

Programowanie równoległe i rozproszone -

Projekt zespołowy - Etap 2.

Adrian Cieśla, Mateusz Król, Bartłomiej Papis, Bartosz Wużyński

Styczeń 2026

1. Wstęp

W niniejszej pracy nasz zespół skupił się na rozwiązaniu zadania 11, polegającego na faktoryzacji Doolittle'a. Przypomnijmy, elementy macierzy dolnej

lewostronnej L i górnej prawostronnej U takich że $A = LU$ są obliczane w następujący sposób k – numer iteracji:

$$u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj}, \quad j = k, k+1, \dots, n$$
$$l_{ik} = \frac{a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk}}{u_{kk}}, \quad i = k, k+1, \dots, n$$

Macierz A powinna być generowana w sposób losowy. Poprawność rozwiązania powinna być liczona z wykorzystaniem normy Frobeniusa.

Praca wykorzystuje rozwiązania zaimplementowane w etapie pierwszym projektu. Na potrzeby etapu drugiego dodano dwa dodatkowe warianty obliczeń:

- Zrównoleglenie wykorzystujące komunikaty
- Obliczenia GPGPU

Większość kodu została napisana w języku C. Do realizacji rozwiązania GPGPU wykorzystano technologię CUDA dostarczaną przez firmę NVIDIA, kompilowaną za pomocą narzędzia nvcc.

Ze względu na sprzętową zależność technologii CUDA od kart graficznych NVIDIA, przygotowano plik Makefile, który podczas kompilacji sprawdza dostępność środowiska CUDA i w zależności od możliwości platformy użytkownika włącza lub wyłącza tryb GPGPU.

2. GPGPU

W ramach drugiego etapu projektu zaimplementowano wariant obliczeń wykorzystujący technikę GPGPU (General-Purpose computing on Graphics Processing Units). GPGPU polega na wykonywaniu obliczeń ogólnego przeznaczenia na procesorze graficznym, który dzięki dużej liczbie równoległe działających rdzeni umożliwia znaczące przyspieszenie obliczeń numerycznych.

W niniejszym projekcie do realizacji obliczeń GPGPU wykorzystano technologię CUDA (Compute Unified Device Architecture) opracowaną przez firmę NVIDIA. CUDA umożliwia programowanie procesora graficznego z poziomu języka C/C++ poprzez definiowanie funkcji wykonywanych na GPU (tzw. kerneli) oraz jawne zarządzanie pamięcią urządzenia.

Model programowania CUDA opiera się na hierarchii wątków, bloków oraz siatek (grid). Każdy kernel uruchamiany jest równoległe przez dużą liczbę wątków, które identyfikowane są za pomocą indeksów wątków i bloków. Dzięki temu możliwe jest rozdzielenie pracy pomiędzy wiele wątków GPU, z których każdy wykonuje obliczenia dla innego fragmentu danych.

Wersja GPGPU stanowi uzupełnienie wcześniej zaimplementowanych wariantów sekwencyjnych, wątkowych oraz procesowych i umożliwia porównanie efektywności obliczeń wykonywanych na CPU i GPU dla różnych rozmiarów macierzy.

2.1. Podział pracy w CUDA

W implementacji GPGPU algorytmu LU (Doolittle) podział pracy pomiędzy wątki procesora graficznego został zrealizowany zgodnie ze strukturą algorytmu sekwencyjnego. W każdej iteracji algorytmu, oznaczonej indeksem k , możliwe jest równoległe obliczanie wielu elementów macierzy, ponieważ wartości wyznaczane w obrębie tego samego etapu nie zależą od siebie nawzajem.

Z tego względu obliczenia zostały podzielone na dwa kolejne etapy. W pierwszym etapie równolegle wyznaczane są elementy wiersza macierzy U dla ustalonego indeksu k , natomiast w drugim etapie równolegle obliczane są elementy kolumny macierzy L dla tego samego indeksu. Taki podział bezpośrednio odpowiada postaci algorytmu Doolittle'a i pozwala na wykorzystanie równoległości dostępnej na poziomie pojedynczych elementów macierzy.

Każdy wątek GPU odpowiada za obliczenie pojedynczego elementu macierzy, a zakres jego pracy określany jest na podstawie indeksów wątków i bloków w modelu programowania CUDA. Przyjęty sposób podziału pracy umożliwia efektywne wykorzystanie dużej liczby wątków dostępnych na procesorze graficznym, przy jednoczesnym zachowaniu zgodności z algorytmem sekwencyjnym.

2.2. Synchronizacja obliczeń

Algorytm LU (Doolittle) charakteryzuje się istotnymi zależnościami danych pomiędzy kolejnymi etapami obliczeń w ramach jednej iteracji k . W szczególności obliczenie elementów kolumny macierzy L wymaga wcześniejszego wyznaczenia elementów wiersza macierzy U , w tym elementu $U[k][k]$, który pełni rolę dzielnika.

W celu zapewnienia poprawnej kolejności obliczeń pomiędzy kolejnymi etapami zastosowano jawną synchronizację po każdym uruchomieniu kernela. Synchronizacja realizowana jest przy użyciu funkcji `cudaDeviceSynchronize()`, która gwarantuje zakończenie obliczeń wykonywanych przez kernel przed przejściem do kolejnego etapu iteracji.

Takie podejście odpowiada barierom synchronizującym stosowanym w wersjach z wątkami oraz procesami współbieżnymi i zapewnia poprawność wyników przy zachowaniu równoległości obliczeń w obrębie pojedynczych etapów.

2.3. Zarządzanie pamięcią GPU i wykorzystane funkcje CUDA

W implementacji GPGPU algorytmu LU (Doolittle) wykorzystano podstawowe mechanizmy zarządzania pamięcią oferowane przez technologię CUDA. Przed rozpoczęciem obliczeń na procesorze graficznym alokowana jest pamięć urządzenia dla macierzy A, L oraz U przy użyciu funkcji *cudaMalloc*.

Dane wejściowe są następnie kopiowane z pamięci hosta do pamięci GPU za pomocą funkcji *cudaMemcpy*. Operacja ta wykonywana jest jednorazowo, przed rozpoczęciem właściwych obliczeń, co pozwala uniknąć wielokrotnego przesyłania danych w trakcie kolejnych iteracji algorytmu.

Obliczenia realizowane są w funkcjach typu kernel, oznaczonych kwalifikatorem *__global__*, które uruchamiane są równolegle przez dużą liczbę wątków GPU. Ze względu na zależności danych występujące w algorytmie LU, po każdym uruchomieniu kernela zastosowano funkcję *cudaDeviceSynchronize*, zapewniającą poprawną kolejność wykonywania obliczeń.

Po zakończeniu pełnej faktoryzacji LU wyniki obliczeń są kopiowane z powrotem do pamięci hosta, a zaalokowana pamięć GPU zwalniana jest przy użyciu funkcji *cudaFree*.

2.4. Poprawność obliczeń i weryfikacja wyników

W celu potwierdzenia poprawności działania implementacji GPGPU wyniki uzyskane w technologii CUDA zostały porównane z wynikami otrzymanymi w wersji sekwencyjnej algorytmu LU (Doolittle). Do weryfikacji poprawności obliczeń zastosowano normę Frobeniusa różnicy pomiędzy macierzą wejściową A oraz iloczynem macierzy L i U.

Norma Frobeniusa obliczana jest zgodnie ze wzorem

$$\|A - LU\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^n (A_{ij} - (LU)_{ij})^2}$$

Po zakończeniu obliczeń GPGPU macierze L oraz U są kopiowane z pamięci procesora graficznego do pamięci hosta, co umożliwia wykonanie weryfikacji w identyczny sposób jak w pozostałych wariantach algorytmu. Dzięki temu porównanie wyników jest spójne dla wszystkich implementacji: sekwencyjnej, wątkowej, procesowej oraz GPGPU.

Przeprowadzone testy wykazały, że wartości normy Frobeniusa dla wersji GPGPU są porównywalne z wartościami uzyskiwanymi w pozostałych wariantach algorytmu i mieszczą się w granicach błędów numerycznych charakterystycznych dla obliczeń zmiennoprzecinkowych. Oznacza to, że implementacja GPGPU poprawnie realizuje algorytm LU (Doolittle) i prowadzi do prawidłowych wyników obliczeń.

2.5. Porównanie z wersją sekwencyjną

W niniejszej części porównamy wyniki czasów wykonania programu w wersji sekwencyjnej oraz w wersji z obliczeniami GPGPU.

n	T (n, 1)[s]	T (n, GPU)[s]	S(n,GPU)
100	0.000390	0.092899	0.004198
400	0.019662	0.304565	0.064558
1000	0.392240	0.540006	0.726362
2000	7.461513	1.064930	7.006576
4000	207.371628	5.385054	38.508737

Tabela 1: Wyniki czasów działania algorytmu w wersji sekwencyjnej i GPGPU oraz przyspieszenia w zależności od rozmiaru macierzy n.

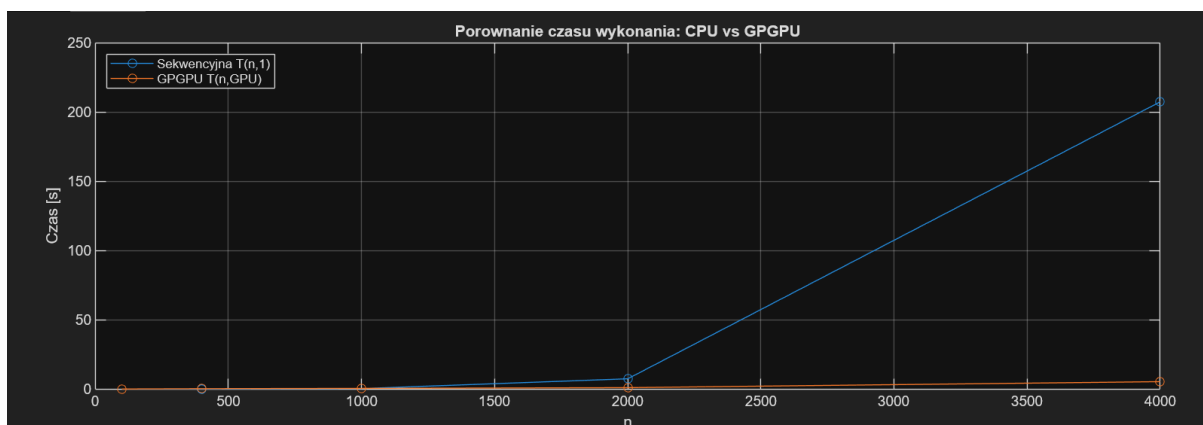
Na podstawie przeprowadzonych pomiarów czasu wykonania można zauważyć istotną zależność pomiędzy rozmiarem macierzy a efektywnością zastosowania obliczeń GPGPU. Dla małych rozmiarów macierzy, takich jak $n=100$ oraz $n=400$, wersja GPGPU osiąga znacznie gorsze wyniki czasowe w porównaniu do implementacji sekwencyjnej. Wynika to z faktu, że całkowity czas obliczeń jest w tym przypadku zdominowany przez narzut związany z

przygotowaniem obliczeń na procesorze graficznym, w szczególności alokację pamięci GPU, kopiowanie danych pomiędzy pamięcią hosta i urządzenia oraz synchronizację wykonywania kerneli.

Wraz ze wzrostem rozmiaru macierzy wpływ narzutu związanego z obsługą GPU staje się coraz mniej istotny w porównaniu do czasu właściwych obliczeń. Dla $n=1000$ czasy wykonania wersji sekwencyjnej i GPGPU są już porównywalne, a dalsze zwiększanie rozmiaru problemu prowadzi do wyraźnej przewagi rozwiązania GPGPU.

Dla dużych rozmiarów macierzy, takich jak $n=2000$ oraz $n=4000$, zastosowanie obliczeń na procesorze graficznym pozwala na uzyskanie znaczącego przyspieszenia względem wersji sekwencyjnej. W przypadku największego testowanego rozmiaru macierzy uzyskane przyspieszenie przekracza kilkadziesiąt razy, co potwierdza zasadność wykorzystania technologii GPGPU do rozwiązywania dużych problemów obliczeniowych.

Należy również zwrócić uwagę, że w przypadku implementacji GPGPU nie występuje bezpośredni odpowiednik parametru p , stosowanego w wersjach wątkowej i procesowej. W technologii CUDA liczba wątków oraz sposób ich mapowania na sprzętowe jednostki wykonawcze są zarządzane automatycznie przez procesor graficzny. Z tego względu w analizie przyjęto jedną konfigurację GPU, a uzyskane przyspieszenie odniesiono bezpośrednio do wersji sekwencyjnej algorytmu.



Rysunek 1: Wykres czasu wykonania programu w zależności od n dla wersji sekwencyjnej i GPGPU

3. Zrównoleglenie wykorzystujące komunikaty

W ramach drugiego etapu projektu zaimplementowano wariant równoległy oparty o przekazywanie komunikatów pomiędzy procesami. W przeciwieństwie do podejścia z pamięcią współdzieloną, w którym kilka jednostek wykonawczych modyfikuje te same struktury danych, tutaj proces nadrzędny, inaczej rodzic, rozdziela pracę na procesy potomne, a następnie odbiera od nich obliczone fragmenty wyników w postaci komunikatów przesyłanych przez potoki.

Celem tego wariantu było wykorzystanie równoległości występującej w algorytmie LU Doolittle wewnątrz pojedynczej iteracji k : dla ustalonego k wiele elementów wiersza macierzy U oraz wiele elementów kolumny macierzy L można wyznaczać niezależnie, o ile znane są wartości z poprzednich iteracji.

3.1. Model obliczeń master-worker i podział pracy

Implementacja realizuje schemat master-worker:

- **Rodzic** steruje przebiegiem iteracji $k = 0..n - 1$, tworzy procesy potomne, a następnie scala wyniki w macierzach L i U .
- **Procesy potomne** obliczają tylko przydzielony fragment, blok indeksów i odsyłają gotowe wartości do procesu nadrzędnego.

Liczba procesów roboczych jest dobierana dynamicznie na podstawie liczby dostępnych rdzeni CPU, a następnie ograniczana do $p \leq n$, aby uniknąć sytuacji, w której liczba procesów przekracza liczbę elementów do policzenia.

Dla każdego k praca jest dzielona na dwa etapy, zgodnie ze strukturą Doolittle'a:

1. Wyznaczenie **wiersza** $U[k][j]$ dla $j = k..n - 1$
2. Wyznaczenie **kolumny** $L[i][k]$ dla $i = k + 1..n - 1$

W każdym etapie zakres indeksów jest dzielony na możliwie równe „kawałki” inaczej „chunk”, np. dla wiersza U :

- $\text{totalU} = n - k$
- $\text{chunkU} = \text{ceil}(\text{totalU} / p)$
- proces w liczy $j \in [k + w \cdot \text{chunkU}, \min(k + (w + 1) \cdot \text{chunkU}, n))$

Analogicznie dla kolumny L w zakresie $i = k + 1..n - 1$.

3.2. Komunikacja przez pipe w formie komunikatów

Do przesyłania wyników z procesów potomnych do rodzica użyto potoków systemowych pipe. Dla każdego procesu roboczego tworzona jest osobna para deskryptorów:

- potomek zamyka koniec do odczytu i zapisuje do potoku ciągły bufor typu *double[]* zawierający obliczone wartości,
- rodzic zamyka koniec do zapisu i odczytuje dokładnie tyle bajtów, ile odpowiada długości przydzielonego fragmentu.

Aby zapewnić poprawność transmisji, czyli brak „uciętych” zapisów lub odczytów, zastosowano pomocnicze funkcje *write_all()* oraz *read_all()*, które:

- obsługują częściowe operacje *write/read*,
- ponawiają wywołania w przypadku przerwania sygnałem (EINTR),
- gwarantują przesłanie/odbiór całego bufora.

Po odebraniu wyników rodzic scala je w odpowiednich miejscach macierzy:

- dla U możliwe jest bezpośrednie *memcpy()* do spójnego fragmentu wiersza,
- dla L wartości są przypisywane do pozycji $L[i \cdot n + k]$, ponieważ jest to kolumna w reprezentacji jednowymiarowej.

3.3. Synchronizacja i zależności danych

Algorytm LU (Doolittle) narzuca istotne zależności:

- elementy $U[k][j]$ zależą od wartości $L[k][m]$ oraz $U[m][j]$ z wcześniejszych iteracji $m < k$,
- elementy $L[i][k]$ zależą od $U[k][k]$ oraz od $L[i][m]$, $U[m][k]$ dla $m < k$.

Z tego powodu w każdej iteracji k zastosowano jawną barierę pomiędzy etapami:

1. rodzic uruchamia procesy liczące wiersz U ,
2. czeka na wyniki (odczyt z potoków + `waitpid()`),
3. dopiero wtedy uruchamia procesy liczące kolumnę L ,
4. ponownie czeka na wyniki.

Takie podejście zapewnia, że wszystkie elementy $U[k][*]$ (w tym kluczowy dzielnik $U[k][k]$) są gotowe, zanim rozpocznie się liczenie $L[*][k]$.

Warto podkreślić, że procesy potomne w tym wariantcie nie modyfikują wspólnych struktur L i U bezpośrednio — jedynie odsyłają wyniki. Dzięki temu unika się wyścigów i potrzeby stosowania blokad, a spójność jest wymuszana przez etapowe „zbieranie” wyników w procesie nadrzędnym.

3.4. Obsługa przypadków brzegowych i stabilność numeryczna

Implementacja zawiera zabezpieczenie przed dzieleniem przez wartość bliską zeru:

- po wyznaczeniu wiersza U sprawdzany jest warunek $|U[k][k]| < 10^{-15}$,
- w takim przypadku zgłaszany jest błąd numeryczny i obliczenia są przerywane.

Jest to istotne, ponieważ w projekcie nie zastosowano pivotingu, a w praktyce dla niektórych macierzy może dojść do sytuacji, w której element diagonalny $U[k][k]$ jest zerowy lub bardzo mały.

3.5. Koszty narzutu i oczekiwane własności wydajnościowe

Równoleglenie na komunikatach zapewnia przyspieszenie obliczeń wewnątrz etapu (wiersz U lub kolumna L), jednak wiąże się z narzutami:

- tworzenie procesów (fork) w każdej iteracji i dla obu etapów,
- tworzenie potoków i kopiowanie wyników przez jądro systemu (IPC),
- konieczność *waitpid()* i etapowej synchronizacji.

Z tego względu wariant ten jest najbardziej sensowny dla większych rozmiarów macierzy, gdzie koszt obliczeń dominuje nad kosztem komunikacji, natomiast dla małych n narzut może zniwelować zyski z równoległości.

3.6. Weryfikacja poprawności

Poprawność wyników w wariacie z komunikatami weryfikowana jest analogicznie jak w pozostałych implementacjach: po wyznaczeniu L i U obliczana jest norma Frobeniusa dla różnicy $A - LU$. Pozwala to porównywać spójnie wszystkie wersje programu (sekwencyjną, wątkową, procesową, komunikatową i GPGPU).