Wrocław 2020.11.26
Autor: Jerzy Sas

Uzupełnienia i korekty: Michał Przewoźniczek

# Konkurs na najefektywniejszą implementację operacji macierzowych

Celem zadania konkursowego jest zaprojektowanie i implementacja kompletnego efektywnego czasowo zestawu komponentów wspierających obliczenia macierzowe, tak aby zaimplementowane zostały operatory pozwalające stosować notację zbliżoną do używanej w algebrze liniowej.

## 1. Wymagania funkcjonalne

Należy zaprojektować i zaimplementować klasę reprezentującą macierz (w szczególności macierz o wymiarach 1xN i Nx1 odpowiadającą wektorowi wierszowemu i kolumnowemu). W klasie tej należy zaimplementować operatory pozwalające w naturalny i zbliżony do przyjętej notacji algebraicznej zapisywać wyrażenia opisujące operacje na macierzach. W szczególności należy zaimplementować operatory do:

- dodawania i odejmowania macierzy
- mnożenia macierzy (w tym mnożenia wektorów i macierzy),
- mnożenia macierzy przez liczbę,
- wyliczania iloczynu skalarnego,
- transponowania macierzy,
- odwracania macierzy.

Dodatkowo należy zaimplementować operacje pozwalające na:

- tworzenie macierzy o zadanych wymiarach (w konstruktorze),
- przypisywanie i odczyt wartości wskazanych elementów macierzy,
- tworzenie wektorów poprzez wybór wskazanego wiersza lub kolumny macierzy,
- przypisywania macierzy kwadratowej wartości odpowiadających macierzy jednostkowej.

Wszystkie operacje (z wyjątkiem tych wykonywanych w naturalny sposób w konstruktorach) należy zaimplementować jako odpowiednio przeciążone operatory. Klasę należy zaimplementować jako klasę szablonową parametryzowaną typem elementu macierzy (int, float, double, complex). Ten ostatni typ odpowiada liczbom zespolonym i **należy go zaimplementować (wraz z odpowiednimi operatorami) samodzielnie.**

W implementacji szczególną uwagę należy zwrócić na efektywność czasową i pamięciową swojej implementacji oraz zapewnić właściwą obsługę błędów. Implementacja nie może nakładać żadnych jawnych ograniczeń na rozmiary przetwarzanych macierzy. W szczególności należy zapewnić aby możliwe było przetwarzania macierzy o wymiarach rzędu 1000x1000 i większych.

## 2. Wymagania pozafunkcjonalne:

- nie można wykorzystywać żadnych obcych komponentów poza standardowym szablonami implementującymi kontenery, w szczególności nie można korzystać z bibliotek LinAlg, MKL, Blas Lapack i innych
- kod powinien być kompilowany w środowisku MSVC 2017 lub 2019,
- nie można wykorzystywać wspomagania przez GPU,
- w celu poprawy efektywności implementacji można stosować instrukcje SIMD (w wersja AXV i AVX2, nie dopuszczamy AVX-512 ze względu na ograniczoną dostępność w starszych procesorach),
- zabronione jest wykorzystywanie kodu pozyskanego metodą copy-and-paste,
- można stosować dowolne algorytmy wykonywania operacji macierzowych pod warunkiem, że ich implementacja jest w pełni dziełem autora oraz autor wykaże pełne zrozumienie wykorzystanej metody (tzn. nie można wykorzystywać obliczeń niezrozumiałych dla programisty),
- można wykorzystywać wielowątkowość przy ograniczeniu maksymalnej liczby wątków do 4.

## 3. Podział implementacji na etapy:

Etap 1:

- opracowanie interfejsu klasy (metody publiczne, metody operatorowe, wybór algorytmów efektywnego mnożenia i odwracania macierzy)
- implementacja prostych metod: dostęp do elementów macierzy, dodawanie, transpozycja, konstruktory i destruktory, mnożenie metodą "naiwną" (z definicji)

Etap 2:

- implementacja wybranej/wybranych metod odwracania macierzy (można eksperymentować z wieloma metodami aby wybrać najefektywniejszą),
- optymalizacja kodu wykonującego odwracanie macierzy

Etap 3:

- implementacja klasy implementującej liczby zespolone,
- implementacja zoptymalizowanych metod mnożenia macierzy (np. z wykorzystaniem instrukcji AVX, z uwzględnieniem ograniczenia programu przez dostęp do pamięci (optymalizacja wykorzystania pamięci podręcznej itp.)

## 4. Konkurs na najszybszą implementację operacji macierzowych

Studenci mogą ze swoimi implementacjami operacji macierzowych przystąpić do konkursu na najefektywniejszą implementację. Kryterium konkursu jest czas wykonania testowych obliczeń z wykorzystaniem metod i funkcji operatorowych zaimplementowanych przez uczestnika konkursu. Zasady konkursu są następujące:

- Czas wykonania obliczeń testowych z wykorzystaniem dostarczonego kodu będzie porównany z czasem identycznych obliczeń wykonanych z wykorzystaniem implementacji referencyjnej opartej o niezoptymalizowane implementacje oparte bezpośrednio na definicjach poszczególnych operacji.
- Kryterium oceny będzie współczynnik przyspieszenia względem implementacji referencyjnej liczony jako stosunek

$$Q = \frac{t_{ref}}{t_{prog}}$$

gdzie $t_{ref}$ to czas obliczeń implementacji referencyjnej a $t_{prog}$ to czas obliczeń implementacji konkursowej.

- Kryterium dopuszczającym do dalszych etapów konkursu jest osiągnięcie współczynnika przyspieszenia na poziomie co najmniej *Q=2.0.* Oceniana będzie tylko efektywność czasowa.
- Nie jest oceniana efektywność pamięciowa (tzn. w celu poprawienia efektywności czasowej można stosować dowolne struktury danych bez względu na zajętość pamięci).
- Ograniczona jest jedynie rozmiar całkowitej pamięci alokowanej przez program. Przy utworzeniu 10 macierzy o wymiarach 10000x10000 pamięć alokowana przez program, który wykonuje dowolne obliczenia na tych macierzach nie powinna być większa niż 10GB.
- Wyniki obliczeń programu konkursowego będą porównane z wynikami obliczeń implementacji referencyjnej. Różnica dowolnej wartości skalarnej należącej do macierzy i wektorów wynikowych powyżej 1.0e-6 dyskwalifikuje zawodnika.
- Procedura konkursowa będzie wykonywana na procesorze Intel i9 w środowisku systemu operacyjnego Windows 7.
- Zawodnik startujący w konkursie zobowiązany jest dostarczyć kod źródłowy programu oraz jego wersję skompilowaną, zdolną do wykonania w środowisku wyspecyfikowanym w poprzednim punkcie.
- Niespełnienie wymagań sprecyzowanych w pkt. 2 (w szczególności – niezrozumienie kodu przedstawionego do konkursu) skutkuje bezwarunkową dyskwalifikacją zawodnika.
- Przykładowy kod źródłowy obliczeń wykorzystywanych do oceny czasu wykonania programów konkursowych będzie dostarczony zawodnikom. Jednak ostateczna postać procedury obliczeniowej użytej w konkursie będzie inna.

## 5. Sugerowane metody optymalizacji czasowej

Najbardziej kosztownymi elementami obliczeń numerycznych są:

- wykonywanie zmiennoprzecinkowych operacji arytmetycznych, w szczególności mnożenia i dzielenia,
- dostęp do pamięci.

Ograniczeń efektywności związanej z wykonywaniem operacji arytmetycznych można unikać stosując następujące techniki:

**Instrukcje maszynowe typu SIMD (Single Instruction, Multiple Data)**. Instrukcje te wykonują równocześnie tę samą operację (np. mnożenie) na całym wektorze operandów. W przypadku mnożenia macierzy gdzie każdy współczynnik macierzy wynikowej jest iloczynem skalarnym wiersza i kolumny wykorzystanie rozkazów tego typu może dać bardzo znaczące przyspieszenie. Warunkiem jest odpowiednie spakowanie wektorów stających się operandami rozkazu SIMD. Większość kompilatorów C++ umożliwia wplatanie w kod C++ rozkazów tego typu. Niektóre kompilatory analizują kod źródłowy i wykorzystują te instrukcje automatycznie. Zainteresowani tą techniką powinni zapoznać się z zasadami korzystania z instrukcji SIMD w dokumentacji używanego kompilatora.

**Obliczenia wielowątkowe.** Skomplikowane obliczenia można powierzyć niezależnie wykonującym się wątkom. Podstawowe warunki efektywnego wykonywania wątków to:

a) brak komunikacji i wewnętrznej synchronizacji pomiędzy wątkami

b) powierzenie wątkom stosunkowo długo trwających sekwencji obliczeń. Mechanizm uruchamiania, zatrzymywania i synchronizacji wątków jest bardzo kosztowny, stąd w przypadku krótkotrwałych wątków ostateczny efekt może być przeciwny do zamierzonego.
c) dostęp wątków do rozłącznych obszarów pamięci. Przy niespełnieniu tego warunku wątki mogą wzajemnie się wstrzymywać z powodu sterylizacji dostępu do pamięci.

**Unikanie kosztownych operacji arytmetycznych.** Należy pamiętać że wykonywanie operacji arytmetycznych na liczbach całkowitych jest znacznie szybsze niż wykonywanie tych operacji na liczbach zmiennoprzecinkowych. Tam gdzie możliwe należy stosować arytmetykę całkowitoliczbową. Należy również pamiętać, że w zakresie arytmetyki zmiennoprzecinkowej operacja mnożenia zajmuje więcej cykli zegarowych niż operacja dodawania a operacja dzielenie zajmuje (zwykle) więcej cykli niż operacja mnożenia. Należy w miarę możliwości unikać operacji dzielenia.

**Dostęp do pamięci** staje się czynnikiem ograniczającym efektywność wtedy gdy program w sekwencji następujących po sobie obliczeń wykorzystuje dane ulokowane w pamięci pod odległymi adresami. Obowiązuje zasada lokalności dostępu, mówiąca, że program (zwykle) działa efektywniej gdy w sekwencji kolejnych następujących po sobie obliczeń wykorzystuje się dane ulokowane w pamięci pod nieodległymi adresami. Gdy zasada ta nie jest zachowana nie jest w pełni wykorzystywana pamięć podręczna. Zasada jej działania polegająca na wczytywaniu do niej obszarów pamięci RAM całymi wierszami prowadzi wówczas nawet do zwiększenia czasu dostępu do danych (z pamięci), nawet w stosunku do architektury w której nie zrealizowano mechanizmu pamięci podręcznej. Należy pamiętać, że kompilator dodatkowo optymalizuje obliczenia tak aby unikać intensywnego dostępu do pamięci, np. wykorzystując przechowywanie wyników pośrednich w rejestrach procesora. Programista jednak musi "dać szansę" kompilatorowi na wykonanie takich optymalizacji odpowiednio aranżując w swoim programie kolejność wykonywania operacji i organizując właściwie struktury danych wykorzystywane w najbardziej intensywnych obliczeniach.

W przypadku operacji macierzowych ta przyczyna nieefektywności najwyraźniej występuje w przypadku wyliczania iloczynu skalarnego wiersza jednej macierzy i kolumny innej macierzy. O ile dostęp do kolejnych elementów wiersza zachowuje zasadę lokalności i nie zakłóca pracy pamięci podręcznej, to w przypadku dużych macierzy dostęp do elementów tej samej kolumny znajdujących się w innych wierszach prowadzi do nieefektywnego wykorzystania pamięci podręcznej. Warto w programie tak zorganizować dane w pamięci aby uniknąć opisanej sytuacji. Istnieje wiele opisanych w literaturze i Internecie schematów mnożenia macierzy, które w znacznym stopniu redukują nieefektywności związane z niezachowywaniem zasady lokalności dostępu.

Dodatkowo na obniżenie efektywności może wpłynąć rozrzutna gospodarka pamięcią dynamiczną. W szczególności należy unikać częstego alokowania i zwalniania niewielkich obszarów pamięci na stercie, gdyż zarządzanie pamięcią jest czasochłonne a ponadto może doprowadzić do niekorygowanej segmentacji pamięci w obszarze sterty, co z kolei może spowodować, że długotrwałe obliczenia doprowadzą do naruszenia ograniczenia na łączną pamięć alokowaną (10GB), pomimo, że rozmiar pamięci rzeczywiście wykorzystywanej będzie znacznie mniejszy.

Aby uniknąć w/w efektów w przypadku implementacji metod i funkcji operatorowych należy w szczególności zadbać o efektywną implementację konstruktorów kopiujących i operatorów przypisania.

## 6. Przykładowe kody i wymagane interfejsy klasy, terminy zgłoszeń

Wymagane interfejsy klas, które należy dostarczyć znajdują się w plikach *CVct.h* i *CMtx.h*. Przykładowa implementacja znajduje się w pliku *Test_main.cpp*.

Żeby wziąć udział w konkursie należy do dnia 2021.01.18 do godziny 12:00 wysłać mail o tytule *Uwielbam C++, więc zgłaszam się do udziału w konkursie*, mail powinien zawierać w załączeniu pliki *CVct.h*, *CMtx.h*, *CVct.cpp* i *CMtx.cpp*. Jedna osoba może wysłać kod więcej niż jeden raz. W takim przypadku będzie brany kod z ostatniego maila.

**Mail zgłoszeniowy należy wysłać na dwa adresy**:

Jerzy.Sas@pwr.edu.pl
michal.przewozniczek@pwr.edu.pl

**Zachęcamy do udziału i życzmy powodzenia!!!**

## Literatura:

1. Fialko S. Modelowanie zagadnień technicznych. (Skrypt). Politechnika Krakowska, 2011 (https://torus.uck.pk.edu.pl/~fialko/text/MZT1/MZT.pdf)
2. Press W. H. et al., Numerical Recipes in C, Cambridge University Press, 2003 (dostępny jako pdf)
3. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial4.html, (dotyczy zabronionego GPU, ale opis problemu i metod również do wykorzystania w CPU)
4. https://gist.github.com/nadavrot/5b35d44e8ba3dd718e595e40184d03f0 (schemat efektywnego mnożenia macierzy)
5. https://en.wikipedia.org/wiki/Adjacency_matrix (zastosowanie operacji macierzowych dla dużych macierzy)
6. http://www.ams.org/publicoutreach/feature-column/fcarc-pagerank (zastosowanie operacji macierzowych dla dużych macierzy)
7. Meyers S. Język C++ bardziej efektywny, WNT, 1998 (pozycja trochę przestarzała, ale ogólne zasady pozostają aktualne).