Sprawozdanie z Laboratorium Projektowania Systemów Informatycznych

Tomasz Lech
Krzysztof Niemiec

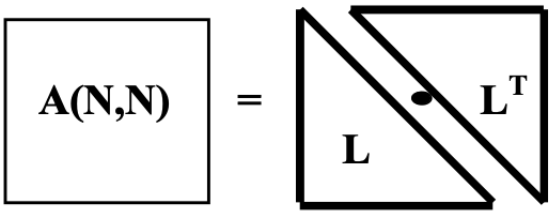
Informatyka
specjalność PKiSI
Semestr V

Koszalin, 22.01.2024

Wprowadzenie

Celem projektu było zaprojektowanie architektury równoległej dla wybranego algorytmu algebry liniowej oraz porównanie parametrów zaprojektowanej architektury z parametrami wykonania danego algorytmu na CPU.

Algorytmem realizowanym przez naszą grupę labolatoryjną był algorytm rozkładu macierzy metodą Cholesky'ego.



Rys 1. Wizualizacja działania dekompozycji Cholesky'ego.

Wynikiem rozkładu Cholesky'ego wykonanego na macierzy A jest macierz trójkątna dolna L, która spełnia następującą zależność:

$$A = LL^T \tag{1}$$

Implementacja algorytmu na CPU

Pierwszą fazą projektu była algorytmu w wybranym języku programowania oraz zmierzenie czasu potrzebnego na wykonanie owego algorytmu na CPU. Razem z poleceniem do zadania otrzymaliśmy pseudokod realizujący rozkład Cholesky'ego:

Listing 1. Pseudokod dla rozkładu Cholesky'ego

```
for i := 1 to N do
begin
a_ii := SQRT(a_ii);
   for j := i+1 to N do
      a_ji := a_ji / a_ii;
   for j := i+1 to N do
      for k := i+1 to j do
         a_jk := a_jk - a_ji * a_ki;
end;
```

Rozkład Cholesky'ego zrealizowaliśmy w języku C++. Pseudokod algorytmu przepisany do składni C++ jest pokazany na listingu 2.

Listing 2. Kod rozkładu Cholesky'ego w C++

```
typedef std::vector<float> vf;
typedef std::vector<vf> vvf;

vvf cholesky(vvf a, int n) {
    for(int i = 0; i < n; i++) {

        a[i][i] = sqrt(a[i][i]);

        for(int j = i+1; j < n; j++) {
            a[j][i] = a[j][i]/a[i][i];
        }

        for(int j = i+1; j < n; j++) {
            for(int k = i+1; k <= j; k++) {
                a[j][k] = a[j][k] - a[j][i]*a[k][i];
            }
        }
    }

    vvf l(n, vf(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(i >= j) {
                l[i][j] = a[i][j];
            }
        }
    }
    return l;
}
```

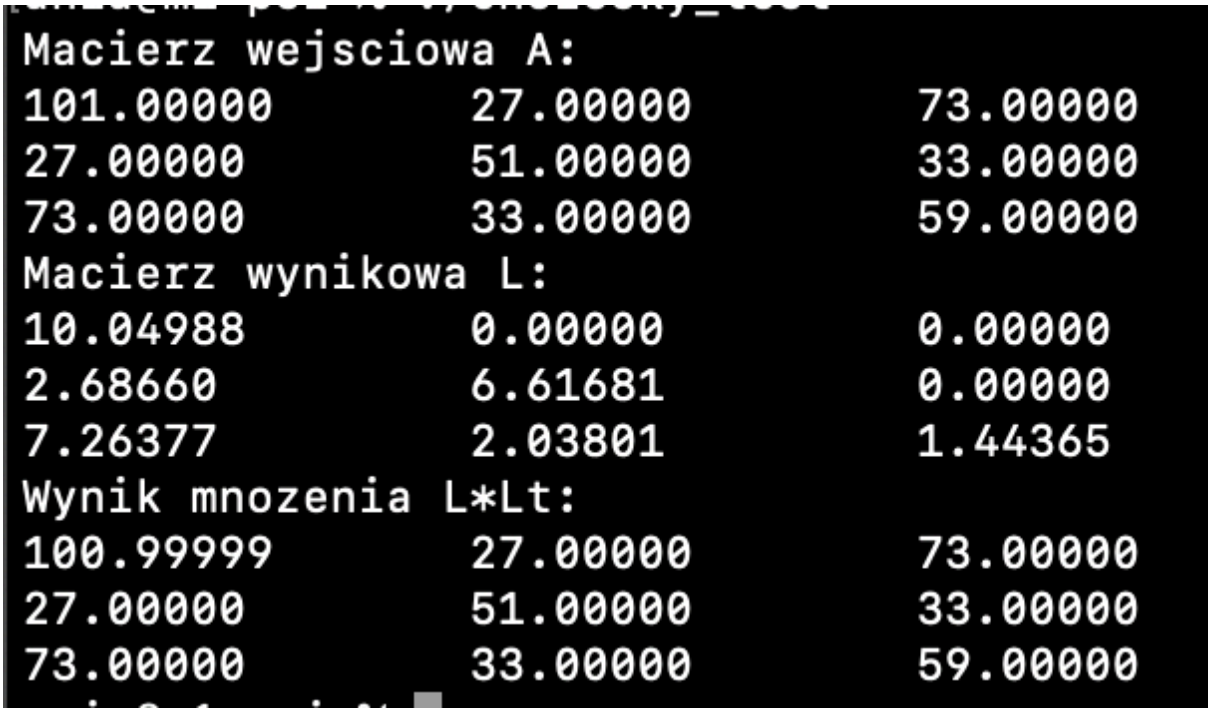
Do generowania danych testowych również wykorzystaliśmy język C++. Rozkład Cholesky'ego można wykonać jedynie na pewnej klasie macierzy zwanymi macierzami symetrycznymi pozytywnie zdefiniowanymi (SPD). Do generowania przypadków testowych wykorzystaliśmy zależność, że dowolna macierz przemnożona przez jej macierz odwrotną jest macierzą SPD[1]. Kod generujący losowe macierze testowe jest pokazany na listingu 3.

[1] https://stackoverflow.com/questions/48736724/generate-matrix-symmetric-and-positive-definite

Listing 3. Kod generujący losową macierz SPD

```
#define RAND_A -10
#define RAND_B 10
typedef std::vector<float> vf;
typedef std::vector<vf> vvf;

int rand_range() {
    return (RAND_A)+std::rand()%((RAND_B)-(RAND_A));
}

vvf matmul_nxn(vvf a, vvf b, int n) {
    vvf res(n, vf(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for(int k = 0; k < n; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return res;
}

vvf inverse(vvf a, int n) {
    vvf res(n, vf(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            res[i][j] = a[j][i];
        }
    }
    return res;
}

vvf random_spd_matrix(int n) {
    vvf res;
    for(int i = 0; i < n; i++) {
        vf row(n);
        std::generate(row.begin(), row.end(), rand_range);
        res.push_back(row);
    }

    vvf res_inv = inverse(res, n);
    res = matmul_nxn(res, res_inv, n);
    return res;
}
```

Ze względu na zależność (1), poprawność algorytmu można zweryfikować poprzez mnożenie macierzy L oraz L^T. Poprawność ta została empirycznie sprawdzona na niewielkich instancjach problemu.



Rys 2. Sprawdzenie poprawności algorytmu. Wartość 100.99999 w macierzy LL^T wynika z błędu użytego typu `float`.

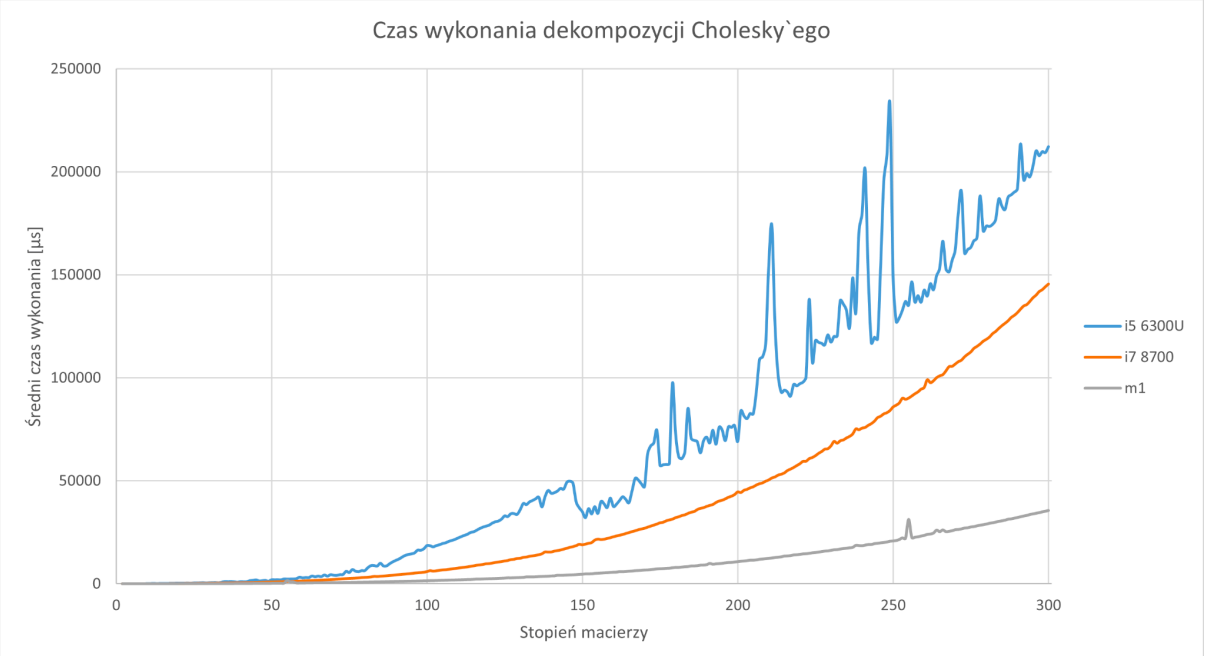
W celach pomiaru prędkości wykonania algorytmu, zmierzyliśmy czas wykonania funkcji `cholesky()` na trzech różnych urządzeniach. Badane procesory to:

- Intel Core i5-6300U
- Intel Core i7-8700
- Apple M1

Algorytm sprawdzono na macierzach o stopniu od 2 do 300. Dla każdego stopnia macierzy wykonano 10 pomiarów i policzono ich średnią. Czas wykonania algorytmu zmierzono w mikrosekundach przy pomocy biblioteki `chrono` języka C++. Funkcję `cholesky_benchmark()` przedstawiono na listingu 4. Porównanie czasów na powyższych procesorach przedstawiono na wykresie na rysunku 3.

Listing 4. Funkcja `cholesky_benchmark()`

```
typedef std::vector<float> vf;
typedef std::vector<vf> vvf;

void cholesky_benchmark(int rank, int rep_per_rank) {
    for(int n = 2; n <= rank; n++) {
        for(int i = 0; i < rep_per_rank; i++ ){
            vvf a = random_spd_matrix(n);
            auto start =
std::chrono::high_resolution_clock::now();
            vvf l = cholesky(a, n);
            auto stop = std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(stop -
start);
            std::cout << duration.count() << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```



Rys 3. Porównanie czasu wykonania funkcji `cholesky()` w zależności od stopnia macierzy wejściowej na badanych procesorach.

Pseudokod został poprawnie zaimplementowany w języku C++. Czas wykonania algorytmu został zmierzony oraz porównany między trzema popularnymi procesorami. Wykres odzwierciedla wielomianową złożoność algorytmu w każdym przypadku.

Interesującym zjawiskiem jest pojawienie się nieprecyzyjności w pomiarach dla większych wartości stopnia macierzy. Potencjalnym wytłumaczeniem jest fakt, że pomiar był różnicą między czasem wywołania funkcji a czasem wyjścia z funkcji - w tym czasie system operacyjny mógł potencjalnie wywłaszczyć wątek na swoje cele, co nie jest uwzględnione w pomiarze. Takie wywłaszczenie ma większą szansę zajścia w przypadku dłuższego wykonywania się funkcji, więc zachodzi częściej dla większego stopnia macierzy.

Wygenerowanie architektury równoległej

Pierwszym krokiem w generowaniu architektury jest rozbicie programu na gniazda pętli programowych. W omawianym algorytmie znajdują się trzy typy instrukcji, więc został rozbity na trzy gniazda:

Listing 5. Gniazda pętli programowych

```
for(int i = 0; i < n; i++) {
    for(int j = i; j <= i; j++) {
        for(int k = i; k <= i; k++) {
            a[i][i] = sqrt(a[i][i]);
        }
    }
}
for(int i = 0; i < n; i++) {
    for(int j = i+1; j < n; j++) {
        for(int k = j; k <= j; k++) {
            a[j][i] = a[j][i]/a[i][i];
        }
    }
}
for(int i = 0; i < n; i++) {
    for(int j = i+1; j < n; j++) {
        for(int k = i+1; k <= j; k++) {
            a[j][k] = a[j][k] - a[j][i]*a[k][i];
        }
    }
}
```

Instrukcje, które są płyciej zagnieżdżone (np. instrukcja `sqrt`) zostały osadzone w “sztucznych” pętlach tak, aby stopień ich zagnieżdżenia był taki sam jak innych instrukcji.

Po rozbiciu algorytmu na gniazda pętli, do każdego gniazda dodano informację o indeksach pętli (współrzędnych), wykorzystywanych indeksach macierzy A oraz typie instrukcji i zestawiono ją w tabeli. Wynik działania programu wykonującego taką tabelę dla $N = 4$ wygląda następująco:

id	i j k	a(i,i)	a(j,i)	a(j,k)	a(k,i)	optype
0	1 1 1	1 1				sqrt
1	1 2 1	1 1	2 1			div
2	1 2 2		2 1	2 2	2 1	sub-mul
3	1 3 1	1 1	3 1			div
4	1 3 2		3 1	3 2	2 1	sub-mul
5	1 3 3		3 1	3 3	3 1	sub-mul
6	1 4 1	1 1	4 1			div
7	1 4 2		4 1	4 2	2 1	sub-mul
8	1 4 3		4 1	4 3	3 1	sub-mul
9	1 4 4		4 1	4 4	4 1	sub-mul
10	2 2 2	2 2				sqrt
11	2 3 2	2 2	3 2			div
12	2 3 3		3 2	3 3	3 2	sub-mul
13	2 4 2	2 2	4 2			div
14	2 4 3		4 2	4 3	3 2	sub-mul
15	2 4 4		4 2	4 4	4 2	sub-mul
16	3 3 3	3 3				sqrt
17	3 4 3	3 3	4 3			div
18	3 4 4		4 3	4 4	4 3	sub-mul
19	4 4 4	4 4				sqrt

Rys 4. Tabela pomocnicza dla macierzy 4×4.

Tabela pomocnicza służy do utworzenia grafu zależności informacyjnych algorytmu (nazywany później grafem DAG (*ang. directed acyclic graph)*, ponieważ spełnia warunek acykliczności oraz jest skierowany). Graf $G = (V, E)$ to parą zbioru wierzchołków oraz zbioru łuków, i programowo zrealizowano tą reprezentację poprzez zapisanie grafu DAG jako listy łuków oraz listy węzłów. Węzły odpowiadają określonym operacjom w algorytmie, a łuki odpowiadają zależnościom między tymi operacjami - tj. jeżeli dwie operacje korzystają z tej samej wartości w danej macierzy, to operacja późniejsza musi koniecznie wykonać się po wcześniejszej (w przeciwnym wypadku można wykonać je równolegle).

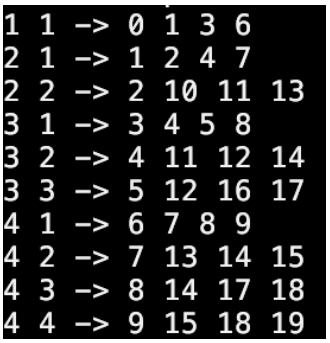
Lista węzłów określa współrzędne danej operacji w przestrzeni określonej poprzez indeksy iteracji pętli (wartości `i`, `j`, `k` można interpretować jako współrzędne w 3-wymiarowej przestrzeni) oraz typ tej operacji. Wartości te są zapisane w drugiej i ostatniej kolumnie tabeli pomocniczej.

Generowanie listy łuków odbywa się poprzez analizę kolumn 3-6. Każdy wpis w kolumnie zawiera parę indeksów, z których korzysta operacja. Algorytm generowania łuków na podstawie tej informacji wygląda następująco:

Listing 5. Generowanie łuków

```
edge_list generate_edge_list(table t) {
    edge_list res;
    indexes empty_indexes = {0, 0};

    std::map<indexes, std::set<int>> res_map;

    for(int i = 0; i < t.size(); i++) {
        indexes_list r_ix = std::get<1>(t[i]);
        for(auto x : r_ix) {
            if(x != empty_indexes) {
                res_map[x].insert(i);
            }
        }
    }

    for(auto m : res_map) {
        vi paths;
        for(auto s : m.second) {
            paths.push_back(s);
        }
        for(int i = 0; i < paths.size()-1; i++) {
            res.push_back({paths[i], paths[i+1]});
        }
    }
    return res;
}
```

Algorytm iteruje po każdym wierszu w tabeli i dynamicznie tworzy listę użytych elementów tablicy A, przypisując w tym czasie do każdego wiersza w liście indeks operacji, która korzysta z danego elementu A. Po zakończeniu iteracji, algorytm przeszukuje listę użytych elementów i na jej podstawie generuje łuki.



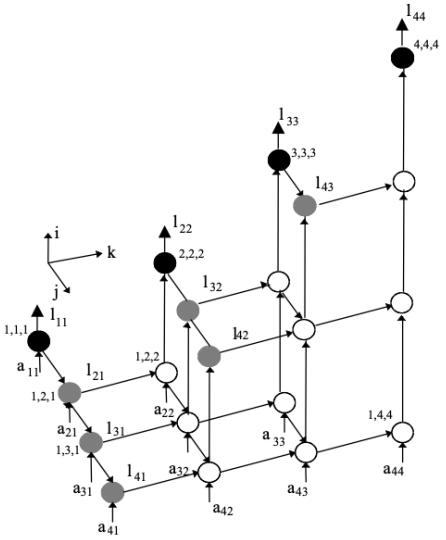
Rys 5. Wizualizacja działania algorytmu tworzenia łuków. Na podstawie wiersza `1 1` zostaną utworzone łuki `0->1`, `1->3` oraz `3->6`.

Lista węzłów oraz lista łuków dla rozmiaru $N = 4$ wygląda następująco:

id	i	j	k	optype
0	1	1	1	sqrt
1	1	2	1	div
2	1	2	2	sub-mul
3	1	3	1	div
4	1	3	2	sub-mul
5	1	3	3	sub-mul
6	1	4	1	div
7	1	4	2	sub-mul
8	1	4	3	sub-mul
9	1	4	4	sub-mul
10	2	2	2	sqrt
11	2	3	2	div
12	2	3	3	sub-mul
13	2	4	2	div
14	2	4	3	sub-mul
15	2	4	4	sub-mul
16	3	3	3	sqrt
17	3	4	3	div
18	3	4	4	sub-mul
19	4	4	4	sqrt

src id	dst id
0	1
1	3
3	6
1	2
2	4
4	7
2	10
10	11
11	13
3	4
4	5
5	8
4	11
11	12
12	14
5	12
12	16
16	17
6	7
7	8
8	9
7	13
13	14
14	15
8	14
14	17
17	18
9	15
15	18
18	19

Rys 6. Lista węzłów (po lewej) oraz lista łuków (po prawej) dla $N = 4$.



Rys 7. Graf DAG dla algorytmu rozkładu Cholesky'ego dla macierzy rzędu $N = 4$.

Alternatywną metodą generowania listy łuków jest szukanie powtarzających się wartości w danej kolumnie, sprawdzając parami węzły, i powtórzenie procedury dla każdej kolumny. Metoda ta daje jednak niepoprawne wyniki dla badanego algorytmu (tj. nie wszystkie łuki są generowane), i choć można temu zapobiec sprawdzając wartości między kolumnami (a nie tylko w obrębie jednej), to podejście jest mało generalizowalne na inne badane algorytmy i konkretne pary kolumn do przeszukania należy znaleźć ręcznie.

Metoda wykorzystana w projekcie posiada dodatkowo lepszą złożoność obliczeniową (iteruje po pętli liniowo w przeciwieństwie do kwadratowego przeszukiwania par węzłów, które mogą korzystać z tych samych danych z macierzy). To znacząco wpłynęło na czas generowania grafu; metoda użyta finalnie pozwala wygenerować graf dla $N = 100$ w mniej niż sekundę, a metoda alternatywna nie wykonała się dla tego samego rozmiaru po upływie 20 minut.

Kolejnym krokiem po wygenerowaniu grafu DAG jest wyznaczenie funkcji odwzorowania F_s, która przeniesie współrzędne z grafu na strukturę wykonującą dane zadanie obliczeniowe. Funkcja odwzorowania $F_s: K^n \rightarrow K_F^m, m < n$ przekształca współrzędne z grafu DAG na współrzędne elementu procesującego EP w zadanej strukturze. W praktyce funkcję F_s można zrealizować jako macierz $m \times n$, a współrzędne operacji k' w generowanej strukturze otrzymuje się równaniem

$$k' = F_s k \tag{2}$$

(k - współrzędne operacji w grafie)

Projektowana architektura jest architekturą systoliczną. Oznacza to, że musi być zachowany warunek lokalności tj. jeżeli między dwoma węzłami w grafie DAG jest łuk, to węzły po odwzorowaniu muszą znajdować się w sąsiednich elementach procesujących; inaczej, współrzędne k' dla obu węzłów muszą różnić się co najwyżej o jedną jednostkę.

Warunek lokalności można zapewnić poprzez sprawdzenie, czy dla każdego wektora reprezentującego łuk współrzędne zmieniają się o wartość ze zbioru $\{-1, 0, 1\}$. Wektory reprezentujące łuk są zapisane w macierzy D. Matematycznie warunek można opisać następująco:

$$\forall d \in D' \colon d \in \{-1, 0, 1\}, D' = F_s D \tag{3}$$

Warto zauważyć, że dla algorytmu Cholesky'ego macierz D jest następująca:

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Jest to macierz jednostkowa, która nie zmienia wartości innej macierzy przy operacji mnożenia. Konsekwencją tego jest fakt, że dowolnie dobrana macierz F_s będzie spełniała warunek lokalności dla algorytmu Cholesky'ego tak długo, jak każda wartość w tej macierzy będzie elementem zbioru $\{-1, 0, 1\}$.

Wartości funkcji F_s wybrane jako funkcje odwzorowania są następujące:

- Architektura 1:

$$F_1 = \begin{bmatrix} -1 & -1 & 0 \\ 1 & -1 & 0 \end{bmatrix}$$

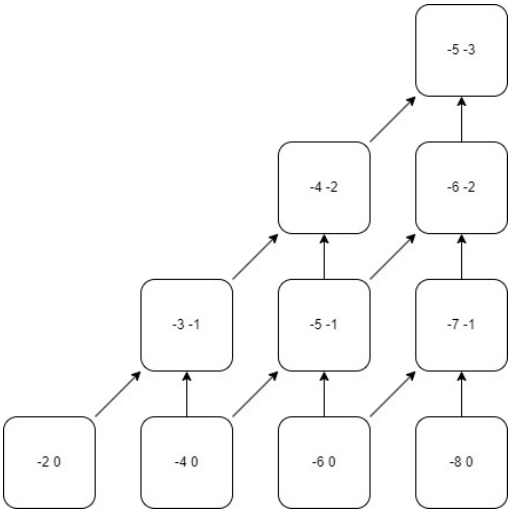
- Architektura 2:

$$F_2 = \begin{bmatrix} -1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

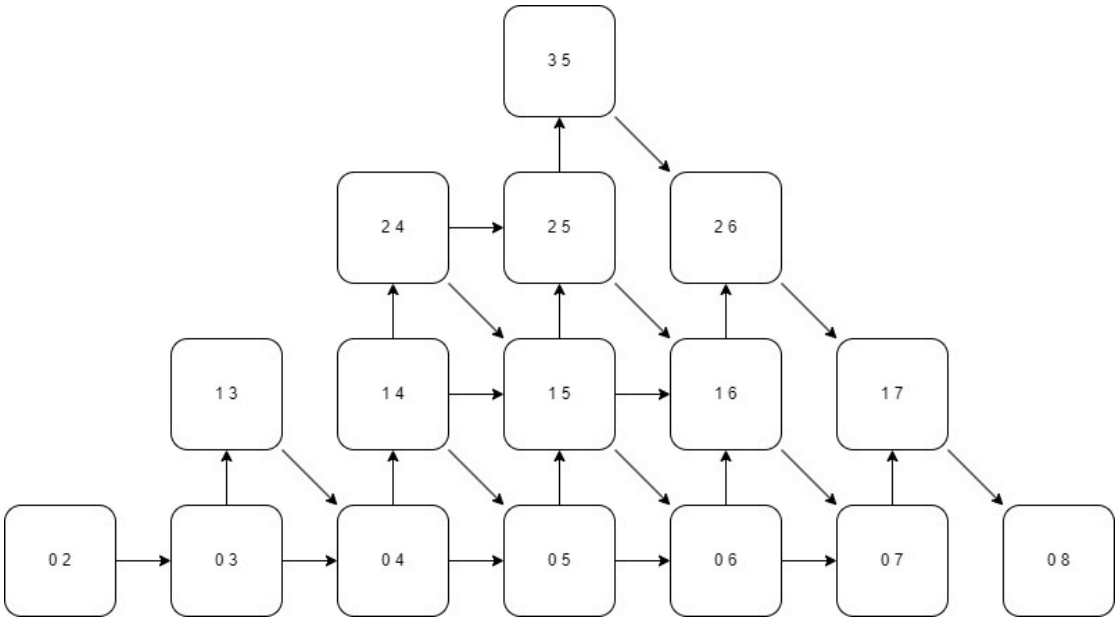
- Architektura 3:

$$F_3 = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}$$

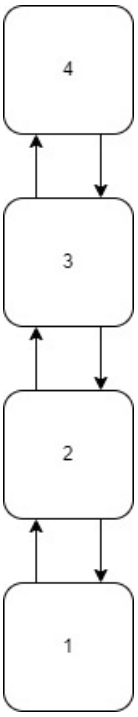
Struktury dla $N = 4$ wyglądają następująco:



Rys 8. Struktura 1



Rys 9. Struktura 2



Rys 10. Struktura 3

Po wygenerowaniu struktury, należy przypisać operacjom numer taktu, w którym dana operacja ma się wykonać. Takie przypisanie jest obarczone dwoma ograniczeniami:

- Jeżeli dwa węzły połączone są łukiem, to operacja z końca łuku musi wykonać się w takcie późniejszym niż operacja z początku łuku,
- Jeżeli dwie operacje po odwzorowaniu znajdują się w tym samym elemencie procesującym, to nie mogą one wykonywać się w tym samym takcie.

Takty w projekcie przypisywane są algorytmem zachłannym, który iteruje po wierzchołkach i przypisuje wierzchołkom połączonym łukiem z wierzchołkiem procesowanym najmniejszy możliwy takt, uwzględniając wymienione wyżej ograniczenia.

Pomiar parametrów wygenerowanych architektur

Trzy podane wyżej architektury przetestowano pod kątem ilości użytych elementów procesujących, liczby taktów, czasu obliczeniowego potrzebnego na wykonanie algorytmu, oraz średniego obciążenia elementu. Obciążenie P zostało policzone ze wzoru:

$$P = \frac{L_{op}}{L_{EP} \cdot T} \tag{5}$$

(L_{op} - liczba operacji w algorytmie, L_{EP} - liczba elementów procesujących, T - liczba taktów)

Czas w sekundach t został wyliczony teoretycznie ze wzoru:

$$t = \frac{l+T}{f} \tag{6}$$

(l - latencja układu, T - liczba taktów, f - częstotliwość)

Latencja układu została obliczona jako długość najdłuższej ścieżki w strukturze razy 28. Jest to szacunkowa wartość dla pewnego układu FPGA. Długość najdłuższej ścieżki w strukturze wynosi $2 \cdot N - 1$ dla architektury 1 oraz architektury 2 oraz N dla architektury 3. Częstotliwość jest równa 320 Mhz i jest to również wartość dla wspomnianego wyżej układu FPGA.

Tab 1. Parametry architektury 1

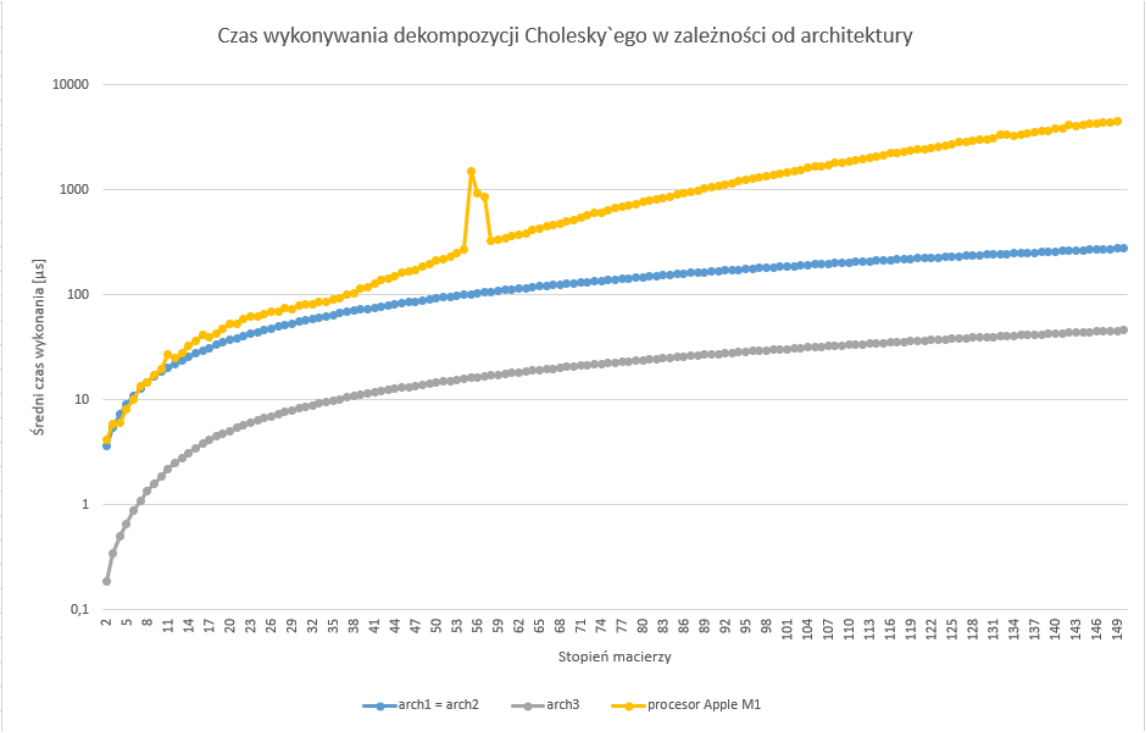
Architektura 1					
rozmiar N	ilość węzłów	liczba EP	ilość taktów	wartość p [%]	czas [µs]
10	220	55	28	14,2857	9,625
20	1540	210	58	12,6437	36,84375
30	4960	465	88	12,1212	81,5625
40	11480	820	118	11,8644	143,78125
50	22100	1275	148	11,7117	223,5
60	37820	1830	178	11,6105	320,71875
70	59640	2485	208	11,5385	435,4375
80	88560	3240	238	11,4846	567,65625
90	125580	4095	268	11,4428	717,375
100	171700	5050	298	11,4094	884,59375

Tab 2. Parametry architektury 2

Architektura 2					
rozmiar N	ilość węzłów	liczba EP	ilość taktów	wartość p [%]	czas [µs]
10	220	100	28	7,85714	17,5
20	1540	400	58	6,63793	70,09375
30	4960	900	88	6,26263	157,6875
40	11480	1600	118	6,08051	280,28125
50	22100	2500	148	5,97297	437,875
60	37820	3600	178	5,902	630,46875
70	59640	4900	208	5,85165	858,0625
80	88560	6400	238	5,81408	1120,65625
90	125580	8100	268	5,78496	1418,25
100	171700	10000	298	5,76174	1750,84375

Tab 3. Parametry architektury 3

Architektura 3					
rozmiar N	ilość węzłów	liczba EP	ilość taktów	wartość p [%]	czas [µs]
10	220	10	50	44	1,03125
20	1540	20	142	54,2254	2,19375
30	4960	30	233	70,9585	3,353125
40	11480	40	324	88,5802	4,5125
50	22100	50	414	106,763	5,66875
60	37820	60	504	125,066	6,825
70	59640	70	594	143,434	7,98125
80	88560	80	684	161,842	9,1375
90	125580	90	774	180,276	10,29375
100	171700	100	864	198,727	11,45



Rys 11. Porównanie czasu obliczeń między wygenerowanymi architekturami a CPU.

Podsumowanie

W ramach projektu zrealizowano program generujący architektury równoległe realizujące algorytm rozkładu macierzy metodą Cholesky'ego. Zbadano je również pod kątem wydajności, porównując z naiwną implementacją algorytmu na CPU. Projektowane architektury wykonują zadanie obliczeniowe ponad rząd wielkości szybciej, a ich przewaga jest tym większa, im większy jest rozmiar danych wejściowych. Projekt pokazuje, jak wykorzystanie architektur równoległych pozwala na przyspieszenie obliczeń w znaczący sposób.