rhstmp ;
    tmp = _mm256_load_pd(&y[c<<2]); //wpakuj 4 wartości LHS
    offs = sell.sliceStart[c]; // poczatkowy offset jest początkiem plastra
    for (j =0; j < sell.cl[c]; j ++){ //pętla w plastrze po wartościach do długści plastra
      val = _mm256_load_pd(&sell.val[offs]); //załaduj 4 wartości macierzy rzadkiej
      rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj pierwszą wartość RHS (wektor)
      rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj drugą wartość RHS (wektor)
      rhs = _mm256_insertf128_pd(rhs , rhstmp ,0); //wstaw spakowane wartości
      rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj trzecią wartość RHS (wektor)
      rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj czwartą wartość RHS (wektor)
      rhs = _mm256_insertf128_pd(rhs , rhstmp ,1); //wstaw spakowane wartości
      tmp = _mm256_add_pd(tmp , _mm256_mul_pd(val , rhs )); //gromadzenie wartości
    }
    _mm256_store_pd(&y[c<<2] , tmp ); //przechowaj 4 wartości LHS (y)
  }
}
```

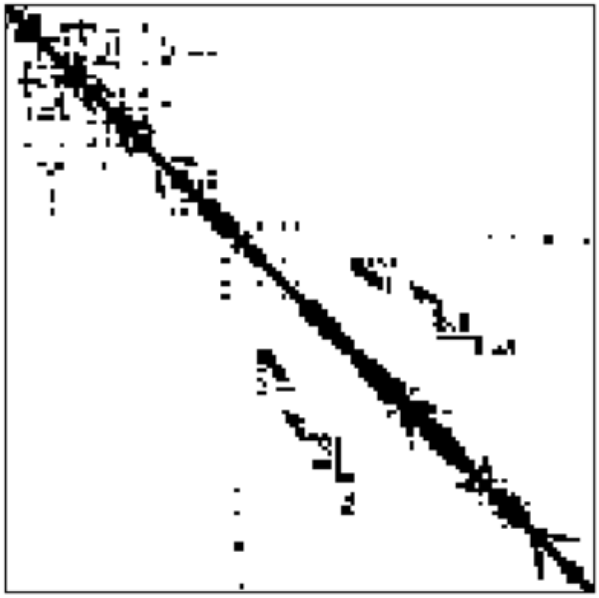
Fragment kodu 11: Funkcja macierz-wektor realizowana w sposób równoległy wektorowy dla formatu SELL

Źródło: opracowanie własne na podstawie [11]

4. Testy programu

Testy implementacji mnożenia macierz-wektor przeprowadzone zostały na dwuprocesorowym serwerze z procesorami Intel Xeon E5-2630 v4, pracującym pod kontrolą systemu operacyjnego Centos7. Jako kompilator wykorzystany został gcc w wersji 4.8.5. W celu wyznaczenia różnic uzyskanych między czasami mnożenia macierz-wektor dla użytych formatów wykorzystana została funkcja zliczania czasu z środowiska OpenMP, omp_get_wtime(). Dodatkowo z faktu, że rejestry w AVX2 są 256 bitowe, a liczba double ma 64 bity, testowanie macierzy w formacie SELL przeprowadzone zostało tylko dla wartości C=4. Program został skompilowany i uruchomiony kilkukrotnie w celu znalezienia najkrótszych czasów wykonania.

Do przedstawienia działania funkcji mnożenia macierz-wektor użyta została macierz – BCSSTK30 o wymiarach 28924x28924, która posiada 1036208 niezerowych elementów. Ilustracja nr 10 przedstawia rozkład elementów w macierzy:



Ilustracja 10: Rozkład niezerowych elementów w macierzy BCSSTK30

Źródło: Bibliografia nr [12]

Po uruchomieniu kodu otrzymano wyniki, które w sposób tabelaryczny przedstawione zostały w tabelach nr 1 i 2:

Tabela 1: Uzyskane czasy dla macierzy BCSSTK30

Równolegle				
Czas [ns]	**Format**			
	CSR	**CSC**	**SELL**	**SELL wektor**
1 wątek	0,0033	0,0026	0,0023	0,0010
2 wątki	0,0018	0,0014	0,0009	0,0005
4 wątki	0,0010	0,0012	0,0006	0,0003
8 wątków	0,0006	0,0008	0,0004	0,0002

Źródło: opracowanie własne

Tabela 2: Uzyskane wyniki jednostki GFLOPS dla macierzy BCSSTK30

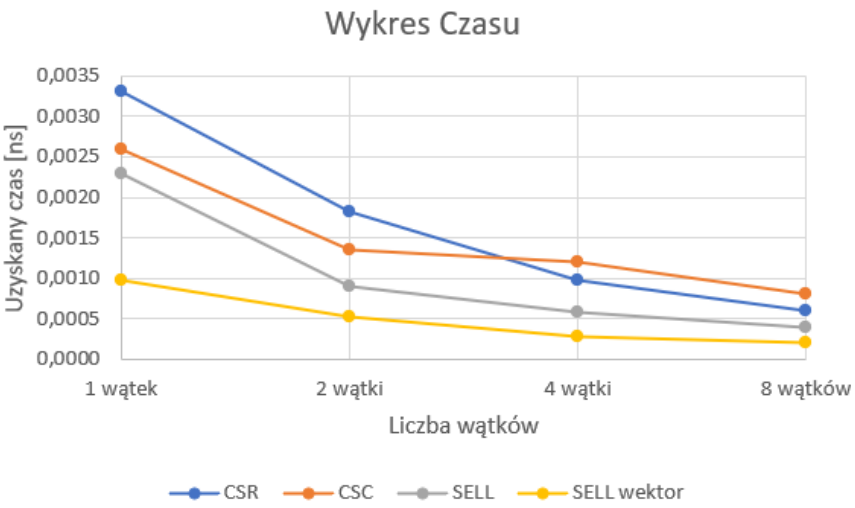
Równolegle				
GFLOPS	**Format**			
	CSR	**CSC**	**SELL**	**SELL wektor**
1 wątek	0,62	0,80	0,90	2,12
2 wątki	1,14	1,53	2,30	3,88
4 wątki	2,13	1,78	3,59	7,48
8 wątków	3,60	2,74	5,21	9,92

Źródło: opracowanie własne

Na podstawie wyników uzyskanych w tabelach stworzone zostały trzy wykresy:

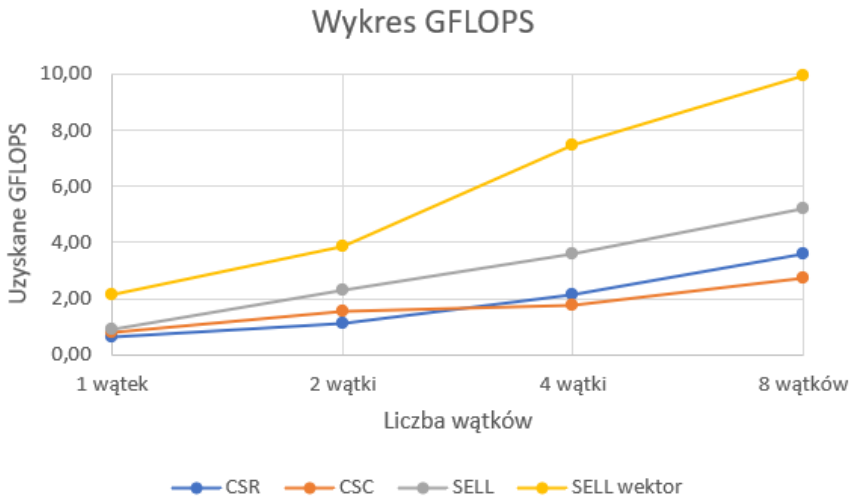
- wykres zależności czasu od liczby wątków (rys. 11),
- wykres zależności uzyskanych GFLOPS od liczby wątków (rys. 12),
- wykres zależności uzyskanego przyspieszenia od liczby wątków (rys. 13).

Wartości przyspieszenia widoczne na ilustracji nr 13 zostały obliczone na podstawie wzoru Tp(1)/Tp(p).



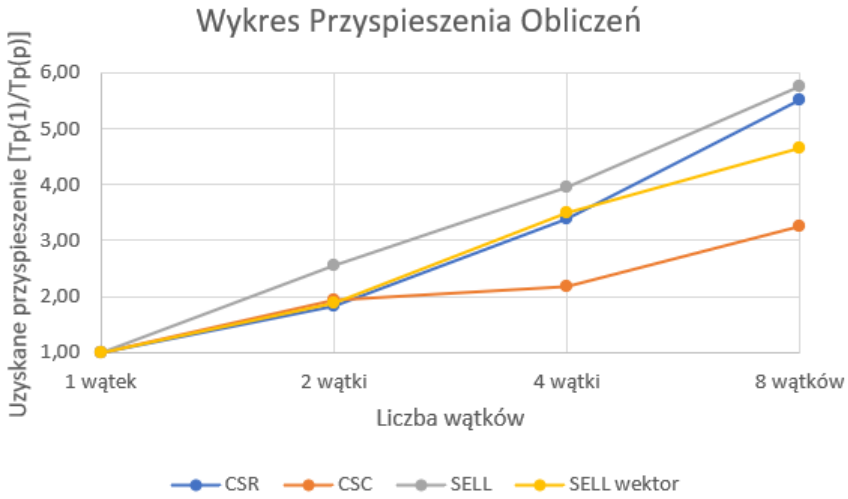
Ilustracja 11: Wykres czasu dla macierzy BCSSTK30

Źródło: opracowanie własne



Ilustracja 12: Wykres uzyskanych GFLOPS dla macierzy BCSSTK30

Źródło: opracowanie własne



Ilustracja 13: Wykres przyspieszenia dla macierzy BCSSTK30

Źródło: opracowanie własne

Analizując otrzymane wyniki na wykresach i w tabelach można zauważyć, że jednostka GFLOPS oraz przyspieszenie obliczeń stale rosną dla wszystkich badanych formatów: CSR, CSC, SELL i SELL zrealizowanego w sposób wektorowy. Jednakże można zauważyć, że wartości GFLOPS oraz przyspieszenie obliczeń realizowane dla formatu CSC rośnie w sposób nierównomierny. Związane to jest z koniecznością wykonania operacji redukcji dla lokalnych wektorów wynikowych **y_local.** Dla formatu SELL zrealizowanego w sposób wektorowy w niniejszej pracy najefetywniejszą realizację zadania udało się uzyskać dla 8 wątków. Dodatkowo analizując uzyskane wartości przyspieszenia obliczeń dla wszystkich przedstawionych przypadków można zauważyć, że przyspieszenie ma tendencję wzrostową. Można więc założyć, że wraz ze wzrostem liczby wątków efektywność obliczeń może nadal wzrastać. W rzeczywistości dla 20 wątków i formatu SELL oraz algorytmu wektorowego udaje się osiągnąć wydajność ok. 20 GFLOPS. Dalszemu wzrostowi wydajności na przeszkodzie stoją ograniczone zasoby maszyny testowej, w szczególności przepustowość magistrali łączącej rdzenie z pamięcią operacyjną

5. Podsumowanie

Celem niniejszej pracy inżynierskiej było utworzenie programu realizującego funkcje zamiany formatów macierzy rzadkiej, funkcje mnożenia macierz-wektor dla badanych formatów, porównanie czasów obliczeń mnożenia macierz-wektor i na ich podstawie wyliczenie uzyskanego przyspieszenia dla wszystkich badanych formatów macierzy rzadkiej. Testowanie przeprowadzone zostało na jednej dużej macierzy, w której liczba niezerowych elementów była większa niż milion. Dodatkowo poprzez zrealizowane w pracy testy można było przeanalizować czasy i wydajność zaimplementowanych funkcji, a tym samym stwierdzić, czy wykorzystywana wektoryzacja kodu przyniosła korzyści w usprawnieniu obliczeń.

Uzyskane wyniki dla wszystkich formatów można uznać za prawidłowe ze względu na widoczny wzrost przyspieszenia dla każdego formatu. Jednak programy nie są idealnie skalowalne, ponieważ ich przyspieszenie odbiega od przyspieszenia idealnego, $S(p) = p$. Widoczna różnica w uzyskanych czasach, wydajności i obliczonych, na podstawie czasu, przyspieszeniach obliczeń pozwala na stwierdzenie, że rekomendowanym formatem dla obliczeń mnożenia macierz-wektor realizowanego w sposób równoległy jest format SELL używający rejestry wektorowe.

Praca ta nie wyczerpuje w pełni realizowanego tematu. Tworzony program ma perspektywy rozwoju w wielu kierunkach, z których najważniejszymi mogą być wykorzystanie procesu wektoryzacji dla wszystkich opisywanych w pracy formatów oraz realizacja obliczeń równoległych przy użyciu większej liczby wątków.

Bibliografia

[1] Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures. [Online], https://jianbinfang.github.io/files/2018-06-28-aspmv.pdf, [dostęp: 20.12.2021]

[2] Publikacja naukowa „Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C-σ formats on NVIDIA GPUs”

[3] Architektura mikroprocesorów, [Online], http://ue.pwr.wroc.pl/wyklad_architektura_mikroprocesorow/AM_1.pdf [dostęp: 28.12.2021]

[4] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone, http://ww1.metal.agh.edu.pl/~banas/PR/PR_W03_Wspolbieznosc.pdf [dostęp: 27.12.2021]

[5] Strona internetowa specyfikacji OpenMP https://www.openmp.org/ [dostęp: 21.12.2021]

[6] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone, http://ww1.metal.agh.edu.pl/~banas/PR/PR_W07_OpenMP_1.pdf [dostęp: 05.01.2022]

[7] A Guide to Vectorization with Intel® C++ Compilers, adres: https://www.intel.com/content/dam/www/public/us/en/documents/guides/compiler-auto-vectorization-guide.pdf [dostęp: 26.12.2021]

[8] Intel® C++ Compiler Classic Developer Guide and Reference Development Reference Guides,

https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions.html [dostęp: 02.01.2022]

[9] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone [Online] http://ww1.metal.agh.edu.pl/~banas/PR/PR_W13_Wydajnosc.pdf

[10] Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, https://www.netlib.org/templates/templates.pdf [dostęp: 05.01.2022]

[11] Sparse Matrix-Vector Multiplication with Wide SIMD Units: Performance Models and a Unified Storage Format [Online], adres: https://blogs.fau.de/essex/files/2012/11/SELL-C-sigma.pdf [dostęp: 04.01.2022]

[12] Macierz BCSST30, https://www.cise.ufl.edu/research/sparse/matrices/HB/bcsstk30.html [dostęp: 05.01.2022]