UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE

ROZPRAWA DOKTORSKA

Równoległe i wektorowe algorytmy rozwiązywania trójdiagonalnych układów równań liniowych typu Toeplitza na współczesnych architekturach wieloprocesorowych

Beata Dmitruk

Praca wykonana pod kierunkiem
dra hab. Przemysława Stpiczyńskiego, prof. UMCS

Lublin 2024

Składam serdeczne podziękowania
Profesorowi Przemysławowi Stpiczyńskiemu
za nieocenioną pomoc podczas
przygotowywania niniejszej rozprawy.
Pragnę również wyrazić wdzięczność za poświęcony
czas i opiekę w całej dotychczasowej karierze naukowej

Spis treści

Streszczenie w języku polskim

Przewodnim celem niniejszej rozprawy doktorskiej jest opracowanie wydajnych i dokładnych algorytmów numerycznego rozwiązywania układów równań liniowych z macierzą współczynników trójdiagonalną typu Toeplitza oraz dokładniejszych metod sumujących bez zmniejszenia wydajności czasowej, których wektorowo-równoległe implementacje umożliwiają dobre wykorzystanie własności współczesnych architektur wieloprocesorowych.

Pierwszym z uzyskanych wyników jest sformułowanie algorytmów rozwiązywania omawianych układów z uwzględnieniem budowy współczesnych komputerów. Następnie utworzone zostały implementacje w języku C z wykorzystaniem interfejsów pozwalających na włączenie procesów równoległości, tj. OpenMP (na CPU) i OpenACC (implementacje przenośne: CPU i GPU). Następnie przedstawiono różne formaty danych, pozwalające na lepszy dostęp do pamięci. Oprócz standardowego kolumnowego formatu danych przedstawiono również format wierszowy, jak i efektywną zamianę pomiędzy nimi z wykorzystaniem pamięci podręcznej oraz implementacje, w której obliczenia są przeprowadzane z wykorzystaniem bloku pamięci podręcznej bez zamiany całej tablicy danych. Wykorzystanie innych formatów znacząco przyspieszyło działanie funkcji na procesorach graficznych, wliczając nawet czas potrzebny na konwersję formatów. Pokazano implementacje przenośne pomiędzy CPU i GPU, heterogeniczne - działające na dwóch kartach graficznych oraz hybrydowe działające jednocześnie na CPU i GPU.

Przedstawiono również implementacje algorytmów Kahana i Gilla-Møllera w sposób pozwalający na włączenie procesów wektorowych za pomocą funkcji wbudowanych (ang. *intrinsics*) oraz procesu równoległości przy użyciu OpenMP. Wyniki dokładności obu algorytmów sumowania z poprawkami są znacząco lepsze niż wykorzystanie zwykłego sumowania, a funkcje zrównoleglone i zwektoryzowane działają w znacząco krótszym czasie niż zwykłe sumowanie. Funkcje tylko zwektoryzowane działają w czasie podobnym do zwykłego sumowania, a jednocześnie poprawiają dokładność wyniku. Następnie podejście to wykorzystano w algorytmach całkowania numerycznego oraz w metodzie rozwiązywania szczególnego przykładu trójdiagonalnego układu równań liniowych typu Toeplitza. W obu rozpatrywanych przykładach wykorzystanie sumowania z poprawkami poprawiło dokładność wyników.

Otrzymane rezultaty potwierdzają słuszność tezy stawianej w niniejszej rozprawie, że zrównoleglenie i wektoryzacja pozwalają na znaczące przyspieszenie działania implementacji algorytmów numerycznych rozwiązujących układy równań o macierzach trójdiagonalnych typu Toeplitza na współczesnych procesorach, procesorach graficznych oraz architekturach hybrydowych.

Streszczenie w języku angielskim

The primary objective of this doctoral dissertation is to develop efficient and accurate algorithms for solving linear systems of equations with a tridiagonal Toeplitz coefficient matrix, as well as more precise summation methods without loss of time efficiency. Vectorization and parallelization of these algorithms aim to utilize modern multiprocessor architectures effectively.

The first set of results involves formulating algorithms for solving the discussed systems, taking into account the architecture properties of modern computers. Implementations have been developed in the C language using interfaces that enable parallel processing, namely OpenMP (on CPU) and OpenACC (portable implementations: CPU and GPU). Novel data formats have been developed to improve memory access. The usage of different formats significantly accelerates the performance of the functions on graphics processors, including the time required for format conversion. Portable implementations between CPU and GPU, heterogeneous implementations operating on two graphics processing units, and hybrid implementations simultaneously running on CPU and GPU have been presented. Apart from the conventional column-wise data format, various data structures have been introduced. These include pure row-wise format and row-wise format with efficient conversion using cache memory and the implementation where computations are performed using cache blocks without swapping entire data arrays.

Implementations of Kahane's and Gill-Moller's algorithms have been also presented. It allows to utilize vector extensions using intrinsic functions and parallel processing using OpenMP. Accuracy results of both compensated summation algorithms are significantly better than using ordinary summation. The vectorized and parallelized functions operate significantly faster than ordinary summation. Vectorized functions run in a time similar to ordinary summation while improving the accuracy of the results. Then, this approach has been applied to the problem of numerical integration and a method for solving a specific case of tridiagonal Toeplitz linear systems. In both cases, the use of summation with corrections improves the accuracy of results.

The obtained results confirm the thesis presented in this dissertation, namely that parallelization and vectorization allow for significant acceleration of the implementation of numerical algorithms for solving systems of equations with tridiagonal Toeplitz matrices on modern processors, graphics processing units, and hybrid architectures.

Wstęp

Obliczenia sekwencyjne polegają na realizacji każdej z instrukcji jedna po drugiej. Współczesne procesory wielordzeniowe i komputery wieloprocesorowe umożliwiają równoległe podejście do wykonywania poleceń. Znacznym ograniczeniem w zrówleglaniu są obliczenia rekurencyjne, które nie zawsze mogą zostać zoptymalizowane przez kompilatory tak, aby wykorzystać własności procesorów wielordzeniowych. Każda oszczędność czasu trwania programu ma potencjał przyniesienia znaczących korzyści, zwłaszcza gdy dana implementacja jest wielokrotnie uruchamiana. Mimo coraz lepszych kompilatorów, ciągle istnieje trudność w automatycznym wykorzystaniu potencjału równoległości i wektorowości, zwłaszcza w przypadku złożonych procesów rekurencyjnych.

Niniejsza praca przedstawia problematykę efektywnego rozwiązywania wybranych algorytmów numerycznych na współczesnych architekturach komputerowych. Przez efektywne rozumie się programy działające szybciej i/lub z lepszą dokładnością niż znane do tej pory metody.

Należy zwrócić uwagę na fundamentalny trójkąt: sprzęt - algorytmy - kompilatory [EGJK04]. Wynika z niego, że powinno się tworzyć algorytmy, które umożliwią kompilatorom adaptację kodu do architektury komputerów. Jest to nieodłączną częścią postępu w dziedzinie informatyki obliczeniowej. Dlatego, aby efektywnie rozwiązywać badane układy równań, poszukuje się nowych metod zapewniających szybkość, wydajność i dokładność rozwiązań, które wykorzystują potencjał wieloprocesorowości.

Znane podejścia do rozwiązywania omawianych problemów nie są łatwo przenośne na programy równoległe i wektorowe. Wcześniejsze rozwiązania dążyły do jak najmniejszej liczby operacji, jednak efektywność algorytmów nie zawsze koreluje z liczbą operacji; często te z większą ich liczbą mogą być efektywnie zrównoleglone lub zwektoryzowane. Proces ten wymaga opracowania nowych struktur danych i implementacji, które mogą w wydajny sposób wykorzystać potencjał równoległości, szczególnie w obliczeniach na kartach graficznych, gdzie podejście SIMD odgrywa kluczową rolę [Ebe14].

W niniejszej pracy zaproponowano efektywne algorytmy rozwiązujące szczególne przypadki układów równań liniowych, tj. z macierzą współczynników trójdiagonalną typu Toeplitza. Układy takie występują w algorytmach numerycznych do rozwiązywania problemów wartości granicznych zwyczajnych i cząstkowych równań różniczkowych [SP11, VA12]. Wykorzystuje się je również w interpolacji kawałkami sześciennej (ang. *piecewise cubic interpolation*), w algorytmach splajnowych (ang. *spline algorithms*) [CY96a, SP06, Ter16], w metodzie ADI (ang. *Alternating Direction Implicit*) [Ter16] i wiele innych [Pot14, SCB05, Saa03, QS04, QS04, WE17, Far11].

Rozwiązywaniu szczególnych układów równań liniowych poświęcono wiele uwagi. Podstawowa idea pochodzi od Rojo [Roj90]. Zaproponował on metodę dla układów trójdiagonalych symetrycznych z wykorzystaniem dekompozycji macierzy współczynników na LU oraz równań Shermana-Morrisona [DSZ18]. Podejście to zostało zmodyfikowane w celu otrzymania nowych solwerów umożliwiających zrówloleglanie obli-

czeń [VA12, GS01]. Zwektoryzowany, lecz nie zrównoleglony alogorytm został zaproponowany w [CY96b]. Inne podejście wykorzystali McNally i in. [MGS08] w skalowalnym algorytmie z ograniczoną komunikacją pomiędzy danymi. Znajduje on przybliżenie dokładnego rozwiązania z zadaną precyzją, lecz nie wykorzystuje wektoryzacji. Terekhov [Ter16] zaproponował wysoce skalowalny, zrównoleglony algorytm do rozwiązywania układów równań z wieloma prawymi stronami.

Zauważyć należy, że znane biblioteki numeryczne takie jak MKL, LAPACK czy NAG nie zapewniają funkcji do rozwiązywania trójdiagonalych układów Toeplitza. Zapewniają natomiast funkcje dla bardziej ogólnych układów.

Dotychczasowe prace badawcze nad rozwiązywaniem trójdiagonalnych układów równań liniowych typu Toeplitza autorka zawarła w trzech artykułach [DS20, DS21, DS22].

Kolejnym elementem niniejszej pracy jest efektywne - pod względem czasowym i dokładności - sumowanie długich ciągów liczb. Jest to jeden z najważniejszych i najczęściej występujący element algorytmów numerycznych, wpływający znacząco na dokładność i stabilność wielu algorytmów. Dlatego powstało wiele algorytmów efektywniejszych niż zwykłe, iteracyjne sumowanie [Hig96]. Jeżeli pożądana jest większa dokładność, a czas wykonania jest mniej istotny, można skorzystać z bardziej wyszukanych algorytmów sumowania [ADN20]. Znane są dwie podstawowe metody sumowania z poprawkami: metoda Kahana [Kah65] i algorytm Gilla-Møllera [Mø65]. Są to względnie proste metody, jednak ich automatyczna optymalizacja przez kompilator nie jest możliwa ze względu na zależności między danymi wykorzystywanymi w kolejnych krokach iteracji. Praca [NDMR20] pokazuje wektoryzacje metody Kahana.

Dotychczasowe prace nad tym zagadnieniem autorka opublikowała w dwóch artykułach [DS23b, DS23a].

Cel i teza rozprawy

Celem rozprawy jest opracowanie wydajnych i dokładniejszych algorytmów numerycznego rozwiązywania układów równań liniowych z macierzą współczynników trójdiagonalną typu Toeplitza oraz dokładniejszych metod sumujących bez zmniejszenia wydajności czasowej, których wektorowo-równoległe implementacje umożliwiałyby dobre wykorzystanie własności współczesnych architektur wieloprocesorowych.

W związku z tak postawionym celem została sformułowana teza: **Zrównoleglenie i wektoryzacja pozwalają na znaczące przyspieszenie działania implementacji algorytmów numerycznych rozwiązujących układy równań o macierzach trójdiagonalnych typu Toeplitza na współczesnych procesorach, procesorach graficznych oraz architekturach hybrydowych.**

Aby osiągnąć tak postawiony cel postuluje się wykonanie następujących kroków:

- sformułowanie algorytmów pozwalających na implementacje, które będą wykorzystywały własności architektur współczesnych systemów komputerowych,
- implementacja ww. algorytmów z wykorzystaniem narzędzi wspierających wektoryzację i obliczenia równoległe oraz przenośność pomiędzy architekturami,
- wykorzystanie nowych reprezentacji danych (macierzy, wektorów) pozwalających na lepszy dostęp do pamięci i zredukowanie liczby niezbędnych synchronizacji,

- zwiększenie dokładności obliczeń poprzez wykorzystanie algorytmów sumowania z poprawkami,
- wykorzystanie obliczeń w różnych formatach reprezentujących liczby zmiennopozycyjne (pojedyncza, podwójna i mieszana precyzja),
- przeprowadzenie testów, analiza wyników czasowych, dokładności, dobór parametrów, próba automatyzacji doboru parametrów.

Struktura rozprawy

Niniejsza praca składa się z sześciu rozdziałów, wstępu, podsumowania oraz bibliografii.

Rozdział pierwszy stanowi wprowadzenie do obliczeń równoległych. Ponadto, w rozdziale tym omówione zostają narzędzia programistyczne wykorzystywane do tworzenia programów zaprezentowanych w niniejszej pracy. Opisano również architektury, na których zebrano wyniki stanowiące część niniejszej rozprawy. Jako ostatni punkt tego rozdziału omówiono sposoby weryfikacji wydajności omawianych implementacji.

Rozdział drugi prezentuje tematykę związaną z układami równań liniowych. Wprowadza w układy z macierzą współczynników trójdiagonalną typu Toeplitza. Przedstawia zastosowania omawianych układów, znane algorytmy do ich rozwiązywania oraz prezentuje ich rozwinięcia oraz inne podejścia do rozwiązywania omawianych układów opisane w literaturze. Omawia również istniejące oprogramowanie pozwalające na rozwiązywanie specjalnych układów równań liniowych.

Kolejne rozdziały, tj. trzeci, czwarty, piąty i szósty przedstawiają oryginalne wyniki badawcze.

Rozdział trzeci zajmuje tematyka związana z trójdiagonalnymi układami równań linowych typu Toeplitza postaci $(1, t_2, t_3)$. Zaprezentowano dwie wersje równoległego solwera typu *dziel i zwyciężaj* (ang. *divide and conquer*), opartego na znanym algorytmie rozwiązywania wyżej wymienionych układów. Pokazano jak zredukować liczbę niezbędnych synchronizacji i jak wyrównywać dane, aby optymalnie wykorzystać pamięć podręczną. Zaprezentowane implementacje osiągają bardzo dobre przyspieszenie na architekturach wielordzeniowych oraz są bardziej energooszczędne niż prosta implementacja sekwencyjna.

Rozdział czwarty kontynuuje tematykę trójdiagonalnych układów równań linowych typu Toeplitza, jednak postać badanych układów to (t_1, t_2, t_3). Przedstawiono implementacje dwóch algorytmów - pierwszy oparty na metodzie Wanga [Wan81] oraz drugi, oparty na sekwencyjnym algorytmie przedstawionym przez Liu i in. [LLYZ20]. Zaproponowano implementacje są przenośne, a więc mogą zostać wykonane zarówno na CPU jak i GPU, a bardziej zaawansowane wersje mogą zostać wykonane na wielu GPU lub na architekturze hybrydowej. Oprócz standardowego kolumnowego formatu danych przedstawiono również format wierszowy, jak i efektywną zamianę pomiędzy nimi z wykorzystaniem pamięci podręcznej oraz implementacje, w której obliczenia są przeprowadzane w bloku pamięci podręcznej bez zamiany całej tablicy danych. Rozważono również sposób predykcji parametrów przedstawionego algorytmu. Przeprowadzone eksperymenty wykazały dobrą wydajność i dokładność algorytmów.

Rozdział piąty związany jest z tematyką efektywnego sumowania. Zawiera wprowadzenie do tematyki sumowania z poprawkami. Przedstawia dwa znane algorytmy: Kahana oraz Gilla-Møllera, pozwalające na osiąganie wysokiej dokładności sumowania dużych ciągów liczbowych. Pokazano również sposób wektoryzacji i zrównoleglenia

z wykorzystaniem funkcji wbudowanych oraz OpenMP. Zaprezentowano wyniki eksperymentów w pojedynczej, w podwójnej oraz w mieszanej precyzji. Pokazały one, że zaproponowane implementacje osiągają znacznie większą dokładność niż iteracyjne sumowanie, a czas działania jest z nim porównywalny.

Rozdział szósty jest ostatnim rozdziałem niniejszej rozprawy. Przedstawia on efekty wykorzystania sumowania z poprawkami w znanych algorytmach. Pierwszy przykład to całkowanie numeryczne. Wybrano kilka znanych metod całkowania numerycznego oraz uzupełniono te algorytmy o sumowanie z poprawkami wykorzystując algorytmy Kahana oraz Gilla-Møllera i zbadano otrzymane dokładności. Drugi przykład to zagadnienie warunków brzegowych równania różniczkowego, czyli rozwiązywanie szczególnego układu równań liniowych wzbogacone o algorytm Kahana. Dla tego przykładu zebrano wyniki dokładności jak i czasu trwania programów.

W podsumowaniu znalazły się konkluzje płynące z niniejszej rozprawy oraz kierunki dalszych badań.

Należy dodać, że wszystkie implementacje przedstawione w niniejszej rozprawie są ogólnodostępne i można jest znaleźć w publicznym repozytorium pod adresem *https://github.com/beatadmitruk/doktorat_kody*.

Rozdział 1

Wprowadzenie do obliczeń równoległych

1.1 Wstęp

Kompilatory mogą samodzielnie przeprowadzać pewne optymalizacje kodu. Występują różne rodzaje takich działań. Podstawowymi rodzajami są optymalizacja maszynowa (dotyczy właściwego wykorzystania architektury) i niezależna sprzętowo optymalizacja skalarna (wykorzystująca m.in. upraszczanie wyrażeń, eliminację wspólnych fragmentów kodów, eliminację nadmiarowych podstawień i inne; dzielona na lokalną - w ramach bloków instrukcji prostych i globalną - w obrębie całych podprogramów) [ASU02, Stp08].

Problem stanowią jednak algorytmy sekwencyjne, polegające na realizacji każdej z instrukcji sekwencyjnie - jedna po drugiej, bez wykonywania innych procesów jednocześnie. W algorytmach tych często występują zależności między danymi przetwarzanymi w kolejnych krokach. Wtedy, nawet w przypadku względnie prostych algorytmów nie jest możliwa ich automatyczna optymalizacja. Zatem implementując takie metody nie wykorzystuje się własności architektury współczesnych komputerów wielordzeniowych i wieloprocesorowych, co jest kluczowe w otrzymywaniu efektywnych programów. Potrzebne jest więc ciągłe doskonalenie metod obliczeniowych tak, aby mogły być efektywnie zaimplementowane na współczesne architektury. Wiodące są w tym dwa kolejne sposoby optymalizacji: wektorowa i równoległa.

Wektoryzacja jest procesem, który przekształca skalarne operacje na pojedynczych elementach (ang. *Single Instruction Single Data—SISD*) na operacje, gdzie jedna instrukcja wykonywana jest jednocześnie dla wielu elementów np. całego wektora lub macierzy (ang. *Single Instruction Multiple Data—SIMD*) [RF15]. Wiele działań występujących w obliczeniach naukowych jest zdefiniowanych jako działania wektorowe lub macierzowe. Aby wykorzystać ten fakt, projektowane są architektury, w których można realizować instrukcje dla całych zbiorów danych [SB11]. Powstają również nowe klasy algorytmów pozwalające na wykorzystywanie jednostek wektorowych [GBWS18, BE93, Sun01, Ter16, ACD+01, AALZ00]. We współczesnych procesorach wielordzeniowych wektoryzacja odbywa się na poziomie rozszerzeń SIMD, czyli AVX, AVX-2, AVX-512 i ma główny wpływ na osiągane przyspieszenia względem algorytmów sekwencyjnych.

Zrównoleglenie jest procesem, który pozwala na przeprowadzenie wielu operacji w tym samym czasie. Programowanie równoległe jest zasadniczo innym typem dzia-

łania niż wektoryzacja. W wektoryzacji - jak wspomniano wyżej - wykonuje się jedną instrukcję na wielu elementach. Natomiast programowanie równoległe polega na dzieleniu dużych zadań na mniejsze działania i wykonywanie tych właśnie mniejszych zadań jednocześnie. Proces ten może zostać wykorzystany dzięki odpowiedniej budowie komputerów, a odbywa się na poziomie odrębnych procesorów lub rdzeni procesorów wielordzeniowych. Zrównoleglanie jest szczególnie istotne dla procesorów graficznych (ang. *Graphics Processing Unit, GPU*) oraz dla architektury MIC (ang. *Many Integrated Core Architecture*) [GBWS18]. Należy jeszcze dodać cytując za [Stp08], że realizując idee równoległości wewnątrz pojedynczego procesora na poziomie wykonywanych równolegle rozkazów (ang. *instruction-level parallelism*) powstała koncepcja budowy procesorów superskalarnych, wyposażonych w kilka jednostek arytmetyczno-logicznych (ALU) oraz jedną lub więcej jednostek realizujących działania zmiennopozycyjne (FPU). Jednostki obliczeniowe otrzymują w tym samym cyklu do wykonania instrukcje pochodzące zwykle z pojedynczego strumienia. Zależność między poszczególnymi instrukcjami jest sprawdzana dynamicznie w trakcie wykonania programu przez odpowiednie układy procesora.

Powstawanie coraz bardziej zaawansowanych architektur komputerowych ze złożoną strukturą pamięci pociąga za sobą potrzebę tworzenia nowych algorytmów, które w wydajny sposób będą korzystały z tych ulepszeń [EGJK04].

1.2 Narzędzia programistyczne

W niniejszej sekcji zostaną wymienione oraz pokrótce omówione narzędzia, które zostały wykorzystane do tworzenia programów przedstawionych w niniejszej rozprawie. Zostanie przedstawiony język programowania jak i wykorzystane interfejsy umożliwiające programowanie równoległe oraz funkcje pozwalające na działania wektorowe.

1.2.1 Język C

C jest wysokopoziomowym, strukturalnym językiem programowania powstałym w roku 1972. Jest to jeden z najpopularniejszych języków programowania, wykorzystywanym bardzo wszechstronnie - od systemów operacyjnych do programów nadzorujących pracę urządzeń przemysłowych. Istotnymi zaletami są przenośność oraz szybkość programów napisanych w tym języku. Natomiast poprawne zarządzanie pamięcią wymaga poświęcenia większej uwagi programisty, niż w innych popularnych językach, takich Python czy Java.

Wszystkie implementacje, które zostały przedstawione w niniejszej rozprawie powstały w języku C. Przykładowy program w języku C znajduje się na listingu 1.1.

```c
#include <stdio.h>

int main() {
   printf("Hello World!");
   return 0;
}
```

Listing 1.1: Przykładowy program w języku C

1.2.2 OpenMP

OpenMP (ang. *Open Multi-Processing*) jest interfejsem pozwalającym na wprowadzenie programowania równoległego do programów napisanych w języku C/C++ oraz Fortran dla systemów wieloprocesorowych z pamięcią współdzieloną [BSHS11]. Pozwala on również na wprowadzenie mechanizmów komunikacji pomiędzy wątkami oraz ich synchronizacji [SB11]. OpenMP oparty jest na dyrektywach - w języku C są to *pragmy*, które specyfikują możliwą równoległość obliczeń. Dyrektywny te mogą również zawierać informację o podziale danych pomiędzy wątkami [Qui03]. Dodatkowo, w OpenMP zdefiniowano także kilka zmiennych środowiskowych oraz funkcji, które m.in. pomagają w mierzeniu czasu wykonania programu [Stp18]. Listing 1.2 przedstawia przykładowy program w języku C wykorzystujący interfejs OpenMP.

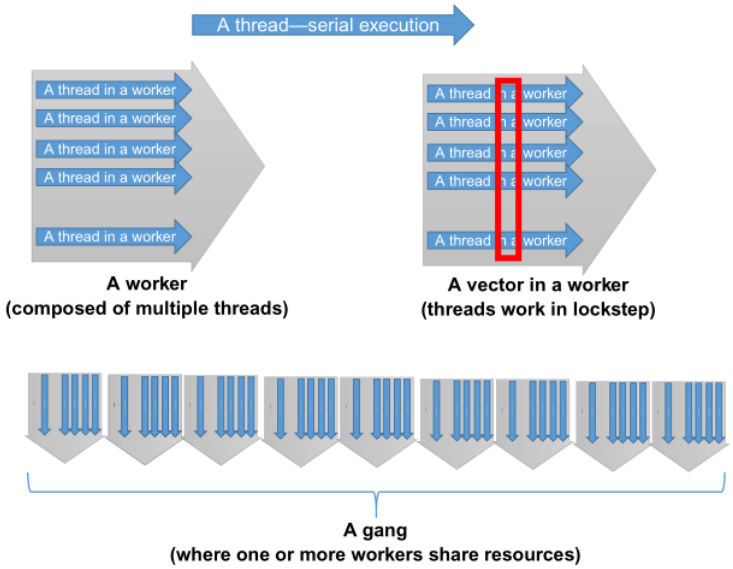
```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(){
6      int i, all;
7  #pragma omp parallel private(i,all)
8      {
9          i=omp_get_thread_num() ;
10         all=omp_get_num_threads() ;
11         printf( "Hello from %d of %d\n" , i , all ) ;
12     }
13     return 0;
14 }
```

Listing 1.2: Przykładowy program w języku C z wykorzystaniem OpenMP

1.2.3 OpenACC

OpenACC (ang.*Open ACCelerators*) jest interfejsem służącym do zrówoleglania obliczeń w programach stworzonych w językach C/C++ lub Fortran [SC17]. Oferuje dyrektywy (*pragmy*), dzięki którym można przekazać informację o możliwościach zrównoleglenia oraz przenoszenia danych z oraz do urządzeń akceleracyjnych (np. kart graficznych) [Far17]. Dyrektywy te służą do oznaczania bloków kodu źródłowego, które mogą zostać automatycznie zakcelerowane. Czasami jednak niezbędne jest również dodanie wysokopoziomych przekształceń kodu źródłowego, aby uzyskać oczekiwaną wydajność [CJ18, Stp16].

Model wykonania OpenACC został przedstawiony na rysunku 1.1. Model ten pozwala użytkownikom wyrazić trzy poziomy równoległości: gang (ang. *gang*), pracownik (ang. *worker*) i wektor (ang. *vector*). Poniżej nastąpi omówienie tych poziomów za pozycją [Far17], zaczynając od najbardziej podstawowego, czyli wątku (ang. *thread*). Pojedynczy, szeregowy wątek wykonawczy to podstawowa koncepcja równoległa. Pracownik, to grupa wątków, które mogą działać razem w trybie SIMD. Wektory to zbiory wątków działających krok po kroku podczas wykonywania instrukcji SIMD. Natomiast gangi to grupy pracowników. Gangi działają niezależnie od siebie. Przykładowy kod z wykorzystaniem OpenACC znajduje się na listingu 1.3.



Rysunek 1.1: Różne formy równoległości OpenACC[1]

```
1  #include <openacc.h>
2  #include <stdio.h>
3
4  int main(){
5      int N=10000;
6      float a = 11.3f;
7      float *x = (float*)malloc(N * sizeof(float));
8      float *y = (float*)malloc(N * sizeof(float));
9      for (int i = 0; i < N; ++i) {
10         x[i] = i;
11         y[i] = N - i;
12     }
13 #pragma acc parallel loop
14     for(int i = 0; i < N; ++i)
15         y[i] = a * x[i] + y[i];
16     return 0;
17 }
```

Listing 1.3: Przykładowy program w języku C z wykorzystaniem OpenACC

1.2.4 Funkcje wbudowane

Funkcje wbudowane (ang. *intrinsics*) dla procesorów CPU to wysokopoziomowe reprezentacje instrukcji maszynowych, pozwalające na manualną wektoryzację obliczeń w językach C/C++. Instrukcje te pozwalają w pełni wykorzystać zalety płynące z rozszerzeń wektorowych (ang. *Advanced Vector Extensions*). Działają na typach danych reprezentujących 512-, 256-, 128-bitowe rejestry (w AVX-512, AVX2, AVX, odpowiednio). Pozwalają programiście na zapis instrukcji SIMD wyglądających tak, jak wywołanie funkcji C/C++. Główną wadą takich programów jest brak przenośności pomiędzy różnymi wersjami rozszerzeń wektorowych [Stp18].

[1]Źródło: [Far17]

```c
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    __m256 a = _mm256_set_ps(1.0 2.0,4.0,8.0,16.0,32.0,64.0,128.0);
    __m256 b = _mm256_set_ps(128.0,64.0,32.0,16.0,8.0,4.0,2.0,1.0);
    __m256 c = _mm256_add_ps(a, b);

    float d[8];
    _mm256_storeu_ps(d, c);
    for(int i=0;i<8;i++)
        printf("%f\n",d[i]);
    return 0;
}
```

Listing 1.4: Przykładowy program w języku C z wykorzystaniem funkcji wbudowanych

1.3 Architektury współczesnych komputerów

Efektywność algorytmów zależy nie tylko od ich złożoności. Wpływa na nią także kompilator oraz sprzęt komputerowy. Dlatego za pracą [EGJK04] można mówić o fundamentalnym trójkącie *"algorytmy - sprzęt komputerowy - kompilatory"* (ang. *algorithms-hardware-compilers triangle* lub *fundamental AHC triangle*). W praktyce przekłada się to na konieczność tworzenia odpowiednich algorytmów, umożliwiających kompilatorom dostosowanie kodu do budowy komputerów [Stp08].

Koncepcja programowania równoległego początkowo realizowana była na bardzo drogich superkomputerach, które służyły do wysokowydajnych obliczeń (ang. *high performance computing*). Urządzenia te można było znaleźć jedynie w centrach naukowych, a przez to dostępne one były jedynie dla wąskiego grona specjalistów [Pot14].

Obecnie zapotrzebowanie na moc obliczeniową stale rośnie. Aby sprostać tej potrzebie nastąpiło przejście od tradycyjnych procesorów jednordzeniowych do architektur wielordzeniowych i wieloprocesorowych. Efektywne wykorzystanie takich architektur pociąga za sobą rozpowszechnienie technologii obliczeń równoległych. Obecnie niemal wszystkie nowoczesne komputery dysponują taką technologią, przez co znacząco rozszerzyła się grupa jej użytkowników [RSW15]. Ostatnie działania producentów architektur komputerowych sugerują, że dalszy rozwój będzie związany z systemami hybrydowymi, a obecnie najpopularniejsze - ze względu na cenę - rozwiązania to heterogeniczne urządzenia, na które składają się zintegrowane z procesorem karty graficzne.

W dalszej części omówione zostaną architektury, na których zebrano wyniki zaprezentowane w niniejszej rozprawie z podziałem na procesory i procesory kart graficznych. Urządzenia te zostały zainstalowane na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie.

1.3.1 Procesory

Procesor (ang. *Central Processing Unit, CPU*) - jednostka, która odpowiedzialna jest za pobieranie danych z pamięci operacyjnej i wykonywanie rozkazów. Następnie takie dane zwracane są do pamięci bądź wysyłane do strumienia wyjściowego. Współczesne

procesory mają budowę wielordzeniową. Wiodącym producentem procesorów jest firma Intel. Przykładowy procesor - Intel Xeon Phi - został przedstawiony na rysunku 1.2.

Rysunek 1.2: Procesor Intel Xeon Phi[2]

Poniżej opisano architektury, które służyły do testów w niniejszej pracy:

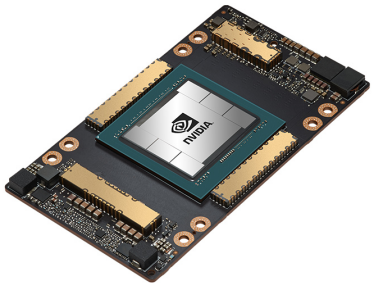
I. Intel Xeon E5-2670 v3 Intel Xeon E5-2670 v3 (łącznie 24 rdzenie z hyperthreadingiem, 2.3 GHz), 128GB RAM, uruchomiony na systemie Linux z CUDA 10.0 oraz kompilatorami Portland Group PGI wersja 19.4 ze wsparciem dla OpenMP i OpenACC,

II. Intel Xeon Phi 7120P (KNC, 61 rdzeni z multithreadingiem, 1.238 GHz, 16GB RAM, 512-bitowe rozszerzenia wektorowe) - w dalszej części nazywany będzie MIC,

III. Intel Xeon Gold 6342 (łącznie 48 rdzeni z hyperthreadingiem, 2.8 GHz, 36 MB pamięci podręcznej, 256 GB RAM) uruchomiony na systemie Linux wraz z OneAPI firmy Intel (wersja 2022) zawierającym kompilatory języków C/C++, Fortran oraz wysokowydajną bibliotekę numeryczną MKL.

1.3.2 Procesory graficzne

Procesor graficzny (ang. *Graphics Processing Unit, GPU*) jest to jednostka obliczeniowa znajdująca się w kartach graficznych.



Rysunek 1.3: Procesor graficzny Ampere A100 firmy NVIDIA[3]

[2]Źródło: https://www.benchmark.pl/aktualnosci/intel-xeon-phi-procesory-dla-superkomputerow-cena-specyfikacja.html, dostęp: 19.09.2023

[3]Źródło: https://www.nvidia.com/en-us/data-center/a100/, dostęp: 20.09.2023

Początkowo karty graficzne służyły głównie do obsługi grafiki, gdzie wymagane było przeprowadzenie wielu, ale raczej bardzo prostych obliczeń. Wprowadzenie architektury NVIDIA G80 oraz opartego na języku C środowiska NVIDIA CUDA umożliwiło powszechne wykorzystanie procesorów graficznych do obliczeń naukowych i inżynieryjnych [Pot14]. Czołowi producenci omawianych podzespołów to NVIDIA i AMD. Przykład karty graficznej - Ampere A100 firmy NVIDIA - został przedstawiony na rysunku 1.3.

Poniżej opisano wykorzystywane w eksperymentach karty graficzne, które uruchomione były na systemie Linux z CUDA 10.0 oraz kompilatorem PGI:

A. Kepler - NVIDIA Tesla K40m GPU (2880 rdzeni, 12GB RAM),

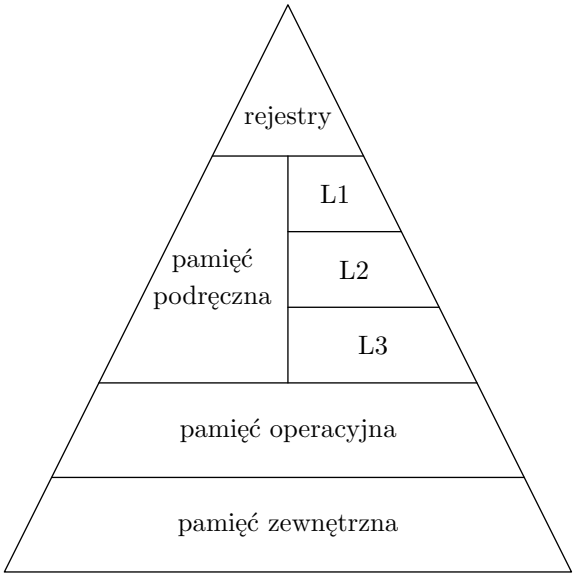
B. Volta - NVIDIA Volta V100 GPU (5120 rdzeni, 32GB RAM),

C. Turing - NVIDIA GEFORCE RTX 2080 SUPER GPU (3072 rdzeni, 8 GB RAM), pracująca na komputerze z procesorem AMD Ryzen 9 3900X (łącznie 12 rdzeni z hyperthreadingiem, 3.8 GHz), 32GB RAM,

D. Ampere - NVIDIA Ampere A100 GPU (6912 rdzeni, 40GB RAM).

1.4 Pamięć komputera

Jednym z fundamentalnych aspektów projektowania efektywnych implementacji jest optymalny dostęp do danych. Dlatego należy poświęcić uwagę na zbadanie systemu pamięci.



Rysunek 1.4: Trójkąt prezentujący hierarchię pamięci komputera

Na szczycie hierarchii pamięci znajdują się rejestry, gdzie mają miejsce obliczenia (zmiennoprzecinkowe, całkowitoliczbowe i logiczne). Mają one najkrótszy czas dostępu, ale też najmniejszą pojemność. Pomiędzy rejestrami i pamięcią główną znajduje się jeden lub więcej poziomów pamięci podręcznej. Najbliższa rejestrom jest pamięć

podręczna pierwszego poziomu (ang. *first-level cache memory (L1 cache)*) - najszybsza, lecz najmniejsza. Następna jest pamięć podręczna drugiego poziomu (ang. *second-level (L2)*). Jest jeszcze możliwy poziom trzeci (ang. *third-level (L3)*). Poniżej pamięci głównej znajdują się poziomy pamięci zewnętrznej z większymi pojemnościami, ale wolniejszym dostępem [EGJK04]. Opisany powyżej schemat prezentuje rysunek 1.4.

1.5 Analiza wydajności

Podstawową cechą opisującą wydajność programu jest jego czas wykonania. Działania programisty mają za zadanie skrócenie tego czasu. Jednak operowanie czasami bezwzględnymi nie jest miarodajną informacją, ponieważ może nie odzwierciedlać osiąganego zysku. Dlatego wygodniej jest posługiwać się miarą przyspieszenia (ang. *speedup*) programu [Stp08]. Wartość ta pokazuje ile razy szybciej działa program zoptymalizowany określoną techniką lub na określoną architekturę względem programu bazowego - na ogół sekwencyjnego.

Definicja 1.1. *Przyspieszenie wyliczane jest jako iloraz czasu działania algorytmu bazowego oraz rozważanego algorytmu, tj.*

$$Przyspieszenie = \frac{Czas(Algorytm\ Bazowy)}{Czas(Algorytm\ Rozważany)}. \tag{1.1}$$

Kolejnym ważnym aspektem osiąganym przez program jest dokładność. Jest ona równie ważna, a czasami ważniejsza niż czas działania programu. Przede wszystkim zmniejszenie czasu działania nie może pociągać za sobą zmniejszenia dokładności.

Niech rozważany układ równań ma postać:

$$T\mathbf{x} = \mathbf{b}, \tag{1.2}$$

gdzie $T \in \mathbb{R}^{n \times n}$ jest macierzą współczynników, $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ szukanym wektorem oraz $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})^T$ prawą stroną równania. Dokładności programów rozwiązujących układy równań postaci (1.2) może zostać wyliczona za pomocą następującego wzoru:

$$Błąd\ względny = \frac{||T\bar{\mathbf{x}} - \mathbf{b}||}{||\mathbf{x}||}, \tag{1.3}$$

gdzie $\bar{\mathbf{x}}$ jest rozwiązaniem wyliczonym przez badany program. W programach będących przedmiotem niniejszej rozprawy wykorzystano normę euklidesową, tj. następujący wzór:

$$Błąd\ względny = \sqrt{\sum_{i=0}^{n-1} \frac{g_i^2}{x_i^2}}, \tag{1.4}$$

gdzie $\mathbf{g} = T\bar{\mathbf{x}} - \mathbf{b}$.

Błąd względny sumowania został sprawdzony w łatwy sposób, ponieważ - jak opisane zostało w rozdziale 5 - znana jest dokładną wartość sumy. Wyliczenie to przedstawia wzór (1.5).

$$Błąd\ względny = \frac{|Suma\ wyliczona - Suma\ dokładna\ |}{|Suma\ dokładna\ |}, \tag{1.5}$$

gdzie *Suma wyliczona* jest wynikiem rozpatrywanego programu, a *Suma dokładna* została wyznaczona za pomocą wzorów matematycznych.

Aspektem wartym uwagi jest również ilość operacji w algorytmie. Jednak w programowaniu równoległym nie zawsze przekłada się to bezpośrednio na efektywność implementacji. Występują sytuacje, w których implementacja z większą liczbą operacji może zostać obliczona szybciej, ponieważ wiele z tych operacji może zostać wykonanych równolegle.

Znając liczbę operacji w algorytmie można wyznaczyć liczbę operacji zmiennoprzecinkowych na sekundę, czyli FLOPS (ang. *Floating Point Operations Per Second*). Jest to jednostka mocy obliczeniowej. Wyznacza się ją poprzez podzielenie ilości operacji dla konkretnego przypadku przez czas wykonania tego właśnie przypadku. Na ogół wygodniej jest podawać liczbę operacji na sekundę w jednostce GFLOPS, czyli podzielonej przez 10^9.

Rozdział 2

Trójdiagonalne układy równań liniowych typu Toeplitza

2.1 Wstęp

Układ równań liniowych o n niewiadomych i m równaniach może zostać zapisany jako:

$$\begin{cases} a_{00}x_0 & + & a_{01}x_1 & + & ... & + & a_{0(n-1)}x_{n-1} & = & b_0 \\ a_{10}x_0 & + & a_{11}x_1 & + & ... & + & a_{1(n-1)}x_{n-1} & = & b_1 \\ \vdots & & & & & & & & \\ a_{(m-1)0}x_0 & + & a_{(m-1)1}x_1 & + & ... & + & a_{(m-1)(n-1)}x_{n-1} & = & b_{m-1} \end{cases} \tag{2.1}$$

Niech $0 \leqslant i \leqslant m-1$ oraz $0 \leqslant j \leqslant n-1$. Wtedy x_j są niewiadomymi, a_{ij} - współczynnikami układu, b_i - stałymi. Przypisując:

$$A = \begin{bmatrix} a_{00} & a_{01} & ... & a_{0(n-1)} \\ a_{10} & a_{11} & ... & a_{1(n-1)} \\ \vdots & & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & ... & a_{(m-1)(n-1)} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix},$$

układ (2.1) może zostać zapisany w następującej, równoważnej formie:

$$A\mathbf{x} = \mathbf{b}. \tag{2.2}$$

Do dalszych rozważań przyjęto założenie, że $n = m$. W przeciwnym przypadku można dopełnić brakujące kolumny lub wiersze elementami zerowymi.

Głównym przedmiotem badań niniejszej rozprawy będą układy, w których macierz A spełnia dodatkowe kryteria. Pierwszym typem jest macierz Toeplitza (2.3), która nazwana została za niemieckim matematykiem Otto Toeplitzem (ur. 1 sierpnia 1881 we Wrocławiu, zm. 15 lutego 1940 w Jerozolimie). Jest to macierz, w której elementy na każdej z przekątnych mają takie same wartości, tj.:

$$\begin{bmatrix} a_0 & a_{-1} & a_{-2} & \dots & \dots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \dots & \dots & a_2 & a_1 & a_0 \end{bmatrix} \tag{2.3}$$

Inny ciekawy przypadek to macierze wąskopasmowe. Oznacza to macierz, w której wszystkie elementy są zerami, z wyjątkiem głównej przekątnej i wąskiego pasma obok niej. Poniżej przykład macierzy wąskopasmowej, a konkretnie trójdiagonalnej (o trzech przekątnych zawierających niezerowe elementy):

$$\begin{bmatrix} a_{00} & a_{01} & 0 & \dots & 0 \\ a_{10} & a_{11} & a_{12} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & a_{(n-2)(n-1)} \\ 0 & \dots & 0 & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} \end{bmatrix}. \tag{2.4}$$

Macierz rozważana w niniejszej rozprawie jest jednocześnie trójdiagonalną i typu Toeplitza, tj. ma następującą postać:

$$\begin{bmatrix} t_2 & t_3 & & & \\ t_1 & t_2 & t_3 & & \\ & . & . & . & \\ & & . & . & t_3 \\ & & & t_1 & t_2 \end{bmatrix}. \tag{2.5}$$

2.2 Macierze blokowe

Koncepcja fundamentalnego trójkąta *"algorytmy - sprzęt komputerowy - kompilatory"*, przedstawiona w rozdziale 1 pociąga za sobą konieczność grupowania danych tak, aby pasowały do hierarchicznej struktury pamięci. W przypadku macierzy można podzielić je na bloki (lub podmacierze, lub klatki). Poniżej podano przykład takiego podziału dla macierzy o rozmiarze 6×9 i rozmiarze bloku 2×3:

$$\left[\begin{array}{ccc|ccc|ccc} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} & a_{07} & a_{08} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ \hline a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ \hline a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \end{array}\right]. \tag{2.6}$$

Przyjmując oznaczenia:

$$A_{00} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}, A_{01} = \begin{bmatrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \end{bmatrix}, A_{02} = \begin{bmatrix} a_{06} & a_{07} & a_{08} \\ a_{16} & a_{17} & a_{18} \end{bmatrix},$$
$$A_{10} = \begin{bmatrix} a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix}, A_{11} = \begin{bmatrix} a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \end{bmatrix}, A_{12} = \begin{bmatrix} a_{26} & a_{27} & a_{28} \\ a_{36} & a_{37} & a_{38} \end{bmatrix}, \quad (2.7)$$
$$A_{20} = \begin{bmatrix} a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{52} \end{bmatrix}, A_{21} = \begin{bmatrix} a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{bmatrix}, A_{22} = \begin{bmatrix} a_{46} & a_{47} & a_{48} \\ a_{56} & a_{57} & a_{58} \end{bmatrix},$$

macierz (2.6) można zapisać jako:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}. \quad (2.8)$$

Korzystając z powyższego sposobu, można przekształcać całe układy na zapis blokowy. Równanie (2.9) przedstawia ogólny zapis układu blokowego:

$$\begin{bmatrix} A_{00} & A_{01} & \dots & A_{0r} \\ A_{10} & A_{11} & \dots & A_{1r} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q0} & A_{q1} & \dots & A_{qr} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_r \end{bmatrix}, \quad (2.9)$$

gdzie $n/r = s$, $m/q = s$ oraz A_{ij}, $i \in \{0, 1, \dots, q\}$, $j \in \{1, 2, \dots, r\}$ są blokami o rozmiarach $r \times s$, natomiast $\mathbf{x}_j$ oraz $\mathbf{b}_j$ są wektorami o rozmiarze s.

Postać taka jest bardzo istotna, ponieważ pozwala uprościć wiele obliczeń macierzowych, np. mogą pojawić się podmacierze zerowe lub będące tożsamościami. Zapis blokowy jest także kluczowy w programowaniu wysokopoziomym [GVL96].

Podziały na bloki macierzy wykorzystywanych w niniejszej rozprawie zostaną szczegółowo przedstawione w dalszych rozdziałach.

2.3 Zastosowania

Układy równań liniowych z macierzą współczynników trójdiagonalną typu Toeplitza występują w wielu praktycznych i teoretycznych zagadnieniach. Poniżej wymieniono wybrane problemy, które sprowadzają się do rozwiązania wyżej wymienionych układów równań. Oczywiście, nie wyczerpują one zbioru możliwych zastosowań takich układów.

A) Numeryczne metody rozwiązywania warunków brzegowych równań różniczkowych zwyczajnych lub cząstkowych [SP11, VA12]. Jako przykład podano następujące równanie różniczkowe:

$$-\frac{d^2u}{dx^2} = f(x), \quad x \in [0, 1] \quad (2.10)$$

z warunkami brzegowymi $u'(0) = 0$ oraz $u(1) = 0$, na siatce punktów $0 = x_0 < x_1 < \ldots < x_n = 1$, gdzie $x_i = (i-1)h$, $h = 1/n$, $i = 0, \dots, n+1$. Niech $f_i = f(x_i)$ oraz $u_i = u(x_i)$. Wykorzystując przybliżenie drugiej pochodnej:

$$u''(x_i) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \quad (2.11)$$

oraz warunki brzegowe:

$$u''(0) \approx \frac{u(0-h) - 2u_1 + u_2}{h^2}, \quad u'(0) \approx \frac{u_2 - u(0-h)}{2h} \tag{2.12}$$

otrzymano następujące równania:

$$\begin{aligned} 2(u_1 - u_2) &= h^2 f_1 \\ -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i,\ i = 2, \ldots, n-1, \\ -u_i + 2u_n &= h^2 f_n. \end{aligned} \tag{2.13}$$

Z tak opisanych warunków wynika, że omawiany problem może zostać zredukowany do następującego trójdiagonalnego układu równań liniowych:

$$\begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}, \tag{2.14}$$

gdzie $d_1 = \frac{1}{2}h^2 f_1$, $d_i = h^2 f_i$ dla $i = 2, \ldots, d_n$. Przykład ten oraz szczegółowe informacje o omawianym zagadnieniu można znaleźć w [SP11, Pot14, SCB05].

B) Układy powstające przy rozwiązywaniu numerycznym zwyczajnych równań różniczkowych drugiego rzędu. Dla przykładu:

$$\begin{cases} y''(x) - py'(x) - qy(x) = 0,\ x \in [a, b] \\ y(a) = A,\ y(b) = B. \end{cases} \tag{2.15}$$

Po wykonaniu odpowiednich kroków, powyższy układ może zostać rozwiązany poprzez rozwiązanie następującego układu równań liniowych:

$$\begin{bmatrix} a_0 & c_0 & & & \\ b_2 & a_1 & c_1 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-3} & a_{n-3} & c_{n-3} \\ & & & b_{n-2} & a_{n-2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-3} \\ y_{n-2} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-3} \\ f_{n-2} \end{bmatrix}. \tag{2.16}$$

C) Różnice skończone dla problemów 1-D (ang. *finite differences for 1-D problems*). Jest to metoda rozwiązywania równań różniczkowych cząstkowych [Saa03, QS04]. Niech:

$$\begin{cases} -u''(x) = f(x),\ x \in (0, 1) \\ u(0) = u(1) = 0. \end{cases} \tag{2.17}$$

Przedział [0, 1] może zostać przybliżony przy pomocy $x_i = ih$, $i = 0, \ldots, n+1$, oraz $h = \frac{1}{n+1}$. Przy tak przedstawionych oznaczeniach powyższe zagadnienie może zostać sprowadzone do następującego układu:

$$A\mathbf{x} = \mathbf{f},$$

gdzie:

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \tag{2.18}$$

D) Różnice skończone dla problemów 2-D (ang. *finite differences for 2-D problems*). Analogicznie jak powyżej jest to metoda rozwiązywania równań różniczkowych cząstkowych, a jej opis przytoczony zostanie za [Saa03, QS04]. Podobnie do poprzedniego przypadku rozważono następujący problem:

$$\begin{cases} -\left(\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial {x_2}^2}\right) = f \text{ w } \Omega \\ u = 0 \text{ na } \Gamma, \end{cases} \tag{2.19}$$

gdzie Ω jest prostokątem $(0, l_1) \times (0, l_2)$, a Γ jego brzegiem. Przy odpowiednich założeniach macierz układu jest blokowa i ma postać:

$$A = \frac{1}{h^2} \begin{bmatrix} B & -I & \\ -I & B & -I \\ & -I & B \end{bmatrix}, \text{ gdzie } B = \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & -1 & 4 & -1 \\ & & -1 & 4 \end{bmatrix}. \tag{2.20}$$

E) Szybkie metody rozwiązywania równań Poissona (ang. *Fast Poisson Solvers, FPS*) za [Saa03] - metoda pozwalająca na rozwiązywanie układów równań liniowych powstających przy dyskretyzacji równania Poissona na siatce prostokątnej za pomocą różnic skończonych. Za pomocą równań Laplaca'a opisać można na przykład potencjał grawitacyjny poza punktami źródeł pola, potencjał prędkości cieczy przy braku źródeł. Gdy do analogicznych związków dodane zostanie źródło pola, to do opisu takich zjawisk można wykorzystać równania Poissona.

Rozważając macierz B z podpunktu D), można wyznaczyć jej wartości własne (λ_i) i wektory własne (q_i) za pomocą metody Fouriera. Definiując $Q = [q1, \ldots, q_p]$ oraz $Q^T B Q = \Lambda = diag(\lambda_j)$, należy rozwiązać następujący blokowy układ równań:

$$\begin{bmatrix} \Lambda & -I & & & \\ -I & \Lambda & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & \Lambda & -I \\ & & & -I & \Lambda \end{bmatrix} \begin{bmatrix} \bar{u}_0 \\ \bar{u}_1 \\ \vdots \\ \bar{u}_{n-2} \\ \bar{u}_{n-1} \end{bmatrix} = \begin{bmatrix} \bar{b}_0 \\ \bar{b}_1 \\ \vdots \\ \bar{b}_{n-2} \\ \bar{b}_{n-1} \end{bmatrix}. \tag{2.21}$$

Przy opisanych założeniach i-ty wiersz każdego z bloków sprowadza się do:

$$\begin{bmatrix} \lambda_i & -1 & & & \\ -1 & \lambda_i & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & \lambda_i & -1 \\ & & & -1 & \lambda_1 \end{bmatrix} \begin{bmatrix} \bar{u}_{i0} \\ \bar{u}_{i1} \\ \vdots \\ \bar{u}_{ip-2} \\ \bar{u}_{ip-1} \end{bmatrix} = \begin{bmatrix} \bar{b}_{i0} \\ \bar{b}_{i1} \\ \vdots \\ \bar{b}_{ip-2} \\ \bar{b}_{ip-1} \end{bmatrix}, \tag{2.22}$$

gdzie $\bar{u}_{ij}$ i $\bar{b}_{ij}$ jest j-tą składową wektorów $\bar{u}_i$ i $\bar{b}_i$ odpowiednio.

F) Schematy wsteczne (ang. *Upwind schemes*). Problem, który pojawia się przy rozwiązywaniu układów równań liniowych powstających dla równań konwekcji-dyfuzji jednowymiarowych [Saa03]. Opisywane schematy wykorzystywane są na przykład przy problemach gromadzenia zanieczyszczeń (smog, dym papierosowy). Niech przykładowy układ będzie postaci:

$$\begin{cases} -au'' + bu' = 0,\ x \in (0,1) \\ u(0) = 0,\ u(1) = 1, \end{cases} \tag{2.23}$$

Sprowadza się do rozwiązania układu z macierzą:

$$A = \frac{1}{h^2} \begin{bmatrix} 2 & -1+c & & & & \\ -1-c & 2 & -1+c & & & \\ & . & . & . & & \\ & & . & . & -1+c & \\ & & & -1-c & 2 & \end{bmatrix}, \tag{2.24}$$

gdy $b < 0$, a gdy $b > 0$ macierz A przyjmuje postać:

$$A = \frac{1}{h^2} \begin{bmatrix} 2+c & -1 & & & \\ -1-c & 2+c & -1 & & \\ & . & . & . & \\ & & . & . & -1 \\ & & & -1-c & 2+c \end{bmatrix}, \tag{2.25}$$

G) Numeryczne wyznaczanie funkcji sklejanych sześciennych (ang. *cubic spline*) podane za [Pot14]. Funkcje sklejane wykorzystywane są w celach interpolacji nieznanej funkcji za pomocą wielomianów - raczej niskiego rzędu. Aby wyznaczyć funkcję sklejaną sześcienną S interpolującą funkcję f na przedziale $[a, b]$ w punktach podziału x_i, $i = 0, ..., n$ takich, że

$$a = x_0 < x_1 < ... < x_n = b,$$

na każdym przedziale $[x_i, x_{i+1}]$, gdzie $0 \leqslant i \leqslant n-1$ konstruuje się wielomian $p_i(x)$ stopnia trzeciego następującej postaci:

$$p_i(x) = w_{1,i} + w_{2,i}(x - x_i) + w_{3,i}(x - x_i)^2 + w_{4,i}(x - x_i)^3.$$

Przy odpowiednich oznaczeniach problem powyższy redukuje się do następującego układu:

$$\begin{bmatrix} 1 & b_0 & & & \\ c_1 & 2 & b_1 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & 2 & b_{n-1} \\ & & & c_n & 2 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}. \tag{2.26}$$

Zaprezentowany układ ma charakter bardziej ogólny, a macierz jego współczynników nie jest typu Toeplitza. Jednak, w przypadku funkcji sklejanych macierze trójdiagonalne pojawiają się bardzo często i zdarza się, że są one również typu Toeplitza. Więcej informacji o funkcjach sklejanych można znaleźć w [Spä95].

H) Metoda kierunków naprzemiennych (ang. *Alternating Direction Implicit, ADI*) [Wac13, Ter16] została wprowadzona jako metoda rozwiązywania równań różnicowych eliptycznych i parabolicznych w [PR55]. Pojawia się przy wyznaczaniu rozwiązania macierzowego równania Sylvestera postaci:

$$AX + XB = C \tag{2.27}$$

lub w metodzie Lapunowa wyznaczania stabilności punktu równowagi.

I) Problemy fizyczne. Trójdiagonalne układy równań liniowych pojawiają się również w wielu problemach opisujących prawa fizyki. Występują one w modelu głowicy rurowej (ang. *tube header model*) oraz w modelu rury grubościennej (ang. *model for a thick-walled tube*). Bez wdawania się w szczegóły natury fizycznej (pełne opisy dostępne w [WE17]) każdy z wymienionych problemów redukuje się do rozwiązania trójdiagonalnego układu równań liniowych, bądź układ trójdiagonalny jest częścią rozwiązania.

J) Dyskretna transformata sinusoidalna (ang. *discrete sine transform*). Trójdiagonalne macierze występują także w dyskretnej transformacie sinusoidalnej, którą wykorzystuje się na przykład w algorytmach z tematyki przetwarzania obrazu. Więcej informacji można znaleźć w [Far11].

2.4 Znane algorytmy

Ze względu na opisane, szerokie wykorzystanie omawianych układów istnieje szereg dedykowanych metod pozwalających na ich rozwiązywanie. W niniejszej sekcji omówiono najważniejsze z nich.

2.4.1 Algorytm Thomasa

Jako pierwszy sposób wymienić należy algorytm Thomasa, zwany też z ang. *Tridiagonal Matrix Algorithm, TMA*. Jest to specjalny przypadek eliminacji Gaussa dla macierzy trójdiagonalnych [CH14]. Zakładamy, że układ, który należy rozwiązać ma postać:

$$T\mathbf{x} = \mathbf{b}, \tag{2.28}$$

gdzie T jest macierzą trójdiagonalną typu Toeplitza. W opisywanym rozwiązaniu przyjęto, że główna przekątna jest dominująca, tzn. nie ma konieczności wyznaczania elementów głównych (ang. *pivoting*). Rozwiązanie opiera się na założeniu, że macierz T może zostać zapisana jako iloczyn macierzy górnotrójkątnej (U) i dolnotrójkątnej (L), tj. $T = LU$. Wtedy układ (2.28) można zapisać równoważnie jako:

$$LU\mathbf{x} = \mathbf{b}. \tag{2.29}$$

Rozwiązanie układów postaci (2.29) możne zostać znalezione poprzez rozwiązanie dwóch prostych układów, co przekłada się na następujące fazy:

- eliminacja do przodu (ang. *forward reduction*):

$$L\mathbf{z} = \mathbf{b} \tag{2.30}$$

- podstawianie odwrotne (ang. *back substitution*):

$$U\mathbf{x} = \mathbf{z}. \tag{2.31}$$

Przedstawiony algorytm jest algorytmem sekwencyjnym i przez to jest wydajny jednie dla małych układów. Gdy rozwiązywany problem ma duży rozmiar, czas wykonywania będzie się znacząco wydłużał, ponieważ algorytmy sekwencyjne nie są w stanie wykorzystywać w pełni architektur współczesnych komputerów. Rozważając większe problemy, należy szukać lepszych metod. Dużo bardziej efektywne są metody, które można zrównoleglić bądź zwektoryzować. Omówione zostaną teraz trzy takie algorytmy: *recursive doubling*, *cyclic reduction* oraz metoda Wanga.

2.4.2 Metoda rekurencyjnego podwajania

Rekurencyjne podwajanie (ang. *recursive doubling*) to algorytm rozwiązywania układów trójdiagonalnych wprowadzony w [KS73]. Opisy omawianej metody można znaleźć w następujących pracach: [DDSV91, Mod88, Pan95, Stp08, KS73]. Poniższy opis pochodzi z [Stp08]. Gdy parametr $m = 1$, kroki opisujące omawianą metodę wyglądają następująco:

$$x_k = f_k + a_{k,1}x_{k-1} = f_k + a_{k,1}(f_{k-1} + a_{k-1,1}x_{k-2}) = \tilde{\tilde{f}}_k + \tilde{a}_{k,1}x_{k-2}. \tag{2.32}$$

Nowe współczynniki $\tilde{\tilde{f}}_k$, $\tilde{a}_{k,1}$ mogą, być obliczane równolegle bądź wektorowo, lecz obliczenia wektorowe nie będą dużym usprawnieniem ze względu na raczej niewielkie długości pętli wektoryzowalnych. Powyższe przekształcenie może być powtarzane: w pierwszym kroku można zastąpić we wzorze na x_n wyraz x_{n-1} korzystając ze wzoru na x_{n-1}, czyli przy użyciu x_{n-2}. Podobnie x_{n-2} można zastąpić przez x_{n-3} itd. Kolejny krok to zastąpienie we wzorze na x_n elementu x_{n-2} wzorem na x_{n-2}, czyli przez x_{n-3} itd. [Stp08].

Sposób działania algorytmu *recursive doubling* można przedstawić również w następujących krokach:

1. Podział danych: podział na mniejsze układy, które można rozwiązać równolegle.
2. Lokalne rozwiązania: każdy proces równoległy znajduje swoją części rozwiązania układu trójdiagonalnego.
3. Redukcja: łączenie wyników lokalnych przez co następuje redukcja rozmiaru problemu.
4. Iteracyjne podwajanie: opisane procesy powtarzane są iteracyjnie.

2.4.3 Metoda cyklicznej redukcji

Kolejną metodą jest cykliczna redukcja (ang. *cyclic reduction*). Oryginalnie wprowadzona jako metoda rozwiązywania równań Poissona na prostokącie [BGN70], natomiast modyfikacja przeprowadzona w [Pol11] prowadzi do algorytmu rozwiązywania układów równań macierzowych. Szczegółowe opisy mogą zostać zaczerpnięte z [LPNJR96, LQC96, Wil92, Stp08].

W każdym kroku tej metody, eliminacji podlegają parzyste lub nieparzyste wiersze bądź kolumny, czyli w każdym kroku rozmiar problemu zmniejsza się o połowę, aż do pojedynczego równania [Wil92]. Proces podziału - choć kosztowny - pozwala na wykorzystanie obliczeń równoległych. Efektywniejszy jest również proces wektoryzacji tej metody.

Działanie (ang. *cyclic reduction*) można przedstawić jako wykonanie następujących kroków:

1. Inicjalizacja: podział macierzy trójdiagonalnej na dwie podmacierze: jedną dla parzystych indeksów, a drugą dla indeksów nieparzystych.

2. Redukcja: każdy krok redukcji polega na operacjach na macierzach o zmniejszonym rozmiarze.

3. Obniżanie stopnia układu w trakcie każdej iteracji. Rozwiązywanie układów o niższym stopniu.

4. Łączenie wyników, co pozwala uzyskać rozwiązanie dla całego układu trójdiagonalnego.

5. Iteracyjność: opisany proces powtarza się aż do uzyskania ostatecznego rozwiązania.

2.4.4 Metoda Wanga

Ostatnią omawianą metodą jest metoda Wanga, znana też jako metoda podziału (ang. *divide and conquer*). Opisy tej metody znajdują się w [Wan81,LPNJR96,Stp93]. Metoda Wanga jest wykorzystywana w niniejszej pracy i praktyczny sposób jej wykorzystania można znaleźć w rozdziałach 3 oraz 4. Polega ona na zapisaniu rozważanego układu w postaci blokowej dwudiagonalnej (w przypadku macierzy Toeplitza), a następnie rozwiązywaniu powstałych składowych układów w sposób możliwie zrównoleglony, bądź zwektoryzowany - choć niektóre kroki mogą zostać wykonane tylko sekwencyjnie.

Ponadto w literaturze spotkać można szereg nowych lub ulepszeń już istniejących metod dedykowanych macierzom trójdiagonalnym typu Toeplitza, bądź macierzom z modyfikacją pojedynczych elementów. Poniżej podano aktualny spis literatury (prace pojawiające się w innych miejscach niniejszej rozprawy nie zostały tutaj powtórzone).

Rojo w [Roj90] zaproponował metodę rozwiązywania układów z symetryczną macierzą Toeplitza wykorzystując dekompozycję LU oraz wzory Shermana-Morrisona [DSZ18]. Podejście to zostało dalej zmodyfikowane w celu uzyskania równoległych implementacji [VA12, GS01]. Zwektoryzowana, lecz niezrównoleglona metoda została zaproponowana w [CY96b]. Inne podejście zostało zaproponowane przez McNally'ego i in. w [MGS08], którzy przedstawili skalowalny algorytm znajdujący przybliżenie dokładnego rozwiązania z daną precyzją. Jednakże algorytm ten nie jest wektoryzowalny. Terekhov w [Ter16] zaproponował wysokoskalowalny algorytm równoległy dla rozwiązywania układów z wieloma prawymi stronami. Yan and Chung w [YC94] przedstawili algorytm dla układów z macierzą symetryczną Toeplitza, diagonalnie dominującą. Algorytm ten został rozwinięty w [MGS00] przez McNally'ego i in. tak, aby omawiany problem mógł zostać podzielony na dwa oddzielne układy, które można rozwiązać

równolegle. W pracy [KR21] zaprezentowano metodę rozwiązywania trójdiagonalnych układów blokowych typu Toeplitza, gdzie każdy blok jest typu Toeplitza (ang. *Toeplitz–Block–Toeplitz, TBT*) za pomocą rozkładu WZ na urządzeniach akcelerujących. Wymienione prace nie wykorzystują w pełni jednocześnie procesów wektorowych i równoległych, przez co nie jest efektywnie wykorzystywana budowa nowoczesnych architektur komputerowych. Stąd potrzeba powstawania nowych algorytmów i ich implementacji.

2.5 Istniejące oprogramowanie

Istnieje wiele dostępnych narzędzi pozwalających na obliczenia numeryczne. W niniejszej sekcji wymienione zostaną te, które mogą być pomocne w tematyce omawianej w rozprawie.

2.5.1 LAPACK

LAPACK jest biblioteką zawierającą funkcje do rozwiązywania najczęściej spotykanych problemów w numerycznej algebrze liniowej m. in. układów równań liniowych, problemów najmniejszych kwadratów, wartości własnych. LAPACK jest akronimem z ang. *Linear Algebra PACKage* [ABB+92]. W bibliotece LAPACK nie ma funkcji rozwiązującej układy jednocześniej trójdiagonalne i typu Toeplitza, nie ma też funkcji dla układów Toeplitza. Istnieje natomiast funkcja do rozwiązywania układów trójdiagonalych - DGTSV, która korzysta z eliminacji Gaussa z częściowym wyborem elementu głównego [GTM].

2.5.2 NAG

Biblioteka NAG jest zbiorem funkcji rozwiązujących problemy numeryczne i statystyczne utworzoną przez *The Numerical Algorithms Group Ltd.* Biblioteka ta podzielona jest na rozdziały, z których każdy poświęcony jest jednej z dziedzin analizy numerycznej bądź statystyki [Phi87]. W dokumentacji biblioteki NAG [NAG] można znaleźć funkcję rozwiązującą układy z macierzą Toeplitza. Funkcja ta nosi nazwę `f04fff`. Można także znaleźć wiele funkcji rozwiązujących układy z macierzą trójdiagonalną z dodatkowymi założeniami (rzeczywistą, zespoloną, symetryczną, dodatnio określoną), np. `f04bcc, f04bcf, f04bgc, f07cac`. Jednak ta biblioteka również nie zapewnia funkcji rozwiązującej układy z macierzą jednocześnie Toeplitza i trójdiagonalną.

2.5.3 MATLAB

MATLAB jest środowiskiem programowania pozwalającym na przeprowadzanie obliczeń numerycznych, wizualizację danych, tworzenie i przeprowadzanie symulacji komputerowych, a także do wielu innych zastosowań związanych z matematyką i inżynierią. Nazwa stanowi skrót od ang. *MATrix LABoratory* [Mol]. MATLAB oferuje bogaty zestaw funkcji matematycznych i narzędzi do pracy z macierzami, wektorami, funkcjami, wykresami oraz różnymi rodzajami danych. Środowisko to zapewnia m.in. funkcję `tridiagonal_matrix(A,d)` [TMA], gdzie:

$$A\mathbf{x} = \mathbf{d} \text{ oraz } \mathbf{x} \in \mathbb{R}^n. \tag{2.33}$$

Funkcja ta rozwiązuje trójdiagonalne układy równań liniowych za pomocą algorytmu Thomasa. Do dyspozycji są również dwie funkcje rozwiązujące układy z macierzą współczynników typu Toeplitza [TM]: symetryczną - funkcja `toeplitz(r)`, gdzie `r` jest - w przypadku macierzy rzeczywistych - pierwszym wierszem oraz niesymetryczną `toeplitz(c,r)`, gdzie `c` jest pierwszą kolumną, a `r` - pierwszym wierszem. Jednak, podobnie jak w przypadku biblioteki LAPACK, środowisko MATLAB nie zapewnia funkcji dla układów będących jednocześnie trójdiagonalnymi oraz typu Toeplitza.

Z powyższych sekcji można zauważyć, że żadna z ogólnodostępnych, znanych bibliotek numerycznych optymalizowanych na współczesne architektury nie zapewnia funkcji do obliczania układów z macierzą trójdiagonalną typu Toeplitza. Jednak dzięki badaniu macierzy spełniających jednocześnie oba warunki można utworzyć dedykowany, wydajny algorytm, a przykłady wymienione w sekcji 2.3 pokazują istniejącą ku temu potrzebę.

Rozdział 3

Rozwiązywanie układów równań liniowych postaci $(1, t_2, t_3)$

W niniejszym rozdziale zaprezentowane zostaną dwie wersje zrównoleglonego i zwektoryzowanego algorytmu typu *dziel i zwyciężaj*, do rozwiązywania układów równań liniowych o specjalnych macierzach. Zbadane zostaną wyniki eksperymentalne: czasowe, dokładności oraz wydajność energetyczna.

3.1 Wstęp

Niech trójdiagonalny układ równań liniowych typu Toeplitza $T\mathbf{x} = \mathbf{b}$ ma następującą postać:

$$\begin{bmatrix} t_2 & t_3 & & & \\ 1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & t_2 & t_3 \\ & & & 1 & t_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{3.1}$$

Dla uproszczenia założono, że $n = 2^k$, $k \in \mathbb{N}$.

3.2 Algorytm równoległy

Pierwszym krokiem do uzyskania efektywnego algorytmu rozwiązywania 3.1 będzie dekompozycja macierzy T. Podążając za algorytmem zaprezentowanym przez Chunga i in. [CY96b] sformułowano twierdzenie 3.1.

Twierdzenie 3.1. *Trójdiagonalną macierz Toeplitza T można zapisać jako:*

$$T = LR + P,$$

lub równoważnie:

$$\begin{bmatrix} t_2 & t_3 & & & \\ 1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & t_2 & t_3 \\ & & & 1 & t_2 \end{bmatrix} =$$

$$\underbrace{\begin{bmatrix} \alpha & & & & \\ 1 & \alpha & & & \\ & \ddots & \ddots & & \\ & & 1 & \alpha & \\ & & & 1 & \alpha \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & \beta & & & \\ & 1 & \beta & & \\ & & \ddots & \ddots & \\ & & & 1 & \beta \\ & & & & 1 \end{bmatrix}}_{R} + \underbrace{\begin{bmatrix} \beta & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}}_{P}, \tag{3.2}$$

gdzie:

$$\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_3}}{2} \quad oraz \quad \beta = t_2 - \alpha$$

Dowód. Po wymnożeniu prawej strony równania (3.2) otrzymano:

$$\begin{cases} \beta + \alpha = t_2 \\ \beta\alpha = t_3. \end{cases} \tag{3.3}$$

Po wyliczeniu $\beta = t_2 - \alpha$ z pierwszego równania i podstawieniu do drugiego, otrzymano równanie kwadratowe postaci:

$$t_2\alpha - (\alpha)^2 - t_3 = 0. \tag{3.4}$$

Z przyjętego założenia o macierzy diagonalnie dominującej wynika, że równanie (3.4) będzie miało rozwiązania. Łatwo policzyć, że będą one postaci:

$$\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_3}}{2}$$

□

Wykorzystując wzór (3.2), układ (3.1) może zostać zapisany następująco:

$$(LR + P)\mathbf{x} = \mathbf{b},$$

a następnie:

$$LR\mathbf{x} + \begin{bmatrix} \beta & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \mathbf{b}. \tag{3.5}$$

Niech $\mathbf{e}_0 = (1, 0 \dots, 0)^T \in \mathbb{R}^n$. Po wymnożeniu macierzy z równania (3.5) i przyjętym założeniu otrzymano:

$$LR\mathbf{x} + \beta x_0 \mathbf{e}_0 = \mathbf{b}.$$

Ostatecznie:

$$\mathbf{x} = (LR)^{-1}\mathbf{b} - \beta x_0 (LR)^{-1}\mathbf{e}_0$$

lub równoważnie:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = (LR)^{-1} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} - \beta x_0 (LR)^{-1} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3.6}$$

Po wprowadzeniu oznaczenia:

$$\mathbf{v} = (LR)^{-1} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} \quad \text{oraz} \quad \mathbf{u} = (LR)^{-1} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.7}$$

rozważany układ można zapisać w prosty sposób jako:

$$\mathbf{x} = \mathbf{v} - \beta x_0 \mathbf{u}. \tag{3.8}$$

Następujące twierdzenie przedstawia rozwiązanie rozważanego układu równań liniowych.

Twierdzenie 3.2. *Niech $T\mathbf{x} = \mathbf{b}$, gdzie:*

$$T = \underbrace{\begin{bmatrix} \alpha & & & & \\ 1 & \alpha & & & \\ & \ddots & \ddots & & \\ & & 1 & \alpha & \\ & & & 1 & \alpha \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & \beta & & & \\ & 1 & \beta & & \\ & & \ddots & \ddots & \\ & & & 1 & \beta \\ & & & & 1 \end{bmatrix}}_{R} + \underbrace{\begin{bmatrix} \beta & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}}_{P}, \tag{3.9}$$

oraz $\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_3}}{2}$, a $\beta = t_2 - \alpha$. Niech również $\mathbf{v} = (LR)^{-1}\mathbf{b}$, $\mathbf{u} = (LR)^{-1}\mathbf{e}_0$, gdzie $\mathbf{e}_0 = (1, 0, \dots, 0)^T$. Wtedy rozwiązanie równania $T\mathbf{x} = \mathbf{b}$, może zostać wyznaczone ze wzoru:

$$\begin{cases} x_0 = \frac{v_0}{1+\beta u_0} \\ x_i = v_i - \beta x_0 u_i, \quad i = 1, \dots, n-1. \end{cases} \tag{3.10}$$

Dowód. Równanie (3.8) można zapisać następująco:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} - \beta x_0 \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{bmatrix} \tag{3.11}$$

Co prowadzi do równoważnej formy:

$$\begin{cases} x_0 = v_0 - \beta x_0 u_0 \\ x_i = v_i - \beta x_0 u_i, \quad i = 1, \dots, n-1. \end{cases} \tag{3.12}$$

Po prostym przekształceniu powyższego wzoru otrzymano wzór (3.10).

□

Zauważmy, że aby rozwiązać układ (3.10) potrzeba jedynie wektorów $\mathbf{u}$ i $\mathbf{v}$, które są odpowiednio rozwiązaniami układów liniowych $LR\mathbf{u} = \mathbf{e}_0$ i $LR\mathbf{v} = \mathbf{b}$. Oczywiście nie należy wyznaczać tych wektorów przez znalezienie macierzy odwrotnej do macierzy LR ze względu na koszt algorytmów wyznaczania odwrotności macierzy oraz to, że macierze odwrotne do macierzy trójdiagonalnych będą składały się w większości z elementów niezerowych, co znacząco zwiększy koszt obliczeń. Korzystając jednak z rozkładu macierzy układu na macierz górno- i dolnotrójkątną można za [Wil92] rozwiązywać w prosty sposób układy postaci $LR\mathbf{y} = \mathbf{f}$ korzystając z algorytmu Thomasa, który został opisany w rozdziale 2.4.1. W opisywanym przypadku należy rozwiązać układy $L\mathbf{z} = \mathbf{f}$ oraz $R\mathbf{y} = \mathbf{z}$, co przekłada się na następujące wzory sekwencyjne:

$$\begin{cases} z_0 = f_0/\alpha \\ z_i = (f_i - z_{i-1})/\alpha, \ i = 1, \ldots, n-1 \end{cases} \tag{3.13}$$

$$\begin{cases} y_{n-1} = z_{n-1} \\ y_i = z_i + \beta y_{i+1}, \ i = n-2, \ldots, 0. \end{cases} \tag{3.14}$$

Aby rozwiązać układ równań (3.1), należy wykorzystać algorytm Thomasa do dwóch układów:

$$LR\mathbf{u} = \mathbf{e}_0 \quad \text{oraz} \quad LR\mathbf{v} = \mathbf{b},$$

a następnie rozwiązać układ 3.10. Implementacja opisanego rozwiązania - rozwiązania sekwencyjnego - została przestawiona na listingu 3.1.

```
1  void sequential(int n, double t2, double t3, double *b, double *u){
2      double tmp;
3      const double alfa=(t2>0)?((t2+sqrt(t2*t2-4*t3))/2):
4                               ((t2-sqrt(t2*t2-4*t3))/2);
5      const double beta=t2-alfa;
6
7      b[0]=b[0]/alfa;
8      for(int i=1;i<n;i++)
9          b[i]=(b[i]-b[i-1])/alfa;
10
11     for(int i=n-2;i>=0;i--)
12         b[i]-=b[i+1]*beta;
13
14     u[0]=1.0/alfa;
15     tmp=-1/alfa;
16     for(int i=1;i<n;i++)
17         u[i]=u[i-1]*tmp;
18
19     for(int i=n-2;i>=0;i--)
20         u[i]-=u[i+1]*beta;
21
22     b[0]/=(1+beta*u[0]);
23     tmp=beta*b[0];
24     for(int i=1;i<n;i++)
25         b[i]-=tmp*u[i];
26 }
```

Listing 3.1: Algorytm sekwencyjny wyznaczania rozwiązania układu $T\mathbf{x} = \mathbf{b}$

Należy zaznaczyć, że rozwiązanie powyższe jest rozwiązaniem sekwencyjnym z oczywistymi zależnościami pomiędzy danymi przetwarzanymi w kolejnych krokach iteracyjnych, które nie mogą zostać ani zrównoleglone ani zwektoryzowane automatycznie przez kompilator. Algorytm wymaga $9n + O(1)$ operacji zmiennopozycyjnych.

W celu stworzenia algorytmu do rozwiązywania $LR\mathbf{y} = \mathbf{f}$, który będzie mógł wykorzystywać rozszerzenia wektorowe nowoczesnych komputerów zastosowano podejście *dziel i zwyciężaj* [SP11, Stp04]. Macierze L oraz R zostały przekształcone w macierze blokowe. Niech $n = rs$, gdzie $r, s > 1$. Podział macierzy L został przeprowadzony w następujący sposób:

$$L = \left[\begin{array}{ccc|ccc|ccc|ccc}
\alpha & & & & & & & & & & & \\
1 & \alpha & & & & & & & & & & \\
 & \ddots & \ddots & & & & & & & & & \\
 & & 1 & \alpha & & & & & & & & \\
\hline
 & & & 1 & \alpha & & & & & & & \\
 & & & & 1 & \alpha & & & & & & \\
 & & & & & \ddots & \ddots & & & & & \\
 & & & & & & 1 & \alpha & & & & \\
\hline
 & & & & & & & \ddots & \ddots & & & \\
 & & & & & & & & \ddots & \ddots & & \\
 & & & & & & & & & \ddots & \ddots & \\
 & & & & & & & & & & \ddots & \ddots \\
\hline
 & & & & & & & & & 1 & \alpha & \\
 & & & & & & & & & & 1 & \alpha \\
 & & & & & & & & & & & \ddots \\
 & & & & & & & & & & & 1
\end{array}\right]. \tag{3.15}$$

Gdy przyjęte zostaną oznaczenia:

$$L_s = \begin{bmatrix} \alpha & & & \\ 1 & \alpha & & \\ & \ddots & \ddots & \\ & & 1 & \alpha \end{bmatrix} \quad \text{oraz} \quad B = \begin{bmatrix} 0 & \dots & 0 & 1 \\ \vdots & & 0 & 0 \\ \vdots & \iddots & & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}, \tag{3.16}$$

macierz L można zapisać jako macierz blokową następującej postaci:

$$L = \begin{bmatrix} L_s & & & \\ B & L_s & & \\ & \ddots & \ddots & \\ & & B & L_s \end{bmatrix}. \tag{3.17}$$

Niech teraz:
$\mathbf{e}_k = (\underbrace{0, \ldots, 0}_{k}, 1, 0, \ldots, 0)^T \in \mathbb{R}^s$, $k = 0, \ldots, s-1$,
$\mathbf{z}_i = (z_{is}, \ldots, z_{(i+1)s-1})^T$,
$\mathbf{f}_i = (f_{is}, \ldots, f_{(i+1)s-1})^T$.

Przy tak przyjętych oznaczeniach można przedstawić twierdzenie o rozwiązaniu układu dolnotrójkątnego, dwudiagonalnego.

Twierdzenie 3.3. *Rozwiązaniem układu dolnotrójkątnego bidiagonalnego* $L\mathbf{z} = \mathbf{f}$ *jest:*

$$\begin{cases} \mathbf{z}_0 = L_s^{-1}\mathbf{f}_0 \\ \mathbf{z}_i = L_s^{-1}\mathbf{f}_i - z_{is-1}L_s^{-1}\mathbf{e}_0, \quad i = 1, \ldots, r-1. \end{cases} \tag{3.18}$$

Dowód. Po wymnożeniu $L\mathbf{z} = \mathbf{f}$ otrzymano:

$$\begin{cases} L_s\mathbf{z}_0 = \mathbf{f}_0 \\ L_s\mathbf{z}_i + B\mathbf{z}_{i-1} = \mathbf{f}_i, \quad i = 1, \ldots, r-1. \end{cases} \tag{3.19}$$

$$B\mathbf{z}_{i-1} = \begin{bmatrix} 0 & \ldots & 0 & 1 \\ \vdots & & 0 & 0 \\ \vdots & \iddots & & \vdots \\ 0 & \ldots & \ldots & 0 \end{bmatrix} \begin{bmatrix} z_{(i-1)s} \\ \vdots \\ \vdots \\ z_{is-1} \end{bmatrix} = z_{is-1} \begin{bmatrix} 1 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} = z_{is-1}\mathbf{e}_0 \tag{3.20}$$

Podstawiając powyższe wyliczenie, a następnie mnożąc lewostronnie przez L_s^{-1} uzyskano układ (3.18). □

Przekształcenie w macierz blokową doprowadziło do równania (3.18), które może zostać efektywnie zrównoleglone.

Pierwszym krokiem do rozwiązania (3.18) jest znalezienie wszystkich wektorów $L_s^{-1}\mathbf{f}_i$ dla $i = 0, \ldots, r-1$. Następnie można wyliczyć wektory $\mathbf{z}_i$ dla $i = 1, \ldots, r-1$ używając konstrukcji OpenMP `for simd` jeden po drugim, ponieważ każdy kolejny wektor $\mathbf{z}_i$ jest obliczany na podstawie ostatniej składowej poprzedniego wektora, czyli $\mathbf{z}_{i-1}$. Dlatego zanim obliczony zostanie kolejny wektor, wszystkie wątki muszą zostać zsynchronizowane. Wektory $\mathbf{z}_i$ można również obliczyć korzystając z innego podejścia, a mianowicie znaleźć wszystkie ostatnie składowe wektorów $\mathbf{z}_i$. Następnie $s-1$ początkowych elementów może zostać obliczona równolegle bez konieczności synchronizacji wątków. Opisane powyżej podejście pierwsze będzie nazywane wersją pierwszą algorytmu, a podejście drugie wersją drugą. Zostaną one szerzej omówione i zaprezentowane w sekcji 3.3.

Analogiczna sytuacja ma miejsce dla macierzy R. Pierwszy krok, to podział na macierze blokowe w sposób pokazany poniżej:

$$R = \left[\begin{array}{ccccc|ccc|ccc|ccc}
1 & \beta & & & & & & & & & & & \\
 & 1 & \ddots & & & & & & & & & & \\
 & & \ddots & \beta & & & & & & & & & \\
 & & & 1 & \beta & & & & & & & & \\
\hline
 & & & & & 1 & \beta & & & & & & \\
 & & & & & & 1 & \ddots & & & & & \\
 & & & & & & & \ddots & \beta & & & & \\
 & & & & & & & & 1 & \beta & & & \\
\hline
 & & & & & & & & \ddots & \ddots & & & \\
 & & & & & & & & & \ddots & \ddots & & \\
 & & & & & & & & & & \ddots & \ddots & \\
 & & & & & & & & & & & \ddots & \\
\hline
 & & & & & & & & & & 1 & \beta & \\
 & & & & & & & & & & & \ddots & \ddots \\
 & & & & & & & & & & & & \ddots \\
 & & & & & & & & & & & & \beta \\
 & & & & & & & & & & & & 1
\end{array}\right]. \tag{3.21}$$

Ponownie, gdy przyjęte zostaną oznaczenia:

$$R_s = \begin{bmatrix} 1 & \beta & & \\ & 1 & \ddots & \\ & & \ddots & \beta \\ & & & 1 \end{bmatrix} \quad \text{oraz} \quad C = \begin{bmatrix} 0 & \dots & \dots & 0 \\ \vdots & & \iddots & \vdots \\ 0 & 0 & & \vdots \\ \beta & 0 & \dots & 0 \end{bmatrix}, \tag{3.22}$$

macierz R można zapisać jako:

$$R = \begin{bmatrix} R_s & C & & \\ & R_s & \ddots & \\ & & \ddots & C \\ & & & R_s \end{bmatrix}. \tag{3.23}$$

Analogicznie do twierdzenia o rozwiązaniu układu dolnotrójkątnego, dwudiagonalnego przedstawić można twierdzenie o rozwiązaniu układu górnotrójkątnego, dwudiagonalnego.

Twierdzenie 3.4. *Rozwiązaniem układu górnotrójkątnego bidiagonalnego $R\mathbf{y} = \mathbf{z}$ jest:*

$$\begin{cases} \mathbf{y}_{r-1} = R_s^{-1}\mathbf{z}_{r-1} \\ \mathbf{y}_i = R_s^{-1}\mathbf{z}_i - \beta y_{(i+1)s} R_s^{-1}\mathbf{e}_{s-1}, \quad i = r-2, \dots, 0. \end{cases} \tag{3.24}$$

Dowód. Po wymnożeniu $R\mathbf{y} = \mathbf{z}$ otrzymano:

$$\begin{cases} R_s\mathbf{y}_{r-1} = \mathbf{z}_{r-1} \\ R_s\mathbf{y}_i = \mathbf{z}_i - C\mathbf{y}_{i+1}, \quad i = r-2, \dots, 0. \end{cases} \tag{3.25}$$

$$C\mathbf{y}_{i+1} = \begin{bmatrix} 0 & \dots & \dots & 0 \\ \vdots & & \ddots & \vdots \\ & & & \\ 0 & 0 & & \vdots \\ \beta & 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} y_{(i+1)s} \\ \vdots \\ \\ \vdots \\ y_{(i+2)s-1} \end{bmatrix} = \beta y_{(i+1)s} \begin{bmatrix} 0 \\ \vdots \\ \\ \vdots \\ 1 \end{bmatrix} = \beta y_{(i+1)s} \mathbf{e}_{s-1}. \tag{3.26}$$

Podstawiając powyższe wyliczenie, a następnie mnożąc lewostronnie przez R_s^{-1} uzyskano układ (3.24). □

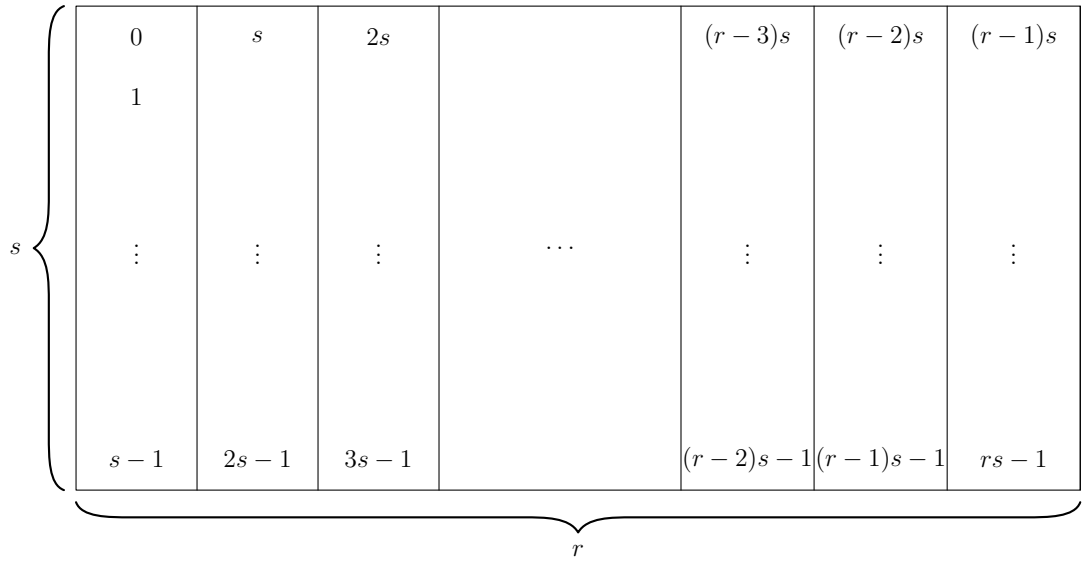
Jak łatwo zauważyć, sytuacja jest analogiczna do układu dolnotrójkątnego. Aby znaleźć wektory $\mathbf{y}_i$ dla $i = 0, \dots, r-2$ potrzeba teraz pierwszych składowych wszystkich wektorów $\mathbf{y}_i$ dla $i = 1, \dots, r-1$. Można w pierwszej wersji obliczać wektory jeden po drugim z synchronizacją lub w wersji drugiej obliczyć wszystkie pierwsze składowe $\mathbf{y}_i$, a pozostałe $s-1$ składowych można wyznaczyć równolegle, bez potrzeby synchronizacji. Algorytm taki wymaga $16n - 3r - 6s + O(1)$ operacji zmiennopozycyjnych.

3.3 Implementacja

Korzystając z opisanych wcześniej wzorów (3.18), (3.24) oraz (3.10) można sformułować algorytm do rozwiązywania układów równań postaci (3.1). Do zaprogramowania wykorzystano język C wraz z interfejsem OpenMP (sekcja 1.2.2). Zaproponowany algorytm - w obu wersjach - wykorzystuje następującą liczbę zmiennych wymienionego typu: $4 \cdot (int) + (2n + s + 6) \cdot (double)$. Poniższy opis przedstawia rolę zmiennych wykorzystanych w implementacjach:

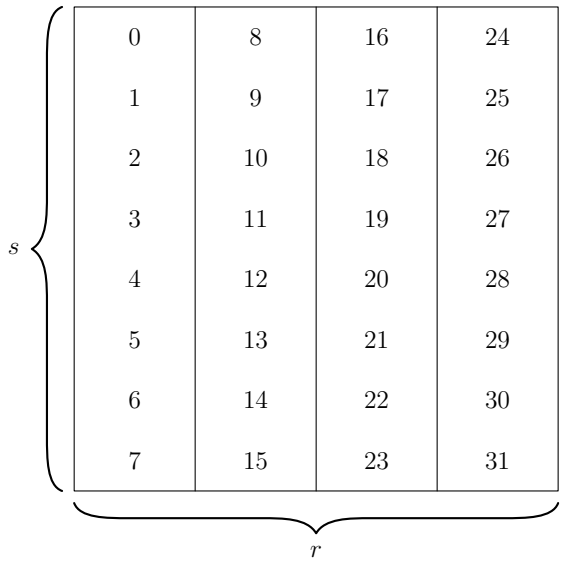
- zmienna `b`, w której przechowano wektor wejściowy $\mathbf{b}$, który następnie został przekształcony w wektor $\mathbf{v}$ ze wzoru (3.7) i ostatecznie w wynik $\mathbf{x}$ ze wzoru (3.10) - n elementów typu double,
- zmienna `u`, w której obliczono wektor $\mathbf{u}$ ze wzoru (3.7) - można zauważyć, że początkowe s elementów wektora $\mathbf{u}$ pokrywa się z wektorem $\mathbf{e}_0$ - n elementów typu double,
- zmienna `es`, w której obliczono wektor $\mathbf{e}_{s-1}$ - s elementów typu double.

Przyjmując, że wektory reprezentowane są za pomocą tablicy jednowymiarowej o rozmiarze $n = rs$, gdzie $r, s > 1$, dane będą interpretowane do potrzeb wzorów (3.18) i (3.24) tak, jak pokazano na rysunku 3.1. Format taki będzie nazywany układem kolumnowym. W kolejnym rozdziale zostanie zaprezentowany układ wierszowy oraz efektywna zamiana pomiędzy układami.



Rysunek 3.1: Układ danych po podziale na macierz blokową, gdzie $n = rs$

Szczegółowy przykład dla $n = 32$, $r = 4$ oraz $s = 8$, znajduje się na rysunku 3.2.



Rysunek 3.2: Układ danych po podziale na macierz blokową, gdzie $32 = 4 \cdot 8$, (tzn. $n = 32$, $r = 4$, $s = 8$)

Skrócony fragment kodu obu wersji został zaprezentowany na rysunku 3.3. Przedstawia on elementy wspólne oraz różnice występujące między wersjami w obliczaniu fragmentu wzoru (3.18) (linie 8-41), a także wzoru (3.10) (linie 41-47). Przyjęto założenie, że n, r i s są potęgami dwójki, dlatego każda z kolumn jest poprawnie wyrównana w pamięci (linie kodu 14 i 32). Ponadto, każda kolumna zajmuje spójny blok w pamięci.

Występują dwa rodzaje pętli. Pierwszy (linie kodu 12-19) nie wykorzystuje rozszerzeń wektorowych, ale może zostać wykonany równolegle. Zauważmy, że pętla wewnętrzna (linie kodu 17-18) korzysta z poprzedniego elementu kolumny, dlatego może zostać wykonana z użyciem pamięci podręcznej. Linie 35-37 zawierają inny rodzaj pętli. Jest to pętla zrównoleglona, która wykorzystuje rozszerzenia wektorowe (przy użyciu konstrukcji OpenMP `for simd`). Różnice pomiędzy wersjami są niewielkie. W przypadku pierwszej wersji umieszczono jawną barierę po pętli wewnętrznej z linii 35-37,

```
void method(int n, int r, double a, double d, double *b, double *u){
    const double alfa=(d>0)?((d+sqrt(d*d-4*a))/2):((d-sqrt(d*d-4*a))/2);
    const double beta=d-alfa, tmp=-1/alfa;
    u[0]=1./alfa;
    int s=n/r;
    double *es=_mm_malloc(s*sizeof(double), 64); es[s-1]=1;

#pragma omp parallel  //rozpoczyna się obszar zrównoleglony
    {
     double tmp=u[0];
#pragma omp for nowait schedule(static)
     for(int j=0;j<r;j++){
       double *col;
       __assume_aligned(col,64);  // każda kolumna jest poprawnie wyrównana
       col=&b[j*s];
       col[0]/=alfa;
       for(int i=1;i<s;i++)
         col[i]=(col[i]-col[i-1])/alfa;
     }
#pragma omp single
     for(int i=1;i<s;i++)
       u[i]=u[i-1]*tmp;
      //bariera niejawna
```

```
//wersja 1




for(int j=1;j<r;j++){
  double *col;
  __assume_aligned(col,64);
  col=&b[j*s];
  double last=b[j*s-1];
#pragma omp for simd schedule(static)
  for(int i=0;i<s;i++)
    col[i]-=last*u[i];
   //bariera niejawna
}
 //...reszta implementacji
```

```
//wersja 2

  #pragma omp single
  for(int j=1;j<r;j++)
    b[(j+1)*s-1]-=b[j*s-1]*u[s-1];

  for(int j=1;j<r;j++){
    double *col;
    __assume_aligned(col,64);
    col=&b[j*s];
    double last=b[j*s-1];
#pragma omp for simd nowait schedule(static)
    for(int i=0;i<s-1;i++)
      col[i]-=last*u[i];
  }
#pragma omp barrier
 //...reszta implementacji
```

```
#pragma omp single
     b[0]/=(1+beta*u[0]);

     tmp=beta*b[0];
#pragma omp for simd schedule(static)
     for(int i=1;i<n;i++)
       b[i]-=tmp*u[i];

 }  //koniec obszaru zrównoleglonego
  _mm_free(es);
}
```

Rysunek 3.3: Fragmenty zrównoleglonych implementacji zaproponowanego algorytmu opierające się na wzorach (3.18) oraz (3.10) (pominięto obliczenia z wzoru (3.24))

ponieważ do następnej iteracji zewnętrznej pętli (linie 30-39) potrzeba ostatniego elementu poprzedniej kolumny, co wynika ze wzoru (3.18).

Jako inne podejście, można znaleźć najpierw wszystkie ostatnie składowe w pętli sekwencyjnej (linie 26-28), a następnie wewnętrzna pętla (linie 35-37) może zostać uruchomiona z klauzulą `nowait`. Jawna bariera musi zostać wtedy umieszczona po zewnętrznej pętli (linia 39), aby upewnić się, że wszystkie niezbędne obliczenia zostały zakończone.

Analogiczna sytuacja ma miejsce ze wzorem (3.24). Jednak w tym przypadku zamiast ostatnich elementów potrzeba pierwszych składowych wektora. Problem ten rozwiązać można dokładnie w ten sam sposób: w pierwszej wersji wyliczyć należy całe wektory, po których występuje niejawna bariera lub - w drugiej wersji - obliczyć wszystkie pierwsze elementy wektorów, a następnie równolegle pozostałe $s - 1$ elementów.

3.4 Wyniki

Wyniki eksperymentalne tego rozdziału zostały zebrane dla dwóch architektur:

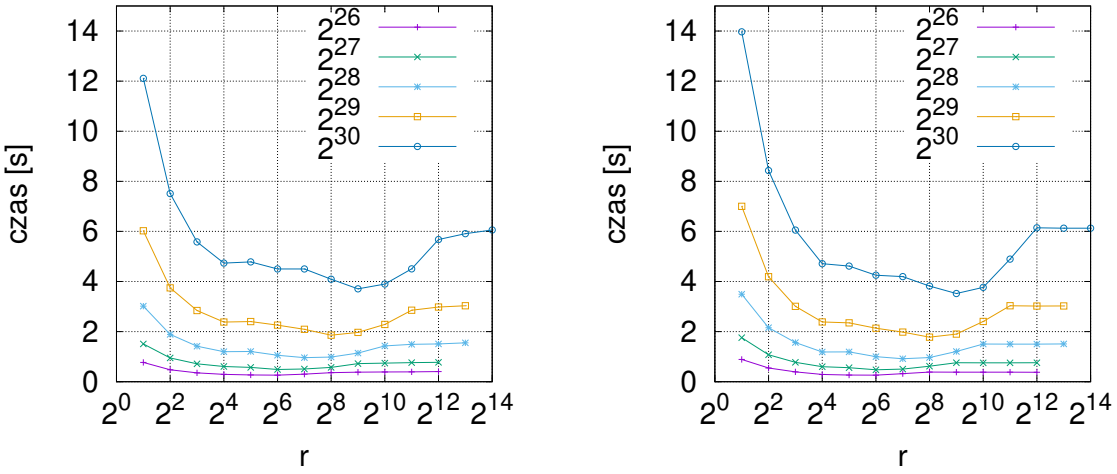
- serwer z dwoma procesorami nazwanymi I w sekcji 1.3 - w dalszej części nazywany CPU,
- serwer z procesorem nazwanym II w sekcji 1.3 - w dalszej części nazywany będzie MIC.

Obie wersje działały pod kontrolą systemu Linux wraz z Intel Parallel Studio wersja 2017. Eksperymenty na procesorze Xeon Phi zostały wykonane przy użyciu jego trybu natywnego [JRS16a].

Przetestowano wszystkie opisane wersje, to znaczy:

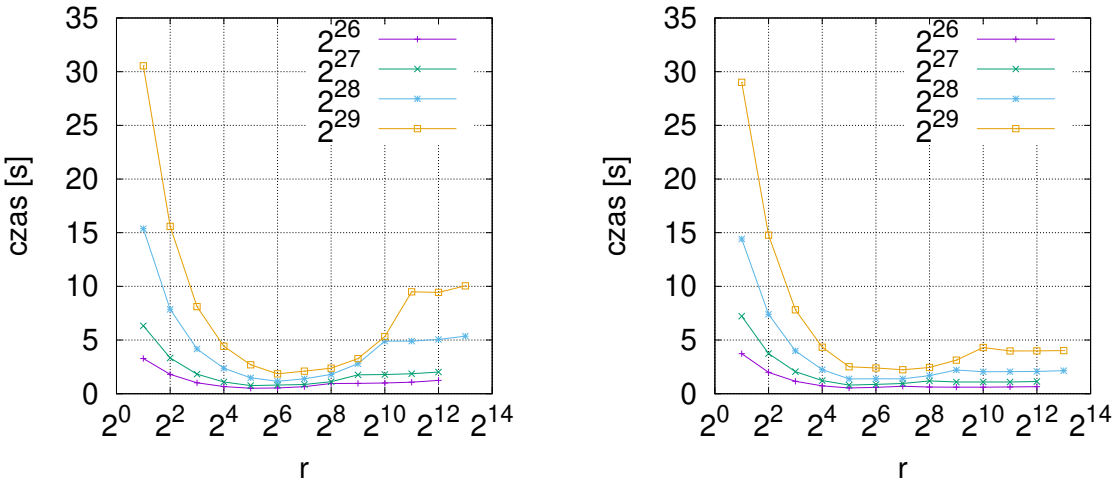
- `Sekwencyjny` - implementacja algorytmu Thomasa opisanego w sekcji 2.4.1, a następnie wykorzystanie wzoru (3.10), zoptymalizowana automatycznie przez kompilator,
- `Wersja_1` - wyliczenie każdej części wektorów, jedna po drugiej,
- `Wersja_2` - wyliczenie pierwszych lub ostatnich składowych oddzielnie, a następnie pozostałe $s - 1$ elementów równolegle.

Na obu rozważanych architekturach (CPU i MIC), w większości przypadków, najlepsze wyniki czasowe dla algorytmów równoległych uzyskano przy wykorzystaniu jednego wątku na rdzeń. Inny efekt zaobserwowano jedynie na MIC dla największych rozmiarów badanego problemu w `Wersji_1` algorytmu. Najlepszy efekt osiągnięto wtedy dla dwóch wątków na rdzeń. Osiągnięte wyniki zostały zaprezentowane na rysunkach 3.4, 3.5 oraz tabelach 3.1 i 3.2.



(a) (b)

Rysunek 3.4: Czas działania dla różnych rozmiarów układu n oraz parametru r na CPU: `Wersja_1` (a), `Wersja_2` (b)



(a) (b)

Rysunek 3.5: Czas działania dla różnych rozmiarów układu n oraz parametru r na MIC: `Wersja_1` (a), `Wersja_2` (b)

Górne ograniczenie rozmiaru badanego problemu różni się ze względu na inną strukturę pamięci obu architektur. CPU pozwala na badania do $n = 2^{30}$, natomiast na MIC występuje ograniczenie do $n = 2^{29}$. Dlatego rysunki 3.4 i 3.5 przedstawiają różne dane. Rysunek 3.4 i tabela 3.1 - CPU - przedstawia dane dla $n \in \{2^{27}, \ldots, 2^{30}\}$, natomiast rysunek 3.5 i tabela 3.2 - MIC - dla $n \in \{2^{26}, \ldots, 2^{29}\}$. Pierwszym krokiem było zbadanie wpływu parametru r, a w konsekwencji również $s = n/r$, na czas działania programu. Wycinek tych danych został zaprezentowany na rysunkach 3.4 i 3.5. Zaobserwowano, że parametr ten ma znaczący wpływ na wydajność czasową. Badanie rozpoczęto od $r = 2$. Zwiększanie tego parametru poprawia efektywność tylko do pewnego stopnia. Większe r oznacza mniejsze s, czyli należy znaleźć optymalny punkt przy zwiększaniu pętli po r. Można również zaobserwować, że dla tej samej architektury wykresy mają podobny kształt, różnią się jednak znacząco wykresy różnych architektur. Oznacza to,

że wybór optymalnego r będzie uzależniony od rodzaju wybranej architektury.

Tabele 3.1 i 3.2 przedstawiają czas trwania programów: `Sekwencyjnego`, `Wersji_1` i `Wersji_2` oraz przyspieszenie dla optymalnej wartości parametru r. Przyspieszenia zostały obliczone według wzoru z definicji 1.1 z sekcji 1.5, gdzie algorytmem bazowym jest wersja sekwencyjna, a algorytmem rozpatrywanym odpowiednio dwie przedstawione w tym rozdziale implementacje równoległe.

Tabela 3.1: Czas działania [s] i przyspieszenie względem wersji sekwencyjnej dla optymalnych wartości parametru r (CPU)

	`Sekwencyjny`	`Wersja_1`			`Wersja_2`		
n	czas	r	czas	przyspieszenie	r	czas	przyspieszenie
2^{20}	0,0160	2^4	0,0095	1,69	2^5	0,0090	1,79
2^{21}	0,0336	2^4	0,0141	2,37	2^6	0,0119	2,83
2^{22}	0,0704	2^6	0,0233	3,02	2^6	0,0219	3,22
2^{23}	0,1393	2^4	0,0409	3,41	2^8	0,0435	3,21
2^{24}	0,2786	2^4	0,0746	3,74	2^4	0,0783	3,56
2^{25}	0,5572	2^4	0,1410	3,95	2^4	0,1433	3,89
2^{26}	1,1143	2^6	0,2609	4,27	2^6	0,2621	4,25
2^{27}	2,2263	2^6	0,4873	4,57	2^6	0,4755	4,68
2^{28}	4,4509	2^7	0,9606	4,63	2^7	0,9183	4,85
2^{29}	8,9077	2^8	1,8586	4,79	2^8	1,7777	5,01
2^{30}	17,8155	2^9	3,7072	4,81	2^9	3,5225	5,06

Tabela 3.2: Czas działania [s] i przyspieszenie względem wersji sekwencyjnej dla optymalnych wartości parametru r (MIC)

	`Sekwencyjny`	`Wersja_1`			`Wersja_2`		
n	czas	r	czas	przyspieszenie	r	czas	przyspieszenie
2^{20}	0,1095	2^6	0,1503	0,73	2^6	0,1391	0,79
2^{21}	0,2177	2^6	0,1711	1,27	2^7	0,1524	1,43
2^{22}	0,4328	2^5	0,1981	2,19	2^7	0,1705	2,54
2^{23}	0,8582	2^4	0,2458	3,49	2^8	0,2008	4,27
2^{24}	1,7095	2^5	0,2974	5,75	2^8	0,2633	6,49
2^{25}	3,4111	2^5	0,3649	9,35	2^9	0,3671	9,29
2^{26}	6,8127	2^5	0,5095	13,37	2^5	0,5357	12,72
2^{27}	13,6193	2^5	0,7563	18,01	2^5	0,8088	16,84
2^{28}	27,2880	2^6	1,1568	23,59	2^5	1,3780	19,80
2^{29}	54,5765	2^6	1,8531	29,45	2^7	2,2310	24,46

Na CPU `Wersja_2` uzyskała lepsze wyniki dla największych rozważanych n. Mniejsze rozmiary dają zbliżone wyniki dla obu wersji. Dla MIC sytuacja jest odwrotna - dla dużych n lepsza okazała się `Wersja_1`, a dla mniejszych nieznacznie lepsza jest `Wersja_2`. Przyspieszenie na CPU sięga około 5 dla obu wersji (nieznacznie lepsze

dla `Wersji_2`). Dla MIC przyspieszenie dla `Wersji_1` sięga prawie 30, a efekt ten zaobserwowano ze względu na kluczowe na tej architekturze wykorzystanie rozszerzeń wektorowych.

Zbadano również dokładność zaproponowanych trzech metod za pomocą sposobu opisanego w sekcji 1.5. Osiągnięte wyniki - zarówno dla algorytmu sekwencyjnego jak i dwóch zaproponowanych zrównoleglonych wersji - są zadowalające, ponieważ sięgają rzędu dokładności użytej precyzji zmiennych (podwójna precyzja), czyli około 10^{-16}.

3.5 Wydajność energetyczna

Jako kolejny element analizy efektywności zaprezentowanego rozwiązania zbadano wydajność energetyczną algorytmu sekwencyjnego oraz obu zaproponowanych wersji równoległych. Dane zostały zebrane na serwerze z dwoma procesorami oznaczonymi jako I w sekcji 1.3, czyli Intel Xeon E5-2670 v3 z wykorzystaniem interfejsu Running Average Power Limit (RAPL) Intela [KHN+18]. Interfejs ten pozwala na mierzenie zużycia energii procesora (CPU) oraz pamięci dynamicznej (DRAM). Tabela 3.3 przedstawia zużycie energii w dżulach przez obie zaproponowane w pracy wersje oraz przez algorytm sekwencyjny dla różnych rozmiarów badanego problemu od $n = 2^{20}$ do $n = 2^{30}$.

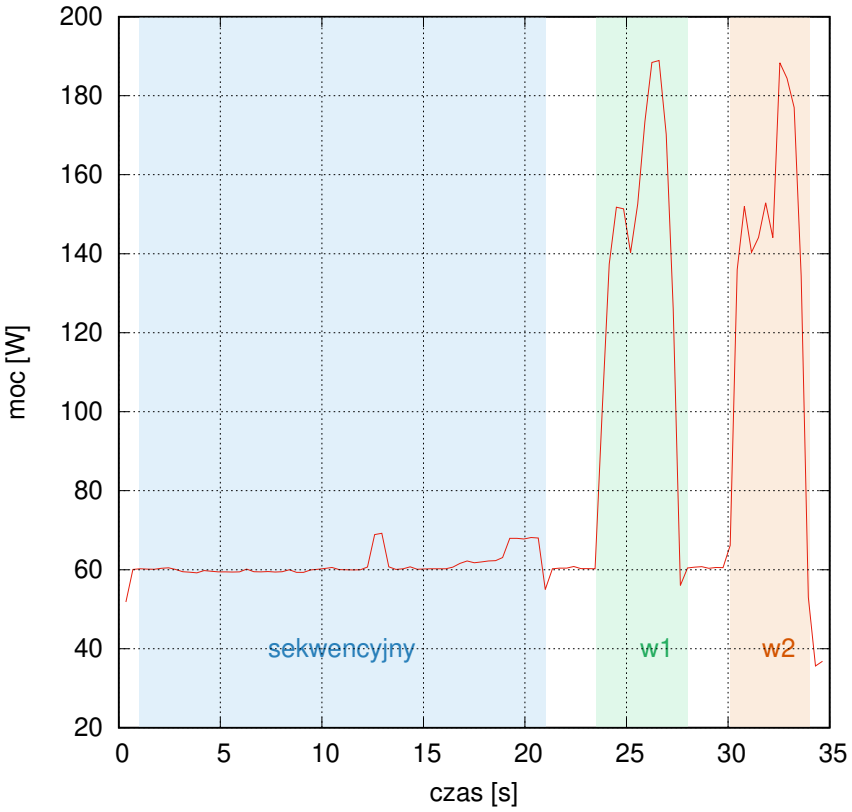
Tabela 3.3: Całkowite zużycie energii [J] przez procesor podczas wykonywania wszystkich rozważanych implementacji i różnych rozmiarów problemu z zaznaczonym najlepszym wynikiem

n	`Sekwencyjny`	`Wersja_1`	`Wersja_2`
2^{20}	1,06	1,25	0,97
2^{21}	2,10	1,98	1,95
2^{22}	4,10	3,17	3,11
2^{23}	8,57	5,44	6,03
2^{24}	17,57	10,26	11,02
2^{25}	35,88	20,80	20,78
2^{26}	70,25	40,17	39,17
2^{27}	139,68	75,03	71,32
2^{28}	277,47	147,28	142,93
2^{29}	544,99	291,43	281,67
2^{30}	1092,91	583,29	543,25

Z wyników przedstawionych w tabeli 3.3 można wnioskować, że algorytm sekwencyjny jest zdecydowanie najmniej wydajny energetycznie dla każdego z badanych rozmiarów. Potrzebuje on w przybliżeniu o 50% więcej energii niż każda z wersji równoległych zaproponowanych w pracy. Dla większości badanych rozmiarów lepszym okazuje się wersja druga zaproponowanego algorytmu, jednak dopiero dla największych z badanych rozmiarów ($n > 2^{26}$) różnice te są znaczące. Dla $n \leqslant 2^{26}$ różnice pomiędzy dwoma wersjami są relatywnie małe.

Rysunek 3.6 prezentuje zużycie energii przez wszystkie badane w pracy wersje, lecz dla ustalonego $n = 2^{30}$ oraz optymalnego r. Badanie rozpoczęło się od wykonania algorytmu sekwencyjnego (trwa około 18 sekund), następnie wykonano wersję pierwszą

(około 4 sekundy) i jako ostatnią wersję drugą (około 4 sekundy). Z rysunku 3.6 wynika, że zużycie energii w trakcie wykonywania `Wersji_1` oraz `Wersji_2` jest znacznie wyższe, lecz trwa jedynie przez krótki czas. Algorytm sekwencyjny zużywa mniej energii w jednostce czasu, lecz trwa o wiele dłużej, co w efekcie przekłada się na znacząco zwiększone całkowite zużycie energii.



Rysunek 3.6: Bieżące zużycie energii oraz całkowite zużycie energii wymagane do wykonania rozważanych metod dla $n = 2^{30}$

Rozdział 4

Rozwiązywanie układów równań liniowych postaci (t_1, t_2, t_3)

Rozdział ten zawiera dwa algorytmy rozwiązywania układów równań liniowych o macierzy współczynników specjalnej postaci, bardziej ogólnej od układu przedstawionego w rozdziale poprzednim (rozdział 3). Przedstawiono algorytm analogiczny do algorytmu z rozdziału 3, jednak występują znaczące różnice. W rozdziale 3 zaprezentowano implementacje z wykorzystaniem OpenMP dla procesorów CPU oraz zbadano zużycie energii. Natomiast w rozdziale bieżącym przedstawiono dwa algorytmy: pierwszy - jak już powiedziano - analogiczny do algorytmu wprowadzonego w rozdziale 3 i drugi - oparty na sekwencyjnej metodzie opisanej w pracy [LLYZ20]. Pokazano różne wersje implementacji przedstawionych algorytmów z wykorzystaniem OpenACC. Wprowadzono różne formaty danych do przedstawiania długich wektorów - utożsamianych z prawą stroną równania, bądź wektorem niewiadomych. Pokazano cztery wersje formatów danych, każdy zapisany w jednowymiarowej tablicy: format kolumny - naturalny, format wierszowy z prostą zamianą danych między formatami, format wierszowy z zamianą wykorzystującą pamięć podręczną oraz format kolumnowy z obliczeniami wykorzystującymi pamięć podręczną. Zawarto również wyniki czasowe zebrane na architekturach CPU, GPU oraz na platformie hybrydowej. Zbadano dokładności przedstawionych funkcji oraz podjęto próbę predykcji parametrów programu.

4.1 Wstęp

Rozważono trójdiagonalny układ równań liniowych typu Toeplitza $T\mathbf{x} = \mathbf{b}$ następującej postaci:

$$\begin{bmatrix} t_2 & t_3 & & & \\ t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & t_1 & t_2 & t_3 \\ & & & t_1 & t_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{4.1}$$

Dla uproszczenia można przyjąć założenie, że $n = 2^k$, $k \in \mathbb{N}$.

4.2 Algorytmy równoległe

W celu sformułowania wydajnego programu rozwiązującego układy postaci (4.1) należy podać algorytm, który będzie mógł zostać zrównoleglony. Jest to kluczowe, aby wykorzystać architektury nowoczesnych komputerów, takie jak np. rozszerzenia wektorowe. Przedstawione zostaną dwa takie algorytmy.

4.2.1 Pierwszy algorytm - modyfikacja metody Wanga

W sekcji tej zaprezentowany zostanie pierwszy algorytm, który oparty jest na metodzie Wanga [Wan81]. Jak już zostało powiedziane w sekcji 2.4.4 metoda ta jest algorytmem typu *dziel i zwyciężaj*. Pierwszym więc krokiem omawianego algorytmu jest podział macierzy współczynników zaprezentowany twierdzeniem 4.1.

Twierdzenie 4.1. *Macierz T postaci:*

$$\begin{bmatrix} t_2 & t_3 & & & \\ t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & t_1 & t_2 & t_3 \\ & & & t_1 & t_2 \end{bmatrix} \tag{4.2}$$

może zostać rozłożona na następujące czynniki:

$$T = \underbrace{\begin{bmatrix} 1 & & & & \\ \alpha & 1 & & & \\ & \ddots & \ddots & & \\ & & \alpha & 1 & \\ & & & \alpha & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \beta & t_3 & & & \\ & \beta & t_3 & & \\ & & \ddots & \ddots & \\ & & & \beta & t_3 \\ & & & & \beta \end{bmatrix}}_{R} + \underbrace{\begin{bmatrix} t_3\alpha & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}}_{P}, \tag{4.3}$$

gdzie

$$\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_1t_3}}{2t_3} \quad \textit{oraz} \quad \beta = t_2 - t_3\alpha. \tag{4.4}$$

Dowód. Po wymnożeniu prawej strony otrzymano dwa następujące równania:

$$\begin{cases} \beta + t_3\alpha = t_2 \\ \beta\alpha = t_1. \end{cases} \tag{4.5}$$

Po wyliczeniu z pierwszego z nich $\beta = t_2 - t_3\alpha$ i podstawieniu do drugiego otrzymano następujące równanie kwadratowe:

$$t_3(\alpha)^2 - t_2\alpha + t_1 = 0,$$

a jego rozwiązania będą postaci:

$$\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_1t_3}}{2t_3}.$$

□

Korzystając z dekompozycji przedstawionej w twierdzeniu 4.1, układ ze wzoru (4.1) może zostać przepisany do następującej postaci:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \underbrace{(LR)^{-1} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}}_{\mathbf{v}} - t_3 \alpha x_0 \underbrace{(LR)^{-1} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\mathbf{u}} \tag{4.6}$$

lub jeszcze prościej:

$$\mathbf{x} = \mathbf{v} - t_3 \alpha x_0 \mathbf{u}. \tag{4.7}$$

Bazując na elementach opisanych do tej pory można sformułować twierdzenie 4.2. Twierdzenie to prezentuje rozwiązanie układu z wzoru (4.1).

Twierdzenie 4.2. *Niech* $T\mathbf{x} = \mathbf{b}$*, gdzie*

$$T = \underbrace{\begin{bmatrix} 1 & & & & \\ \alpha & 1 & & & \\ & \ddots & \ddots & & \\ & & \alpha & 1 & \\ & & & \alpha & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \beta & t_3 & & & \\ & \beta & t_3 & & \\ & & \ddots & \ddots & \\ & & & \beta & t_3 \\ & & & & \beta \end{bmatrix}}_{R} + \underbrace{\begin{bmatrix} t_3\alpha & 0 & \dots & 0 \\ 0 & 0 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix}}_{P}, \tag{4.8}$$

$\alpha = \frac{t_2 \pm \sqrt{(t_2)^2 - 4t_1t_3}}{2t_3}$, *a* $\beta = t_2 - t_3\alpha$. *Niech również* $\mathbf{v} = (LR)^{-1}\mathbf{b}$, $\mathbf{u} = (LR)^{-1}\mathbf{e}_0$, *gdzie* $\mathbf{e}_0 = (1, 0, \dots, 0)^T$. *Wtedy rozwiązanie równania* $T\mathbf{x} = \mathbf{b}$*, może zostać wyznaczone ze wzoru:*

$$\begin{cases} x_0 = v_0/(1 + t_3\alpha u_0) \\ x_i = v_i - t_3\alpha x_0 u_i, \quad i = 1, \dots, n-1. \end{cases} \tag{4.9}$$

Dowód. Układ (4.7) można zapisać następująco:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} - t_3\alpha x_0 \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{bmatrix}, \tag{4.10}$$

co prowadzi do równoważnej formy:

$$\begin{cases} x_0 = v_0 - t_3\alpha x_0 u_0 \\ x_i = v_i - t_3\alpha x_0 u_i, \quad i = 1, \dots, n-1. \end{cases} \tag{4.11}$$

Po prostym przekształceniu powyższego wzoru otrzymano wzór (4.9) z twierdzenia 4.2. □

Aby znaleźć rozwiązanie wzoru (4.1) korzystając z twierdzenia 4.2, należy znaleźć dwa wektory: $\mathbf{v}$ oraz $\mathbf{u}$, rozwiązując dwa układy z taką samą macierzą współczynników, tzn. LR. Rozwiązania takich układów mogą zostać znalezione za pomocą algorytmu Thomasa przedstawionego w sekcji 2.4.1. Niech układ, który należy rozwiązać będzie postaci $LR\mathbf{y} = \mathbf{f}$. Wtedy rozwiązanie można otrzymać poprzez rozwiązanie dwóch

układów: $L\mathbf{z} = \mathbf{f}$, a następnie $R\mathbf{y} = \mathbf{z}$. Przekłada się to na następujące wzory sekwencyjne:

$$\begin{cases} z_0 = f_0 \\ z_i = f_i - \alpha z_{i-1}, \quad i = 1, \ldots, n-1 \end{cases} \tag{4.12}$$

oraz

$$\begin{cases} y_{n-1} = z_{n-1}/\beta \\ y_i = (z_i - t_3 y_{i+1})/\beta, \quad i = n-2, \ldots, 0. \end{cases} \tag{4.13}$$

Jak już wspomniano wcześniej, taki sekwencyjny algorytm może być efektywny tylko dla małych wartości n, ponieważ nie wykorzystuje on możliwości architektury współczesnych procesorów oraz procesorów graficznych. Ponadto, nie może on zostać automatycznie zoptymalizowany przez kompilatory ze względu na zależności między danymi. W celu otrzymania efektywnego rozwiązania zastosowano podejście analogiczne do zaprezentowanego w rozdziale 3, a mianowicie metodę *dziel i zwyciężaj* [DS20,DS21].

Jako pierwszy krok należy wybrać dwie liczby naturalne $r, s > 1$, $rs = n$, a następnie zapisać układ (4.1) w formie blokowej korzystając z dekompozycji przedstawionej w twierdzeniu 4.1. Analogicznie do sposobu zaprezentowanego jako wzór (3.15) w rozdziale 3, zapisano macierz L w formie blokowej:

$$L = \begin{bmatrix} L_s & & & \\ B & L_s & & \\ & \ddots & \ddots & \\ & & B & L_s \end{bmatrix}, \tag{4.14}$$

gdzie

$$L_s = \begin{bmatrix} 1 & & & \\ \alpha & 1 & & \\ & \ddots & \ddots & \\ & & \alpha & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 & \ldots & 0 & \alpha \\ \vdots & & 0 & 0 \\ \vdots & \iddots & & \vdots \\ 0 & \ldots & \ldots & 0 \end{bmatrix}. \tag{4.15}$$

Macierze kolumnowe $\mathbf{f}$ oraz $\mathbf{z}$ podzielone zostały na następujące wektory, odpowiednio:

$$\mathbf{f}_i = (f_{is}, \ldots, f_{(i+1)s-1})^T \quad \text{oraz} \quad \mathbf{z}_i = (z_{is}, \ldots, z_{(i+1)s-1})^T \in \mathbb{R}^s, \quad i = 0, \ldots, r-1.$$

Niech $\mathbf{e}_k = (\underbrace{0, \ldots, 0}_{k}, 1, 0, \ldots, 0)^T \in \mathbb{R}^s,\ k = 0, \ldots, s-1.$

Mając powyższe oznaczenia można przejść do sformułowania twierdzenia o rozwiązaniu układu blokowego $L\mathbf{z} = \mathbf{f}$.

Twierdzenie 4.3. *Rozwiązaniem układu $L\mathbf{z} = \mathbf{f}$ jest:*

$$\begin{cases} \mathbf{z}_0 = L_s^{-1}\mathbf{f}_0 \\ \mathbf{z}_i = L_s^{-1}\mathbf{f}_i - \alpha z_{is-1} L_s^{-1}\mathbf{e}_0, \quad i = 1, \ldots, r-1. \end{cases} \tag{4.16}$$

Dowód. Po wymnożeniu blokowej wersji $L\mathbf{z} = \mathbf{f}$, to znaczy:

$$\begin{bmatrix} L_s & & & \\ B & L_s & & \\ & \ddots & \ddots & \\ & & B & L_s \end{bmatrix} \begin{bmatrix} \mathbf{z}_0 \\ \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_{r-1} \end{bmatrix} \tag{4.17}$$

otrzymano:

$$\begin{cases} L_s\mathbf{z}_0 = \mathbf{f}_0 \\ L_s\mathbf{z}_i + B\mathbf{z}_{i-1} = \mathbf{f}_i, \quad i = 1, \ldots, r-1. \end{cases} \tag{4.18}$$

Upraszczając:

$$B\mathbf{z}_{i-1} = \begin{bmatrix} 0 & \ldots & 0 & \alpha \\ \vdots & & 0 & 0 \\ \vdots & \iddots & & \vdots \\ 0 & \ldots & \ldots & 0 \end{bmatrix} \begin{bmatrix} z_{(i-1)s} \\ \vdots \\ \vdots \\ z_{is-1} \end{bmatrix} = \alpha z_{is-1} \begin{bmatrix} 1 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} = \alpha z_{is-1}\mathbf{e}_0. \tag{4.19}$$

Podstawiając powyższe wyliczenie, a następnie mnożąc lewostronnie przez L_s^{-1} uzyskano układ z twierdzenia (4.3). □

Podobnie jest w przypadku górnej bidiagonalnej macierzy R. Przy tych samych założeniach, otrzymano następującą wersję blokową:

$$R = \begin{bmatrix} R_s & C & & \\ & R_s & \ddots & \\ & & \ddots & C \\ & & & R_s \end{bmatrix}, \tag{4.20}$$

gdzie

$$R_s = \begin{bmatrix} \beta & t_3 & & \\ & \beta & \ddots & \\ & & \ddots & t_3 \\ & & & \beta \end{bmatrix}, \qquad C = \begin{bmatrix} 0 & \ldots & \ldots & 0 \\ \vdots & & \iddots & \vdots \\ 0 & 0 & & \vdots \\ t_3 & 0 & \ldots & 0 \end{bmatrix}. \tag{4.21}$$

Można teraz wprowadzić twierdzenie opisujące rozwiązaniu układu $R\mathbf{y} = \mathbf{z}$.

Twierdzenie 4.4. *Rozwiązaniem blokowej wersji układu $R\mathbf{y} = \mathbf{z}$ jest:*

$$\begin{cases} \mathbf{y}_{r-1} = R_s^{-1}\mathbf{z}_{r-1} \\ \mathbf{y}_i = R_s^{-1}\mathbf{z}_i - t_3 y_{(i+1)s} R_s^{-1}\mathbf{e}_{s-1}, \quad i = r-2, \ldots, 0. \end{cases} \tag{4.22}$$

Dowód. Po wymnożeniu $R\mathbf{y} = \mathbf{z}$, czyli

$$\begin{bmatrix} R_s & C & & \\ & R_s & \ddots & \\ & & \ddots & C \\ & & & R_s \end{bmatrix} \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{z}_0 \\ \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_{r-1} \end{bmatrix} \tag{4.23}$$

otrzymano:

$$\begin{cases} R_s\mathbf{y}_{r-1} = \mathbf{z}_{r-1} \\ R_s\mathbf{y}_i = \mathbf{z}_i - C\mathbf{y}_{i+1}, \quad i = r-2, \ldots, 0. \end{cases} \tag{4.24}$$

Upraszczając:

$$C\mathbf{y}_{i+1} = \begin{bmatrix} 0 & \ldots & \ldots & 0 \\ \vdots & & \iddots & \vdots \\ 0 & 0 & & \vdots \\ t_3 & 0 & \ldots & 0 \end{bmatrix} \begin{bmatrix} y_{(i+1)s} \\ \vdots \\ \vdots \\ y_{(i+2)s-1} \end{bmatrix} = t_3 y_{(i+1)s} \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 1 \end{bmatrix} = t_3 y_{(i+1)s}\mathbf{e}_{s-1}. \tag{4.25}$$

Podstawiając powyższe wyliczenie, a następnie mnożąc lewostronnie przez R_s^{-1} uzyskano układ z twierdzenia 4.4. □

Algorytm rozwiązywania układu $LR\mathbf{y} = \mathbf{d}$ bazujący na wzorach z twierdzeń 4.3 i 4.4 składa się z dwóch głównych etapów (A i B), z których każdy zawiera trzy kroki (1, 2, 3). Wszystkie wektory $L_s^{-1}\mathbf{f}_i$, $i = 0, \ldots, r-1$ mogą zostać znalezione równolegle (krok A1). Niech $F = [\mathbf{f}_0, \ldots, \mathbf{f}_{r-1}]$, wtedy wszystkie kolumny macierzy $X = L_s^{-1}F$ mogą zostać znalezione przy użyciu następującego wzoru:

$$X_{i,*} \leftarrow F_{i,*} - \alpha * X_{i-1,*}, \quad i = 1, \ldots, s-1, \tag{4.26}$$

gdzie $X_{i,*}$ oraz $F_{i,*}$ oznaczają i-ty wiersz X lub F, odpowiednio. Następnie mogą zostać znalezione ostatnie składowe każdego z wektorów $\mathbf{z}_i$, $i = 1, \ldots, r-1$ sekwencyjnie (krok A2), poprzez zastosowanie wzoru z twierdzenia 4.3. Ostatecznie, ponownie równolegle, można obliczyć $s-1$ pozostałych składowych wektorów $\mathbf{z}_1, \ldots, \mathbf{z}_{r-1}$ (krok A3).

W przypadku wzoru z twierdzenia 4.4 postępowanie wygląda analogicznie. Niech $Z = [\mathbf{z}_0, \ldots, \mathbf{z}_{r-1}]$, wtedy $Z \leftarrow R_s^{-1}Z$ znaleźć można - wykonując krok B1 - za pomocą wzoru:

$$Z_{i,*} \leftarrow (Z_{i,*} - t_3 * Z_{i+1,*})/\beta, \quad i = s-2, \ldots, 0. \tag{4.27}$$

Następnie podczas części sekwencyjnej (krok B2), program wyznacza pierwsze składowe każdego z wektorów $\mathbf{y}_i$, $i = r-2, \ldots, 0$. Na koniec, w kroku B3, korzystając ze wzoru z twierdzenia 4.4, wyznaczono równolegle wszystkie pozostałe składowe wektorów $Y = [\mathbf{y}_0, \ldots, \mathbf{y}_{r-1}]$.

Tak opisany sposób rozwiązania należy zastosować dwa razy: raz, aby znaleźć wektor $\mathbf{v}$ oraz drugi raz w celu znalezienia wektora $\mathbf{u}$. Jako ostatni krok zaproponowanego algorytmu należy wykorzystać twierdzenie 4.2, co pozwoli wyznaczyć rozwiązanie wzoru (4.1). Należy zauważyć, że wzór z twierdzenia 4.2 również może zostać w łatwy sposób zrównoleglony i zwektoryzowany.

4.2.2 Drugi algorytm - zrównoleglony algorytm Liu

Liu i in. w pracy [LLYZ20] przedstawili następujący sekwencyjny sposób rozwiązywania omawianego problemu. Dla uproszczenia założyć można, że $n = 2^k + 1$, $k \in \mathbb{N}$. Niech:

$$P = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ 1 & 0 & 0 & \ldots & 0 \end{bmatrix}. \tag{4.28}$$

Wtedy rozwiązanie (4.1) jest dokładnie takie same jak rozwiązanie $PT\mathbf{x} = P\mathbf{f}$, które po wymnożeniu ma następującą postać:

$$\left[\begin{array}{ccccc|c} t_1 & t_2 & t_3 & & & \\ & \ddots & \ddots & \ddots & & \\ & & t_1 & t_2 & t_3 & \\ & & & t_1 & t_2 & t_3 \\ & & & & t_1 & t_2 \\ \hline t_2 & t_3 & & & & 0 \end{array}\right] \left[\begin{array}{c} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_{n-2} \\ \hline x_{n-1} \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_{n-1} \\ \hline b_0 \end{array}\right], \tag{4.29}$$

oraz może zostać zapisane w równoważnej formie blokowej:

$$\left[\begin{array}{c|c} T_{11} & \mathbf{p} \\ \hline \mathbf{w}^T & 0 \end{array}\right] \left[\begin{array}{c} \mathbf{x}_0 \\ \hline x_{n-1} \end{array}\right] = \left[\begin{array}{c} \mathbf{b}_1 \\ \hline b_0 \end{array}\right] \tag{4.30}$$

lub za pomocą układu równań:

$$\begin{cases} T_{11}\mathbf{x}_0 + x_{n-1}\mathbf{p} = \mathbf{b}_1 \\ \mathbf{w}^T\mathbf{x}_0 = b_0 \end{cases}. \tag{4.31}$$

Poniższe twierdzenie wprowadza rozwiązanie układu z wzoru (4.29).

Twierdzenie 4.5. *Niech* $\mathbf{v}$ *oraz* $\mathbf{u}$ *spełniają odpowiednio następujące równości:*

$$T_{11}\mathbf{v} = \mathbf{b}_1 \quad \textit{oraz} \quad T_{11}\mathbf{u} = \mathbf{p}.$$

Przy powyższych oznaczeniach rozwiązanie układu (4.29) ma postać:

$$\begin{cases} \mathbf{x}_0 = \mathbf{v} - x_{n-1}\mathbf{u} \\ x_{n-1} = (t_2v_0 + t_3v_1 - b_0)/(t_2u_0 + t_3u_1). \end{cases} \tag{4.32}$$

Dowód. Pierwsze równanie z układu (4.31) może zostać przekształcone do następującej postaci:

$$\mathbf{x}_0 = T_{11}^{-1}\mathbf{b}_1 - x_{n-1}T_{11}^{-1}\mathbf{p}.$$

Co, przy oznaczeniach z twierdzenia 4.5, można zapisać równoważnie jako:

$$\mathbf{x}_0 = \mathbf{v} - x_{n-1}\mathbf{u}.$$

Następnie, tak wyznaczone $\mathbf{x}_0$ można wstawić do równania drugiego z (4.31), co da następujący rezultat:

$$\mathbf{w}^T(\mathbf{v} - x_{n-1}\mathbf{u}) = b_0.$$

Rozwijając:

$$\mathbf{w}^T\mathbf{v} - x_{n-1}\mathbf{w}^T\mathbf{u} = b_0$$

i wyznaczając x_{n-1} otrzymano:

$$x_{n-1} = \frac{\mathbf{w}^T\mathbf{v} - b_0}{\mathbf{w}^T\mathbf{u}}. \tag{4.33}$$

Można obliczyć, że:

$$\mathbf{w}^T\mathbf{v} = [t_2, t_3, 0, \ldots, 0] \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \end{bmatrix} = t_2v_0 + t_3v_1$$

oraz

$$\mathbf{w}^T\mathbf{u} = [t_2, t_3, 0, \ldots, 0] \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \end{bmatrix} = t_2u_0 + t_3u_1.$$

Podstawiając powyższe wyliczenia do wzoru (4.33) otrzymano:

$$x_{n-1} = \frac{t_2v_0 + t_3v_1 - b_0}{t_2u_0 + t_3u_1},$$

czyli drugie równanie ze wzoru (4.32). □

Aby móc skorzystać ze wzoru z twierdzenia 4.5, konieczne jest rozwiązanie dwóch układów równań o rozmiarze $n-1=2^k$, $k\in\mathbb{N}$, tzn. $T_{11}\mathbf{v}=\mathbf{b}_1$ oraz $T_{11}\mathbf{u}=\mathbf{p}$. Następnie można wyznaczyć x_{n-1} oraz, jako ostatni krok, równolegle wyznaczyć wszystkie pozostałe składowe $\mathbf{x}_0$.

Kolejnym zagadnieniem, niezbędnym do rozwiązania układu (4.29) jest rozwiązywanie układów postaci $T_{11}\mathbf{z}=\mathbf{f}$, gdzie $T_{11}\in\mathbb{R}^{(n-1)\times(n-1)}$, $n-1=2^k$, $k\in\mathbb{N}$, czyli układów następującej postaci:

$$\begin{bmatrix} t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & t_1 & t_2 & t_3 \\ & & & t_1 & t_2 \\ & & & & t_1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ \vdots \\ z_{n-2} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ \vdots \\ f_{n-2} \end{bmatrix}. \tag{4.34}$$

Po prostych przekształceniach można zapisać sekwencyjne rozwiązanie powyższego równania jako:

$$\begin{cases} z_{n-2} = f_{n-2}/t_1 \\ z_{n-3} = (f_{n-3} - t_2 z_{n-2})/t_1 \\ z_i = (f_i - t_2 z_{i+1} + t_3 z_{i+2})/t_1, \quad i = n-4, n-5, \ldots, 0. \end{cases} \tag{4.35}$$

Podobnie jak w rozdziale 3, tutaj także rozwiązanie sekwencyjne jako efektywne, może zostać zastosowane jedynie do niewielkich rozmiarów problemu. Wydajna implementacja rozpatrywanego problemu musi zakładać, że wektory $\mathbf{v}$ oraz $\mathbf{u}$ zostaną znalezione za pomocą równoległego algorytmu wykorzystującego budowę nowoczesnych komputerów. Możliwe jest sformułowanie algorytmu bardziej złożonego, lecz poddającego się w łatwy sposób zrównolegleniu [Stp92]. W tym celu wykorzystano postać blokową. Niech $(n-1)=rs$, $r,s>1$. Wtedy macierz T_{11} można podzielić następująco:

$$\left[\begin{array}{ccccc|ccccc|ccccc} t_1 & t_2 & t_3 & & & & & & & & & & & & \\ & \ddots & \ddots & \ddots & & & & & & & & & & & \\ & & t_1 & t_2 & t_3 & & & & & & & & & & \\ & & & t_1 & t_2 & t_3 & & & & & & & & & \\ & & & & t_1 & t_2 & t_3 & & & & & & & & \\ \hline & & & & & t_1 & t_2 & t_3 & & & & & & & \\ & & & & & & \ddots & \ddots & \ddots & & & & & & \\ & & & & & & & t_1 & t_2 & t_3 & & & & & \\ & & & & & & & & t_1 & t_2 & t_3 & & & & \\ & & & & & & & & & t_1 & t_2 & t_3 & & & \\ \hline & & & & & & & & & & t_1 & t_2 & t_3 & & \\ & & & & & & & & & & & \ddots & \ddots & \ddots & \\ & & & & & & & & & & & & t_1 & t_2 & t_3 \\ & & & & & & & & & & & & & t_1 & t_2 \\ & & & & & & & & & & & & & & t_1 \end{array}\right]. \tag{4.36}$$

Niech:

$$D = \begin{bmatrix} t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & t_1 & t_2 & t_3 \\ & & & t_1 & t_2 \\ & & & & t_1 \end{bmatrix}, \quad U = \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ t_3 & & & & \\ t_2 & t_3 & & & \end{bmatrix}, \tag{4.37}$$

oraz $\mathbf{z}_i = (z_{is}, z_{is+1}, \ldots, z_{(i+1)s-1})^T$, $\mathbf{d}_i = (d_{is}, d_{is+1}, \ldots, d_{(i+1)s-1})^T$ Przy powyższych oznaczeniach układ $T_{11}\mathbf{z} = \mathbf{f}$ może zostać zapisany w równoważny sposób w postaci blokowej jako:

$$\begin{bmatrix} D & U & & & \\ & D & U & & \\ & & \ddots & \ddots & \\ & & & D & U \\ & & & & D \end{bmatrix} \begin{bmatrix} \mathbf{z}_0 \\ \mathbf{z}_1 \\ \vdots \\ \vdots \\ \mathbf{z}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \vdots \\ \mathbf{d}_{r-1} \end{bmatrix}. \tag{4.38}$$

Po powyższym opisie można wprowadzić twierdzenie o rozwiązaniu układu $T_{11}\mathbf{z} = \mathbf{f}$.

Twierdzenie 4.6. *Rozwiązanie układu $T_{11}\mathbf{z} = \mathbf{f}$, gdzie $T_{11} \in \mathbb{R}^{(n-1)\times(n-1)}$, $n-1 = 2^k$, $k \in \mathbb{N}$, oraz $(n-1) = rs$, $r, s > 1$, zapisanego w postaci blokowej ma następującą postać:*

$$\begin{cases} \mathbf{z}_{r-1} = D^{-1}\mathbf{f}_{r-1} \\ \mathbf{z}_i = D^{-1}\mathbf{f}_i - (t_2 z_{(i+1)s} + t_3 z_{(i+1)s+1})\boldsymbol{y}_{r-1} - t_3 z_{(i+1)s}\boldsymbol{y}_{r-2}, \quad i = r-2, \ldots, 0. \end{cases} \tag{4.39}$$

Dowód. Po wymnożeniu układu (4.38) otrzymano:

$$\begin{cases} D\mathbf{z}_{r-1} = \mathbf{f}_{r-1} \\ D\mathbf{z}_i + U\mathbf{z}_{i+1} = D^{-1}\mathbf{f}_i, \quad i = r-2, \ldots, 0, \end{cases}$$

czyli:

$$\begin{cases} \mathbf{z}_{r-1} = D^{-1}\mathbf{f}_{r-1} \\ \mathbf{z}_i = D^{-1}\mathbf{f}_i - D^{-1}U\mathbf{z}_{i+1}, \quad i = r-2, \ldots, 0. \end{cases} \tag{4.40}$$

Niech

$$\mathbf{e}_k = (\underbrace{0, \ldots, 0}_{k}, 1, 0, \ldots, 0)^T \in \mathbb{R}^s,\ k = 0, \ldots, s-1.$$

Wtedy macierz U może zostać zapisana jako następująca suma:

$$U = t_2\mathbf{e}_{r-1}\mathbf{e}_0^T + t_3\mathbf{e}_{r-2}\mathbf{e}_0^T + t_3\mathbf{e}_{r-1}\mathbf{e}_1^T.$$

Dzięki powyższemu zapisowi, można dokonać kolejnego przekształcenia. Powyżej podano, że:

$$\mathbf{z}_i = (z_{is}, z_{is+1}, \ldots, z_{(i+1)s-1})^T.$$

Stąd wynika, że:

$$\mathbf{z}_{i+1} = (z_{(i+1)s}, z_{(i+1)s+1}, \ldots, z_{(i+2)s-1})^T.$$

Mając powyższe wyliczenia można wyznaczyć $U\mathbf{z}_{i+1}$:

$$U\mathbf{z}_{i+1} = (t_2\mathbf{e}_{r-1}\mathbf{e}_0^T + t_3\mathbf{e}_{r-2}\mathbf{e}_0^T + t_3\mathbf{e}_{r-1}\mathbf{e}_1^T)(z_{(i+1)s}, z_{(i+1)s+1}, \ldots, z_{(i+2)s-1})^T$$

i ostatecznie:

$$U\mathbf{z}_{i+1} = t_2 z_{(i+1)s}\mathbf{e}_{r-1} + t_3 z_{(i+1)s}\mathbf{e}_{r-2} + t_3 z_{(i+1)s+1}\mathbf{e}_{r-1}.$$

Wzór na $U\mathbf{z}_{i+1}$, otrzymany powyżej, można wstawić do równania (4.40). Wtedy:

$$\mathbf{z}_i = D^{-1}\mathbf{f}_i - D^{-1}(t_2 z_{(i+1)s}\mathbf{e}_{r-1} + t_3 z_{(i+1)s}\mathbf{e}_{r-2} + t_3 z_{(i+1)s+1}\mathbf{e}_{r-1})$$

i dalej:

$$\mathbf{z}_i = D^{-1}\mathbf{f}_i - t_2 z_{(i+1)s}D^{-1}\mathbf{e}_{r-1} - t_3 z_{(i+1)s}D^{-1}\mathbf{e}_{r-2} - t_3 z_{(i+1)s+1}D^{-1}\mathbf{e}_{r-1}. \tag{4.41}$$

Niech:

$$D\mathbf{y}_{r-1} = \mathbf{e}_{r-1} \quad \text{oraz} \quad D\mathbf{y}_{r-2} = \mathbf{e}_{r-2}.$$

Wtedy wzór (4.41) można zapisać jako:

$$\mathbf{z}_i = D^{-1}\mathbf{f}_i - t_2 z_{(i+1)s}\mathbf{y}_{r-1} - t_3 z_{(i+1)s}\mathbf{y}_{r-2} - t_3 z_{(i+1)s+1}\mathbf{y}_{r-1}$$

i w efekcie:

$$\mathbf{z}_i = D^{-1}\mathbf{f}_i - (t_2 z_{(i+1)s} + t_3 z_{(i+1)s+1})\mathbf{y}_{r-1} - t_3 z_{(i+1)s}\mathbf{y}_{r-2},$$

co jest drugim równaniem układu z twierdzenia (4.6). □

Należy dodać, że jeżeli $\mathbf{y}_{r-1} = (y_0, y_1, \ldots, y_{r-1})^T$, to $\mathbf{y}_{r-2} = (y_1, y_2, \ldots, y_{r-1}, 0)^T$. Ponadto, wszystkie wektory $D^{-1}\mathbf{f}_i$, $i = 0, \ldots, r-1$ mogą zostać znalezione równolegle. Aby unaocznić ten fakt, stworzono następującą macierz:

$$F = [\mathbf{f}_0, \ldots, \mathbf{f}_{r-1}].$$

Wtedy wszystkie kolumny $Z = D^{-1}F$ mogą zostać znalezione korzystając z następującego wzoru wektorowego:

$$Z_{i,*} \leftarrow (F_{i,*} - t_2 Z_{i+1,*} - t_3 Z_{i+2,*})/t_1, \quad i = s-3, \ldots, 0. \tag{4.42}$$

Następnie można sekwencyjnie wyznaczyć dwie pierwsze składowe dla każdego wektora $\mathbf{z}_i$, $i = r-2, \ldots, 0$, stosując wzór (4.39). Ostatecznie, znów równolegle, obliczyć można $s-2$ pozostałych składowych wektorów $\mathbf{z}_{r-2}, \ldots, \mathbf{z}_0$.

Algorytm ten powinien zostać zastosowany dwa razy: raz - by znaleźć wektor $\mathbf{v}$ oraz ponownie w celu znalezienie wektora $\mathbf{u}$. Ostatecznie, wzór (4.32) może zostać zastosowany w celu znalezienia rozwiązania wzoru (4.29) - takiego samego jak rozwiązanie wzoru (4.1). Należy zauważyć, że wzór (4.32) może w łatwy sposób zostać zrównoleglony i zwektoryzowany.

4.3 Implementacja na procesory graficzne

Algorytmy rozwiązujące układy postaci (4.1) zaprezentowane w sekcji 4.2 mogą zostać zaimplementowane z wykorzystaniem OpenACC, dzięki czemu będą mogły być wykonywane zarówno na akceleratorach (GPU) jak i na procesorach (CPU). OpenACC został szerzej omówiony w sekcji 1.2.3. Istnieje również możliwość użycia OpenACC i OpenMP w jednym programie w celu wykorzystania CPU i GPU w tym samym czasie [DS21] dla realizacji obliczeń w modelach hybrydowych.

4.3.1 Pierwszy algorytm - modyfikacja metody Wanga

Kroki równoległe A1, A3, B1, B3 algorytmu zaprezentowanego w sekcji 4.2.1 mogą zostać zwektoryzowane i zrównoleglone poprzez wykorzystanie dyrektywy `"pragma acc parallel loop"`. Oznacza to, że niezależne pętle mogą zostać rozłożone pomiędzy gangi (ang. *gangs*), które pracują w trybie SIMD i wykorzystują dostępną architekturę. Kroki A2 oraz B2 są sekwencyjne i powinny zostać wykonane przez jeden gang. Listing 4.2 przedstawia implementację kroków A1, A2, A3 z wykorzystaniem formatu kolumnowego dla macierzy $\mathbf{x}$ przechowywanej jako jednowymiarowa tablica danych podwójnej precyzji. Format taki jest najbardziej naturalnym, ponieważ nie występuje potrzeba modyfikacji ułożenia danych po wybraniu wartości dla parametrów r oraz s. Niestety format ten nie pozwala na wykorzystanie łączonego dostępu do pamięci globalnej urządzenia [CGM14] podczas kroków A1 oraz B1. Dlatego można oczekiwać, że wydajność programów używających takich formatów danych będzie znacząco gorsza w porównaniu do implementacji wykorzystującej format wierszowy (listing 4.3). W tym przypadku wszystkie odwołania do pamięci globalnej są połączone. Listing 4.1 pokazuje prostą zrównoleglona pętlę, która może zostać wykorzystana w celu zmiany pomiędzy formatami kolumnowym i wierszowym (i na odwrót). Niestety, pętla ta może znacząco wydłużyć działanie programu, ze względu na niepołączone dostępy do pamięci globalnej (ang. *non-coalesced memory access*) [SGMHS17].

```
// z formatu kolumnowego do wierszowego
#pragma acc parallel loop independent present(b,x)
    for(int i=0;i<n;i++)
        x[i]=b[i/r+(i%r)*s];
  ...
// z formatu wierszowego do kolumnowego
#pragma acc parallel loop independent present(b,x)
    for(int i=0;i<n;i++)
        b[i]=x[i/s+(i%s)*r];
```

Listing 4.1: Prosta konwersja tablicy w formacie kolumnowym na format wierszowy przy użyciu dyrektywy OpenACC `"pragma acc parallel loop"`

W celu zwiększenia wydajności implementacji, podjęto próbę znalezienia efektywniejszej metody zamiany pomiędzy formatami kolumnowym i wierszowym, która wykorzystuje pamięć podręczną, aby zapewnić łączony dostęp do pamięci globalnej (listing 4.4). Każdy gang składający się z $BSIZE$ wątków jest odpowiedzialny za odczytanie bloku $VL \times BSIZE$ elementów z tablicy w formacie kolumnowym wykorzystując łączony dostęp do pamięci i zapisanie tych elementów do pamięci podręcznej. Wtedy taki blok elementów przenoszony jest do nowej tablicy przechowywanej w formacie wierszowym (rysunek 4.2 i dokładniej na rysunku 4.1 oraz listing 4.4). Gang wątków działa na ciągu VL bloków, każdy po $VL \times NC$ elementów, gdzie $VL \times NC = BSIZE$. Każda kolumna, tak opisanego bloku jest ładowana (ang. *load*) przez VL wątków, gdzie VL jest rozmiarem warpu (ang. *warp*) [CGM14]. Gdy wszystkie bloki są w pamięci podręcznej wszystkie wątki wewnątrz gangu zapisują wiersze zawierające $BSIZE$ elementów do pamięci globalnej. Takie dostępy do globalnej pamięci są łączone w jak najmniej możliwych transakcji do wykonania, aby optymalizować przepustowość DRAM [CGM14]. Ostatecznie, należy zauważyć, że w obu przypadkach (w formacie kolumnowym i wier-

szowym) kroki B1, B2 oraz B3 zostały zaimplementowane analogicznie do swoich odpowiedników zaprezentowanych na listingach 4.2 oraz 4.3. Natomiast obliczenia z twierdzenia 4.2 zostały zaimplementowane przy użyciu dyrektywy `pragma acc parallel loop`.

```
1  // format kolumnowy
2  // Krok A1
3  #pragma acc parallel present(x)
4  {
5  #pragma acc loop  independent
6   for(int j=0;j<r;j++){
7      for(int i=1;i<s;i++)
8       x[j*s+i]-=x[j*s+i-1]*alpha;
9   }
10 }
11 // Krok A2
12 #pragma acc parallel num_gangs(1)
       present(x)
13 {
14  for(int j=1;j<r;j++)
15    x[(j+1)*s-1]-=x[j*s-1]*e0_last
          *alpha;
16 }
17 // Krok A3
18 #pragma acc parallel deviceptr
       (e0) present(x)
19 {
20 #pragma acc loop independent
21  for(int j=1;j<r;j++){
22     last1=x[j*s-1]*alpha;
23 #pragma acc loop  independent
24     for(int i=0;i<s-1;i++)
25       x[j*s+i]-=last1*e0[i];
26  }
27 }
```

Listing 4.2: Implementacja algorytmu pierwszego z wykorzystaniem OpenACC oparta na wzorze z twierdzenia 4.3 w formacie kolumnowym

```
// format wierszowy
// Krok A1
#pragma acc parallel present(x)
{
 for(int i=1;i<s;i++){
#pragma acc loop  independent
   for(int j=0;j<r;j++)
     x[i*r+j]-=x[(i-1)*r+j]*alpha;
 }
}
// Krok A2
#pragma acc parallel num_gangs(1)
   present(x)
{
 for(int j=1;j<r;j++)
   x[n-r+j]-=x[n-r+j-1]*e0_last*
      alpha;
}
// Krok A3
#pragma acc parallel deviceptr(e0)
   present(x)
{
#pragma acc loop independent
 for(int i=0;i<s-1;i++){
   last1=e0[i]*alpha;
#pragma acc loop independent
   for(int j=1;j<r;j++)
    x[i*r+j]-=x[(s-1)*r+j-1]*last1;
 }
}
```

Listing 4.3: Implementacja algorytmu pierwszego z wykorzystaniem OpenACC oparta na wzorze z twierdzenia 4.3 w formacie wierszowym

Inny możliwy sposób usprawnienia implementacji został przedstawiony na rysunku 4.3. Wykorzystuje on format kolumnowy, lecz zakłada, że blok $VL \times BSIZE$ elementów jest pobrany z pamięci globalnej do pamięci podręcznej, z wykorzystywaniem łączonego dostępu do pamięci. Oznacza to, że za pomocą pojedynczego dostępu do pamięci można pobrać wiele zmiennych, o ile są one pogrupowane [NVI15]. Wtedy wszystkie niezbędne obliczenia przeprowadzane są w pamięci podręcznej. Ostatecznie, taki blok po przeprowadzeniu obliczeń zapisywany jest z powrotem do pamięci globalnej. Należy zauważyć, że w pamięci podręcznej powinien zostać zaalokowany blok $(VL + 1) \times BSIZE$ elementów, ponieważ jest to niezbędne, aby móc odwoływać się do ostatniego rzędu poprzedniego bloku w przypadku wzoru z twierdzenia 4.3 oraz pierwszego rzędu poprzedniego bloku w przypadku wzoru z twierdzenia 4.4.

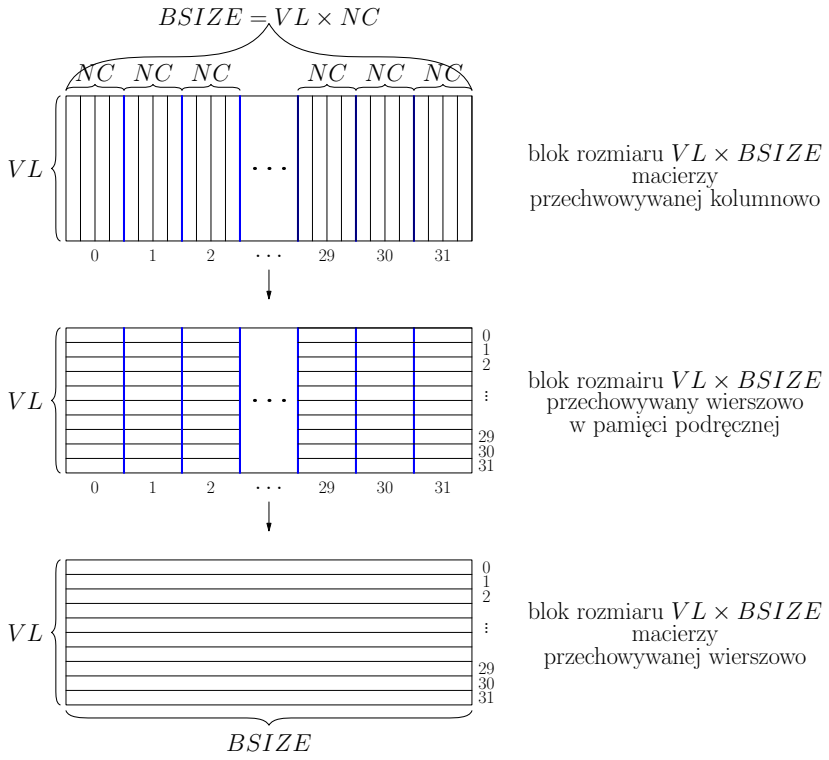
```
#pragma acc parallel  present(b) deviceptr(x) vector_length(BSIZE)
{
    float xc[VL][BSIZE];
#pragma acc cache(xc)

    for(int k=0;k<s;k+=VL){
// kazdy gang wczytuje blok rozmiaru VL x BSIZE
// z formatu kolumnowego do pamieci podrecznej
#pragma acc loop gang
        for(int j=0;j<r;j+=BSIZE){
#pragma acc loop seq
            for(int l=0;l<BSIZE;l+=NW)
#pragma acc loop vector
            for(int i=0;i<BSIZE;i++)
               xc[i%VL][l+i/VL]= x[(j+l+i/VL)*s+k+i%VL];
        }
// kazdy gang zapisuje blok rozmiaru VL x BSIZE
// z pamieci podrecznej do formatu wierszowego
#pragma acc loop gang vector
        for(int j=0;j<r;j++){
#pragma acc loop seq
           for(int i=0;i<VL;i++)
              b[(k+i)*r+j]=xc[i][j%BSIZE];
        }
    }
}
```

Listing 4.4: Implementacji OpenACC konwersji z formatu kolumnowego do wierszowego z wykorzystaniem pamięci podręcznej

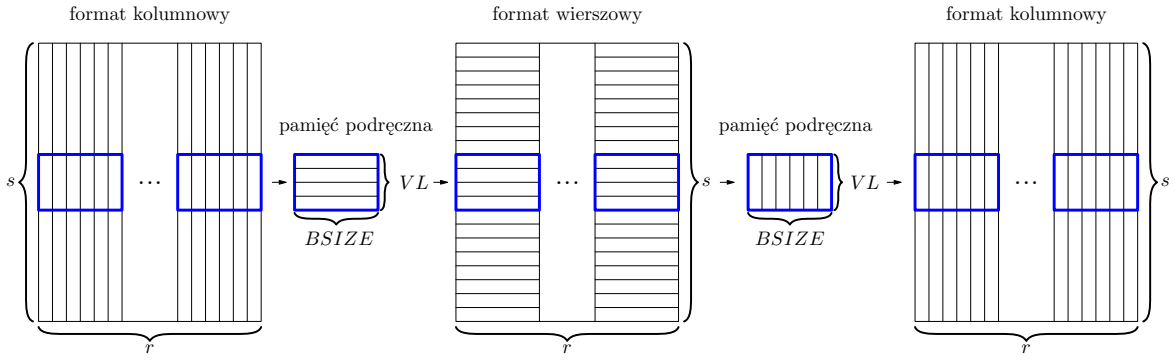
4.3.2 Drugi algorytm - zrównoleglony algorytm Liu

Rysunek 4.4 przedstawia fragment implementacji algorytmu wprowadzonego w sekcji 4.2.2. Implementacja ta zakłada wykorzystanie formatu kolumnowego. Inna możliwa modyfikacja wykorzystuje blok $(VL+2)\times BSIZE$ elementów zaalokowanych w pamięci podręcznej, ponieważ niezbędne są odwołania do dwóch górnych wierszy z poprzedniego bloku. Należy zauważyć, że w przypadku tego algorytmu, możliwa implementacja wykorzystująca format wierszowy osiąga znacznie gorsze wyniki czasowe, dlatego też nie została przedstawiona w niniejszej rozprawie.

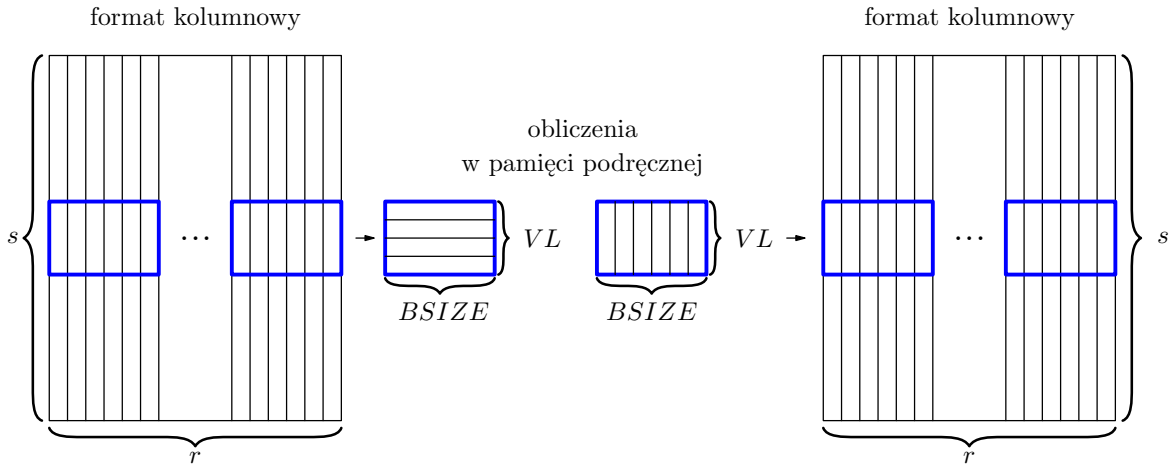
Należy dodać, że oba przedstawione algorytmy mają zbliżoną złożoność pamięciową do algorytmów z rozdziału 3. Dokładniej - algorytm pierwszy potrzebuje (w różnych wersjach formatowych) $4\cdot(int)+(2n+2s+13)\cdot(double)$, a algorytm drugi potrzebuje $4\cdot(int)+(2n+s+10)\cdot(double)$.



Rysunek 4.1: Zamiana bloku o rozmiarze $VL \times BSIZE$ przechowywanego kolumnowo do formatu wierszowego przeprowadzona przez pojedynczy gang wątków ($VL = 32$, $BSIZE = 256$)



Rysunek 4.2: Zmiana formatu kolumnowego na wierszowy i ponownie do kolumnowego z wykorzystaniem pamięci podręcznej



Rysunek 4.3: Wykorzystanie formatu kolumnowego wraz z blokiem pamięci podręcznej

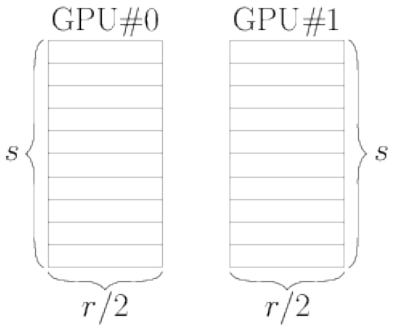
```
1  //format kolumnowy
2  #pragma acc parallel deviceptr(v)
3  {
4    #pragma acc loop independent
5    for(int j=r-1;j>=0;j--){
6      v[j*s+s-1]/=t1;
7      v[j*s+s-2]=(v[j*s+s-2]-t2*v[j*s+s-1])/t1;
8    }
9  }
10 #pragma acc parallel deviceptr(v)
11 {
12   #pragma acc loop independent
13   for(int j=r-1;j>=0;j--){
14       for(int i=s-3;i>=0;i--)
15          v[j*s+i]=(v[j*s+i]-t2*v[j*s+i+1]
16                    -t3*v[j*s+i+2])/t1;
17   }
18 }
19 #pragma acc parallel num_gangs(1)\
20                 deviceptr(v,y)
21 {
22    for(int j=r-2;j>=0;j--){
23       tmp1=t2*v[(j+1)*s]+t3*v[(j+1)*s+1];
24       tmp2=t3*v[(j+1)*s];
25       v[j*s]-=(tmp1*y[0]+tmp2*y[1]);
26       v[j*s+1]-=(tmp1*y[1]+tmp2*y[2]);
27    }
28 }
29 #pragma acc parallel deviceptr(v,y)
30 {
31    #pragma acc loop independent
32    for(int j=r-2;j>=0;j--){
33       tmp1=t2*v[(j+1)*s]+t3*v[(j+1)*s+1];
34       tmp2=t3*v[(j+1)*s];
35 #pragma acc loop independent
36       for(int i=2;i<s;i++)
37          v[j*s+i]-=(tmp1*y[i]+tmp2*y[i+1]);
38       }
39 }
```

```
//wykorzystanie pamięci podręcznej
#pragma acc parallel deviceptr(v)
{
    double vc[VL+2][BSIZE];
    #pragma acc cache(vc)

    #pragma acc loop gang
    for(int j=0;j<r;j+=BSIZE){
        #pragma avv loop vector
        for(int k=0;k<BSIZE;k++)
            vc[VL][k]=vc[VL+1][k]=0.0;
        for(int k=s-VL;k>=0;k-=VL){
            for(int l=0;l<BSIZE;l+=NC){
                #pragma acc loop vector
                for(int i=0;i<BSIZE;i++)
                    vc[i%VL][l+i/VL]=
                    v[(j+l+i/VL)*s+k+i%VL];
            }
            #pragma acc loop vector
            for(int j=0;j<BSIZE;j++){
                for(int i=VL-1;i>=0;i--)
                    vc[i][j]=(vc[i][j]
                      -t2*vc[i+1][j]
                      -t3*vc[i+2][j])/t1;
            }
            for(int l=0;l<BSIZE;l+=NC){
                #pragma acc loop vector
                for(int i=0;i<BSIZE;i++)
                v[(j+l+i/VL)*s+k+i%VL]=
                  vc[i%VL][l+i/VL];
            }
            #pragma acc loop vector
            for(int kk=0;kk<BSIZE;kk++){
                vc[VL][kk]=vc[0][kk];
                vc[VL+1][kk]=vc[1][kk];
            }
        }
    }
}
```

Rysunek 4.4: Implementacja algorytmu drugiego opartego na wzorze (4.39) z wykorzystaniem OpenACC w formacie kolumnowym (po lewej) oraz wykorzystującym pamięć podręczną do obliczenia pierwszego kroku algorytmu (po prawej) - równoważnie do lini 2-18 po lewej stronie

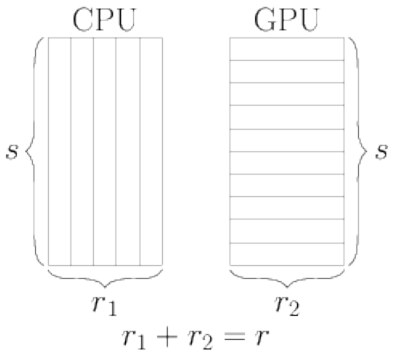
4.3.3 Implementacja na wiele urządzeń

Rozważane implementacje mogą zostać zmodyfikowane w sposób taki, aby wykorzystywały wiele urządzeń GPU [DS21]. Rysunek 4.5 pokazuje, w jaki sposób wykorzystać wspólnie OpenMP i OpenACC, aby prezentowany program mógł zostać uruchomiony na dwóch GPU. Dane wejściowe zostają podzielone na dwie równe części. Każda z połówek przechowywana jest w pamięci jednej z kart graficznych w formacie wierszowym. Utworzono dwa wątki OpenMP, każdy odpowiedzialny za kontrolę jednego GPU. Procesory graficzne dzielą dane poprzez pamięć hosta (ang. *host*). Tablica o nazwie `shared_data` zaalokowana jest w pamięci hosta, a każde z GPU ma kopię tej tablicy. Dyrektywa `update self` jest wykorzystana, aby zaktualizować dane na hoście podczas, gdy `update device` aktualizuje dane urządzenia (ang. *device*). Niezbędna jest również synchronizacja wątków za pomocą `omp barrier`.




```
#pragma omp parallel num_threads(2)
{ int dn=omp_get_thread_num();
  #pragma acc set device_num(dn)
  #pragma acc enter data create(shared_data[0:size])
  stepA1(...);
  if(dn==0){ // GPU#0
    stepA2(dn,shared_data,...);
    #pragma acc update self(shared_data[0:size])
  }
  #pragma omp barrier
  if(dn==1){ // GPU#1
    #pragma acc update device(shared_data[0:size])
    stepA2(dn,shared_data,...);
  }
  // dalsze kroki ...
```

Rysunek 4.5: Rozmieszczenie danych i ogólny schemat działania dla dwóch GPU




```
#pragma omp parallel num_threads(2)
{ int dn=omp_get_thread_num();
  if(dn==0){ // CPU
    stepA1_host(...); // wykorzystanie OpenMP
    stepA2_host(shared_data,...);
  }else{ // GPU
    #pragma acc enter data create(shared_data[0:size])
    stepA1_device(...); // wykorzystanie OpenACC
   }
  }
  #pragma omp barrier
  if(dn==1){ // GPU
    #pragma acc update device(shared_data[0:size])
    stepA2_device(shared_data,...);
  }
  // dalsze kroki ...
```

Rysunek 4.6: Hybrydowa implementacja dla hosta (OpenMP) i GPU (OpenACC)

Opisany powyżej sposób może również posłużyć do sformułowania implementacji wykorzystującej GPU oraz CPU w tym samym czasie. Rysunek 4.6 prezentuje ideę hybrydowej implementacji z wykorzystaniem OpenMP oraz OpenACC. CPU jest odpowiedzialne za znalezienie r_1 początkowych kolumn, natomiast GPU wykorzystano do znalezienie pozostałych r_2 kolumn, gdzie $r_1 + r_2 = r$. Należy zauważyć, że w tym przypadku, w OpenMP powinno zostać włączone zagnieżdżone zrównoleglenie [JRS16b]. W przypadku takiego wykorzystania lepszym rozwiązaniem jest wykorzystanie obsługi OpenACC dla pamięci ujednoliconej (ang. *unified memory*), która

upraszcza zarządzanie obszarami pamięci współdzielonymi pomiędzy procesorami graficznymi i procesorami. W programach skompilowanych przy użyciu kompilatora PGI z opcją `-ta=tesla:managed` wszystkie wywołania funkcji `malloc()` i `calloc()` są automatycznie zastępowane przez `cudaMallocManaged()`, która zwraca wskaźnik dostępny z poziomu CPU i GPU.

4.4 Wyniki

Wszystkie wyniki zostały zebrane na pięciu różnych architekturach z urządzeniami akcelerującymi pozwalającymi na wykorzystanie OpenMP oraz OpenACC. Są to:

I. **Multicore**

A. **Kepler**

B. **Volta**

C. **Ampere**

D. **Turing**

Architektury te zostały szerzej omówione w sekcji 1.3. Oznaczenia z sekcji 1.3 zostały zachowane.

Przetestowano wszystkie zaprezentowane implementacje oparte na OpenACC. Dla usystematyzowania poniżej wymieniono nazwy i krótkie opisy sześciu rozpatrywanych implementacji wraz ze skrótowcami używanymi w tabelach oraz na rysunkach przedstawianych w niniejszym rozdziale.

- Algorytm pierwszy:
  - format kolumnowy - **kol.**,
  - format wierszowy - **wiersz.**,
  - format wierszowy z zamianą formatów z wykorzystaniem pamięci podręcznej - **wiersz. cache**,
  - format kolumnowy z obliczeniami przeprowadzanymi z wykorzystaniem bloku pamięci podręcznej - **blok cache**.
- Algorytm drugi:
  - format kolumnowy - **kol.**,
  - format kolumnowy z obliczeniami przeprowadzanymi z wykorzystaniem bloku pamięci podręcznej - **blok cache**.

Należy teraz rozwinąć powyższy opis. Dla pierwszego algorytmu zaprezentowano cztery implementacje. Pierwsza wykorzystuje standardowy format kolumnowy - brak konieczności zmian formatu. Dwie kolejne działają w formacie wierszowym: wykorzystując prostą konwersję między formatami (pokazaną na listingu 4.1) oraz drugą - bardziej zaawansowaną zamianę wykorzystującą pamięć podręczną pokazaną na listingu

4.4 oraz na rysunku 4.2 i dokładniej na rysunku 4.1. Ostatnia funkcja dla algorytmu pierwszego korzysta z formatu kolumnowego, jednak obliczenia przeprowadzane są z wykorzystaniem bloku pamięci podręcznej (rysunek 4.3).

Dla drugiego algorytmu przygotowano analogiczne implementacje, jednak wersje wierszowe (również z wykorzystaniem pamięci podręcznej do zamiany formatów) dały zdecydowanie gorsze wyniki. Z tego powodu rozważono jedynie dwie wersje: ze standardowym formatem kolumnowy oraz formatem kolumnowym przeprowadzającym obliczenia z wykorzystaniem bloku pamięci podręcznej.

Eksperymenty przeprowadzono dla różnych rozmiarów problemów i różnych wartości parametrów r, s. Ze względu na ograniczenia pamięci w przypadku architektury Kepler oraz Turing maksymalny możliwy rozmiar badanego układu to $n = 2^{28}$. Z tego powodu, dla ujednolicenia, wyniki dla wszystkich architektur ograniczono do tej właśnie wartości. Należy jeszcze zwrócić uwagę, że rozmiary układów dla drugiego algorytmu są powiększone o jedność, tj. dla algorytmu drugiego do n prezentowanego w tabelach oraz na rysunkach należy dodać jeden.

Rozważano problem F) z sekcji 2.3, a dokładniej przypadek dla $b > 0$ i dokładnych parametrów $t_1 = -1 - c$, $t_2 = 2 + c$, $t_3 = -1$ oraz $c = 9$. Ten sam przypadek został również wybrany przez Liu i in. w pracy [LLYZ20] i oznaczony tam jako przykład drugi (ang. *Example 2*). Przykład taki zapewni prawidłowe wyniki dla obu algorytmów, ponieważ takie parametry spełniają następujące zależność $|t_2| = |t_1| + |t_3|$, a także $t_2^2 - 4t_1t_3 > 0$ oraz $|t_1| > |t_3|$.

Testy zostały przeprowadzone dla dwóch różnych prawych stron:

A. $\mathbf{f} = T\mathbf{x}^*$, gdzie $\mathbf{x}^* = \text{rand}(0, 1)$,

B. $\mathbf{f} = T\mathbf{e}$, gdzie $\mathbf{e} = (1, 0, \ldots, 0)$.

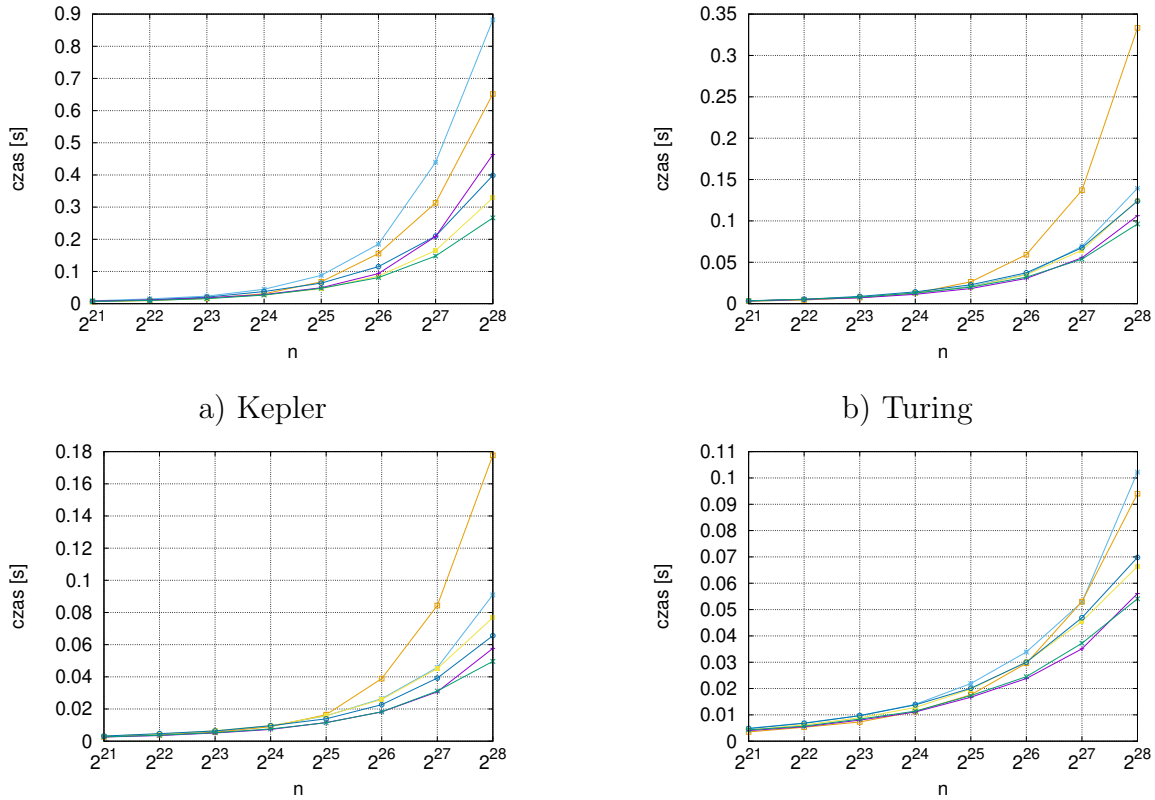
Jednak, po zebraniu wyników, nie zauważono znaczących rozbieżności w czasie trwania programów (w przeciwieństwie do wyników dokładności, co omówione zostanie w sekcji następnej - 4.4.2). Z tego powodu w niniejszej sekcji zaprezentowano jedynie wyniki dla prawej strony B.

4.4.1 Czas działania

Tabela 4.1 prezentuje wyniki czasowe otrzymane dla wszystkich rozważanych metod i architektur dla najlepszej wartości parametru r. Dla lepszego uwidocznienia różnic, wyniki zawarte w tabeli 4.1 zostały również przedstawione na wykresach przedstawionych na rysunku 4.7. Dla pierwszego algorytmu, w większości przypadków testowanych na GPU, implementacje używające formatu wierszowego uzyskiwały dużo lepszą wydajność niż implementacje z formatem kolumnowym. Ponadto, można zwrócić uwagę, że wykonywanie obliczeń w bloku pamięci podręcznej jest zawsze lepsze niż wykorzystanie zwykłego formatu kolumnowego, a dla niektórych GPU lepsze niż obliczenia w formacie wierszowym wykorzystującym pamięć podręczną. Wykorzystanie pamięci podręcznej do konwersji pomiędzy formatami kolumnowym i wierszowym zwiększa wydajność, zwłaszcza na karcie graficznej Kepler. W tym przypadku wersja z formatem wierszowym oraz pamięcią podręczną jest o około 50% szybsza niż format wierszowy bez użycia pamięci podręcznej dla największych rozważanych rozmiarów układów. Ponadto, dla tej architektury format kolumnowy z obliczeniami w pamięci podręcznej jest lepszy niż zwykły format kolumnowy. Karty graficzne Turing i Volta dają zyski z użycia pamięci podręcznej jedynie dla większych rozmiarów układów. Przy największych

Tabela 4.1: Czas działania [s] wszystkich rozważanych implementacji na różnych architekturach. Zaprezentowany parametr r pozwolił osiągnąć najlepszy wynik czasowy

	n	Algorytm Pierwszy								Algorytm Drugi			
		kol.		wiersz.		wiersz. cache		blok cache		kol.		blok cache	
		r	czas	r	czas	r	czas	r	czas	r	czas	r	czas
	2^{20}	2^{11}	0,0061	2^{11}	**0,0042**	2^{11}	0,0052	2^{11}	0,0052	2^{10}	0,0055	2^{10}	**0,0050**
K	2^{21}	2^{11}	0,0090	2^{12}	**0,0061**	2^{11}	0,0073	2^{11}	0,0073	2^{11}	0,0080	2^{10}	**0,0072**
E	2^{22}	2^{12}	0,0149	2^{12}	**0,0095**	2^{12}	0,0107	2^{11}	0,0115	2^{11}	0,0114	2^{11}	**0,0100**
P	2^{23}	2^{12}	0,0234	2^{13}	**0,0164**	2^{12}	0,0164	2^{11}	0,0202	2^{12}	0,0181	2^{11}	**0,0154**
L	2^{24}	2^{12}	0,0448	2^{13}	0,0305	2^{13}	**0,0273**	2^{11}	0,0376	2^{12}	0,0283	2^{11}	**0,0266**
E	2^{25}	2^{12}	0,0878	2^{14}	0,0680	2^{13}	**0,0466**	2^{13}	0,0638	2^{12}	0,0495	2^{12}	**0,0479**
R	2^{26}	2^{12}	0,1850	2^{14}	0,1558	2^{13}	**0,0858**	2^{14}	0,1151	2^{12}	0,0934	2^{13}	**0,0810**
	2^{27}	2^{15}	0,4393	2^{14}	0,3133	2^{14}	**0,1652**	2^{14}	0,2098	2^{12}	0,2089	2^{13}	**0,1479**
	2^{28}	2^{16}	0,8819	2^{14}	0,6516	2^{14}	**0,3286**	2^{15}	0,3985	2^{15}	0,4633	2^{14}	**0,2665**
	2^{20}	2^{10}	0,0021	2^{11}	**0,0020**	2^{11}	0,0024	2^{11}	0,0025	2^{10}	**0,0025**	2^{11}	0,0026
T	2^{21}	2^{11}	0,0030	2^{11}	**0,0030**	2^{11}	0,0034	2^{11}	0,0034	2^{11}	**0,0033**	2^{11}	0,0035
U	2^{22}	2^{11}	0,0045	2^{12}	**0,0044**	2^{12}	0,0051	2^{11}	0,0052	2^{11}	**0,0051**	2^{12}	0,0054
R	2^{23}	2^{12}	0,0074	2^{12}	**0,0074**	2^{12}	0,0078	2^{11}	0,0088	2^{12}	**0,0071**	2^{12}	0,0078
I	2^{24}	2^{12}	**0,0122**	2^{13}	0,0127	2^{13}	0,0127	2^{12}	0,0143	2^{12}	**0,0113**	2^{12}	0,0129
N	2^{25}	2^{12}	0,0209	2^{13}	0,0264	2^{13}	**0,0201**	2^{12}	0,0228	2^{12}	**0,0183**	2^{12}	0,0200
G	2^{26}	2^{12}	0,0356	2^{15}	0,0592	2^{14}	**0,0350**	2^{13}	0,0373	2^{13}	**0,0305**	2^{13}	0,0320
	2^{27}	2^{12}	0,0692	2^{16}	0,1371	2^{14}	**0,0645**	2^{13}	0,0676	2^{13}	0,0554	2^{14}	**0,0538**
	2^{28}	2^{13}	0,1394	2^{17}	0,3333	2^{14}	0,1249	2^{14}	**0,1239**	2^{13}	0,1063	2^{14}	**0,0963**
	2^{20}	2^{10}	0,0024	2^{11}	**0,0020**	2^{11}	0,0023	2^{10}	0,0025	2^{10}	**0,0019**	2^{10}	0,0020
	2^{21}	2^{11}	0,0031	2^{12}	**0,0027**	2^{12}	0,0032	2^{11}	0,0032	2^{11}	**0,0026**	2^{11}	0,0027
V	2^{22}	2^{11}	0,0044	2^{12}	**0,0039**	2^{12}	0,0044	2^{11}	0,0047	2^{11}	**0,0035**	2^{11}	0,0037
O	2^{23}	2^{12}	0,0062	2^{13}	**0,0056**	2^{13}	0,0065	2^{12}	0,0064	2^{12}	**0,0051**	2^{12}	0,0053
L	2^{24}	2^{12}	0,0095	2^{13}	**0,0090**	2^{13}	0,0096	2^{12}	0,0096	2^{12}	**0,0073**	2^{12}	0,0076
T	2^{25}	2^{12}	0,0158	2^{13}	0,0165	2^{13}	0,0159	2^{13}	**0,0139**	2^{12}	0,0116	2^{13}	**0,0115**
A	2^{26}	2^{13}	0,0264	2^{14}	0,0389	2^{14}	0,0259	2^{13}	**0,0227**	2^{13}	**0,0182**	2^{13}	0,0183
	2^{27}	2^{13}	0,0458	2^{16}	0,0843	2^{15}	0,0451	2^{13}	**0,0392**	2^{13}	**0,0307**	2^{14}	0,0312
	2^{28}	2^{13}	0,0910	2^{16}	0,1777	2^{15}	0,0768	2^{14}	**0,0656**	2^{14}	0,0577	2^{14}	**0,0496**
	2^{20}	2^{10}	0,0036	2^{11}	**0,0023**	2^{11}	0,0027	2^{10}	0,0035	2^{10}	**0,0028**	2^{10}	0,0030
A	2^{21}	2^{11}	0,0049	2^{12}	**0,0035**	2^{12}	0,0043	2^{11}	0,0048	2^{10}	**0,0040**	2^{10}	0,0043
M	2^{22}	2^{11}	0,0069	2^{12}	**0,0053**	2^{12}	0,0062	2^{11}	0,0068	2^{11}	**0,0055**	2^{11}	0,0057
P	2^{23}	2^{12}	0,0097	2^{13}	**0,0073**	2^{13}	0,0089	2^{12}	0,0097	2^{11}	**0,0080**	2^{12}	0,0084
E	2^{24}	2^{12}	0,0140	2^{13}	**0,0113**	2^{13}	0,0128	2^{12}	0,0138	2^{12}	**0,0111**	2^{12}	0,0114
R	2^{25}	2^{13}	0,0219	2^{14}	**0,0176**	2^{14}	0,0198	2^{13}	0,0201	2^{12}	**0,0167**	2^{13}	0,0173
E	2^{26}	2^{13}	0,0339	2^{14}	**0,0296**	2^{14}	0,0301	2^{13}	0,0300	2^{13}	**0,0238**	2^{13}	0,0245
	2^{27}	2^{13}	0,0528	2^{14}	0,0530	2^{15}	**0,0455**	2^{13}	0,0469	2^{13}	**0,0351**	2^{13}	0,0372
	2^{28}	2^{13}	0,1022	2^{15}	0,0940	2^{15}	**0,0664**	2^{13}	0,0698	2^{14}	0,0561	2^{14}	**0,0541**



a) Kepler

b) Turing

c) Volta

d) Ampere

Rysunek 4.7: Wykresy prezentujące czasy działania [s] wszystkich rozważanych implementacji z podziałem na wykorzystane karty graficzne

rozważanych układach najkrótszy czas obliczeń osiągano z wykorzystaniem pamięci podręcznej. Należy dodać, że wykonanie drugiego algorytmu zawsze będzie szybsze, ponieważ algorytm pierwszy wymaga więcej operacji niż algorytm drugi.

Tabela 4.2: Najlepsze wyniki czasowe [s] dla algorytmu sekwencyjnego, implementacji skompilowanych na systemy bez urządzeń akcelerujących - **multicore**, funkcji `dgtsv` z biblioteki LAPACK oraz wyniki dla wszystkich GPU dla najlepszej funkcji (dla pierwszego i drugiego algorytmu). Wszystkie wyniki uwzględniają transfer danych

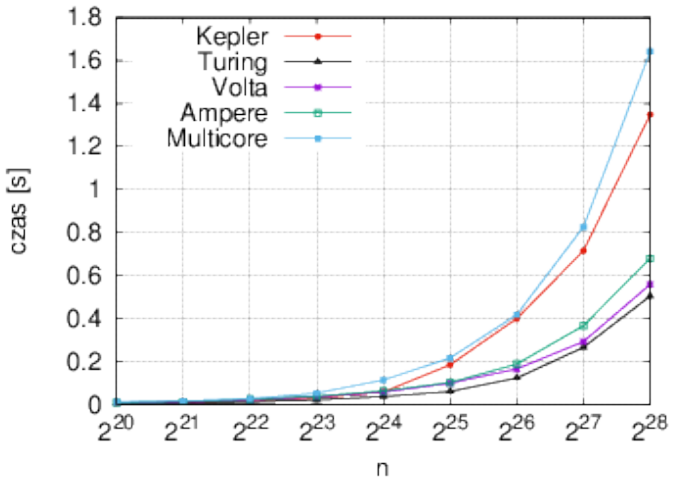
n	Kepler		Turing		Volta		Ampere		Multicore		`dgtsv`
	copy	managed	copy	managed	copy	managed	copy	managed			
	alg1 wiersz. cache		alg1 blok cache		alg1 blok cache		alg1 blok cache		alg1 sek.	alg1 kol.	
2^{20}	0,0443	**0,0074**	0,0177	**0,0042**	0,0411	**0,0079**	0,0385	**0,0088**	0,0347	0,0120	0,0343
2^{21}	0,0518	**0,0115**	0,0204	**0,0075**	0,0451	**0,0121**	0,0458	**0,0141**	0,0711	0,0165	0,0671
2^{22}	0,0631	**0,0188**	0,0255	**0,0115**	0,0549	**0,0208**	0,0531	**0,0238**	0,1455	0,0267	0,1328
2^{23}	0,0866	**0,0319**	0,0358	**0,0211**	0,0710	**0,0355**	0,0739	**0,0385**	0,2899	0,0531	0,2636
2^{24}	0,1320	**0,0580**	0,0550	**0,0366**	0,1006	**0,0571**	0,1062	**0,0629**	0,5767	0,1134	0,5229
2^{25}	0,2153	**0,1843**	0,0898	**0,0604**	0,1591	**0,0987**	0,1684	**0,1022**	1,1540	0,2161	1,0423
2^{26}	**0,3759**	0,3996	0,1575	**0,1229**	0,2752	**0,1652**	0,2861	**0,1879**	2,3063	0,4176	2,0837
2^{27}	0,7457	**0,7150**	0,2954	**0,2655**	0,4618	**0,2923**	0,5051	**0,3663**	4,6085	0,8257	4,1613
2^{28}	1,4211	**1,3490**	0,5665	**0,5039**	0,9357	**0,5582**	0,9189	**0,6799**	9,2176	1,6417	8,3316
n	alg2 blok cache		alg2 blok cache		alg2 kol.		alg2 kol.		alg2 sek	alg2 kol.	
2^{20}	0,0442	**0,0069**	0,0178	**0,0047**	0,0405	**0,0089**	0,0370	**0,0106**	0,0291	0,0103	
2^{21}	0,0511	**0,0110**	0,0206	**0,0082**	0,0431	**0,0155**	0,0451	**0,0174**	0,0583	0,0119	
2^{22}	0,0626	**0,0173**	0,0259	**0,0158**	0,0533	**0,0263**	0,0554	**0,0305**	0,1206	0,0155	
2^{23}	0,0855	**0,0302**	0,0348	**0,0292**	0,0688	**0,0444**	0,0709	**0,0546**	0,2408	0,0251	
2^{24}	0,1295	**0,0564**	0,0536	**0,0497**	0,0980	**0,0698**	0,1021	**0,0848**	0,4808	0,0459	
2^{25}	0,2078	**0,1848**	**0,0873**	0,1041	0,1567	**0,1284**	0,1587	**0,1362**	0,9618	0,0826	
2^{26}	**0,3825**	0,3924	**0,1523**	0,1951	0,2793	**0,2207**	0,2811	**0,2442**	1,9187	0,1543	
2^{27}	0,7045	**0,6960**	**0,2827**	0,3349	0,4565	**0,3977**	0,5020	**0,4331**	3,8326	0,2976	
2^{28}	1,3788	**1,2862**	**0,5408**	0,6783	0,8161	**0,7394**	0,9106	**0,8024**	7,6694	0,5812	

Do wyników zaprezentowanych w tabeli 4.1 nie został wliczony czas potrzebny do kopiowania danych pomiędzy hostem i GPU. Rozwiązywanie układów postaci (4.1) na ogół jest częścią większego problemu, a to może wiązać się z koniecznością przeprowadzenia dalszych obliczeń na GPU. Pomimo tego, przeprowadzono również testy, w których zmierzono czas potrzebny do transferu danych z hosta na GPU oraz z powrotem. Rozważono dwie możliwe opcje: jawny transfer danych za pomocą dyrektywy `pragma acc data copy` (tabela 4.2 "copy") oraz niejawny transfer danych wykorzystujący zarządzany transfer danych włączony przez `-ta=tesla:managed` (tabela 4.2 "managed"). Zebrane dane wskazują, że dla prawie wszystkich przypadków użycie pamięci zarządzanej jest wydajniejsze.

Tabela 4.2 przedstawia również wyniki dla implementacji korzystających z OpenACC skompilowanych z opcją `multicore` (tzn. dla systemów bez urządzeń akcelerujących) oraz dla implementacji sekwencyjnej, czyli algorytmu Thomasa (sekcja 2.4.1). Po zebraniu wszystkich wyników zaobserwowano, że dla opcji `multicore` format kolumnowy jest najlepszy, dlatego zaprezentowano wyniki jedynie dla tego formatu. W tym przypadku nie ma konieczności zamiany formatu na wierszowy, dlatego konwersja taka powinna zostać przeprowadzono tylko w przypadku, gdy jest dostępne urządzenie akcelerujące. Fakt ten można łatwo sprawdzić poprzez skorzystanie z instrukcji `acc_get_num_devices()` z biblioteki Runtime z OpenACC. Oznacza to, że prezentowane kody źródłowe mogą zostać zastosowane zarówno na architekturach z GPU lub bez GPU bez konieczności jakichkolwiek zmian.

Dla lepszego zobrazowania opisanych wyników, najlepsze implementacje dla każdej architektury zostały umieszczone na rysunku 4.8. Wybrane implementacje to format

kolumnowy dla architektury Multicore, format wierszowy z wykorzystaniem pamięci podręcznej dla karty graficznej Kepler oraz obliczenia w bloku pamięci podręcznej dla pozostałych GPU.



Rysunek 4.8: Czas [s] działania z uwzględnieniem ewentualnego kopiowania danych na/z GPU

W tabeli 4.2 zawarto czas dla funkcji `dgtsv` z biblioteki LAPACK (omówienie biblioteki LAPACK dostępne w sekcji 2.5.1). Zdecydowano się na porównanie z tą właśnie funkcję, ponieważ - jak opisano w sekcji 1.3 - ogólnodostępne narzędzia nie oferują funkcji dla układów z macierzą trójdiagonalną typu Toeplitza. Ponadto, ciekawą opcją jest też porównanie dokładności zaproponowanych implementacji z dokładnością uzyskaną przez funkcję biblioteczną, które zostało opisane w sekcji 4.4.2.

Zaproponowana implementacja skompilowana na systemy bez urządzeń akcelerujących (tzn. z opcją `multicore`) osiągają bardzo dobre przyspieszenie względem algorytmu sekwencyjnego - do 5,6 dla algorytmu pierwszego i do 13,2 dla algorytmu drugiego. Jest ona również o około 50% szybsza niż implementacja z wykorzystaniem OpenMP podobnego problemu - zaprezentowana w rozdziale 3. Jednak, należy zauważyć, że wyniki dla wszystkich zaprezentowanych GPU są znacząco lepsze niż te osiągnięte dla `multicore`. Dlatego, wykorzystanie hybrydowej implementacji może być korzystne jedynie wtedy, gdy pamięć oferowana przez GPU jest zbyt mała na potrzeby rozwiązania danego problemu. Wyniki dla takiej sytuacji przedstawia tabela 4.3. Zaprezentowano tam implementację uruchomioną na CPU + GPU (architektury I (Multicore) + A (Kepler)). Implementacja powstała zgodnie z opisem przedstawionym na rysunku 4.6. Wykorzystano format wierszowy z użyciem pamięci podręcznej - najlepszy dla karty graficznej Kepler.

Tabela 4.3: Najlepsze wyniki czasowe [s] dla implementacji hybrydowej w formacie wierszowym z użyciem pamięci podręcznej

n	CPU + GPU	
	r	czas
2^{29}	2^{16}	1,2689
2^{30}	2^{16}	2,7807

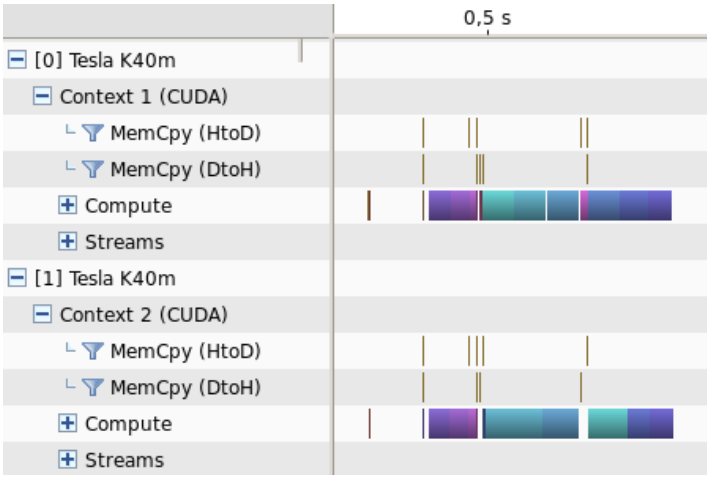
Zaimplementowano i przetestowano również opcję przedstawioną na rysunku 4.5 dla układu 2×A (2×Kepler), a osiągnięte wyniki przedstawia tabela 4.4. Porównując

dane z tabel 4.4 oraz 4.1 zauważyć można, że wyniki dla dwóch procesorów graficznych skalują się bardzo dobrze - są na ogół o 50% lepsze niż wyniki dla jednego procesora graficznego dla odpowiadającego rozmiaru układu.

Tabela 4.4: Najlepsze wyniki czasowe [s] dla implementacji na 2×GPU w formacie wierszowym z użyciem pamięci podręcznej

n	2 x Kepler	
	r	czas
2^{15}	2^{8}	0,0013
2^{16}	2^{9}	0,0016
2^{17}	2^{10}	0,0025
2^{18}	2^{10}	0,0031
2^{19}	2^{10}	0,0044
2^{20}	2^{11}	0,0056
2^{21}	2^{12}	0,0082
2^{22}	2^{12}	0,0112
2^{23}	2^{12}	0,0173
2^{24}	2^{13}	0,0241
2^{25}	2^{13}	0,0384
2^{26}	2^{14}	0,0607
2^{27}	2^{14}	0,1029
2^{28}	2^{15}	0,1856
2^{29}	2^{15}	0,3491

W przypadku implementacji działającej na dwóch, kartach graficznych nie zauważono również znaczących kosztów ogólnej wydajności, co można zaobserwować na rysunku 4.9, który przedstawia wyniki z programu NVIDIA Visual Profiler.



Rysunek 4.9: Wyniki z programu NVIDIA Visual Profiler dla implementacji na dwa GPU

Policzono również wartości GFLOPS (opis w sekcji 1.5) dla prezentowanych danych, a poniżej opisano wnioski z otrzymanych obliczeń. Pierwszy algorytm skompilowany z opcją `multicore` osiąga wydajność do 4.2 GFLOPS (format kolumnowy), a na GPU 95.9 GFLOPS (karta graficzna Volta) (blok z wykorzystaniem pamięci podręcznej).

Natomiast drugi algorytm - 11.7 GFLOPS na `multicore` (format kolumnowy) oraz 126.8 GFLOPS na GPU (również karta graficzna Volta) (blok z wykorzystaniem pamięci podręcznej). Osiągnięte wyniki są dalekie od maksymalnej wydajności (ang. *peak performance*) dla tych architektur, co jest normalnym zjawiskiem w problemach, gdzie stosunek odwołań do pamięci i liczby operacji arytmetycznych jest rzędu $O(1)$. Należy zaznaczyć, że wydajność na wszystkich rozważanych procesorach graficznych znacząco przewyższa wydajność osiągniętą na `multicore`, nawet jeżeli uwzględniony zostanie czas potrzebny na transfer danych. Oznacza to, że użycie GPU jest korzystniejsze nawet w przypadkach jak opisane, tzn. z małą intensywnością obliczeń. Ponadto, lepsza wydajność w implementacjach z OpenACC może zostać osiągnięta w prosty sposób, jeżeli wykorzystana zostanie wektoryzacja.

4.4.2 Błąd względny

Błąd względny został wyliczony zgodnie z wzorem (1.4) podanym w sekcji 1.5. Parametry r i s wpływają na wyniki czasowe w większym stopniu niż na otrzymywaną dokładność. Zmiana pozostałych parametrów nie wpływa na zaprezentowane wcześniej wyniki czasowe, lecz można zaobserwować ich wpływ na otrzymaną dokładności. Błąd różnicuje się dla różnych wartości prawych stron oraz różnych współczynników macierzy, tj. t_1, t_2, t_3. Z tego względu przeprowadzono i zaprezentowano wyniki dla dwóch różnych wymienionych już prawych stron układ. Dla przypomnienia:

A. $\mathbf{f} = T\mathbf{x}^*$, gdzie $\mathbf{x}^* = \text{rand}(0, 1)$,

B. $\mathbf{f} = T\mathbf{e}$, gdzie $\mathbf{e} = (1, 0, \ldots, 0)$.

Uzyskane wyniki dla prawej strony A zostały przedstawione w tabeli 4.5, a dla prawej strony B w tabeli 4.6. Tabele te przedstawiają błąd względny wyliczony dla wszystkich rozważanych implementacji równoległych oraz sekwencyjnych. Dodatkowo przedstawiono również wynik uzyskany dla funkcji `dgtsv` z biblioteki LAPACK przedstawionej w sekcji 2.5.1.

Tabela 4.5: Błąd względny wszystkich rozważanych implementacji dla najlepszych wartości r oraz prawej strony A

		Algorytm Pierwszy					**Algorytm Drugi**		
n	dgtsv	**sek.**	**kol**	**wiersz.**	**wiersz. cache**	**blok cache**	**sek.**	**kol.**	**blok cache**
2^{20}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{21}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	2e-16	2e-16	2e-16
2^{22}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{23}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{24}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{25}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	2e-16	2e-16	2e-16
2^{26}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{27}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16
2^{28}	2e-16	2e-16	2e-16	2e-16	1e-16	2e-16	1e-16	2e-16	2e-16

Z tabeli 4.5 można zaobserwować, że wyniki dla prawej strony A są tak dokładne jak pozwala nam precyzja przeprowadzanych obliczeń (podwójna precyzja). Natomiast w tabeli 4.5 można zauważyć lekki spadek dokładności dla większych układów. Spadek ten jest jednak akceptowalny, podobny dla wszystkich przedstawionych funkcji - włącznie z funkcją z biblioteki LAPACK.

Tabela 4.6: Błąd względny wszystkich rozważanych implementacji dla najlepszych wartości r oraz prawej strony B

		Algorytm Pierwszy					**Algorytm Drugi**		
n	dgtsv	**sek.**	**kol**	**wiersz.**	**wiersz. cache**	**blok cache**	**sek.**	**kol.**	**blok cache**
2^{20}	4e-14	4e-14	6e-14	4e-14	4e-14	6e-14	4e-14	6e-14	6e-14
2^{21}	6e-14	6e-14	9e-14	6e-14	6e-14	9e-14	6e-14	9e-14	8e-14
2^{22}	8e-14	8e-14	1e-13	9e-14	9e-14	1e-13	8e-14	1e-13	1e-13
2^{23}	1e-13	1e-13	2e-13	1e-13	1e-13	2e-13	1e-13	2e-13	2e-13
2^{24}	2e-13	2e-13	3e-13	2e-13	2e-13	3e-13	2e-13	2e-13	2e-13
2^{25}	2e-13	2e-13	4e-13	2e-13	2e-13	4e-13	2e-13	4e-13	3e-13
2^{26}	3e-13	3e-13	5e-13	3e-13	3e-13	5e-13	3e-13	5e-13	5e-13
2^{27}	5e-13	5e-13	7e-13	5e-13	5e-13	7e-13	5e-13	7e-13	7e-13
2^{28}	7e-13	7e-13	1e-12	7e-13	7e-13	1e-12	7e-13	1e-12	9e-13

4.4.3 Predykcja parametru r

Jak wspomniano wcześniej, dobór parametru r, gdzie $n = rs$ wpływa na otrzymywane wyniki, zarówno czasowe jak i dokładność. W celu zebrania optymalnych wyników przeprowadzono testy dla wielu różnych parametrów r, s. Część tych wyników została przedstawiona na rysunku 4.10 - dla $n = 2^{22}$ oraz na rysunku 4.11 dla $n = 2^{28}$. Rysunki te pokazują zmiany czasu wykonania programu związane ze zmianą parametru r. Można zaobserwować, że dla wszystkich rozważanych funkcji i wszystkich rozważanych architektur, najlepsze r jest takie same lub bardzo bliskie.

Szukanie optymalnych parametrów krok po kroku jest uciążliwe z perspektywy użytkownika takiego programu, który chciałby otrzymać gotowy program, bez potrzeby wyszukiwania lepszych parametrów. Dlatego podjęto próbę zautomatyzowania wyznaczania parametrów programu.

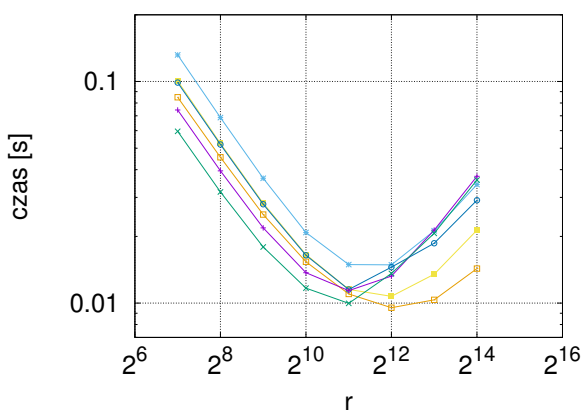
Największe różnice pomiędzy funkcjami można zauważyć w przypadku największych rozważanych układów (patrz rysunek 4.11, gdzie $n = 2^{28}$). Metody odpowiadające sobie mają podobne wykresy, np. format wierszowy dla algorytmu pierwszego i drugiego, użycie bloku pamięci podręcznej dla algorytmu pierwszego i drugiego. Ponadto, dla architektury `multicore` zmiany w czasie trwania funkcji wywołane zmianą parametru r są w większości przypadków nieznaczne. Efekt ten można zaobserwować zwłaszcza dla najbardziej wydajnego formatu dla tej architektury, tj. dla formatu kolumnowego, gdzie wykresy dla funkcji w tych właśnie formatach są niemal stałe zarówno na rysunku 4.10e jak i na rysunku 4.11e.

Biorąc pod uwagę otrzymane wyniki, zaprezentowane w tabeli 4.1 oraz na rysunkach 4.10, 4.11 można podjąć próbę predykcji parametru r. Bazując na wartościach r zaprezentowanych w tabeli 4.1 utworzono funkcję trendu, która wygląda następująco:

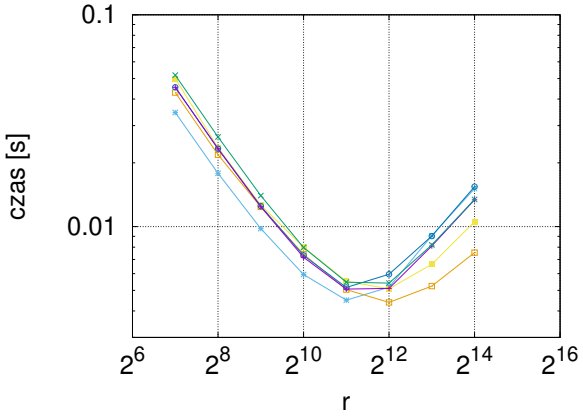
$$r(n) = \text{round}(an + b), \tag{4.43}$$

gdzie r i n są wykładnikami odpowiednich potęg liczby 2. Dla każdej z implementacji dobrano odpowiednie wartości parametrów funkcji liniowej (a, b). Rysunek 4.12 prezentuje różnice pomiędzy najlepszym czasem (wyniki z tabeli 4.1) oraz czasem otrzymanym dla estymowanej wartości parametru r.

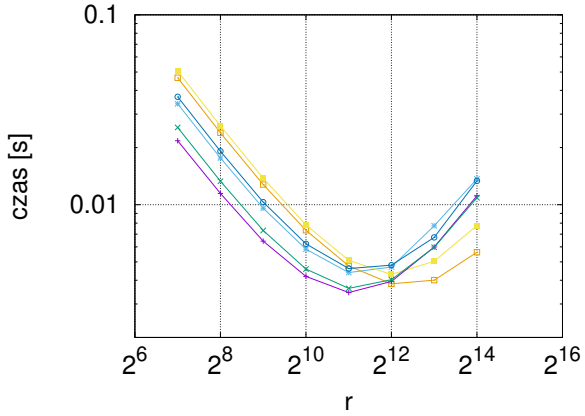
Zauważyć można, że - w większości przypadków - otrzymane wyniki są akceptowalne, a osiągane błędy nie przekraczają 4%. Dwa przypadki, w których osiągnięto najgorsze wyniki to: dla pierwszego algorytmu - około 8%, dla drugiego algorytmu - prawie 19%. Dodać należy, że owe 19%, to w wartościach bezwzględnych 0,003 s. Oba



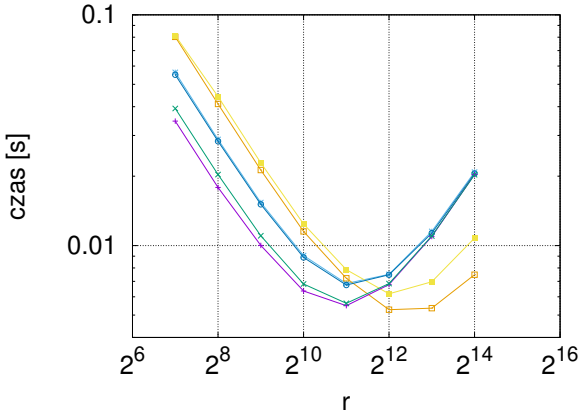
a) Kepler



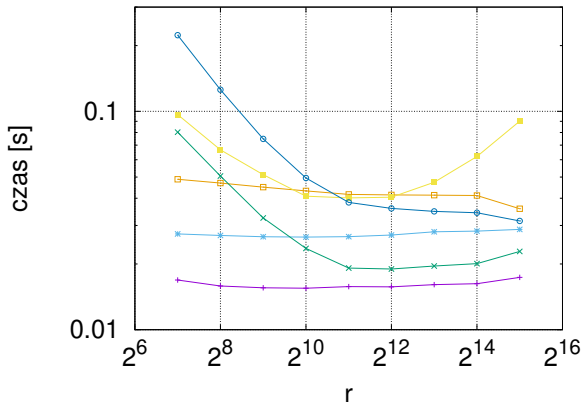
b) Turing



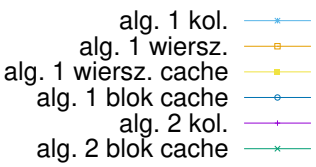
c) Volta



d) Ampere

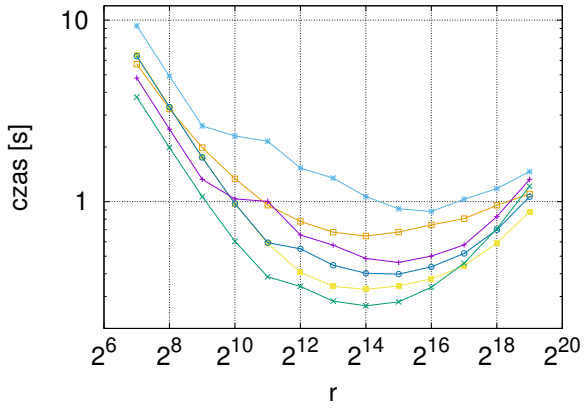


e) Multicore

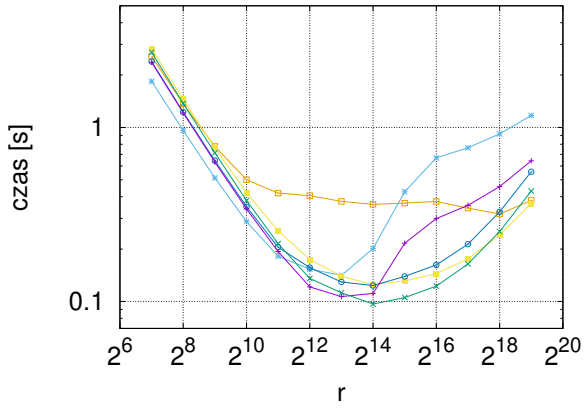


f) legenda

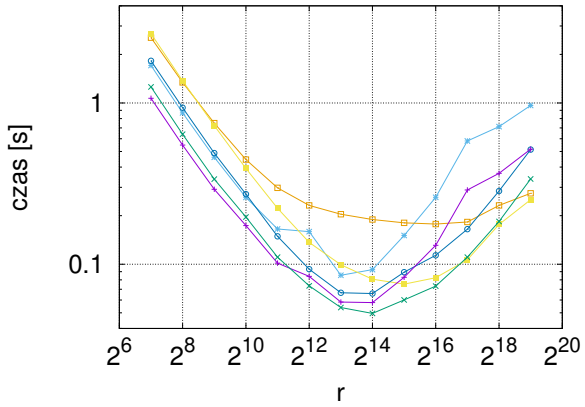
Rysunek 4.10: Wyniki czasowe dla różnych wartości parametru r dla $n = 2^{22}$



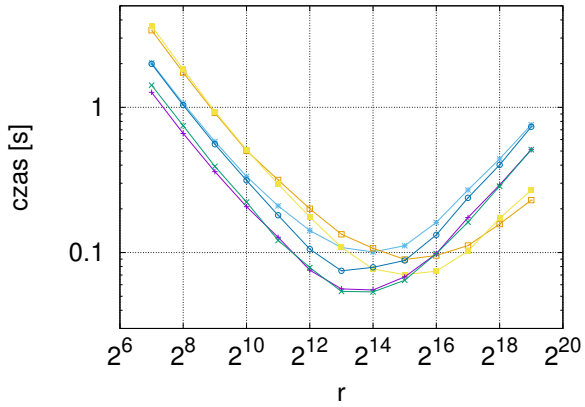
a) Kepler



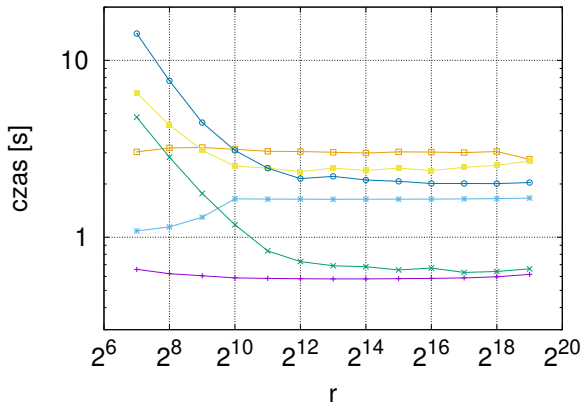
b) Turing



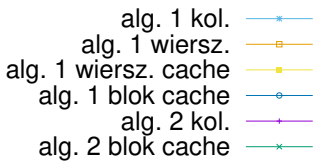
c) Volta



d) Ampere



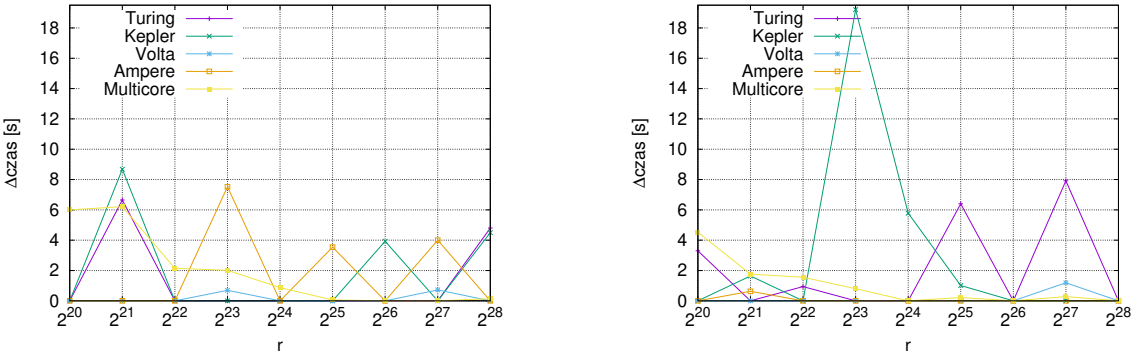
e) Multicore



f) legenda

Rysunek 4.11: Wyniki czasowe dla różnych wartości parametru r dla $n = 2^{28}$

powyższe wyniki osiągnięto na karcie graficznej Kepler. Choć, jak pokazano, omówiona metoda nie daje najlepszych wartości czasowych, to osiągane przez nią wyniki są akceptowalne. Metoda ta jest dobrym narzędziem, pozwalającym na znaczne skrócenie czasu poświęconego programowi, który miałby zostać przeznaczony na eksperymentalne znalezienie optymalnego r.



a) Algorytm pierwszy

b) Algorytm drugi

Rysunek 4.12: Różnica [%] pomiędzy wynikiem najlepszym, a wynikiem otrzymanym dla estymowanej wartości r

Rozdział 5

Sumowanie z poprawkami

5.1 Wstęp

Sumowanie ciągów liczb zmiennoprzecinkowych o dużej liczbie elementów jest jednym z najczęściej spotykanych, a przez to bardzo ważnym elementem wielu algorytmów numerycznych. Działanie to występuje między innymi w iloczynie skalarnym, obliczaniu wartości średnich lub norm, wariancji, wartości oczekiwanej i wielu innych. Algorytm 5.1 przedstawia zwykłe sumowanie iteracyjne (ang. *ordinary summation*), a listing 5.1 jego implementację w podwójnej precyzji nazwaną `DSumOrd`[1].

Algorytm 5.1 Sumowanie iteracyjne

$s \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $s \leftarrow s + a_i$
return s

```
1 //DSumOrd
2 double s=0;
3 for(int i=0;i<n;i++){
4   s+=a[i]
5 }
6 return s;
```

Listing 5.1: Implementacja Algorytmu 5.1 w podwójnej precyzji

[1]Nazwa `DSumOrd` powstała jako skrótowiec z ang. *Ordinary Summation* poprzedzona literą `D` oznaczającą podwójną precyzję (ang. *double*). W analogiczny sposób powstaną wszystkie nazwy funkcji w dalszej części rozdziału. Przy czym kolejne litery przyjmą następujące znaczenie: `F` - pojedyncza precyzja (ang. *float*), `M` - mieszana precyzja (ang. *mixed*), `K` - algorytm Kahana, `GM` - algorytm Gilla-Møllera, `V` - implementacja zwektoryzowana (ang. *vectorized*) oraz `P` - implementacja zrównoleglona (ang. *parallelized*). Wszystkie powyższe funkcje zostaną opisane w dalszej części rozdziału.

Ponadto, sumowanie może być traktowane jako rozwiązanie szczególnego przypadku układu równań typu Toeplitza. Niech:

$$\begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{5.1}$$

Rozwiązanie układu (5.1) można zapisać jako:

$$\begin{cases} x_0 = b_0 \\ x_i = b_i + x_{i-1}, \quad i = 1, \ldots, n-1. \end{cases}$$

Jeżeli w pamięci komputera wykorzystana zostanie zmienna **b** (wektor prawej strony) do wyniku (wektor **x**), to rozwiązanie układu (5.1) sprowadzi się do następującego fragmentu kodu:

```
for(int k=1;k<n;k++)
  b[k] += b[k-1];
```

Listing 5.2: Implementacja algorytmu rozwiązywania układów postaci 5.1

Jak łatwo zauważyć i-ty element takiego wynikowego wektora jest sumą częściową i poprzednich elementów, a ostatnia składowa $(n-1)$ jest sumą wszystkich składowych wektora.

Analogicznie można postąpić z układem:

$$\begin{bmatrix} 1 & -1 & & \\ & 1 & \ddots & \\ & & \ddots & -1 \\ & & & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}, \tag{5.2}$$

który przy tych samych założeniach można implementować następująco:

```
for(int k=n-2;k>=0;k--)
  b[k] += b[k+1];
```

Listing 5.3: Implementacja algorytmu rozwiązywania układów postaci 5.2

W tym przypadku w wektorze wynikowym również powstały sumy częściowe, lecz są one zapisane od końca, a suma całościowa znajduje się w elemencie `b[0]`.

Pozornie sumowanie wydaje się prostym działaniem, bez dużego pola do manewru, lecz dokładność i stabilność wielu, bardziej złożonych metod opiera się na jakości wykorzystanego sumowania. Dlatego powstało wiele algorytmów pozwalających uzyskać dużo lepszą dokładność niż proste, iteracyjne sumowanie [Hig93, Hig96]. Jeżeli pożądanym efektem jest jedynie zwiększenie dokładności - bez względu na wydajność użytej metody - do dyspozycji jest wiele bardziej złożonych sposobów sumowania [ADN20, CDGI15, LGG+22, HD01, Lef17, LH17, UDD17].

Algorytmy sumowania z poprawkami pozwalają zwiększyć dokładność sumowań dużych ciągów liczb w formatach wykorzystywanych przez standardowe oprogramowanie [Hig96]. Opierają się one na rekurencyjnym sumowaniu uzupełnionym o wyliczane wartości poprawek. Poprawki te mają na celu skompensowanie błędów zaokrągleń.

Istnieją dwie podstawowe metody stosujące opisane podejście, a mianowicie algorytm sumacyjny Kahana [Kah65] oraz algorytm Gilla-Møllera [Mø65]. Chociaż oba wymienione algorytmy są względnie proste, nie jest możliwa ich automatyczna optymalizacja przez kompilator ze względu na występujące zależności pomiędzy danymi wykorzystywanymi w kolejnych krokach obliczeniowych. Z drugiej jednak strony wektoryzacja i zrównoleglanie są niezbędne, aby wykorzystać potencjał nowoczesnych architektur wieloprocesorowych [STRP18, WWT+14, Stp20, AS20, Stp18], co pokazano w poprzednich rozdziałach.

Niniejszy rozdział prezentuje badania nad dokładnością i wydajnością sumowania iteracyjnego oraz sumowania algorytmami Kahana i Gilla-Møllera. Oba algorytmy zostały zwektoryzowane oraz zrównoleglone. Przeprowadzono testy zarówno w pojedynczej jak i podwójnej precyzji. W celu ulepszenia algorytmu Gilla-Møllera w pojedynczej precyzji, wprowadzono dodatkowo precyzję mieszaną. Opisano sposób wektoryzacji oparty na podejściu *dziel i zwyciężaj* oraz funkcjach wbudowanych (sekcja 1.2.4) Intel AVX-512. Pokazano zastosowanie OpenMP w celu otrzymania wysokowydajnych (ang. *high performance*) implementacji, które będą wykorzystywać budowę współczesnych procesorów.

5.2 Przegląd literatury

Ze względu na opisaną wcześniej istotność zagadnienia sumowania, wielu badaczy zajmowało się zwiększeniem zarówno jego jakości jak i wydajności. Szczególnym wyzwaniem jest osiągnięcie dobrej dokładności z wykorzystaniem pojedynczej precyzji, a także uzyskanie powtarzalności wyników metod zrównoleglonych.

Wydajne i odtwarzalne sumowanie zostało zaproponowane w pracach [ADN20, CDGI15, HD01, LH17]. [ADN20] prezentuje specjalną strukturę danych i oparty na niej algorytm do sumowania binarnych liczb zmiennoprzecinkowych, natomiast [CDGI15] wprowadza podejście do sumowania liczb zmiennoprzecinkowych, na które składają się dwie fazy: filtrowanie i akumulacja. W [HD01] wykorzystano dwie metody: sumowanie z poprawkami Kahana oraz sumowanie Bailey'a z użyciem formatu liczbowego double-double [HLB08]. Możliwa jest również poprawa własności numerycznych sumowania, gdy wystarczające jest sumowanie w pojedynczej precyzji liczb z mniejszych zakresów [LH17]. Biblioteka GNU MPFR zapewnia sumowanie liczb zmiennoprzecinkowych w różnych precyzjach z akceptowalnym zaokrągleniem [Lef17].

Wiele prac pokazuje sposoby zwiększenia wydajności i dokładności zrównoleglonego sumowania liczb zmiennoprzecinkowych. Między innymi algorytm `PAccSumK` zaproponowany w pracy [LGG+22] jest zrównoleglonego wersją algorytmu `AccSum` pokazanego w pracy [ROO08]. Natomiast w [UDD17] następuje próba połączenia wydajności aplikacji FPGA i łatwości użycia syntezy wysokiego poziomu (ang. *High Level Synthesis*). Rozwiązanie to jest specjalnie ukierunkowane na wzorzec sumowania-redukcji zmiennoprzecinkowej.

Chociaż wymienione rozwiązania i algorytmy pozwalają na otrzymywanie bardzo dokładnych wyników sumowania nie wykorzystują one jednak rozszerzeń wektorowych nowoczesnych procesorów. Dlatego ich wydajność może być niezadowalająca. Wekto-

ryzowanie dokładniejszych metod sumowania zostało zaproponowane w [NDMR20]. Pokazano tam wektoryzowanie sumowania z poprawkami Kahana w taki sposób, aby wykorzystać rozszerzenia SIMD procesorów Intel. Prace [HFR+15, HFR+17] pokazują zastosowanie niskopoziomowych optymalizacji pamięci podręcznej w celu zwiększenia wydajność algorytmu obliczającego iloczyn skalarny.

5.3 Algorytm Kahana

Algorytm 5.2 prezentuje sumowanie za pomocą algorytmu Kahana [Kah65, Hig96, Gol91], a implementację wymienionego algorytmu prezentuje listing 5.4. W każdym kroku iteracji wyliczana jest poprawka e. Po wyliczeniu częściowej sumy, wartość poprawki dodawana jest do następnego składnika zanim składnik ten zostanie dodany do sumy częściowej.

Algorytm 5.2 Algorytm Kahana

$s \leftarrow 0$
$e \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  $temp \leftarrow s$
  $y \leftarrow a_i + e$
  $s \leftarrow temp + y$
  $e \leftarrow (temp - s) + y$
return s

Suma uzyskana algorytmem Kahana spełnia następującą zależność [Hig93, Hig96, Gol91]:

$$\hat{s}'_n = \sum_{i=1}^{n} a_i(1 + \varepsilon'_i), \quad |\varepsilon'_i| \leqslant 2u + O(nu^2), \tag{5.3}$$

gdzie jednostka zaokrąglenia (ang. *unit roundoff*) jest równa 2^{-24} lub 2^{-53} dla formatów IEEE odpowiednio pojedynczej lub podwójnej precyzji. Oznacza to, że wartość błędu można ograniczyć następująco:

$$|s_n - \hat{s}'_n| \leqslant (2u + O(nu^2)) \sum_{i=1}^{n} |a_i|. \tag{5.4}$$

Stąd, jeżeli $nu \leqslant 1$, stała w równaniu (5.4) jest niezależna od n.

```
1 //DSumK
2 double s=0, e=0;
3 for(int i=0;i<n;i++){
4    double temp=s;
5    double y=a[i]+e;
6    s=temp+y;
7    e=(temp-s)+y;
8 }
9 return s;
```

Listing 5.4: Implementacja algorytmu 5.2 w podwójnej precyzji

5.4 Algorytm Gilla-Møllera

Algorytm 5.3 prezentuje pseudokod algorytmu Gilla-Møllera sumowania z poprawkami, natomiast jego implementacja w podwójnej precyzji została pokazana na listingu 5.5. W metodzie tej wartości poprawek są akumulowane niezależnie od wartości sumy, a dodawane są do wyniku tuż przed zwróceniem przez funkcję wyliczonej sumy [Mø65, Kie73].

Algorytm 5.3 Algorytm Gilla-Møllera

$s \leftarrow 0$
$p \leftarrow 0$
$sold \leftarrow 0$
for $i = 1, \ldots, n$ **do**
    $s \leftarrow sold + a_i$
    $p \leftarrow p + (a_i - (s - sold))$
    $sold \leftarrow s$
return $s + p$

Właściwość dla sumy wyliczonej algorytmem Gilla-Møllera wygląda następująco [Hig93, Hig96, Kie73, JSW83, JW85]:

$$\hat{s}''_n = \sum_{i=1}^{n} a_i(1 + \varepsilon''_i), \quad |\varepsilon''_i| \leqslant 2u + n^2u^2. \tag{5.5}$$

Przy założeniu $n^2u \leqslant 0{,}1$ następująca nierówność jest prawdziwa $|\varepsilon''_i| \leqslant 2{,}1u$. Oznacza to, że jeżeli n nie jest zbyt duże, to suma wyliczona przez algorytm 5.2 lub 5.3 jest tak dokładną sumą, jaką można osiągnąć w danej precyzji, natomiast dla większych wartości n algorytm Kahana ma lepsze właściwości numeryczne niż algorytm Gilla-Møllera. Ponadto, jeżeli wszystkie wyrazy sumy $a_i > 0$, to błąd względny dla obu algorytmów jest takiego samego rzędu jak u. Gdy zachodzi nierówność:

$$\sum_{i=1}^{n}|a_i| \ll |\sum_{i=1}^{n} a_i|, \tag{5.6}$$

to algorytmy sumowania z poprawkami nie gwarantują, że błąd względny zdefiniowany jak w sekcji 1.5 będzie małą wartością.

```
1 // DSumGM
2 double s=0, p=0;
3 double sold=0;
4 for(int i=0;i<n;i++){
5    s=sold+a[i];
6    p=p+((a[i]-(s-sold)));
7    sold = s;
8 }
9 return s+p;
```

Listing 5.5: Implementacja algorytmu 5.3 w podwójnej precyzji

5.5 Mieszana precyzja w algorytmie Gill-Møllera

Analiza właściwości numerycznych obu rozważanych w niniejszej rozprawie algorytmów sumowania z poprawkami pokazuje, że ich dokładność jest zależna od wielkości badanego problemu. Ponadto, błąd względny może osiągać bardzo duże wartości, jeżeli nie zostaną spełnione nierówności pokazane w rozdziałach wcześniejszych, tj. $nu \leqslant 1$ dla algorytmu Kahana lub $n^2u \leqslant 0{,}1$ dla Gilla-Møllera. Problem ten jest szczególnie widoczny, gdy implementacje korzystają z pojedynczej precyzji, natomiast użycie podwójnej precyzji sprowadza się do podobnych właściwości numerycznych obu algorytmów [DS23b]. Ponadto, dokładność algorytmu Gilla-Møllera w pojedynczej precyzji może być gorsza niż algorytmu Kahana nawet dla mniejszych rozmiarów problemu. W celu poprawienia ww. problemu rozważono użycie precyzji mieszanej.

Algorytm Gilla-Møllera może być również interpretowany [JSW83] jako wyniki pierwszego kroku metody iteracyjnego poprawiania (ang. *iterative refinement*) [Hig96] zastosowanego do układu równań liniowych postaci:

$$\underbrace{\begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{n-1} \end{bmatrix}}_{\mathbf{s}} = \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{\mathbf{a}}, \tag{5.7}$$

lub równoważnie:

$$L\mathbf{s} = \mathbf{a}. \tag{5.8}$$

Każda składowa wektora $\mathbf{s}$ spełnia następującą zależność:

$$s_k = \sum_{i=0}^{k} a_i. \tag{5.9}$$

Każdy krok algorytmu tworzy następujący wektor:

$$\mathbf{s}^\star = \mathbf{s} + \mathbf{p}, \tag{5.10}$$

gdzie wektor poprawek $\mathbf{p}$ spełnia:

$$L\mathbf{p} = \mathbf{r}. \tag{5.11}$$

a residuum $\mathbf{r}$ jest zdefiniowane następująco:

$$\mathbf{r} := \mathbf{a} - L\mathbf{s} = [a_0 - s_0, a_1 - (s_1 - s_0), \ldots, a_{n-1} - (s_{n-1} - s_{n-2})]^T. \tag{5.12}$$

Każda składowa r_i dla $i = 1, \ldots, n$, reprezentuje pojedynczą poprawkę w sumie $s_{i-1} + a_i$, stąd:

$$p_k = \sum_{i=1}^{k} r_i. \tag{5.13}$$

Ostatecznie ostatnia składowa $\mathbf{s}^\star$ jest wynikiem algorytmu.

Sumowanie algorytmem Gilla-Møllera może wyglądać następująco: układ (5.8) jest rozwiązywany w niższej precyzji podczas, gdy residuum $\mathbf{r}$ oraz poprawki $\mathbf{p}$ wyznaczane są w precyzji wyższej. Listing 5.6 prezentuje implementacje omówionego algorytmu z wykorzystaniem typów danych `float` (FP32) oraz `double` (FP64).

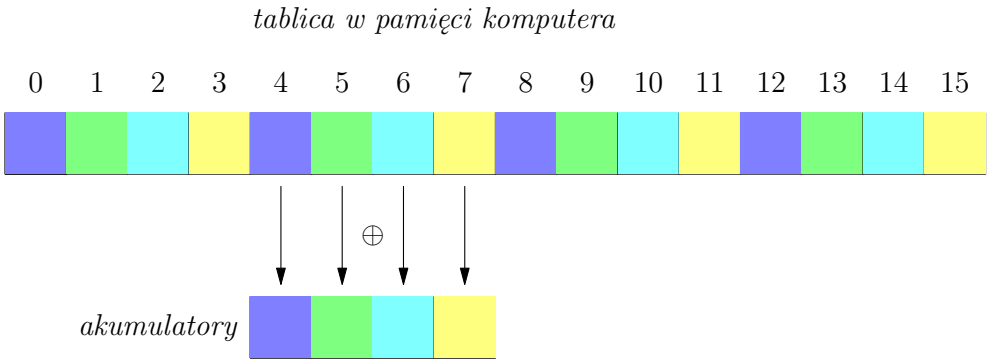
```
1  //MSumGM
2  float s = 0, sold=0;
3  double p = 0, tmp;
4  for(int i = 0; i < n; i++){
5    s=(sold+a[i]);  //dodaj kolejne elementy (FP32)
6    tmp=s-sold;     //zrzutuj aproksymacje na double
7    p=p+(a[i]-tmp); //uaktualnij poprawki (FP64)
8    sold=s;
9  }
10 return (s+p); // wynik
```

Listing 5.6: Algorytm Gilla-Møllera w mieszanej precyzji

5.6 Wektoryzacja

Zwykłe sumowanie przedstawione za pomocą algorytmu 5.1 może zostać automatycznie zoptymalizowane przez kompilator tak, aby wykorzystać rozszerzenia wektorowe nowoczesnych procesorów [DS23b]. Może również w łatwy sposób zostać manualnie zrównoleglone przy wykorzystaniu instrukcji OpenMP `parallel for` wraz z klauzulą `reduction` [PST17]. Jednakże w przypadku sumowania algorytmami Kahana i Gilla-Møllera optymalizacja nie jest tak prosta. Pętle w obu wymienionych metodach zawierają oczywiste zależności między danymi.

Faza 1: wyliczenie częściowych sum z wykorzystaniem sumowania z poprawkami



Faza 2: dodanie częściowych sum z wykorzystaniem sumowania z poprawkami

Rysunek 5.1: Podejście *dziel i zwyciężaj* do sumowania

Ogólna idea, która zostanie zastosowana w celu zbudowania zwektoryzowanego algorytmu sumowania z poprawkami, to podejście *dziel i zwyciężaj*, które zostało przedstawione na rysunku 5.1. Dane wejściowe przechowywane w pamięci są dzielone na wiele mniejszych kawałków, które są następnie ładowane z pamięci do rejestrów wektorowych, aby wykonać jeden krok algorytmu Kahana lub Gilla-Møllera - oznaczonego na rysunku 5.1 jako $\oplus$ - wykorzystując operacje wektorowe. Wynika z tego fakt, że kilka sum częściowych może zostać obliczonych jednocześnie, a ostatecznie może zostać

wykorzystane sumowanie z poprawkami, aby dodać obliczone sumy częściowe. Główna pętla funkcji `DSumK` oraz`DSumGM` może zostać podzielona na v niezależnych pętli, gdzie v jest długością wektora wykorzystanego w danym rozszerzeniu SIMD. Dla uproszenia można przyjąć założenie, że n jest wielokrotnością v. W przypadku 512 bitowych rozszerzeń wektorowych firmy Intel (ang. *Intel Advanced Vector Extensions*) (AVX-512) dla podwójnej precyzji $v = 8$, a dla precyzji pojedynczej $v = 16$. Wtedy iterator pętli - wartość k, gdzie $k = 0, \ldots, v-1$ - będzie odpowiedzialny za posumowanie liczb a_{k+iv}, gdzie $i = 1, \ldots, n/v$. Ostatecznie, v sum częściowych zostanie dodane do siebie z wykorzystaniem odpowiedniego algorytmu: Kahana lub Gilla-Møllera.

W celu spożytkowania korzyści z użycia AVX-512 oraz w celu utworzenia wektoryzowalnej implementacji rozważanych algorytmów, zastosowano funkcje wbudowane (sekcja 1.2.4) do instrukcji SIMD. Użycie funkcji wbudowanych umożliwia zapis w kodzie podobny do wywoływania funkcji w C/C++, który pozwala na odwołanie się do odpowiednich instrukcji AVX-512 [JRS16b]. Wywołania te zostają automatycznie zamienione na kod maszynowy.

Implementacja zwektoryzowanego algorytmu Kahana w podwójnej precyzji została przedstawiona na listingu 5.7, natomiast zwektoryzowany algorytm Gilla-Møllera - także w podwójnej precyzji - przedstawia listing 5.8.

```
1  //DVSumK
2  __m512d vx,vs,ve,vy,vt;
3  vs = ve = _mm512_setzero_pd();
4  for (int k=0;k<n;k=k+8){
5     vt = vs;
6     vx = _mm512_load_pd(&a[k]);  // pobierz kolejny fragment tablicy
7     vy = _mm512_add_pd(vx,ve);   // vy:=vx+ve
8     vs = _mm512_add_pd(vt,vy);   // vs:=vt+vy
9     vt = _mm512_sub_pd(vt,vs);   // vt:=vt-vs
10    ve = _mm512_add_pd(vt,vy);   // ve:=vt+vy
11 }
12 // dalej zastosuj DSumK do vs
```

Listing 5.7: Zwektoryzowany algorytm Kahana

```
1  //DVSumGM
2  __m512d vx,vs,vp,vsold,vt;
3  vp = vs = vsold = _mm512_setzero_pd();
4  for (int k = 0; k<n; k=k+8){
5     vx = _mm512_load_pd(&a[k]);   //pobierz kolejny fragment tablicy
6     vs = _mm512_add_pd(vsold,vx); // vs:=vsold+vx
7     vt = _mm512_sub_pd(vs,vsold); // vt:=vs-vsold
8     vt = _mm512_sub_pd(vx,vt);    // vt:=vx-vt
9     vp = _mm512_add_pd(vp,vt);    // vp:=vp+vt
10    vsold = vs;
11 }
12 vs = _mm512_add_pd(vs,vp);
13 // dalej zastosuj DSumGM do vs
```

Listing 5.8: Zwektoryzowany algorytm Gilla-Møllera

Obie implementacje (zarówno `DVSumK` jak i `DVSumGM`) używają dwóch zmiennych typu `__m512d` zaalokowanych w pamięci rejestru procesora: `vx` wykorzystane jest, aby przechowywać ciąg $v = 8$ kolejnych elementów załadowanych do pamięci za pomocą instrukcji `_mm512_load_pd()` oraz `vs` pełniący zadanie akumulatora. Funkcja wbudowana arytmetyczna `_mm512_add_pd()` jest wykorzystywana do przeprowadzenia zwektoryzowanego sumowania. Funkcje `DVSumK` oraz `DVSumGM` potrzebują dodatkowych zmiennych, także typu `__m512d`. Są to zmienne za pomocą których wykonano odejmowania przy użyciu `_mm512_sub_pd()` - funkcji wbudowanej, która w zwektoryzowany sposób odejmuje dwa wektory. Główną różnicą pomiędzy algorytmami jest fakt, iż algorytm Gilla-Møllera potrzebuje jeszcze jednego wywołania `_mm512_add_pd()` poza główną pętlą, aby dodać do siebie wektor z wyliczoną sumą oraz wektor poprawek.

```
1  //MVSumGM
2  __m512 vx,vs,vsold,vt;
3  __m512d vp0,vp1,vx0,vx1,vt0,vt1;
4  __mmask16 mask = 0xff00;
5  vp0=vp1=_mm512_setzero_pd();
6  vs=vsold=_mm512_setzero_ps();
7
8  for(int k = 0; k < n; k=k+16){
9    vx=_mm512_load_ps(&a[k]);  //vx:=a[k] (FP32)
10   vs=_mm512_add_ps(vsold,vx);//vs:=vsold+vx
11   // (vx0,vx1):=vx   (FP32->FP64)
12   vx0=_mm512_cvtpslo_pd(vx);
13   vx =_mm512_maskz_compress_ps(mask,vx);
14   vx1=_mm512_cvtpslo_pd(vx);
15
16   vt=_mm512_sub_ps(vs,vsold);//vt:=vs-vsold(FP32)
17   // (vt0,vt1):=vt   (FP32->FP64)
18   vt0=_mm512_cvtpslo_pd(vt);
19   vt =_mm512_maskz_compress_ps(mask,vt);
20   vt1=_mm512_cvtpslo_pd(vt);
21
22   vt0=_mm512_sub_pd(vx0,vt0);//vt:=vx-vt (FP64)
23   vt1=_mm512_sub_pd(vx1,vt1);
24   vp0=_mm512_add_pd(vp0,vt0); //vp:=vp+vt(FP64)
25   vp1=_mm512_add_pd(vp1,vt1);
26   vsold = vs;
27 }
28 vx0=_mm512_cvtpslo_pd(vs); //vx:=vs(FP32->FP64)
29 vs =_mm512_maskz_compress_ps(mask,vs);
30 vx1=_mm512_cvtpslo_pd(vs);
31
32 vp0=_mm512_add_pd(vp0,vx0);  //vp:=vp+vx   (FP64)
33 vp1=_mm512_add_pd(vp1,vx1);
34
35 // zastosuj DSumGM w celu dodania vp0, vp1
36 ...
```

Listing 5.9: Zwektoryzowany algorytm Gilla-Møllera w mieszanej precyzji

Implementacje w pojedynczej precyzji są analogiczne do opisanych implementacji w precyzji podwójnej. Różnią się one typem użytych zmiennych - w pojedynczej precyzji będzie to typ `__m512` oraz odpowiadające funkcje wbudowane, które działają na wektorze elementów w pojedynczej precyzji o długości $v = 16$. Funkcje te to:

_mm512_load_ps() - do załadowania elementów z tablicy, _mm512_add_ps() - do zwektoryzowanego dodawania dwóch wektorów oraz _mm512_sub_ps() - służąca do zwektoryzowanego odejmowania dwóch wektorów.

Implementacja algorytmu Gilla-Møllera w mieszanej precyzji została zaprezentowana na listingu 5.9. Jest ona dużo bardziej skomplikowana od implementacji przedstawionych na listingach 5.7 oraz 5.8. Sumy częściowe akumulowane są w wektorze szesnastu elementów pojedynczej precyzji. Każdy z 16-elementowych wektorów ma swojego odpowiednika w postaci pary wektorów 8-elementowych w podwójnej precyzji, w celu wyliczania poprawek. Funkcja wbudowana _mm512_cvtpslo_pd() wykorzystana jest w celu rzutowania początkowej połowy wektora pojedynczej precyzji na wektor w precyzji podwójnej, natomiast _mm512_maskz_compress_ps() służy do przesunięcia drugiej połowy elementów na miejsce początkowej połowy. Wtedy może zostać wykorzystane _mm512_cvtpslo_pd(), aby rzutować pozostałą część na wektor elementów w podwójnej precyzji.

5.7 Zrównoleglenie

Oprócz tego, że - jak powiedziano w sekcji 5.6 - zwykłe sumowanie z Algorytmu 5.1 może zostać automatycznie zoptymalizowane przez kompilator, może ono również w łatwy sposób zostać manualnie zrównoleglone przy wykorzystaniu instrukcji OpenMP `parallel for` wraz z klauzulą `reduction` [PST17]. Otrzymaną funkcję przedstawia listing 5.10.

```
1 //DPSumOrd
2 double s = 0;
3 #pragma omp parallel for reduction(+:s) schedule (static)
4 for (int i = 0; i<n; i++)
5    s += a[i];
6 return s;
```

Listing 5.10: Implementacja zwykłego, zrównoleglonego sumowania w podwójnej precyzji

Kolejna część niniejszej sekcji przedstawi zrównoleglenie zwektoryzowanej wersji algorytmów Kahana (`DVSumK`) oraz Gilla-Møllera (`DVSumGM`). W efekcie otrzymane zostaną funkcje `DPVSumK` (listing 5.11) oraz `DPVSumGM` (listing 5.12), odpowiednio. W tym celu wykorzystana zostanie konstrukcja OpenMP `parallel for` wraz z klauzulą `reduction`, która to - jeżeli nie są wykorzystywane standardowe operatory, takie jak np. sumowanie dwóch liczb zmiennoprzecinkowych - wymaga zdefiniowania operacji redukcji. W tym celu wykorzystana została dyrektywa `declare reduction`. Wymaga ona zdefiniowania nowego operatora oraz zapewnienia elementu inicjalizującego (ang. *initializer*), czyli elemetu neutralnego dla definiowanego działania. W takim przypadku należy również zdefiniować działanie kumulujące (ang. *combiner*), które pokazuje w jaki sposób będzie przebiegała redukcja dwóch elementów. Deklaracje te występują w obu metodach. W przypadku algorytmu Kahana zdefiniowano działanie `vkadd` (listing 5.11, linia 11), gdzie inicjalizatorem jest zmienna typu _mm512_setzero_pd(), a działaniem kumulującym - samodzielnie zdefiniowana funkcja - `avkadd` (ang. *vectorized Kahan addition*) (listing 5.11, linie 2-10). Funkcja ta jest odpowiedzialna za dodawanie dwóch wektorów sum częściowych obliczonych przez dwa wątki OpenMP. Należy zwrócić uwagę, że

zarówno *initializer* jak i *combiner* działają na predefiniowanych zmiennych `omp_priv`, `omp_in` oraz `omp_out`.

```
1  //DPVSumK
2  void avkadd(__m512d *vnew,__m512d *vold){
3     __m512d vs,ve,vy,vt;
4     vt = *vold;
5     vy = *vnew;
6     vs = _mm512_add_pd(vt,vy);
7     vt = _mm512_sub_pd(vt,vs);
8     ve = _mm512_add_pd(vt,vy);
9     *vnew = _mm512_add_pd(vs,ve);
10 }
11 #pragma omp declare reduction (vkadd : __m512d : avkadd (&omp_out,
       ↪ &omp_in) initializer (omp_priv=_mm512_setzero_pd())
12
13 double DPVSumK(int n, double *a){
14    __m512d vsold,vx,vs,ve,vy,vt;
15    ve = _mm512_setzero_pd();
16    vs = _mm512_setzero_pd();
17 #pragma omp parallel for firstprivate (vx, vt, vy, ve) reduction
       ↪ (vkadd : vs) schedule (static)
18    for (int k = 0; k < n; k=k+8){
19     ...
20    }
21    // dalej jak w DVSumK
22 }
```

Listing 5.11: Implementacja zwektoryzowanego i zrównoleglonego sumowania algorytmem Kahana w podwójnej precyzji

Zrównoleglenie zwektoryzowanego algorytmu Gilla-Møllera jest bardziej skomplikowane. W tym celu utworzono strukturę `GMSum`, na którą składają się dwie dane typu `__m512d` - jedna na wartość sumy, a druga na zsumowaną wartość poprawek. Ponadto, potrzebne było utworzenie dwóch dodatkowych funkcji: `avzero` - odpowiedzialnej za element neutralny oraz `avgmadd` - opisującej działanie kumulatora. W efekcie zdefiniowano działanie `vgmadd` (ang. *vectorized Gill-Møller addition*) (listing 5.12, linia 18), którego funkcjonalność opisują wspomniane wcześniej funkcje: `avzero` (listing 5.12, linie 5-8) oraz `avgmadd` (listing 5.12, linie 9-17). Kolejną różnicą w funkcji `DPVSumGM` jest wykonanie dodawania wektora sum i wektora poprawek za pomocą `_mm512_add_pd()`. Dodawanie to odbywa się poza główną pętlą, co w przypadku omawianej funkcji oznacza, że odbywa się również poza regionem zrównoleglonym.

```
1  //DPVSumGM
2  typedef struct GMSum{
3    __m512d vs,vp;
4  } GMSum;
5  void avzero(GMSum *vnew){
6    vnew->vp = _mm512_setzero_pd();
7    vnew->vs = _mm512_setzero_pd();
8  }
9  void avgmadd(GMSum *vnew,GMSum *vold){
10     __m512d tvs,tvp;
11     tvs = _mm512_add_pd(vold->vs,vnew->vs);
12     tvp = _mm512_sub_pd(tvs,vold->vs);
13     tvp = _mm512_sub_pd(vnew->vs,tvp);
14     tvp = _mm512_sub_pd(vnew->vp,tvp);
15     vnew->vp = _mm512_add_pd(vold->vp,tvp);
16     vnew->vs = tvs;
17 }
18 #pragma omp declare reduction (vgmadd : GMSum : avgmadd (&omp_out,
       ↪ &omp_in) initializer (avzero (&omp_priv))
19
20 double DPVSumGM(int n, double *a){
21  __m512d vx,vs,vt,vp,old;
22   GMSum vsold;  avzero(&vsold);
23 #pragma omp parallel for private(vx,vt,vs) reduction(vgmadd:vsold)
       ↪ schedule(static)
24   for (int k = 0; k < n; k=k+8){
25      vx = _mm512_load_pd(&a[k]);
26      vs = _mm512_add_pd(vsold.vs,vx);
27      vt = _mm512_sub_pd(vs,vsold.vs);
28      vt = _mm512_sub_pd(vx,vt);
29      vsold.vp = _mm512_add_pd(vsold.vp,vt);
30      vsold.vs = vs;
31   }
32   vs = _mm512_add_pd(vsold.vs,vsold.vp);
33  // dalej jak w DVSumGM
34     ...
35 }
```

Listing 5.12: Implementacja zwektoryzowanego i zrównoleglonego sumowania algorytmem Gilla-Møllera w podwójnej precyzji

5.8 Wyniki

Wszystkie wyniki przedstawione w tym rozdziale zostały zebrane na serwerze z dwoma procesorami *Intel Xeon Gold 6342* (oznaczenie III. z sekcji 1.3, gdzie również szczegółowe dane) uruchomionym na systemie Linux wraz z OneAPI firmy Intel (wersja 2022) zawierającym kompilatory języków C/C++, Fortran oraz wysokowydajną biblioteką numeryczną MKL.

Jako problem testowy wybrano wyznaczenie wartości sumy zdefiniowanej przez następujący wzór [DS23b]:

$$S_n = \sum_{k=0}^{n-1} a_k = \sum_{k=0}^{n-1} \frac{1}{(k \bmod m + 1)(k \bmod m + 2)}, \tag{5.14}$$

gdzie - dla uproszczenia - przyjęto założenie, że n i m są potęgami liczby 2.

Twierdzenie 5.1. *Prawdziwa jest następująca równość:*

$$S_n = \sum_{k=0}^{n-1} \frac{1}{(k \bmod m + 1)(k \bmod m + 2)} = \frac{n}{m+1}. \tag{5.15}$$

Dowód. Korzystając z faktu, że

$$\frac{1}{(k+1)(k+2)} = \frac{1}{k+1} - \frac{1}{k+2}, \tag{5.16}$$

można zapisać wartość sumy:

$$\begin{aligned} s_m = &\sum_{k=0}^{m-1} \frac{1}{(k+1)(k+2)} = \sum_{k=0}^{m-1} \frac{1}{(k+1)} - \sum_{k=0}^{m-1} \frac{1}{(k+2)} \\ =&1 + \frac{1}{2} + \ldots + \frac{1}{m-1} + \frac{1}{m} - \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{m} + \frac{1}{m+1}\right) \\ =&1 - \frac{1}{m+1} = \frac{m}{m+1}. \end{aligned} \tag{5.17}$$

Dokładając dzielenie z resztą w mianowniku otrzymano n/m powtórzeń takiego wzoru, czyli ostatecznie:

$$S_n = \frac{m}{m+1} \cdot \frac{n}{m} = \frac{n}{m+1}. \tag{5.18}$$

□

Liczby wygenerowane na podstawie wzoru (5.14) zostały poddane $2n$ losowym przestawieniom dwóch elementów. Rozważane funkcje zostały przetestowane dla $n = 2^k$, $k = 15, \ldots, 30$ oraz $m = 2^j$, $j = 2, \ldots, 6$. Dla każdej implementacji zmierzony został czas wykonania oraz wyznaczone zostały przyspieszenia i dokładności. Sekcja 1.5 prezentuje wyliczenia dokładności (wzór (1.5)) i przyspieszenia (definicja 1.1). Definicję przyspieszenia można rozwinąć następująco: *Algorytm_Bazowy* to jeden z następujących algorytmów: `SumOrd`, `SumK` lub `SumGM`, a *Algorytm_Rozważany* to wszystkie analogiczne do odpowiadającego mu algorytmu bazowego, tj. powstały następujące pary: `SumOrd` z wszytkim pozostałymi, `SumK` z `VSumK`, `PVSumK` oraz `SumGM` z `VSumGM`, `PVSumGM`, `MSumGM` oraz `MVSumGM`.

Wszystkie powyższe wyniki zgromadzono zarówno dla pojedynczej jak i dla podwójnej precyzji. Zostały one zaprezentowane w tabelach 5.1, 5.2, 5.3 i 5.4 oraz na rysunkach 5.2, 5.3, 5.4 i 5.5.

Rozważone zostały dwie implementacje zwykłego sumowania: `xSumOrd`[2] - wersja zwektoryzowana automatycznie przez kompilator (w tabelach 5.1, 5.2, 5.3, 5.4 kolumna `V`) oraz `xPSumOrd` - implementacja dodatkowo zrównoleglona przy użyciu OpenMP (w ww. tabelach kolumna `P+V`). W przypadku algorytmów sumowania z poprawkami testowano trzy implementacje: skalarną (w ww. tabelach kolumna `S`, czyli funkcje `xSumK`, `xSumGM`), zwektoryzowaną przy użyciu funkcji wbudowanych (w ww. tabelach kolumna `V`, czyli funkcje `xVSumK`, `xVSumGM`) oraz zwektoryzowaną (funkcje wbudowane) i zrównolegloną (OpenMP) (w ww. tabelach kolumna `V+P`, czyli funkcje `xPVSumK`, `xPVSumGM`).

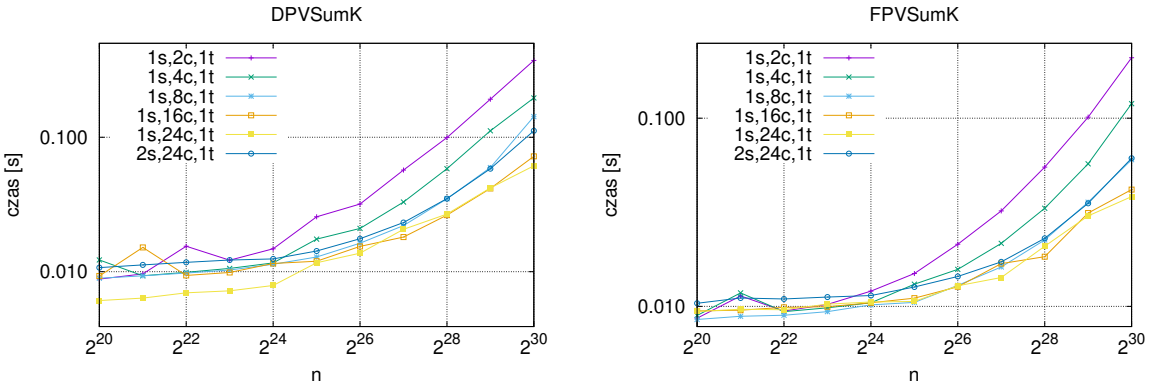
[2]prefix `x` odpowiada za precyzję: `F` - pojedyncza precyzja lub `D` - podwójna precyzja. To samo tyczyć się będzie dalszej części opisu.

Dodatkowo przetestowano dwie implementacje algorytmu Gilla-Møllera w mieszanej precyzji (w ww. tabelach kolumna `M` dla funkcji `MSumGM` - skalarnej oraz kolumna `M+V` dla funkcji `MVSumGM` - zwektoryzowanej).

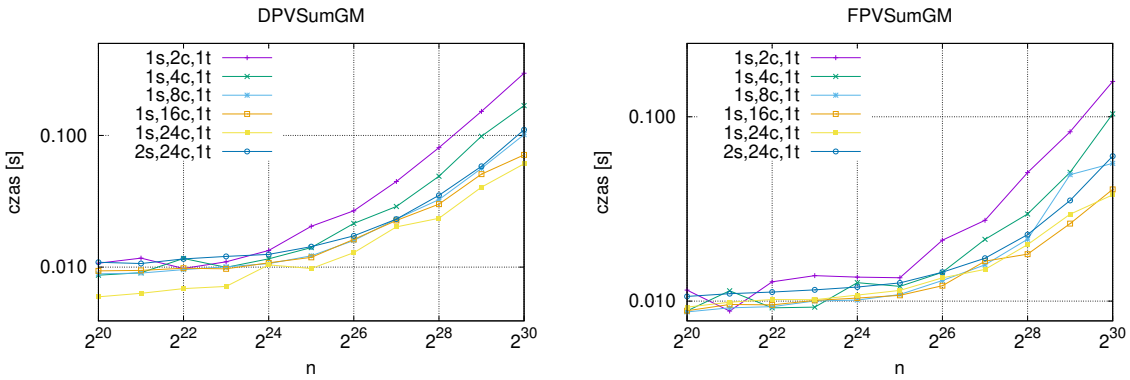
Wszystkie wymienione metody zostały skompilowane przy użyciu opcji kompilatora `O3`. Oznacza to, że włączony został najwyższy stopień optymalizacji. Zawiera on wektoryzację, dołączenie funkcji wbudowanych (ang. *intrinsic*) oraz jest rekomendowany dla implementacji, które wykonują obliczenia w zmiennych typu `float`. Należy dodać, że w przypadku kompilatora OneAPI firmy Intel, konieczna jest kompilacja przedstawionych funkcji z użyciem opcji `-fprotect-parens`, która nakazuje uwzględnianie nawiasów podczas obliczania wyrażeń zmiennoprzecinkowych. W przeciwnym przypadku funkcje opytmalizujące kompilatora mogą przekształcić obliczane wyrażenia w taki sposób, aby uzyskać szybsze działanie. Odbędzie się to jednak kosztem dokładności otrzymanego wyniku.

Dla każdego rozważanego rozmiaru problemu n, wybrano jedną wartość parametru m dla którego sumowanie zwykłe (`DSumOrd`) osiągnęło najgorszą dokładność. Wyboru takiego dokonano, aby pokazać wpływ wykorzystania sumowania z poprawkami na poprawę dokładności. Jednakże, dla poszczególnych funkcji i dla ustalonego rozmiaru n dokładność zmienia się nieznacznie wraz ze zmianą parametru m - prawie zawsze jest tego samego rzędu, a w nielicznych przypadkach zmienia się o jeden rząd wielkości. Przyjęta metodologia nie ma również znaczącego wpływu na czas wykonania funkcji. Dlatego, w celu ujednolicenia wyników, wszystkie dane zostały zebrane dla tych samych (najgorszych dla zwykłego sumowania) wartości m. Dokładne wartości m przedstawione zostały w tabelach 5.1, 5.2, 5.3 i 5.4.

W przypadku implementacji zrównoleglonych przeprowadzono testy, gdzie wykorzystywano różne liczby gniazd (ang. *socket*), rdzeni (ang. *core*) i wątków (ang. *thread*). Wartości te ustawiane były za pomocą zmiennej środowiskowej `KMP_HW_SUBSET`. Wykorzystano również ustawienie `KMP_AFFINITY=scatter`. Ta zmienna środowiskowa pozwala na kontrolę sposobu wykorzystania wątków procesora. Dzięki powyższemu ustawieniu rdzenie zostają równomiernie wykorzystane - wątki równomiernie rozdzielone.



Rysunek 5.2: Czas wykonania zwektoryzowanej i zrównoleglonej implementacji algorytmu Kahana w podwójnej precyzji (DPVSumK) (z lewej) oraz w pojedynczej precyzji (FPVSumK) (z prawej) uruchomionej na różnej liczbie gniazd i rdzeni

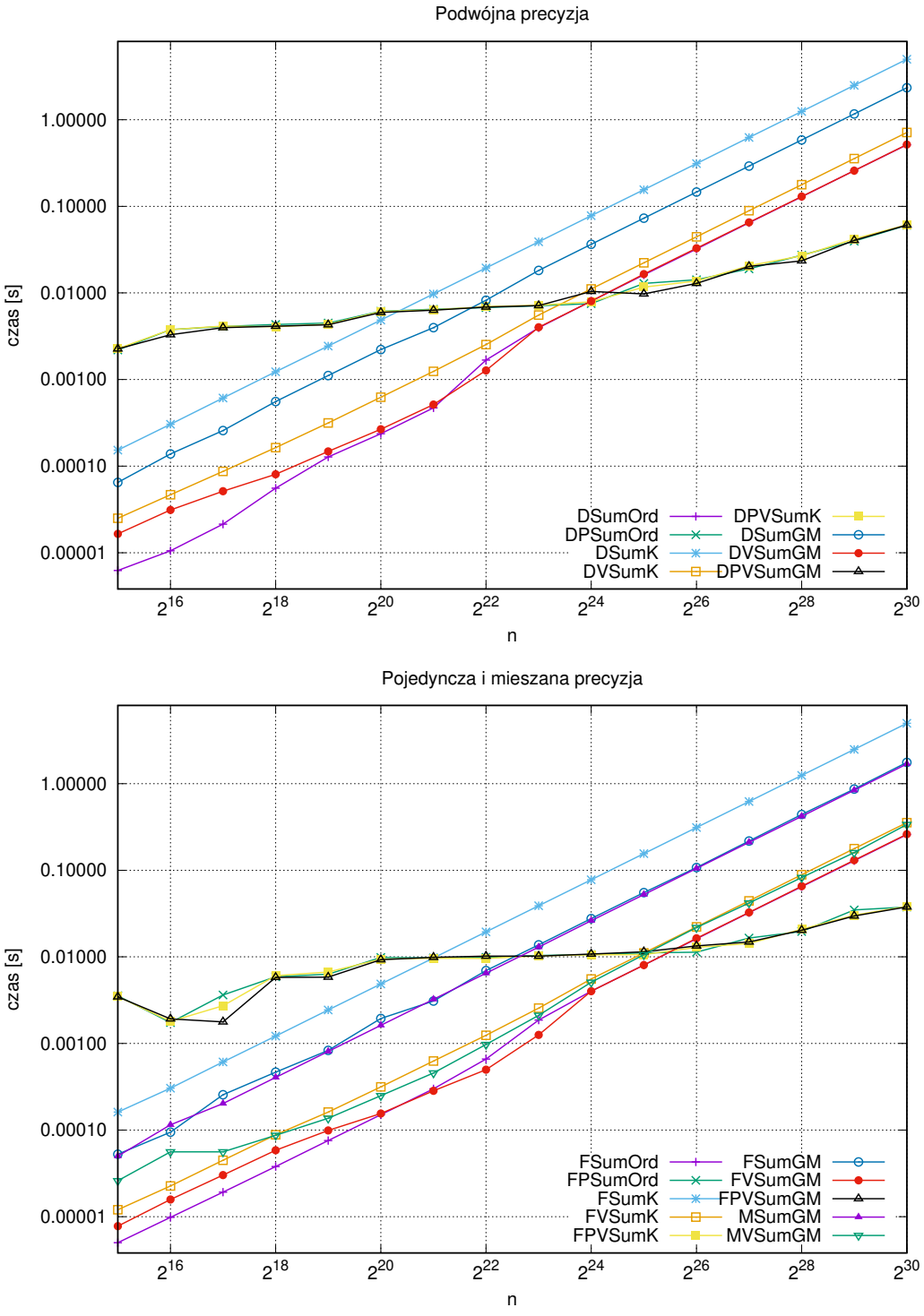


Rysunek 5.3: Czas wykonania zwektoryzowanej i zrównoleglonej implementacji algorytmu Gilla-Møllera w podwójnej precyzji (DPVSumGM) (z lewej) oraz w pojedynczej precyzji (FPVSumGM) (z prawej) uruchomionej na różnej liczbie gniazd i rdzeni

Po zebraniu wyników odrzucono - jako znacząco gorsze - wyniki dla innej liczby wątków niż jeden na rdzeń. Wyniki zabrane dla następujących ustawień: {`1s,2c,1t`}, {`1s,4c,1t`}, {`1s,8c,1t`}, {`1s,16c,1t`}, {`1s,24c,1t`}, {`2s,24c,1t`} przedstawione zostały na rysunkach 5.2 oraz 5.3. Z rysunków 5.2 i 5.3 wynika, że dla mniejszych z rozważanych wartości n, tj. dla $n < 2^{25}$ różnice są nieznaczące. Dla $n \geqslant 2^{25}$ dla obu rozważanych precyzji i implementacji najlepsze wyniki osiągnięto przy ustawieniu `KMP_HW_SUBSET=1s,24c,1t`. Natomiast, ustawienie `KMP_HW_SUBSET=1s,16c,1t` dało nieznacznie gorsze wyniki. Stąd, wszystkie wyniki zaprezentowane w dalszej części rozdziału zebrane zostały dla `KMP_HW_SUBSET=1s,24c,1t`.

Wyniki obejmujące czas trwania prezentowanych programów zostały zebrane w tabelach 5.1 i 5.2 oraz na rysunku 5.4. Z otrzymanych wyników płyną następujące wnioski. Wydajność `xSumK` oraz `xSumGM` jest znacząco gorsza niż wydajność `xSumOrd`. Wynika to z faktu, że z wymienionych funkcji tylko `xSumOrd` może zostać zwektoryzowana automatycznie przez kompilator. Jako zaskoczenie można odebrać fakt, iż `xSumGM` jest znacząco szybszy niż `xSumK` (do $3,3\times$ w pojedynczej precyzji). Podobna zależność zachodzi dla `xVSumK` oraz `xVSumGM` (w tym przypadku do około $1,4\times$ dla obu precyzji). Fakt ten jest niespodziewany dlatego, że oba algorytmy mają podobną liczbę operacji (algorytm Kahana $8n + 2$, a algorytm Gilla-Møllera $7n + 4$). Prawdopodobnym wyjaśnieniem powyższego faktu może być lepsze wykorzystanie jednostek skalarnych procesora w przypadku implementacji algorytmu Gilla-Møllera. Obie wersje zwektoryzowane algorytmów sumowania z poprawkami są do $15\times$ szybsze niż ich skalarne odpowiedniki. Można również zaobserwować, że dla większych rozmiarów problemu wydajność `xVSumGM` jest porównywalna z wydajnością `xSumOrd`.

Wszystkie przyspieszenia zostały zaprezentowane na rysunku 5.5. Przyspieszenie `xPVSumGM` względem `xSumGM` (do $38\times$ w podwójnej precyzji oraz do $39\times$ w precyzji pojedynczej) jest gorsze niż przyspieszenie `xPVSumK` nad `xSumK` (do $81\times$ w podwójnej precyzji oraz do $130\times$ w pojedynczej). Wynika to z faktu, iż zrównoleglona implementacja algorytmu Gilla-Møllera jest bardziej skomplikowana - szczególnie operacja redukcji. Warto zauważyć, że przyspieszenie `xPVSumK` względem `xSumOrd` oraz przyspieszenie `xPVSumGM` względem `xSumOrd` są prawie dokładnie takie same jak przyspieszenie `xPSumOrd` względem `xSumOrd`, to znaczy około $8,5\times$ w podwójnej oraz do $6,8\times$ w pojedynczej precyzji dla większych rozmiarów badanych problemów.



Rysunek 5.4: Czas [s] wykonania rozważanych metod

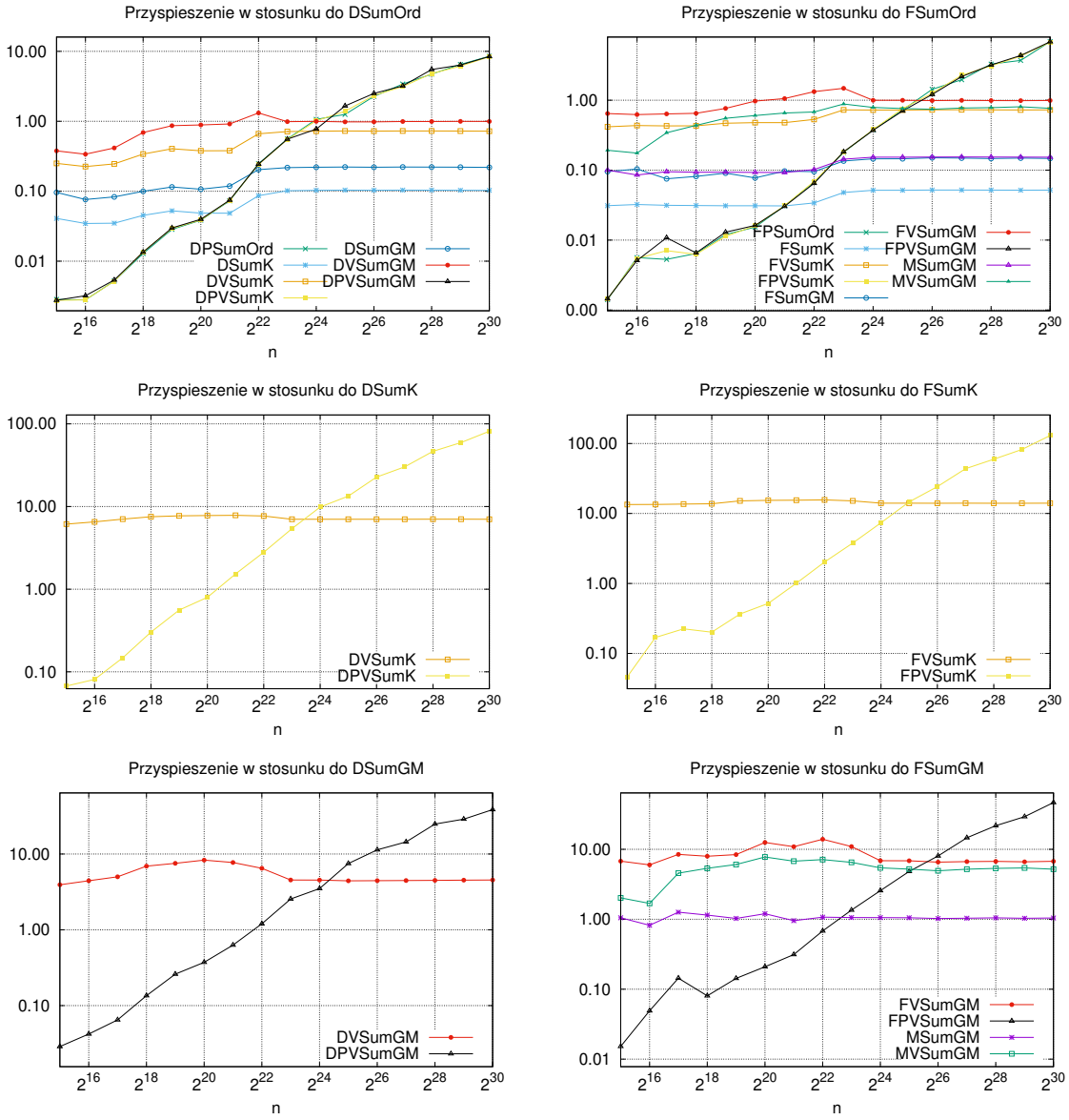
Tabela 5.1: Czas [s] wykonania wszystkich rozważanych implementacji w podwójnej precyzji

		standardowe		**Kahan**			**Gill-Møller**		
n	m	V	P+V	S	V	P+V	S	V	P+V
2^{15}	2^3	6,2e-06	0,0022	0,0002	2,5e-05	0,0023	0,0001	1,7e-05	0,0023
2^{16}	2^2	1,1e-05	0,0038	0,0003	4,6e-05	0,0038	0,0001	3,1e-05	0,0033
2^{17}	2^2	2,1e-05	0,0041	0,0006	0,0001	0,0041	0,0003	0,0001	0,0040
2^{18}	2^2	5,5e-05	0,0043	0,0012	0,0002	0,0041	0,0006	0,0001	0,0041
2^{19}	2^2	0,0001	0,0045	0,0024	0,0003	0,0044	0,0011	0,0001	0,0043
2^{20}	2^3	0,0002	0,0061	0,0049	0,0006	0,0061	0,0022	0,0003	0,0060
2^{21}	2^4	0,0005	0,0065	0,0097	0,0012	0,0064	0,0040	0,0005	0,0063
2^{22}	2^2	0,0017	0,0068	0,0194	0,0025	0,0070	0,0082	0,0013	0,0069
2^{23}	2^3	0,0040	0,0071	0,0389	0,0056	0,0072	0,0182	0,0040	0,0071
2^{24}	2^4	0,0080	0,0075	0,0779	0,0111	0,0079	0,0364	0,0080	0,0104
2^{25}	2^3	0,0162	0,0129	0,1558	0,0222	0,0117	0,0730	0,0165	0,0098
2^{26}	2^2	0,0322	0,0142	0,3116	0,0445	0,0137	0,1461	0,0328	0,0129
2^{27}	2^3	0,0647	0,0190	0,6234	0,0889	0,0206	0,2917	0,0652	0,0202
2^{28}	2^2	0,1290	0,0273	1,2468	0,1775	0,0268	0,5833	0,1299	0,0235
2^{29}	2^5	0,2577	0,0397	2,4950	0,3552	0,0420	1,1664	0,2582	0,0405
2^{30}	2^4	0,5138	0,0604	4,9954	0,7125	0,0613	2,3437	0,5165	0,0609

Tabela 5.2: Czas [s] wykonania wszystkich rozważanych implementacji w pojedynczej i mieszanej precyzji

		standardowe			**Kahan**			**Gill-Møller**			
n	m	V	P+V	S	V	P+V	S	V	P+V	M	M+V
2^{15}	2^3	5,0e-06	0,0036	0,0002	1,2e-05	0,0035	4,4e-05	7,8e-06	0,0035	0,0001	2,6e-05
2^{16}	2^2	9,8e-06	0,0017	0,0003	2,3e-05	0,0018	0,0001	1,6e-05	0,0019	0,0001	0,0001
2^{17}	2^2	1,9e-05	0,0036	0,0006	4,5e-05	0,0027	0,0002	3,0e-05	0,0018	0,0002	0,0001
2^{18}	2^2	3,8e-05	0,0059	0,0012	0,0001	0,0061	0,0004	0,0001	0,0058	0,0004	0,0001
2^{19}	2^2	0,0001	0,0063	0,0024	0,0002	0,0067	0,0007	0,0001	0,0058	0,0008	0,0001
2^{20}	2^3	0,0002	0,0098	0,0049	0,0003	0,0094	0,0014	0,0002	0,0093	0,0016	0,0002
2^{21}	2^4	0,0003	0,0097	0,0097	0,0006	0,0097	0,0028	0,0003	0,0099	0,0032	0,0005
2^{22}	2^2	0,0007	0,0099	0,0194	0,0012	0,0096	0,0057	0,0005	0,0102	0,0065	0,0010
2^{23}	2^3	0,0019	0,0104	0,0389	0,0026	0,0103	0,0115	0,0013	0,0102	0,0130	0,0021
2^{24}	2^4	0,0040	0,0106	0,0778	0,0055	0,0105	0,0232	0,0040	0,0108	0,0262	0,0051
2^{25}	2^3	0,0080	0,0111	0,1556	0,0111	0,0106	0,0463	0,0080	0,0114	0,0524	0,0106
2^{26}	2^2	0,0162	0,0113	0,3111	0,0222	0,0129	0,0925	0,0164	0,0134	0,1050	0,0218
2^{27}	2^3	0,0324	0,0165	0,6220	0,0443	0,0142	0,1854	0,0326	0,0149	0,2099	0,0418
2^{28}	2^2	0,0646	0,0197	1,2441	0,0888	0,0209	0,3704	0,0654	0,0203	0,4197	0,0825
2^{29}	2^5	0,1289	0,0348	2,4884	0,1776	0,0302	0,7407	0,1308	0,0296	0,8395	0,1600
2^{30}	2^4	0,2585	0,0376	4,9810	0,3550	0,0383	1,4898	0,2610	0,0377	1,6853	0,3380

Wykorzystanie zrównoleglonych implementacji rozważanych algorytmów, a mianowicie funkcji `xPSumOrd`, `xPVSumGM` oraz `xPVSumK`, jest korzystne dla największych z rozważanych problemów, tj. $n > 2^{24}$. Wtedy czas wykonania trzech wymienionych funkcji jest niemal identyczny (około 0,06 s w podwójnej precyzji oraz około 0,038 s w precyzji pojedynczej). Korzyść jaka płynie z obliczeń równoległych nie jest wysoka, ponieważ wykonywane pętle nie są intensywne obliczeniowo. Czas wykonania `MSumGM` oraz `FSumGM` jest prawie taki sam, jednak funkcja `MVSumGM` jest już nieco szybsza niż `FVSumK`. Jest też szybsza niż `DVSumGM` czy `DVSumK`. Dlatego, jeśli obliczenia w pojedynczej precyzji są wystarczające, to można również wykorzystać precyzję mieszaną.



Rysunek 5.5: Wybrane przyspieszenia

Tabela 5.3: Błąd względny wszystkich rozważanych implementacji w podwójnej precyzji

		standardowe		**Kahan**			**Gill-Møller**		
n	m	V	P+V	S	V	P+V	S	V	P+V
2^{15}	2^3	3,7e-14	1,0e-15	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	1,2e-16
2^{16}	2^2	3,6e-14	1,7e-15	0,0e-00	1,4e-16	0,0e-00	0,0e-00	1,4e-16	1,4e-16
2^{17}	2^2	6,1e-14	1,5e-15	0,0e-00	0,0e-00	1,4e-16	0,0e-00	1,4e-16	0,0e-00
2^{18}	2^2	1,4e-13	6,1e-15	0,0e-00	0,0e-00	1,4e-16	0,0e-00	0,0e-00	1,4e-16
2^{19}	2^2	2,4e-13	5,8e-15	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	1,4e-16
2^{20}	2^3	1,0e-12	2,1e-14	0,0e-00	0,0e-00	1,3e-16	0,0e-00	0,0e-00	0,0e-00
2^{21}	2^4	1,3e-12	2,5e-14	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	1,2e-16
2^{22}	2^2	2,3e-12	1,0e-13	0,0e-00	0,0e-00	0,0e-00	1,4e-16	0,0e-00	1,4e-16
2^{23}	2^3	8,6e-12	8,4e-14	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	1,2e-16
2^{24}	2^4	1,6e-11	5,5e-15	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00
2^{25}	2^3	3,6e-11	6,9e-13	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00
2^{26}	2^2	3,7e-11	1,6e-12	0,0e-00	0,0e-00	1,4e-16	0,0e-00	1,4e-16	0,0e-00
2^{27}	2^3	1,5e-10	3,7e-12	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	1,2e-16
2^{28}	2^2	1,5e-10	6,5e-12	0,0e-00	0,0e-00	1,4e-16	0,0e-00	0,0e-00	1,4e-16
2^{29}	2^5	3,3e-10	1,2e-11	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00
2^{30}	2^4	4,2e-10	3,5e-11	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00	0,0e-00

Tabela 5.4: Błąd względny wszystkich rozważanych implementacji w pojedynczej i mieszanej precyzji

		standardowe			**Kahan**			**Gill-Møller**			
n	m	V	P+V	S	V	P+V	S	V	P+V	M	M+V
2^{15}	2^3	1,7e-05	3,4e-07	6,0e-08	6,0e-08	6,0e-08	6,0e-08	5,5e-08	6,0e-08	6,0e-08	6,0e-08
2^{16}	2^2	1,6e-05	3,6e-07	6,0e-08	6,0e-08	6,0e-08	6,0e-08	5,0e-08	2,2e-08	6,0e-08	6,0e-08
2^{17}	2^2	3,9e-05	1,7e-06	6,0e-08	6,0e-08	6,0e-08	2,6e-08	5,0e-08	4,1e-08	6,0e-08	6,0e-08
2^{18}	2^2	6,4e-05	1,6e-06	6,0e-08	6,0e-08	6,0e-08	2,7e-07	3,2e-08	6,4e-08	6,0e-08	6,0e-08
2^{19}	2^2	1,5e-04	6,9e-06	6,0e-08	6,0e-08	6,0e-08	1,1e-08	8,3e-08	6,0e-08	6,0e-08	6,0e-08
2^{20}	2^3	6,1e-04	1,2e-05	6,0e-08	6,0e-08	6,0e-08	6,3e-07	4,3e-08	7,2e-08	6,0e-08	6,0e-08
2^{21}	2^4	1,1e-04	2,5e-05	6,0e-08	6,0e-08	6,0e-08	1,2e-06	1,5e-07	5,2e-08	5,9e-08	5,9e-08
2^{22}	2^2	1,0e-03	2,5e-05	6,0e-08	6,0e-08	6,0e-08	6,7e-05	2,4e-07	6,9e-08	6,0e-08	6,0e-08
2^{23}	2^3	1,4e-03	9,8e-05	6,0e-08	6,0e-08	6,0e-08	5,6e-05	1,1e-06	6,0e-08	6,0e-08	6,0e-08
2^{24}	2^4	6,6e-04	3,5e-04	6,0e-08	6,0e-08	6,0e-08	7,8e-04	3,1e-07	5,6e-08	5,9e-08	5,9e-08
2^{25}	2^3	8,5e-03	6,2e-04	6,0e-08	6,0e-08	6,0e-08	5,3e-03	2,0e-06	2,6e-08	6,0e-08	6,0e-08
2^{26}	2^2	1,7e-02	4,0e-04	6,0e-08	6,0e-08	6,0e-08	4,1e-02	6,6e-05	2,0e-08	6,0e-08	6,0e-08
2^{27}	2^3	7,0e-02	1,4e-03	6,0e-08	6,0e-08	6,0e-08	1,2e-01	6,2e-05	6,9e-07	6,0e-08	6,0e-08
2^{28}	2^2	1,6e-01	1,7e-03	6,0e-08	6,0e-08	6,0e-08	6,9e-01	1,1e-03	9,4e-07	6,0e-08	6,0e-08
2^{29}	2^5	1,4e-01	7,5e-03	9,3e-08	3,2e-08	3,2e-08	1,9e-01	2,3e-03	7,8e-07	5,8e-08	5,8e-08
2^{30}	2^4	7,1e-01	2,4e-02	6,0e-08	6,0e-08	3,7e-09	7,3e-01	3,9e-01	1,2e-06	6,0e-08	5,9e-08

Błąd względny dla wszystkich rozważanych metod i precyzji został przedstawiony w tabelach 5.3 (podwójna precyzja) oraz 5.4 (pojedyncza i mieszana precyzja). Z otrzymanych wyników można wyciągnąć następujące wnioski. Dokładność `xSumOrd` oraz `xPSumOrd` zmniejsza się wraz ze wzrostem rozmiaru problemu. Za pomocą funkcji `xPSumOrd` otrzymano nieco lepsze wyniki ze względu na konstrukcję `parallel for` wraz z klauzulą redukcji, przez co otrzymano podejście w stylu sumowania parami (ang. *pairwise summation*) [Hig93]. Z tabeli 5.3 można zauważyć, że wszystkie zaprezentowane implementacje algorytmów zarówna Kahana jak i Gilla-Møllera dają dokładne wyniki. Jedynie w kilku przypadkach błąd względny jest rzędu dokładności użytej precyzji (ang. *unit roundoff*) - u. Natomiast z tabeli 5.4 widać, że algorytm Gilla-Møllera osiąga dużo gorszą dokładność. Fakt ten wynika z teoretycznych właściwości algorytmów opisanych w sekcjach 5.3 oraz 5.4. Dla wszystkich rozważanych rozmiarów problemów $nu < 1$, a co za tym idzie błąd względny algorytmu Kahana nie powinien przekroczyć $O(u)$. Natomiast w przypadku Gilla-Møllera nie jest spełniona nierówność $n^2u \leqslant 0.1$ dla problemów o rozmiarze $n \geqslant 2^{25}$.

Obie zaproponowane implementacje algorytmu Gilla-Møllera z użytą mieszaną precyzją uzyskują precyzję rzędu dokładności pojedynczej precyzji, co dodatkowo (oprócz czasu wykonania) przemawia na korzyść mieszanych precyzji.

Rozdział 6

Przykłady wykorzystania sumowania z poprawkami

W niniejszym rozdziale zaprezentowane zostaną przykłady implementacji algorytmów wykorzystujących sumowanie z poprawkami. Przykłady te mają na celu zademonstrowanie zmian w dokładności wybranych algorytmów spowodowanych zastąpieniem zwykłego sumowania poprzez sumowanie z poprawkami. Jak powiedziano w rozdziale 5, sumowanie jest rozwiązaniem szczególnego układu równań liniowych. Dlatego jeden z zaprezentowanych przykładów łączy oba zagadnienia poruszane w niniejszej rozprawie, tj. wykorzystanie sumowania z poprawkami w rozwiązywaniu szczególnego, trójdiagonalnego układu równań liniowych typu Toeplitza.

6.1 Całkowanie numeryczne

Pierwszym omówionym przykładem są numeryczne metody wyznaczania wartości całki oznaczonej, tj. całkowanie numeryczne. Istnieje wiele dedykowanych temu metod, a większość z nich korzysta z addytywności całki względem przedziału całkowania, tzn:

$$\forall a \leqslant b \leqslant c : \int_a^c f(x)\,dx = \int_a^b f(x)\,dx + \int_b^c f(x)\,dx. \tag{6.1}$$

Dzięki tej własności można podzielić przedział całkowania na wiele mniejszych przedziałów, obliczać wartości dla tak wyznaczonych mniejszych przedziałów, a następnie sumować je [MW12]. Przedstawione zostaną wybrane metody całkowania numerycznego.

- Metoda prostokątów

  Jako pierwszą wybrano jedną z najprostszych metod - metodę prostokątów. Polega ona na podziale przedziału całkowania na n podprzedziałów, a następnie na każdym z tych podprzedziałów przybliżana jest wartość całki za pomocą pola prostokąta. Przedstawia to następujący wzór [MW12]:

  $$\int_a^b f(x)\,dx \approx \frac{b-a}{n} \sum_{i=0}^{n-1} f\left(a + \left(i + \frac{1}{2}\right)\frac{b-a}{n}\right). \tag{6.2}$$

  Implementację metody opisanej wzorem (6.2) przedstawia listing 6.1.

```
float rectangle(int n, float a, float b, float(*f)(float)){
    float dx = (b - a) / n;
    float sum = 0;
    float x = a + (b-a)/(2*n);
    for (int i = 0; i < n; i++) {
        sum += f(x);
        x += dx;
    }
    return sum*dx;
}
```

Listing 6.1: Implementacja metody prostokątów

- Metoda trapezów

  Kolejny sposób całkowania numerycznego to metoda trapezów. Polega ona na podziale przedziału całkowania na n podprzedziałów. Następnie na każdym z tych podprzedziałów przybliżana jest wartość całki za pomocą pola trapezu. Opisaną zależność można przedstawić za pomocą wzoru [MW12]:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2n}\sum_{i=0}^{n-1}\left(f\left(a+i\frac{b-a}{n}\right)+f\left(a+(i+1)\frac{b-a}{n}\right)\right). \tag{6.3}$$

  Implementację metody opisanej wzorem (6.3) przedstawia listing 6.2.

```
float trapezoidal(int n, float a, float b, float(*f)(float)){
    float dx = (b-a)/n;
    float sum = 0;
    float a_base = f(a), b_base,x=a;
    for(int i=0;i<n;i++){
        x+=dx;
        b_base = f(x);
        sum += (a_base+b_base);
        a_base = b_base;
    }
    return 0.5*sum*dx;
}
```

Listing 6.2: Implementacja metody trapezów

- Metoda Simpsona

  Ostatnią omawianą w tym rozdziale metodą jest metoda Simpsona. Polega ona na podziale przedziału całkowania na n podprzedziałów. Następnie na każdym z tych podprzedziałów przybliżana jest wartość całki za pomocą wielomianu interpolacyjnego Lagrange'a. Wzór opisujący tę metodę wygląda następująco [MW12]:

$$\int_a^b f(x)\,dx \approx \frac{h}{6}\left(f(a)+4\sum_{i=0}^{n-1}f\left(a+ih+\frac{h}{2}\right)+2\sum_{i=1}^{n-1}f(a+ih)+f(b)\right), \tag{6.4}$$

  gdzie $h=(b-a)/n$.

  Implementację metody opisanej wzorem (6.4) przedstawia listing 6.3.

```
1  float simpson(int n, float a, float b, float(*f)(float)){
2      float dx = (b - a) / n;
3      float sum1 = f(a+dx/2), sum2=0.,x=a;
4
5      for(int i=1;i<n;i++){
6          x+=dx;
7          sum1+=f(x+dx/2);
8          sum2+=f(x);
9      }
10     return dx/6*(f(a) + 4*sum1+2*sum2 +f(b));
11 }
```

Listing 6.3: Implementacja metody Simpsona

Wszystkie opisane metody mogą zostać uzupełnione o algorytmy sumowania z poprawkami. W każdym z omówionych algorytmów występują dwa typy sumowania:

1. obliczenie sumy/sum,
2. zmiana przedziału całkowania postaci: `x += dx`.

O ile do obliczenia sumy z punktu 1. można wykorzystać zarówno algorytm Kahana jak i Gilla-Møllera, tak suma z punktu 2. może być obliczona tylko z algorytmu Kahana. Wynika to z faktu, że algorytm Kahana aktualizuje w każdym kroku wartość sumy o wartość poprawki, natomiast algorytm Gilla-Møllera wartość poprawek akumuluje w oddzielnej zmiennej, której wartość zostaje dodana do sumy w ostatnim kroku algorytmu. Jednak, aby osiągnąć dokładny wynik w każdym kroku całkowania numerycznego, należy wykorzystać jak najdokładniejszą wartość `x`. Oczywiście, zmiana przedziału całkowania `x += dx`, może zostać zastąpiona poprzez działanie wymagające znacznie mniej operacji arytmetycznych, tzn. poprzez mnożenie postaci `x = i * dx`, dla odpowiednich wartości parametru `i` w zależności od wybranego algorytmu. Omówiony sposób jest również jedną z porównywanych implementacji. Powstało zatem wiele kombinacji implementacji w zależności od wykorzystanego sposobu sumowania:

- Zwykłe sumowanie - implementacja wprost metody całkowania, gdzie zmiana przedziału całkowania wykonana jest przez sumowanie, tj. `x += dx`,
- Mnożenie - implementacja wprost metody całkowania, gdzie zmiana przedziału całkowania wykonana jest przez mnożenie, tj. `x = i * dx`,
- Sumowanie Kahana - zmiana przedziału całkowania za pomocą algorytmu Kahana, zwykłe sumowanie wartości całki,
- Sumowanie Kahana x2 - zmiana przedziału całkowania oraz sumowanie wartości całki za pomocą algorytmu Kahana,
- Sumowanie Kahana x3 - (dotyczy tylko metody Simpsona) zmiana przedziału całkowania oraz obie sumy sumowane za pomocą algorytmu Kahana,
- Sumowanie Gilla-Møllera - sumowanie wartości całki za pomocą algorytmu Gilla-Møllera oraz zwykłe sumowanie do zmiany przedziału całkowania,

- Sumowanie Gilla-Møllera x2 - (dotyczy tylko metody Simpsona) obie sumy sumowane za pomocą algorytmu Gilla-Møllera oraz zwykłe sumowanie do zmiany przedziału całkowania,
- Sumowanie Gilla-Møllera + Kahana - sumowanie wartości całki za pomocą algorytmu Gilla-Møllera oraz zmiana przedziału całkowania za pomocą algorytmu Kahana,
- Sumowanie Gilla-Møllera x2 + Kahana - (dotyczy tylko metody Simpsona) obie sumy sumowane za pomocą algorytmu Gilla-Møllera oraz zmiana przedziału całkowania za pomocą algorytmu Kahana.

Dla przykładu podano jedną z modyfikacji, tj. metodę prostokątów z wykorzystaniem sumowania Kahana do wyznaczania wartości przedziałów całkowania oraz sumowanie Gilla-Møllera do wyznaczania wartości sumy. Rezultat można zobaczyć na listingu 6.4.

```
1  float rectangleGM_Kahan(int n, float a, float b, float(*f)(float)){
2      float dx = (b - a) / n;
3      float x = a + (b-a)/(2*n);
4      float sum=0, p=0,p2=0, sold=0, tmp3,y;
5
6      for (int i = 0; i < n; i++) {
7          //sum+=f(x); zamieniono na sumowanie algorytmem G-M
8          float term=f(x);
9          sum=sold+term;
10         float tmp=sum-sold;
11         float tmp2=term-tmp;
12         p=p+tmp2;
13         sold=sum;
14
15         //x +=dx; zamieniono na sumowanie algorytmem Kahana
16         tmp3=x;
17         y=dx +p2;
18         x=tmp3 + y;
19         p2 = (tmp3 - x) +y;
20     }
21     return (sum+p)*(dx);
22 }
```

Listing 6.4: Implementacja metody prostokątów z wykorzystaniem algorytmu Kahana

Przeprowadzono testy dla kilku wybranych, następujących całek oznaczonych:

1. $$\int_0^1 x^2\,dx = \frac{1}{3},$$
2. $$\int_0^1 e^x\,dx = e - 1,$$
3. $$\int_0^{2\Pi} sin(x)\,dx = 0,$$

4.

$$\int_0^{\Pi/2} cos(x)\,dx = 1.$$

oraz różnej liczby kroków algorytmów ($n = \{100, 1\,000, 10\,000\}$). Osiągnięte wyniki zaprezentowano w tabelach 6.1, 6.2 oraz 6.3 z podziałem na liczbę kroków algorytmów. Każdy wynik przestawiony w wyżej wymienionych tabelach to wartość bezwzględna różnicy pomiędzy otrzymanym wynikiem a wynikiem dokładnym. Łatwo zauważyć, że najlepsze wyniki mają wartość najbliższą zeru.

Tabela 6.1: Błąd dla zaprezentowanych całek i metod osiągnięty dla $n = 100$

Metoda prostokątów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	8,73e-06	4,98e-03	8,37e-06	8,37e-06	8,70e-06	8,37e-06
2	7,39e-06	8,58e-03	7,27e-06	7,27e-06	7,63e-06	7,27e-06
3	1,54e-07	8,75e-08	5,10e-08	2,11e-08	1,16e-07	1,90e-08
4	1,00e-05	7,83e-03	1,03e-05	1,03e-05	1,00e-05	1,03e-05
Metoda trapezów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	1,63e-05	1,67e-05	1,67e-05	1,66e-05	1,63e-05	1,66e-05
2	1,38e-05	1,44e-05	1,44e-05	1,42e-05	1,37e-05	1,42e-05
3	2,29e-07	1,30e-07	1,30e-07	2,29e-08	9,57e-08	2,36e-08
4	2,07e-05	2,06e-05	2,06e-05	2,06e-05	2,08e-05	2,06e-05
Metoda Simpsona						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie Kahan x3	
1	3,87e-07	5,96e-08	5,96e-08	5,96e-08	5,96e-08	
2	4,77e-07	1,19e-07	1,19e-07	1,19e-07	1,19e-07	
3	1,11e-07	1,70e-08	1,70e-08	2,45e-08	1,05e-08	
4	2,98e-07	1,19e-07	1,19e-07	1,19e-07	0	
Metoda Simpsona						
Całka	Sumowanie G-M	Sumowanie G-M x2	Sumowanie G-M+Kahan	Sumowanie GM x2+Kahan		
1	3,87e-07	3,87e-07	5,96e-08	2,98e-08		
2	2,38e-07	4,77e-07	1,19e-07	1,19e-07		
3	1,53e-07	1,08e-07	3,81e-08	1,54e-08		
4	2,38e-07	2,38e-07	0	0		

Z tabeli 6.1 można wywnioskować, że sto kroków algorytmu - czyli sumowanie stu elementów - nie jest wystarczające, aby zauważyć znaczącą poprawę. Większość wyników osiągniętych dla metod wykorzystujących sumowanie z poprawkami nie zwiększa rzędu dokładności bądź nie zwiększa dokładności wcale, a jeżeli już osiągnięto wzrost, to co najwyżej o jeden rząd wielkości.

Przy tysiącu kroków (tabela 6.2) można zauważyć znaczący wpływ sumowania z poprawkami na wynik. Każdy z najlepszych wyników był lepszy niż wynik zwykłego sumowania i tylko dla jednego przypadku najlepszy wynik sumowania z poprawkami zrównał się z wynikiem funkcji wykorzystującej mnożenie. Można z tego wnioskować, że 1000 kroków jest już wystarczającą liczbą, aby wykorzystywać sumowanie z poprawkami.

Przeprowadzono również testy dla 10 000 kroków. W tym przypadku każdy z najlepszy wyników był lepszy od wyników osiągniętych przez funkcje sumujące lub mnożące.

Jednak różnice pomiędzy wynikami osiągniętymi przy $n = 1\,000$ a $n = 10\,000$ były na tyle małe, że można przyjąć, że wykonanie tysiąca kroków będzie wystarczające.

Ponadto, dla wszystkich badanych liczb kroków wyniki najlepsze bądź bardzo bliskie najlepszym dawała metoda wykorzystująca dwa sumowania za pomocą algorytmu Kahana. Stąd można wysnuć wniosek, że wykorzystanie tej właśnie metody najprawdopodobniej da najlepszy wynik również dla innych całek oznaczonych.

Tabela 6.2: Błąd dla zaprezentowanych całek i metod osiągnięty dla $n = 1\,000$

Metoda prostokątów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	5,33e-06	5,00e-04	8,94e-08	5,96e-08	5,22e-06	5,96e-08
2	7,63e-06	8,59e-04	7,15e-07	1,19e-07	7,27e-06	1,19e-07
3	2,64e-05	2,45e-07	2,73e-07	1,03e-09	2,63e-05	7,46e-10
4	7,03e-06	7,85e-04	1,19e-07	1,19e-07	6,79e-06	1,19e-07

Metoda trapezów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	5,07e-06	2,38e-07	2,38e-07	2,09e-07	4,92e-06	2,09e-07
2	7,51e-06	9,54e-07	9,54e-07	2,38e-07	7,15e-06	2,38e-07
3	2,61e-05	5,47e-08	5,47e-08	3,76e-10	2,62e-05	4,58e-10
4	7,21e-06	1,19e-07	1,19e-07	1,79e-07	7,09e-06	1,79e-07

Metoda Simpsona					
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie Kahan x3
1	5,19e-06	5,96e-08	5,96e-08	5,96e-08	5,96e-08
2	7,63e-06	7,15e-07	7,15e-07	3,58e-07	1,19e-07
3	2,58e-05	1,26e-07	1,26e-07	6,82e-08	1,50e-07
4	6,91e-06	1,79e-07	1,79e-07	1,79e-07	0

Metoda Simpsona				
Całka	Sumowanie G-M	Sumowanie G-M x2	Sumowanie G-M+Kahan	Sumowanie GM x2+Kahan
1	5,19e-06	5,10e-06	2,98e-08	2,98e-08
2	7,27e-06	7,15e-06	4,77e-07	1,19e-07
3	2,62e-05	2,62e-05	7,00e-08	1,55e-07
4	7,03e-06	6,85e-06	1,19e-07	0

Tabela 6.3: Błąd dla zaprezentowanych całek i metod osiągnięty dla $n = 10\,000$

Metoda prostokątów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	9,66e-06	4,97e-05	5,96e-08	2,98e-08	9,92e-06	2,98e-08
2	1,29e-05	8,64e-05	1,43e-06	1,19e-07	1,20e-05	1,19e-07
3	4,02e-04	4,68e-08	5,03e-08	3,99e-10	3,97e-04	7,03e-10
4	2,74e-06	7,87e-05	1,25e-06	0	2,86e-06	0
Metoda trapezów						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie G-M	Sumowanie G-M+Kahan
1	9,66e-06	8,94e-08	8,94e-08	2,98e-08	9,92e-06	2,98e-08
2	1,29e-05	1,19e-06	1,19e-06	1,19e-07	1,20e-05	1,19e-07
3	4,02e-04	1,25e-07	1,25e-07	1,19e-09	3,96e-04	1,10e-09
4	2,74e-06	1,25e-06	1,25e-06	0	2,86e-06	0
Metoda Simpsona						
Całka	Zwykłe sumowanie	Mnożenie	Sumowanie Kahan	Sumowanie Kahan x2	Sumowanie Kahan x3	
1	9,75e-06	8,94e-08	8,94e-08	8,94e-08	0	
2	1,29e-05	1,07e-06	1,07e-06	1,19e-07	1,19e-07	
3	3,99e-04	9,89e-08	9,89e-08	6,15e-08	4,64e-08	
4	2,50e-06	7,15e-07	7,15e-07	1,19e-07	0	
Metoda Simpsona						
Całka	Sumowanie G-M	Sumowanie G-M x2	Sumowanie G-M+Kahan	Sumowanie GM x2+Kahan		
1	9,72e-06	9,89e-06	2,98e-08	2,98e-08		
2	1,26e-05	1,22e-05	8,34e-07	1,19e-07		
3	4,00e-04	3,97e-04	3,25e-08	4,44e-08		
4	2,74e-06	2,86e-06	8,34e-07	0		

6.2 Układy równań liniowych

Jako kolejny przykład rozważono rozwiązanie problemu omówionego w sekcji 2.3 oznaczonego przez A). Dla przypomnienia, należy rozwiązać następujący układ równań:

$$\underbrace{\begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{bmatrix}}_{\mathbf{d}}. \tag{6.5}$$

Układ (6.5) może zostać rozwiązany przy wykorzystaniu uproszczonej wersji eliminacji Gaussa bez pivotingu stosowanej do macierzy trójdiagonalnych, to znaczy za pomocą algorytmu Thomasa opisanego w rozdziale 2.4.1. Pierwszy krok to wyznaczenie macierzy L i R takich, że $A = LR$. W opisywanym przypadku otrzymano następujący podział:

$$A = \underbrace{\begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \\ & & & & 1 \end{bmatrix}}_{R}. \tag{6.6}$$

Jak łatwo zauważyć, przedstawione powyżej macierze zostały wspomniane w rozdziale 5. Rozwiązania układów równań powstających z takich macierzy współczynników sprowadzają się do zagadnienia sumowania, lub, inaczej mówiąc, sumowanie jest rozwiązaniem szczególnego układu równań.

Aby rozwiązać układ $LR\mathbf{u} = \mathbf{d}$, należy rozwiązać najpierw $L\mathbf{y} = \mathbf{d}$, a następnie $R\mathbf{u} = \mathbf{y}$. Przekłada się to na dwa proste układy równań:

$$\begin{cases} y_0 = d_0 \\ z_i = d_i + y_{i-1}\, i = 1, \ldots, n-1, \end{cases} \tag{6.7}$$

$$\begin{cases} u_{n-1} = y_{n-1} \\ u_i = y_i + u_{i+1},\ i = n-2, \ldots, 0. \end{cases} \tag{6.8}$$

Implementacja takiego rozwiązania wykorzystującego zwykłe sumowanie została przedstawiona na listingu 6.5.

```
1 void bvp_solve_ord(float *u, int N){
2   for(int k=1;k<N;k++)
3     u[k] += u[k-1];
4
5   for(int k=N-2;k>=0;k--)
6     u[k] += u[k+1];
7 }
```

Listing 6.5: Implementacja algorytmu rozwiązywania układów postaci 6.5 z wykorzystaniem zwykłego sumowania

W celu poprawy dokładności otrzymanego rozwiązania zaproponowano wykorzystanie sumowania algorytmem Kahana. Można zauważyć, że wybór metody sumowania jest oczywisty. W omawianym zagadnieniu oprócz posumowania całego ciągu liczb, sumy cząstkowe stanowią kolejne elementy wektora u. Z tego powodu, w omawianym zagadnieniu nie można wykorzystać sumowania metodą Gilla-Møllera, ponieważ ten algorytm kumuluje wartości poprawek w oddzielnej zmiennej i dodaje je w ostatnim kroku algorytmu. W przeciwieństwie do algorytmu Kahana, który do sumy dodaje wartości poprawek osiągnięte w każdym kroku. Otrzymaną implementację w pojedynczej precyzji przedstawia listing 6.6.

```
1  void bvp_solve_Kahan(float *u, int N){
2      int k;
3      float temp;
4      float s=u[0];
5      float e=0;
6      float y;
7      for(k=1;k<N;k++){
8          temp=s;
9          y=u[k]+e;
10         u[k]=s=temp+y;
11         e=(temp-s)+y;
12     }
13
14     s=u[N-1];
15     e=0;
16     for(k=N-2;k>=0;k--){
17         temp=s;
18         y=u[k]+e;
19         u[k]=s=temp+y;
20         e=(temp-s)+y;
21     }
22 }
```

Listing 6.6: Implementacja algorytmu rozwiązywania układów postaci 6.5 z wykorzystaniem sumowania Kahana

Implementację przedstawioną na listingu 6.5 można przekształcić na implementację dającą dużo większy potencjał do zrównoleglenia za pomocą metody opisanej w [SP11]. Pierwszy krok to podział macierzy L oraz R na macierze blokowe w sposób analogiczny do podziału przeprowadzonego w rozdziałach 3 oraz 4. Niech $n = rs$ oraz $r, s > 1$. Po wykonaniu podziału równanie $L\mathbf{y} = \mathbf{d}$ można zapisać równoważnie jako:

$$\begin{bmatrix} L_s & & & \\ B & L_s & & \\ & \ddots & \ddots & \\ & & B & L_s \end{bmatrix} \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{r-1} \end{bmatrix}, \tag{6.9}$$

gdzie macierz $L_s \in \mathbb{R}^{s\times s}$ jest takiej samej postaci jak macierz L, a

$$B = \begin{bmatrix} 0 & \dots & 0 & -1 \\ \vdots & & 0 & 0 \\ \vdots & \iddots & & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix} \in \mathbb{R}^{s\times s} \tag{6.10}$$

oraz $\mathbf{y}_i = (y_{is}, \ldots, y_{(i+1)s-1})^T$ i $\mathbf{d}_i = (d_{is}, \ldots, d_{(i+1)s-1})^T$. Niech $\mathbf{e} = (1, 1, \ldots, 1)^T \in \mathbb{R}^s$, wtedy rozwiązaniem $L\mathbf{y} = \mathbf{d}$ jest:

$$\begin{cases} \mathbf{y}_0 = L_s^{-1}\mathbf{d}_0 \\ \mathbf{y}_i = L_s^{-1}\mathbf{d}_i + y_{i-1}s\mathbf{e}, \quad i = 1, \ldots, r-1. \end{cases} \tag{6.11}$$

Analogiczna sytuacja zachodzi w przypadku układu $R\mathbf{u} = \mathbf{y}$, który zapisano równoważnie jako:

$$\begin{bmatrix} R_s & C & & \\ & R_s & \ddots & \\ & & \ddots & C \\ & & & R_s \end{bmatrix} \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{r-1} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{r-1} \end{bmatrix}, \tag{6.12}$$

gdzie macierz $R_s \in \mathbb{R}^{s \times s}$ jest takiej samej postaci jak macierz R, a $C = B^T$ oraz $\mathbf{u}_i = (u_{is}, \ldots, u_{(i+1)s-1})^T$. Wtedy rozwiązaniem $R\mathbf{u} = \mathbf{y}$ jest:

$$\begin{cases} \mathbf{u}_{r-1} = R_s^{-1}\mathbf{y}_{r-1} \\ \mathbf{u}_i = R_s^{-1}\mathbf{y}_i + u_{is+1}\mathbf{e}, \quad i = r-2, \ldots, 0. \end{cases} \tag{6.13}$$

Stosując tak opisane podejście można stworzyć implementacje mającą duży potencjał w zrównoleglaniu. Proces ten został przeprowadzony za pomocą OpenACC (patrz rozdział 1.2.3) oraz przedstawiony na listingu 6.7.

Następnie w algorytmie przedstawionym na listingu 6.7 również można zwiększyć dokładność sumowania. Łatwo zauważyć, że modyfikacja niektórych kroków nie przyniesie większych korzyści. I tak, w kroku 1C oraz 2C do każdego elementu kolumny dodawany jest ten sam element, więc występuje tutaj zwykłe dodawanie dwóch elementów. W pozostałych krokach można wykorzystać sumowanie z poprawkami. W krokach 1A i 2A kumulują się kolejne wiersze, więc wiersz ostatni/pierwszy (odpowiednio w kroku 1A/2A) jest sumą wszystkich pozostałych wierszy. W krokach 1B i 2B sumowane są elementy w odpowiednio ostatnim/pierwszym wierszu. Listing 6.8 prezentuje kroki algorytmy, w których wszystkie możliwe zwykłe sumowania zmodyfikowano na sumowania z poprawkami za pomocą algorytmu Kahana. Ze względu opisanego wcześniej wykorzystanie algorytmu Gilla-Møller jest nieefektywne w omawianym przypadku.

Podsumowując, poniżej wypisano testowane funkcje, których wyniki zawarto w niniejszej rozprawie:

- Zwykły - implementacja przedstawiona na listingu 6.5,
- Zwykły Kahan - implementacja przedstawiona na listingu 6.6,
- Równoległy - implementacja przedstawiona na listingu 6.7,
- Zwykły Kahan - implementacja przedstawiona na listingu 6.8.

Jako przykład testowy wybrano jedno z zagadnień testowanych przez [SP11] oznaczony tam jako P2, a mianowicie:

$$-\frac{d^2u}{dx^2} = 20000e^{-100x^2}(1 - 200x^2), x \in [0, 1], \tag{6.14}$$

z warunkami brzegowymi $u'(0) = 0$, $u(1) = 0$ oraz dokładnym rozwiązaniem $u(x) = 100e^{-100x^2} - 100e^{-100}$.

```
1  void parallel_bvp_solve(float *u, int s, int r){
2  // 1A
3  #pragma acc parallel present(u)
4      {
5  #pragma acc loop  independent
6          for(int j=0;j<r;j++) {
7              for(int k=1;k<s;k++)
8                  u[j*s+k]+=u[j*s+k-1];
9          }
10     }
11 // 1B
12 #pragma acc parallel num_gangs(1) present(u)
13     {
14         for (int k=1;k<r;k++)
15             u[(k+1)*s-1]+=u[k*s-1];
16     }
17 // 1C
18 #pragma acc parallel present(u)
19     {
20         for(int j=1;j<r;j++) {
21             float a=u[j*s-1];
22 #pragma acc loop  independent
23             for (int k=0;k<s-1;k++)
24                 u[j*s+k]+=a;
25         }
26     }
27 // 2A
28 #pragma acc parallel present(u)
29     {
30 #pragma acc loop  independent
31         for(int j=0;j<r;j++) {
32             for(int k=s-2;k>=0;k--)
33                 u[j*s+k]+=u[j*s+k+1];
34         }
35     }
36 // 2B
37 #pragma acc parallel num_gangs(1) present(u)
38     {
39         for (int k=r-2;k>=0;k--)
40             u[k*s]+=u[k*s+s];
41     }
42 //2C
43 #pragma acc parallel present(u)
44     for(int j=0;j<r-1;j++) {
45         float a=u[j*s+s];
46 #pragma acc loop independent
47         {
48             for (int k=1;k<s;k++)
49                 u[j*s+k]+=a;
50         }
51     }
52 }
```

Listing 6.7: Zrównoleglona implementacja algorytmu rozwiązywania układów postaci 6.5 z wykorzystaniem zwykłego sumowania

```
1  void acc_bvp_solve_Kahan(float *u, int s, int r){
2      float sum, y,e,tmp;
3
4  // 1A
5  #pragma acc parallel present(u)
6      {
7  #pragma acc loop independent
8          for(int j=0;j<r;j++) {
9              sum=u[j*s],e=0;
10             for(int k=1;k<s;k++){
11                 tmp=sum;
12                 y=u[j*s+k]+e;
13                 u[j*s+k]=sum=tmp+y;
14                 e=(tmp-sum)+y;
15             }
16         }
17     }
18
19 // 1B
20 #pragma acc parallel num_gangs(1) present(u)
21     {
22         sum=u[s-1],e=0;
23         for (int k=1;k<r;k++) {
24             tmp=sum;
25             y=u[(k+1)*s-1]+e;
26             u[(k+1)*s-1]=sum=tmp+y;
27             e=(tmp-sum)+y;
28         }
29     }
30
31 // 1C
32 #pragma acc parallel present(u)
33     {
34         for(int j=1;j<r;j++) {
35             float a=u[j*s-1];
36 #pragma acc loop  independent
37             for (int k=0;k<s-1;k++) {
38                 u[j*s+k]+=a;
39             }
40         }
41     }
42
43 // 2A
44 #pragma acc parallel present(u)
45     {
46 #pragma acc loop independent
47         for(int j=0;j<r;j++) {
48             sum=u[(j+1)*s-1],e=0;
49             for(int k=s-2;k>=0;k--){
50                 tmp=sum;
51                 y=u[j*s+k]+e;
52                 u[j*s+k]=sum=tmp+y;
53                 e=(tmp-sum)+y;
54             }
55         }
56     }
57
```

```
58 // 2B
59 #pragma acc parallel num_gangs(1) present(u)
60     {
61         sum=u[(r-1)*s],e=0;
62         for (int k=r-2;k>=0;k--) {
63             tmp=sum;
64             y=u[k*s]+e;
65             u[k*s]=sum=tmp+y;
66             e=(tmp-sum)+y;
67         }
68     }
69
70 //2C
71 #pragma acc parallel present(u)
72     for(int j=0;j<r-1;j++) {
73         float a=u[j*s+s];
74 #pragma acc loop independent
75         {
76             for (int k=1;k<s;k++) {
77                 u[j*s+k]+=a;
78             }
79         }
80     }
81 }
```

Listing 6.8: Zrównoleglona implementacja algorytmu rozwiązywania układów postaci 6.5 z wykorzystaniem metody Kahana

Wyniki zebrano na architekturze opisanej w sekcji 1.3 oznaczonej jako C. Programy zostały skompilowane za pomocą kompilatora NVC w wersji 23.11 [nvc]. Zebrano zarówno wyniki dotyczące dokładności jak i czasu trwania poszczególnych programów. Dokładność działania opisanych programów została zbadana przy użyciu wzoru (1.4) opisanego w sekcji 1.5. Otrzymane wynik przedstawia tabela 6.4 dla pojedynczej precyzji oraz tabela 6.5 dla precyzji podwójnej.

Z tabeli 6.4 wynika, że wykorzystanie sumowania z poprawkami przy algorytmie sekwencyjnym zwiększa dokładność do poziomu dokładności precyzji (w przypadku pojedynczej precyzji około 10^{-8}), lecz jednocześnie zwiększa znacząca czas trwania programów (około trzykrotnie). Problem ten zostaje praktycznie wyeliminowany w przypadku metod zrównoleglonych. Oczywiście, algorytm równoległy nawet bez wykorzystania sumowania z poprawkami ma lepszą dokładność od algorytmu zwykłego, jednak dodanie algorytmu Kahana zwiększa dokładność do poziomu użytej precyzji i jednocześnie nie wpływa znacząco na czas trwania programu. Przyspieszenia zaprezentowane w omawianej tabeli (Równoległy Kahan względem Zwykły) sięgają rzędu 8,5 dla największych badanych rozmiarów. Stąd można przyjąć wniosek, że zarówno pod względem czasowym jak i dokładności najlepszym wyborem w pojedynczej precyzji będzie wykorzystanie algorytmu wzbogaconego o sumowanie Kahana.

Natomiast wyniki z tabeli 6.5 prezentującej wyniki w podwójnej precyzji nie dają tak jednoznacznej konkluzji. Czas trwania sekwencyjnego algorytmu wzbogaconego o sumowanie Kahana również jest trzykrotnie większy niż algorytmu wykorzystującego zwykłe sumowanie, jednak nie ma w tym przypadku tak jednoznacznej korzyści w dokładności. Zauważyć ją można dopiero dla największych rozpatrywanych układów, tj. dla $n \geqslant 2^{24}$. Jednocześnie w przypadku algorytmu zrównoleglonego, sam proces

równoległości pozwala na osiągnięcie bardzo dobrej dokładności. Jednak również dla podwójnej precyzji przeprowadzono testy funkcji równoległej z algorytmem Kahana, które pokazały, że osiągnięto bardzo podobne wyniki jak dla algorytmu równoległego ze zwykłym sumowaniem. Wyjątkiem jest przypadek dla $n = 2^{28}$, gdzie algorytm Kahana poprawił wynik do poziomu dokładności podwójnej precyzji. Jednak - dla ujednolicenia - przyspieszenia podane w tabeli 6.5 zostały także wyliczone jako stosunek czasu trwania algorytmu zwykłego do równoległej wersji z sumowaniem Kahana. Osiągnięto w ten sposób przyspieszenie rzędu 6,3.

Tabela 6.4: Wyniki czasowe [s] oraz dokładności wszystkich rozważanych metod w pojedynczej precyzji

n	Zwykły		Zwykły Kahan		r	Równoległy		Równoległy Kahan		Speedup
	czas	prec	czas	prec		czas	prec	czas	prec	
2^{20}	0,0020	2,62e-03	0,0059	2,70e-08	2^{9}	0,0015	4,63e-06	0,0013	1,16e-07	1,52
2^{21}	0,0040	4,62e-03	0,0117	2,75e-08	2^{10}	0,0022	2,46e-06	0,0020	4,76e-08	1,97
2^{22}	0,0079	8,26e-03	0,0232	2,75e-08	2^{10}	0,0032	3,80e-06	0,0032	1,01e-07	2,50
2^{23}	0,0162	4,44e-02	0,0462	2,65e-08	2^{11}	0,0044	5,51e-07	0,0045	8,63e-08	3,63
2^{24}	0,0316	4,82e-02	0,0922	2,64e-08	2^{11}	0,0067	3,23e-05	0,0067	4,52e-08	4,68
2^{25}	0,0632	2,24e-01	0,1870	2,59e-08	2^{11}	0,0111	1,18e-04	0,0111	6,56e-08	5,68
2^{26}	0,1274	1,72e-01	0,3698	2,59e-08	2^{12}	0,0207	7,74e-05	0,0214	5,88e-08	5,95
2^{27}	0,2502	9,72e-02	0,7254	3,09e-08	2^{13}	0,0320	3,68e-05	0,0336	4,89e-08	7,45
2^{28}	0,5011	2,74e-01	1,4648	1,05e-07	2^{13}	0,0569	1,54e-04	0,0585	5,75e-08	8,57

Tabela 6.5: Wyniki czasowe [s] oraz dokładności wszystkich rozważanych metod w podwójnej precyzji

n	Zwykły		Zwykły Kahan		r	Równoległy		Równoległy Kahan		Speedup
	czas	prec	czas	prec		czas	prec	czas	prec	
2^{20}	0,0020	1,31e-11	0,0058	1,31e-11	2^{9}	0,0020	1,31e-11	0,0017	1,31e-11	1,12
2^{21}	0,0040	3,34e-12	0,0115	3,28e-12	2^{10}	0,0025	3,28e-12	0,0023	3,28e-12	1,73
2^{22}	0,0081	9,95e-13	0,0231	8,20e-13	2^{10}	0,0035	8,20e-13	0,0040	8,20e-13	2,01
2^{23}	0,0162	4,19e-13	0,0457	2,05e-13	2^{11}	0,0049	2,05e-13	0,0057	2,05e-13	2,86
2^{24}	0,0325	5,65e-13	0,0916	5,13e-14	2^{11}	0,0074	5,14e-14	0,0085	5,13e-14	3,85
2^{25}	0,0635	1,29e-12	0,1804	1,28e-14	2^{11}	0,0120	1,34e-14	0,0140	1,28e-14	4,53
2^{26}	0,1279	1,88e-12	0,3645	3,16e-15	2^{12}	0,0198	7,94e-15	0,0204	3,16e-15	6,27
2^{27}	0,2569	7,83e-13	0,7265	1,00e-15	2^{13}	0,0376	8,06e-15	0,0425	1,00e-15	6,04
2^{28}	0,5101	6,62e-13	1,4732	0	2^{13}	0,0732	3,26e-14	0,0839	0	6,08

Podsumowanie i kontyuacje badań

W niniejszym rozdziale zaprezentowane zostaną wnioski płynące z rozprawy oraz wyznaczone zostaną dalsze kierunki badań.

Podsumowanie

Przedstawiona rozprawa prezentuje algorytmy rozwiązywania trójdiagonalnych układów równań typu Toeplitza oraz ich implementacje na współczesne architektury wieloprocesorowe. Zaprezentowano również badania nad zagadnieniem sumowania długich ciągów liczbowych oraz ich zastosowaniem w rozwiązywaniu trójdiagonalnych układów równań liniowych typu Toeplitza. Celem niniejszej rozprawy, przedstawionym we wstępie było **opracowanie wydajnych i dokładniejszych algorytmów numerycznego rozwiązywania układów równań liniowych z macierzą współczynników trójdiagonalną typu Toeplitza oraz dokładniejszych metod sumujących bez zmniejszenia wydajności czasowej, których wektorowo-równoległe implementacje umożliwiałyby dobre wykorzystanie własności współczesnych architektur wieloprocesorowych**. Aby osiągną tak przedstawiony cel wykonano szereg kroków, których wyniki są oryginalnym osiągnięciami niniejszej rozprawy, a mianowicie:

1. Sformułowanie dwóch algorytmów wyznaczania rozwiązania układu równań liniowych z macierzą współczynników trojdiagonalną i typu Toeplitza z uwzględnieniem budowy nowoczesnych architektur komputerowych.

2. Sformułowanie różnych formatów danych pozwalających na lepsze wykorzystanie dostępu do pamięci. Pokazano - oprócz standardowego formatu kolumnowego - format wierszowy, wierszowy z wykorzystaniem pamięci podręcznej do zamiany formatów oraz format kolumnowy z wykorzystaniem bloku pamięci podręcznej na obliczenia.

3. Implementacja algorytmów z punktu (1) z wykorzystaniem formatów danych z punktu 2 oraz procesów równoległości i wektorowości. Procesy te zostały włączone przy pomocy interfejsów OpenMP lub OpenACC. Pokazano implementacje przenośne pomiędzy CPU i GPU, heterogeniczne - działające na dwóch kartach graficznych oraz hybrydowe działające jednocześnie na CPU i GPU.

4. Implementacja algorytmów Kahana i Gilla-Møllera w sposób pozwalający na włączenie procesów wektorowych za pomocą funkcji wbudowanych (ang. *intrinsic*) oraz procesu równoległości obliczeń przy użyciu OpenMP.

5. Wykorzystanie algorytmów sumowania z poprawkami w algorytmach całkowania numerycznego oraz w metodzie rozwiązywania szczególnego przykładu trójdiagonalnego układu równań liniowych typu Toeplitza.

6. Przeprowadzenie eksperymentów oceniających prezentowane implementacje. Zbadanie zarówno czas działania, jak i dokładność prezentowanych funkcji w pojedynczej jak i podwójnej precyzji. Dla funkcji z rozdziału 3 zbadano również wydajność energetyczną. Dla implementacji z rozdziału 4 przeprowadzono automatyzację doboru parametrów.

7. Poprawa czasu działania algorytmów. W wynikach dotyczących czasu działania programów otrzymano korzyści płynące z wykorzystania implementacji algorytmów sformułowanych w punkcie (1). Otrzymano również korzyści z zamiany formatu danych (punkt 2) w obliczeniach na kartach graficznych względem formatu kolumnowego uwzględniając nawet czas potrzebny do zamiany formatów. Implementacje działające na dwóch GPU skalowały się zgodnie z założeniem, tzn. dla dużych rozmiarów czas trwania programu wykonany na dwóch GPU był dwa razy krótszy niż czas wykonania na jednym urządzeniu akcelerującym. Natomiast implementacje heterogeniczne działają znacząco dłużej niż analogiczne funkcje wykonane na dwóch GPU (tak samo jak implementacje uruchamiane tylko na CPU względem implementacji działających na GPU), jednak mogą być pomocne, gdy rozmiar rozważanego problemu będzie przekraczał możliwości karty graficznej.

8. Poprawa dokładności wyników otrzymywanych z przedstawionych implementacji poprzez wykorzystanie sumowania z poprawkami. Wyniki obu algorytmów sumujących z poprawkami są znacząco lepsze niż wykorzystanie zwykłego sumowania. Pokazuje to rozdział 5, jak również przykłady wykorzystania sumowania z poprawkami przedstawione w rozdziale 6, a mianowicie metody całkowania numerycznego i rozwiązywanie szczególnego układu równań liniowych. Zauważono również gorsze własności numeryczne algorytmu Gilla-Møllera, które uwidoczniły się szczególnie w pojedynczej precyzji. Rozwiązano ten problem wykorzystując precyzję mieszaną.

Otrzymane wyniki implikują prawdziwość postawionej we wstępie tezy, iż **zrównoleglenie i wektoryzacja pozwalają na znaczące przyspieszenie działania implementacji algorytmów numerycznych rozwiązujących układy równań o macierzach trójdiagonalnych typu Toeplitza na współczesnych procesorach, procesorach graficznych oraz architekturach hybrydowych.**

Kontyuacje badań

Zagadnienia opisywane w niniejszej rozprawie mają potencjał do dalszych badań, szczególnie temat rozwiązywania układów równań liniowych. Naturalnym kierunkiem jest stworzenie kolejnych algorytmów i struktur danych do badania innych typów układów równań linowych, w których macierze współczynników będą spełniały dodatkowe kryteria. Przykładem może być układ z macierzą trójdiagonalną typu Toeplitza z dodatkowymi elementami w prawym górnym i lewym dolnym rogu. Zaczynając od naj-

prostszego układu postaci:

$$\begin{bmatrix} t_2 & 1 & & & 1 \\ 1 & t_2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & 1 \\ 1 & & & 1 & t_2 \end{bmatrix}, \tag{6.15}$$

a następnie przechodząc do:

$$\begin{bmatrix} t_2 & t_3 & & & 1 \\ t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & t_3 \\ 1 & & & t_1 & t_2 \end{bmatrix} \quad \text{lub ogólnie} \quad \begin{bmatrix} t_2 & t_3 & & & t_4 \\ t_1 & t_2 & t_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & t_3 \\ t_0 & & & t_1 & t_2 \end{bmatrix}. \tag{6.16}$$

Inny, ciekawy przypadek, warty dalszego badania to dwie macierze postaci:

$$\begin{bmatrix} a_0 & & & & b_0 \\ b_1 & a_1 & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & & b_{n-1} & a_{n-1} \end{bmatrix} \quad \text{oraz} \quad \begin{bmatrix} a_0 & c_0 & & & \\ & a_1 & c_1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & c_{n-2} \\ c_{n-1} & & & & a_{n-1} \end{bmatrix}. \tag{6.17}$$

Jako kolejny przykład można podać macierze układów blokowo trójdiagonalnych, w których każdy blok jest typu Toeplitza. Przykład może wyglądać następująco:

$$\begin{bmatrix} \mathbf{b}_0 & \mathbf{c}_0 & & & & \\ \mathbf{a}_1 & \mathbf{b}_1 & \mathbf{c}_1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \mathbf{c}_{n_2} \\ & & & & \mathbf{a}_{n-1} & \mathbf{b}_{n-1} \end{bmatrix}, \tag{6.18}$$

gdzie każdy z elementów $\mathbf{a}_i$, dla $i = 1, \ldots, n-1$, $\mathbf{b}_i$, dla $i = 0, \ldots, n-1$, $\mathbf{c}_i$, dla $i = 0, \ldots, n-2$ jest typu Toeplitza.

Rozwijaniu może podlegać sam temat wąskopasmowości, czyli kolejnym typem będzie macierz pentadiagonalna:

$$\begin{bmatrix} t_3 & t_4 & t_5 & & & \\ t_2 & t_3 & t_4 & \ddots & & \\ t_1 & t_2 & t_3 & \ddots & t_5 & \\ & t_1 & \ddots & \ddots & t_4 & t_5 \\ & & \ddots & t_2 & t_3 & t_4 \\ & & & t_1 & t_2 & t_3 \end{bmatrix}. \tag{6.19}$$

Na dalsze prace zasługuje również podejście do wyszukiwania rozwiązania. Jako kolejny sposób można rozważyć skorzystanie z metody iteracyjnego poprawiania (ang.

iterative refinement) [CH18]. Polega ona na powtarzaniu pewnych kroków określonego algorytmu w celu poprawienia dokładności rozwiązania do osiągnięcia zadanej precyzji bądź do osiągnięcia nieznaczących różnic pomiędzy rozwiązaniami znalezionymi w kolejnych krokach. Takie podejście może być szczególnie istotne w przypadku implementacji w pojedynczej bądź mieszanej precyzji.

Na dalsze prace zasługuje również temat sumowania z poprawkami. Pokazano, że przedstawione w niniejszej rozprawie algorytmy poprawiają dokładność sumowania bez istotnego wpływu na czas trwania programu. Dlatego, należy przejść do zastosowania pokazanego sumowania w znanych algorytmach na przykład iloczynie skalarnym, w obliczaniu wartości średnich lub norm, wariancji, wartości oczekiwanej lub innych.

Można podjąć również kontynuację nad badaniem rozpoczętym w rozdziale 6, gdzie należy zająć się zagadnieniem czasu trwania programów numerycznego wyznaczania wartości całek oznaczonych. Po zbadaniu czasu należy podjąć badania nad jego zmniejszeniem poprzez zrównoleglenie lub wektoryzację.

Ponadto architektury komputerów podlegają bardzo szybkiemu rozwojowi, szczególnie dotyczy to procesorów graficznych. Proces ten wpływa na konieczność tworzenia nowych algorytmów w pełni wykorzystujących nowoczesne architektury.

Jako element łączący oba zagadnienia prezentowane w niniejszej rozprawie można podjąć próbę wykorzystania implementacji sumowania z poprawkami przedstawionymi w rozdziale 5 w implementacjach algorytmów rozwiązywania trójdiagonalnych układów równań typu Toeplitza. Początek tych badań (rozdział 6) dał bardzo obiecujące wyniki, szczególnie dla pojedynczej precyzji. Zaprezentowane podejście można w dalszym kroku zastosować w algorytmach wykorzystanych w rozdziałach 3 i 4. Warto zbadać w jakim stopniu wykorzystanie sumowania z poprawkami w pojedynczej precyzji poprawi dokładności otrzymywanych wyników i jak wpłynie na czas działania programów.

Bibliografia

[AALZ00] Amor, Margarita ; Argüello, Francisco ; López, Juan ; Zapata, Emilio L.: Parallelization of a Recursive Decoupling Method for Solving Tridiagonal Linear Systems on Distributed Memory Computer. In: *Vector and Parallel Processing - VECPAR 2000, 4th International Conference, Porto, Portugal, June 21-23, 2000, Selected Papers and Invited Talks*, 2000, S. 344–354

[ABB+92] Anderson, E. ; Bai, Z. ; Bischof, C. ; Demmel, J. ; Dongarra, J. ; Du Croz, J. ; Greenbaum, A. ; Hammarling, S. ; McKenney, A. ; Ostruchov, S. ; Sorensen, D.: *LAPACK User's Guide*. Philadelphia : SIAM, 1992

[ACD+01] Alonso, Pedro ; Cortina, Raquel ; Díaz, Irene ; Hernández, Vicente ; Ranilla, José: A Columnwise Block Striping in Neville Elimination. In: *Parallel Processing and Applied Mathematics, 4th International Conference, PPAM 2001 Naleczow, Poland, September 9-12, 2001, Revised Papers*, 2001, 379–386

[ADN20] Ahrens, Peter ; Demmel, James ; Nguyen, Hong D.: Algorithms for Efficient Reproducible Floating Point Summation. In: *ACM Transactions on Mathematical Software* 46 (2020), S. 22:1–22:49. `http://dx.doi.org/10.1145/3389360`. – DOI 10.1145/3389360

[AS20] Amiri, Hossein ; Shahbahrami, Asadollah: SIMD programming using Intel vector extensions. In: *Journal of Parallel and Distributed Computing* 135 (2020), S. 83–100. `http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2019.09.012`. – DOI https://doi.org/10.1016/j.jpdc.2019.09.012

[ASU02] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Kompilatory: reguły, metody i narzędzia*. Warszawa : WNT, 2002

[BE93] Bekakos, M. P. ; Evans, David J.: Parallel Cyclic Odd-Even Reduction Algorithms for Solving Toeplitz Tridiagonal Equations on MIMD Computers. In: *Parallel Computing* 19 (1993), Nr. 5, 545–561. `http://dx.doi.org/10.1016/0167-8191(93)90005-6`. – DOI 10.1016/0167–8191(93)90005–6

[BGN70] Buzbee, B. L. ; Golub, G. H. ; Nielson, C. W.: On Direct Methods for Solving Poisson's Equations. In: *SIAM Journal on Numerical Analysis* 7 (1970), Nr. 4, 627–656. `http://www.jstor.org/stable/2949380`. – ISSN 00361429

[BSHS11] Beyer, James C. ; Stotzer, Eric J. ; Hart, Alistair ; Supinski, Bronis R.: OpenMP for Accelerators, 2011 (Lecture Notes in Computer Science), 108-121

[CDGI15] Collange, Sylvain ; Defour, David ; Graillat, Stef ; Iakymchuk, Roman: Numerical reproducibility for the parallel reduction on multi- and many-core architectures. In: *Parallel Computing* 49 (2015), S. 83–97. http://dx.doi.org/10.1016/j.parco.2015.09.001. – DOI 10.1016/j.parco.2015.09.001

[CGM14] Cheng, John (Hrsg.) ; Grossman, Max (Hrsg.) ; McKercher, Ty (Hrsg.): *Professional CUDA C Programming*. Wiley and Sons, 2014

[CH14] In: Chang, Li-Wen ; Hwu, Wen-mei W.: *A Guide for Implementing Tridiagonal Solvers on GPUs*. Cham : Springer International Publishing, 2014. – ISBN 978–3–319–06548–9, S. 29–44

[CH18] Carson, Erin ; Higham, Nicholas J.: Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. In: *SIAM Journal on Scientific Computing* 40 (2018), Nr. 2, S. A817–A847. http://dx.doi.org/10.1137/17M1140819. – DOI 10.1137/17M1140819

[CJ18] Chandrasekaran, Sunita (Hrsg.) ; Juckeland, Guido (Hrsg.): *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley, 2018

[CY96a] Chung, Kuo-Liang ; Yan, Wen-Ming: Parallel B-Spline Surface Fitting on Mesh-Connected Computers. In: *Journal of Parallel and Distributed Computing* 35 (1996), S. 205 – 210

[CY96b] Chung, Kuo-Liang ; Yan, Wen-Ming: Vectorized algorithms for solving special tridiagonal systems. In: *Computers and Mathematics with Applications* 32 (1996), S. 1 – 14

[DDSV91] Dongarra, J. ; Duff, I. ; Sorensen, D. ; Van der Vorst, H.: *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia : SIAM, 1991

[DS20] Dmitruk, Beata ; Stpiczyński, Przemysław: Vectorized Parallel Solver for Tridiagonal Toeplitz Systems of Linear Equations. In: *Parallel Processing and Applied Mathematics, 13th International Conference, PPAM 2019, Białystok, Poland, September 8-11, 2019* Bd. 12043, Springer, 2020 (Lecture Notes in Computer Science), S. 93–103

[DS21] Dmitruk, Beata ; Stpiczyński, Przemysław: High Performance Portable Solver for Tridiagonal Toeplitz Systems of Linear Equations. In: *Euro-Par 2020: Parallel Processing Workshops* Bd. 12480, Springer, 2021 (Lecture Notes in Computer Science), S. 172–184

[DS22] Dmitruk, Beata ; Stpiczynski, Przemyslaw: Solving tridiagonal Toeplitz systems of linear equations on GPU-accelerated computers. In: *Concurrency and Computation Practice and Experience* 34 (2022). http://dx.doi.org/10.1002/cpe.6449. – DOI 10.1002/cpe.6449

[DS23a] Dmitruk, Beata ; Stpiczyński, Przemysław: Improving accuracy of summation using parallel vectorized Kahan's and Gill-Møller algorithms. In: *Concurrency and Computation Practice and Experience* (2023), S. 1–13. http://dx.doi.org/10.1002/cpe.7763. – DOI 10.1002/cpe.7763

[DS23b] Dmitruk, Beata ; Stpiczyński, Przemysław: Parallel Vectorized Implementations of Compensated Summation Algorithms. In: *Parallel Processing and Applied Mathematics, 14th International Conference, PPAM 2022, Gdańsk, Poland, September 11-14, 2022* Bd. 13827, Springer Nature, 2023 (Lecture Notes in Computer Science), S. 63–74

[DSZ18] Du, Lei ; Sogabe, Tomohiro ; Zhang, Shao-Liang: A fast algorithm for solving tridiagonal quasi-Toeplitz linear systems. In: *Applied Mathematics Letters* 75 (2018), S. 74 – 81. – ISSN 0893–9659

[Ebe14] Eberly, D.H.: *GPGPU Programming for Games and Science*. Taylor & Francis, 2014 https://books.google.pl/books?id=yphBBAAAQBAJ. – ISBN 9781466595354

[EGJK04] Elmroth, Erik ; Gustavson, Fred ; Jonsson, Isak ; Kågström, Bo: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. In: *SIAM Rev.* 46 (2004), 03, S. 3–45. http://dx.doi.org/10.1137/S0036144503428693. – DOI 10.1137/S0036144503428693

[Far11] Farrelle, Paul M.: *Recursive Block Coding for Image Data Compression*. Springer New York, NY, 2011. http://dx.doi.org/10.1007/978-1-4613-9676-5. http://dx.doi.org/10.1007/978-1-4613-9676-5. – ISBN 978–1–4613–9678–9

[Far17] Farber, Rob (Hrsg.): *Parallel Programming with OpenACC*. Morgan Kaufmann, 2017

[GBWS18] Gajos-Balińska, Anna ; Wojcik, Grzegorz M. ; Stpiczyński, Przemysław: Performance comparison of parallel fastICA algorithm in the PLGrid structures. In: *ITM Web of Conferences* 21 (2018), S. 00026. http://dx.doi.org/10.1051/itmconf/20182100026. – DOI 10.1051/itmconf/20182100026

[Gol91] Goldberg, David: What Every Computer Scientist Should Know About Floating-Point Arithmetic. In: *ACM Computing Survey* 23 (1991), S. 5–48. http://dx.doi.org/10.1145/103162.103163. – DOI 10.1145/103162.103163

[GS01] Garey, L.E. ; Shaw, R.E.: A parallel method for linear equations with tridiagonal Toeplitz coefficient matrices. In: *Computers and Mathematics with Applications* 42 (2001), Nr. 1, S. 1 – 11. – ISSN 0898–1221

[GTM] *General tridiagonal Matrix*. https://netlib.org/lapack/explore-html/dd/da3/group___g_t.html, . – Dostęp: 4-09-2023

[GVL96] Golub, Gene H. ; Van Loan, Charles F.: *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA : Johns Hopkins University Press, 1996. – ISBN 0–8018–5414–8

[HD01] He, Yun ; Ding, Chris H. Q.: Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. In: *The Journal of Supercomputing* 18 (2001), S. 259–277. http://dx.doi.org/10.1023/A:1008153532043. – DOI 10.1023/A:1008153532043

[HFR+15] Hofmann, Johannes ; Fey, Dietmar ; Riedmann, Michael ; Eitzinger, Jan ; Hager, Georg ; Wellein, Gerhard: Performance Analysis of the Kahan-Enhanced Scalar Product on Current Multicore Processors. 9573 (2015), S. 63–73. http://dx.doi.org/10.1007/978-3-319-32149-3_7. – DOI 10.1007/978–3–319–32149–3_7

[HFR+17] Hofmann, Johannes ; Fey, Dietmar ; Riedmann, Michael ; Eitzinger, Jan ; Hager, Georg ; Wellein, Gerhard: Performance analysis of the Kahan-enhanced scalar product on current multi-core and many-core processors. In: *Concurr. Comput. Pract. Exp.* 29 (2017), Nr. 9. http://dx.doi.org/10.1002/cpe.3921. – DOI 10.1002/cpe.3921

[Hig93] Higham, Nicholas J.: The Accuracy of Floating Point Summation. In: *SIAM Journal of Scientific Computing* 14 (1993), S. 783–799. http://dx.doi.org/10.1137/0914050. – DOI 10.1137/0914050

[Hig96] Higham, N.J.: *Accuracy and Stability of Numerical Algorithms*. Philadelphia : SIAM, 1996

[HLB08] Hida, Yozo ; Li, Sherry ; Bailey, David: Library for Double-Double and Quad-Double Arithmetic. (2008), 01

[JRS16a] Jeffers, James ; Reinders, James ; Sodani, Avinash: *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*. 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2016. – ISBN 0128091940

[JRS16b] Jeffers, Jim ; Reinders, James ; Sodani, Avinash: *Intel Xeon Phi Processor High-Performance Programming. Knights Landing Edition*. Cambridge, MA, USA : Morgan Kaufman, 2016

[JSW83] Jankowski, M. ; Smoktunowicz, Alicja ; Woźniakowski, Henryk: A Note on Floating-point Summation of Very Many Terms. In: *Elektronische Informationsverarbeitung und Kybernetik* 19 (1983), S. 435–440

[JW85] Jankowski, M. ; Woźniakowski, H.: The accurate solution of certain continuous problems using only single precision arithmetic. In: *BIT* (1985). http://dx.doi.org/10.1007/BF01936142. – DOI 10.1007/BF01936142

[Kah65] Kahan, William: Pracniques: further remarks on reducing truncation errors. In: *Communications of ACM* 8 (1965), S. 40. http://dx.doi.org/10.1145/363707.363723. – DOI 10.1145/363707.363723

[KHN+18] Khan, Kashif N. ; Hirki, Mikael ; Niemi, Tapio ; Nurminen, Jukka K. ; Ou, Zhonghong: RAPL in Action: Experiences in Using RAPL for Power Measurements. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3 (2018), Nr. 2, S. 9:1–9:26. http://dx.doi.org/10.1145/3177754. – DOI 10.1145/3177754

[Kie73] Kiełbasiński, A.: The summation algorithm with correction and their applications. In: *Mathematica Applicanda (Matematyka Stosowana)* (1973). http://dx.doi.org/10.14708/ma.v1i1.295. – DOI 10.14708/ma.v1i1.295

[KR21] Kamra, Rabia ; Rao, S. Chandra S.: A stable parallel algorithm for block tridiagonal toeplitz–block–toeplitz linear systems. In: *Mathematics and Computers in Simulation* 190 (2021), 1415-1440. http://dx.doi.org/https://doi.org/10.1016/j.matcom.2021.07.019. – DOI https://doi.org/10.1016/j.matcom.2021.07.019. – ISSN 0378–4754

[KS73] Kogge, Peter M. ; Stone, Harold S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. In: *IEEE Transactions on Computers* C-22 (1973), Nr. 8, S. 786–793. http://dx.doi.org/10.1109/TC.1973.5009159. – DOI 10.1109/TC.1973.5009159

[Lef17] Lefèvre, Vincent: Correctly Rounded Arbitrary-Precision Floating-Point Summation. In: *IEEE Transactions on Computers* 66 (2017), S. 2111–2124. http://dx.doi.org/10.1109/TC.2017.2690632. – DOI 10.1109/TC.2017.2690632

[LGG+22] Lei, Xiaojun ; Gu, Tongxiang ; Graillat, Stef ; Jiang, Hao ; Qi, Jin: A fast parallel high-precision summation algorithm based on AccSumK. In: *Journal of Compututational and Applied Mathematics* 406 (2022), S. 113827. http://dx.doi.org/10.1016/j.cam.2021.113827. – DOI 10.1016/j.cam.2021.113827

[LH17] Lutz, David R. ; Hinds, Christopher N.: High-Precision Anchored Accumulators for Reproducible Floating-Point Summation. (2017), S. 98–105. http://dx.doi.org/10.1109/ARITH.2017.20. – DOI 10.1109/ARITH.2017.20

[LLYZ20] Liu, Zhongyun ; Li, Shan ; Yin, Yi ; Zhang, Yulin: Fast solvers for tridiagonal Toeplitz linear systems. In: *Computational and Applied Mathematics* (2020). http://dx.doi.org/10.1007/s40314-020-01369-3. – DOI 10.1007/s40314–020–01369–3

[LPNJR96] Larriba-Pey, Josep-Lluis ; Navarro, Juan J. ; Jorba, Angel ; Roig, Oriol: Review of general and Toeplitz vector bidiagonal solvers. In: *Parallel Comput.* 22 (1996), Nr. 8, S. 1091–1125

[LQC96] Lu, Mi ; Qiao, Xiangzhen ; Chen, Guanrong: A parallel algorithm for evaluating general linear recurrence equations. In: *Circuits Syst. Signal Process.* 15 (1996), Nr. 4, S. 481–504

[MGS00] McNally, J. M. ; Garey, L. E. ; Shaw, R. E.: A split-correct parallel algorithm for solving tridiagonal symmetric Toeplitz systems. In: *International Journal of Computer Mathematics* 75 (2000), Nr. 3, S. 303–313

[MGS08] McNally, Jeffrey M. ; Garey, L.E. ; Shaw, R.E.: A communication-less parallel algorithm for tridiagonal Toeplitz systems. In: *Journal of Computational and Applied Mathematics* 212 (2008), S. 260 – 271

[Mø65] Møller, Ole: Quasi Double-Precision in Floating Point Addition. In: *BIT* 5 (1965), S. 37–50. http://dx.doi.org/10.1007/BF01975722. – DOI 10.1007/BF01975722

[Mod88] Modi, J.: *Parallel Algorithms and Matrix Computation*. Oxford : Oxford University Press, 1988

[Mol] Moler, Cleve: *A Brief History of MATLAB*. https://www.mathworks.com/company/newsletters/articles/a-brief-history-of-matlab.html, . – Dostęp: 4-09-2023

[MW12] Mikołajczak, Paweł ; Ważny, Marcin: *Metody numeryczne w C++*. Uniwersytet Marii Curie-Skłodowskiej w Lublinie, 2012

[NAG] *NAG Library Manual*. https://www.nag.com/numeric/nl/nagdoc_latest/nlhtml/frontmatter/manconts.html, . – Dostęp: 4-09-2023

[NDMR20] Neuman, Brett ; Dubois, Andy ; Monroe, Laura ; Robey, Robert W.: Fast, good, and repeatable: Summations, vectorization, and reproducibility. In: *International Journal of High Performance Computing Applications* 34 (2020). http://dx.doi.org/10.1177/1094342020938425. – DOI 10.1177/1094342020938425

[nvc] *HPC SDK Documentation*. https://docs.nvidia.com/hpc-sdk//index.html, . – Dostęp: 5-01-2024

[NVI15] NVIDIA Corporation: *CUDA Programming Guide*. NVIDIA Corporation, 2015. – available at http://www.nvidia.com/

[Pan95] Pan, Xiaosu: An improved recursive doubling algorithm for the parallel solution of linear recurrence $R\langle n, 1\rangle$. In: *Trans. Nanjing Univ. Aeronaut. Astronaut.* 12 (1995), Nr. 2, S. 218–220

[Phi87] Phillips, Jen: *The NAG Library: A Beginners Guide*. USA : Oxford University Press, Inc., 1987. – ISBN 0198532636

[Pol11] Poloni, Federico.: *Algorithms for Quadratic Matrix and Vector Equations by Federico Poloni.* 1st ed. 2011. Pisa : Scuola Normale Superiore, 2011 (Theses (Scuola Normale Superiore), 16). – ISBN 88–7642–384–2

[Pot14] Potiopa, Joanna: *Algorytmy rozwiązywania wybranych typów układów równań liniowych o macierzach wąskopasmowych na komputerach z procesorami wielordzeniowymi*. Lublin : Uniwersytet Marii Curie-Skłodowskiej, 2014. – Rozprawa doktorska

[PR55] Peaceman, D. W. ; Rachford, H. H.: The Numerical Solution of Parabolic and Elliptic Differential Equations. In: *Journal of the Society for Industrial and Applied Mathematics* 3 (1955), Nr. 1, 28–41. http://www.jstor.org/stable/2098834. – ISSN 03684245

[PST17] Pas, Ruud van d. ; Stotzer, Eric ; Terboven, Christian: *Using OpenMP – The next step. Affinity, accelerators, tasking, and SIMD*. Cambridge MA : MIT Press, 2017

[QS04] Quarteroni, Alfio. ; Saleri, Fausto.: *Scientific Computing with MATLAB by Alfio Quarteroni, Fausto Saleri*. 1st ed. 2004. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004 (Texts in Computational Science and Engineering, 2). – ISBN 3–642–59339–9

[Qui03] Quinn, Michael J.: *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. – ISBN 0071232656

[RF15] Raskulinec, George M. ; Fiksman, Evgeny: Chapter 22 - SIMD Functions Via OpenMP. Version: 2015. http://dx.doi.org/https://doi.org/10.1016/B978-0-12-803819-2.00006-9. In: Reinders, James (Hrsg.) ; Jeffers, Jim (Hrsg.): *High Performance Parallelism Pearls*. Boston : Morgan Kaufmann, 2015. – DOI https://doi.org/10.1016/B978–0–12–803819–2.00006–9. – ISBN 978–0–12–803819–2, S. 421–440

[Roj90] Rojo, O.: A new method for solving symmetric circulant tridiagonal systems of linear equations. In: *Comput. Math. Appl.* 20 (1990), S. 61–67. http://dx.doi.org/10.1016/0898-1221(90)90165-G. – DOI 10.1016/0898–1221(90)90165–G

[ROO08] Rump, Siegfried M. ; Ogita, Takeshi ; Oishi, Shin'ichi: Accurate Floating-Point Summation, Part I: Faithful Rounding. In: *SIAM Journal on Scientific Computing* 31 (2008), S. 189–224. http://dx.doi.org/10.1137/050645671. – DOI 10.1137/050645671

[RSW15] Rojek, Krzysztof ; Szustak, Łukasz ; Wyrzykowski, Roman: *Zrównoleglanie i automatyczne dostosowanie algorytmów numerycznych do architektur hybrydowych z akceleratorami GPU*. Warszawa : Wydawnictwo Naukowe PWN, 2015. – ISBN 9788301181192

[Saa03] Saad, Yousef: *Iterative methods for sparse linear systems*. Second. Society for Industrial and Applied Mathematics, 2003. http://dx.doi.org/10.1137/1.9780898718003. http://dx.doi.org/10.1137/1.9780898718003

[SB11] Stpiczyński, Przemysław ; Brzuszek, Marcin: *Podstawy programowania obliczeń równoległych*. Uniwersytet Marii Curie-Skłodowskiej w Lublinie, 2011 http://informatyka.umcs.lublin.pl/files/stpiczynski.pdf

[SC17] Sunita Chandrasekaran, Guido J.: *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley, 2017

[SCB05] Scott, L. R. ; Clark, Terry ; Bagheri, Babak: *Scientific Parallel Computing*. Princeton University Press, 2005

[SGMHS17] Schmidt, B. ; Gonzalez-Martinez, J. ; Hundt, C. ; Schlarb, M.: *Parallel Programming: Concepts and Practice*. Elsevier Science, 2017 https://books.google.pl/books?id=-y9HDgAAQBAJ. – ISBN 9780128044865

[SP06] Stpiczyński, Przemysław ; Potiopa, Joanna: Piecewise Cubic Interpolation on Distributed Memory Parallel Computers and Clusters of Workstations. In: *Fifth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2006), 13-17 September 2006, Bialystok, Poland*, IEEE Computer Society, 2006. – ISBN 0–7695–2554–7, S. 284–289

[SP11] Stpiczyński, Przemysław ; Potiopa, Joanna: Solving a kind of boundary-value problem for ordinary differential equations using Fermi - The next generation CUDA computing architecture. In: *Journal of Computational and Applied Mathematics* 236 (2011), Nr. 3, S. 384–393. http://dx.doi.org/10.1016/j.cam.2011.07.028. – DOI 10.1016/j.cam.2011.07.028

[Spä95] Späth, Helmuth: *One Dimensional Spline Interpolation Algorithms*. 1st ed. New York : A K Peters/CRC Press., 1995. – ISBN 9780429065378

[Stp92] Stpiczyński, Przemysław: Parallel Algorithms for Solving Linear Recurrence Systems. In: *Parallel Processing: CONPAR 92 - VAPP V, Second Joint International Conference on Vector and Parallel Processing, Lyon, France, September 1-4, 1992, Proceedings* Bd. 634, Springer, 1992 (Lecture Notes in Computer Science), S. 343–348

[Stp93] Stpiczyński, Przemysław: Error Analysis of Two Parallel Algorithms for Solving Linear Recurrence Systems. In: *Parallel Computing* 19 (1993), Nr. 8, S. 917–923. http://dx.doi.org/10.1016/0167-8191(93)90074-U. – DOI 10.1016/0167–8191(93)90074–U

[Stp04] Stpiczyński, Przemysław: Solving Linear Recurrence Systems Using Level 2 and 3 BLAS Routines. In: *Parallel Processing and Applied Mathematics, 5th International Conference, PPAM 2003, Czestochowa, Poland, September 7-10, 2003. Revised Papers* Bd. 3019, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–21946–3, S. 1059–1066

[Stp08] Stpiczyński, Przemysław: *Optymalizacja obliczeń rekurencyjnych na komputerach wektorowych i równoległych*. Silesian University of Technology Press, 2008 http://studiainformatica.polsl.pl/index.php/SI/article/download/521/520

[Stp16] Stpiczyński, Przemysław: Semiautomatic Acceleration of Sparse Matrix-Vector Product Using OpenACC. In: *Parallel Processing and Applied Mathematics, 11th International Conference, PPAM 2015, Kracow, Poland, September 6-9, 2015, Revised Selected Papers, Part II* Bd. 9574, Springer, 2016 (Lecture Notes in Computer Science), S. 143–152

[Stp18] Stpiczyński, Przemysław: Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. In: *The Journal of Supercomputing* 74 (2018), Nr. 4, S. 1461–1472. http://dx.doi.org/10.1007/s11227-017-2231-3. – DOI 10.1007/s11227–017–2231–3

[Stp20] Stpiczyński, Przemysław: Algorithmic and language-based optimization of Marsa-LFIB4 pseudorandom number generator using OpenMP, OpenACC and CUDA. In: *Journal of Parallel and Distributed Computing* 137 (2020), S. 238–245. http://dx.doi.org/10.1016/j.jpdc.2019.12.004. – DOI 10.1016/j.jpdc.2019.12.004

[STRP18] Stojanov, Alen ; Toskov, Ivaylo ; Rompf, Tiark ; Pueschel, Markus: SIMD Intrinsics on Managed Language Runtimes. (2018), S. 2–15. http://dx.doi.org/10.1145/3168810. – DOI 10.1145/3168810

[Sun01] Sun, Xian-He: Progress in Computer Research. Version: 2001. http://dl.acm.org/citation.cfm?id=644339.644351. Commack, NY, USA : Nova Science Publishers, Inc., 2001. – ISBN 1–59033–011–0, Kapitel A Scalable Parallel Algorithm for Periodic Symmetric Toeplitz Tridiagonal Systems, 149–156

[Ter16] Terekhov, Andrew V.: A highly scalable parallel algorithm for solving Toeplitz tridiagonal systems of linear equations. In: *J. Parallel Distrib. Comput.* 87 (2016), S. 102–108. http://dx.doi.org/10.1016/j.jpdc.2015.10.004. – DOI 10.1016/j.jpdc.2015.10.004

[TM] *Toeplitz Matrix.* https://www.mathworks.com/help/matlab/ref/toeplitz.html, . – Dostęp: 4-09-2023

[TMA] *Tridiagonal Matrix Algorithm.* https://www.mathworks.com/matlabcentral/fileexchange/85438-tridiagonal-matrix-algorithm, . – Dostęp: 4-09-2023

[UDD17] Uguen, Yohann ; Dinechin, Florent de ; Derrien, Steven: Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations. (2017), S. 1–8. http://dx.doi.org/10.23919/FPL.2017.8056792. – DOI 10.23919/FPL.2017.8056792

[VA12] Vidal, Antonio M. ; Alonso, Pedro: Solving systems of symmetric Toeplitz tridiagonal equations: Rojo's algorithm revisited. In: *Applied Mathematics and Computation* 219 (2012), S. 1874–1889. http://dx.doi.org/10.1016/j.amc.2012.08.030. – DOI 10.1016/j.amc.2012.08.030

[Wac13] Wachspress, Eugene L.: *The ADI Model Problem by Eugene Wachspress.* 1st ed. 2013. New York, NY : Springer New York, 2013. – ISBN 1–4614–5122–1

[Wan81] Wang, H.H.: A parallel method for tridiagonal equations. In: *ACM Trans. Math. Softw.* 7 (1981), S. 170–183

[WE17] Walter, Heimo ; Epple, Bernd: *Numerical Simulation of Power Plants and Firing Systems.* 1st ed. 2017 edition. Vienna : Springer Wien, 2017. – ISBN 3709148537

[Wil92] William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery: *Numerical Recipes in C, 2nd Edition*. Cambridge University Press, 1992

[WWT+14] Wang, Haichuan ; Wu, Peng ; Tanase, Ilie G. ; Serrano, Mauricio J. ; Moreira, José E.: Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library. (2014), S. 9–16. http://dx.doi.org/10.1145/2568058.2568059. – DOI 10.1145/2568058.2568059

[YC94] Yan, W. M. ; Chung, K. L.: A fast algorithm for solving special tridiagonal systems. In: *Computing* (1994). http://dx.doi.org/10.1007/BF02238076. – DOI 10.1007/BF02238076