

**Adam K. Dziekoński**

**Optymalizacja wydajności obliczeniowej  
metody elementów skończonych  
w architekturze CUDA**

Rozprawa doktorska

Promotor:

prof. dr hab. inż. Michał Mrozowski,  
prof. zw. PG  
Wydział Elektroniki, Telekomunikacji  
i Informatyki  
Politechnika Gdańska

Promotor pomocniczy:

dr inż. Adam Lamęcki  
Wydział Elektroniki, Telekomunikacji  
i Informatyki  
Politechnika Gdańska

Gdańsk, 2015



# Spis treści

<b>Wykaz ważniejszych symboli i oznaczeń</b>	<b>5</b>
<b>1 Wprowadzenie</b>	<b>7</b>
1.1 Obecny stan wiedzy . . . . .	10
1.2 Cel i tezy rozprawy . . . . .	12
1.3 Plan rozprawy . . . . .	13
<b>2 Metoda elementów skończonych</b>	<b>15</b>
2.1 Lokalne macierze elementów . . . . .	20
2.2 Konstrukcja macierzy globalnych . . . . .	23
2.3 Rozwiązanie układu równań . . . . .	25
2.3.1 Hierarchiczny wielopoziomowy operator ściskający . . . . .	29
2.4 Podsumowanie metody elementów skończonych . . . . .	32
<b>3 Architektura CUDA</b>	<b>35</b>
<b>4 Budowa macierzy sztywności i bezwładności</b>	<b>41</b>
4.1 Podstawowy wariant generacji macierzy na pojedynczym akceleratorze .	43
4.1.1 Całkowanie numeryczne . . . . .	43
4.1.1.1 Całkowanie numeryczne z wykorzystaniem mieszanych kwadratur Gaussa . . . . .	45
4.1.1.2 Całkowanie w przypadku innych typów ośrodków . . .	50
4.1.2 Strategie masywnego zrównoleglenia wątków w składaniu globalnych macierzy sztywności i bezwładności na GPU . . . . .	54
4.1.2.1 Składanie macierzy globalnych w formacie COO . . . .	56
4.1.2.2 Konwersja z formatu COO do CRS z eliminacją duplikatów . . . . .	59
4.1.3 Podsumowanie podstawowego wariantu generacji macierzy na GPU.	64
<b>5 Konstrukcja dużych globalnych macierzy sztywności i bezwładności</b>	<b>67</b>
5.1 Iteracyjny wariant konstrukcji macierzy globalnych dla kilku akceleratorów graficznych . . . . .	72
5.2 Podsumowanie zrównoleglenia generacji globalnych macierzy sztywności i bezwładności . . . . .	75
<b>6 Metody rozwiązywania układów równań</b>	<b>77</b>
6.1 Metoda bezpośredniego rozwiązywania układu równań . . . . .	77
6.2 Metoda iteracyjnego rozwiązywania układu równań . . . . .	81

6.2.1	Mnożenie macierzy rzadkiej przez wektor na akceleratorze graficznym . . . . .	83
6.2.2	Sliced ELLR-T - nowy formatu zapisu macierzy rzadkiej . . . .	86
6.2.2.1	Wydażność formatu Sliced ELLR-T . . . . .	89
6.2.2.2	Rozwój formatu Sliced ELLR-T . . . . .	91
6.2.3	Rozwiązanie problemu liniowego w operatorze ściskającym . . .	91
6.3	Implementacja metody PCG z wielopoziomowym operatorem ściskającym dla jednego akceleratora graficznego . . . . .	95
6.3.1	Analiza układu w paśmie częstotliwości . . . . .	104
<b>7</b>	<b>Strategie iteracyjnego rozwiązywania układów równań dla kilku GPU</b>	<b>109</b>
7.1	Wariant z podziałem macierzy ze względu na wiersze . . . . .	109
7.2	Wariant z podziałem macierzy ze względu na dziedziny obliczeniowe . .	111
7.3	Podsumowanie strategii iteracyjnego rozwiązywania układów równań dla kilku GPU . . . . .	116
<b>8</b>	<b>Pełny cykl analizy z użyciem MES na CPU i GPU</b>	<b>121</b>
<b>9</b>	<b>Podsumowanie</b>	<b>127</b>
<b>A</b>	<b>Metoda elementów skończonych</b>	<b>131</b>
A.1	Simpleksowy układ współrzędnych . . . . .	131
A.2	Metoda residuów ważonych . . . . .	133
A.3	Element Whitneya . . . . .	134
A.4	Hierarchiczne funkcje bazowe - element Webba . . . . .	135
<b>B</b>	<b>Programowanie GPU</b>	<b>137</b>
B.1	Skuteczne programowanie GPU . . . . .	137
B.2	Przykład uruchomienia obliczeń na GPU. . . . .	138
<b>C</b>	<b>Schematy, kody i pseudokody implementacji na GPU</b>	<b>141</b>
<b>D</b>	<b>FSMA - algorytm sumowania macierzy rzadkich</b>	<b>153</b>
<b>E</b>	<b>Konwersja macierzy z formatu COO do formatów CCS lub CRS</b>	<b>155</b>
E.1	Konwersja sekwencyjna . . . . .	155
E.2	Zrównoleglenie konwersji z biblioteki UMFPACK . . . . .	159
	<b>Podziękowania</b>	<b>161</b>
	<b>Bibliografia</b>	<b>162</b>
	<b>Prawo rozpowszechniania</b>	<b>173</b>
	<b>Sylwetka autora</b>	<b>175</b>

# Wykaz ważniejszych symboli i oznaczeń

## Symbole matematyczne:

$\mathbf{A}, \mathbf{B}, \mathbf{C}, \dots$  - macierze

$\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots$  - wektory kolumnowe

$\mathbf{I}$  - macierz jednostkowa

$\mathbf{A}_{m \times n}$  - macierz zawierające  $m$  wierszy i  $n$  kolumn

$\mathbf{A}^T$  - transpozycja macierzy  $\mathbf{A}$

$\mathbf{a}_{ij}$  - element macierzy  $\mathbf{A}$ , znajdujący się w  $i$ -tym wierszu i  $j$ -tej kolumnie

$\mathbf{a}_i$  - element wektora  $\mathbf{a}$ , znajdujący się w  $i$ -tym wierszu

$(\cdot)^T$  - transpozycja macierzy lub wektora

$(\cdot)^{-T}$  - transpozycja odwrotności macierzy

$\overline{(\cdot)}$  - sprzężenie macierzy lub wektora zespolonego

$\Lambda$  - wartość własna macierzy

$\|\cdot\|$  - norma

$\|\cdot\|_2$  - norma euklidesowa

$|\cdot|$  - wartość bezwzględna

## Symbole wielkości fizycznych oraz stałe matematyczne:

$\vec{E}$  - wektor natężenia pola elektrycznego

$\vec{H}$  - wektor natężenia pola magnetycznego

$k_0$  - liczba falowa

$c = 299792458 \text{ [m/s]}$  - prędkości światła w próżni

$\pi = 3,1415926535$  - liczba Pi

$\varepsilon$  - przenikalność elektryczna

$\mu$  - przenikalność magnetyczna

## Symbole użyte w opisie metody elementów skończonych:

$\vec{N}$  - wektorowe funkcje bazowe

$\vec{W}$  - wektorowe funkcje wagowe

$N_{dofs}$  - liczba stopni swobody

$K$  - liczba funkcji bazowych

$\Omega$  - objętość dziedziny obliczeniowej

$V$  - objętość czworościanu

$R$  - rząd kwadratury Gaussa

$Q$  - liczba punktów kwadratury Gaussa

$\lambda$  - współrzędna w układzie simpleksowym

$f$  - częstotliwość

$f_{bw}$  - pasmo częstotliwości

$f_m$  - liczba częstotliwości w paśmie  
 $f_{pt}$  - indeks częstotliwości w paśmie  
 $S^{(e)}$  - lokalna macierz sztywności  
 $T^{(e)}$  - lokalna macierz bezwładności  
 $SE$  - grupa lokalnych macierzy elementów sztywności  
 $TE$  - grupa lokalnych macierzy elementów bezwładności  
 $S$  - globalna macierz sztywności  
 $T$  - globalna macierz bezwładności

### Skróty:

**ALU** - jednostka arytmetyczno-logiczna (ang. Arithmetic Logic Unit)  
**CEM** - elektrodynamika obliczeniowa (ang. Computational Electromagnetics)  
**CG** - metoda gradientów sprzężonych (ang. Conjugate Gradient)  
**COO** - format przechowywania macierzy rzadkiej w postaci wektorów: wartości niezerowych, indeksów kolumn i indeksów wierszy (ang. Coordinate format)  
**CRS** - format przechowywania macierzy rzadkiej w porządku wierszowym w postaci wektorów: wartości niezerowych, indeksów kolumn i skompresowanego wektora indeksów wierszy (ang. Compressed Row Storage)  
**CPU** - procesor centralny (ang. Central Processing Unit)  
**CUDA** - softwarowa i hardwarowa architektura dedykowana obliczeniom na GPU (ang. Compute Unified Device Architecture)  
**DG-TD** - nieciągła metoda Galerkina w dziedzinie czasu (ang. Discontinuous Galerkin Time Domain)  
**DOF** - stopień swobody (ang. Degree of Freedom)  
**DtoD** - przesłanie danych z pamięci GPU (Device) do pamięci innego GPU (ang. Device to Device)  
**DtoH** - przesłanie danych z pamięci GPU (Device) do pamięci CPU (ang. Device to Host)  
**FDTD** - metoda różnic skończonych w dziedzinie czasu (ang. Finite Difference Time Domain)  
**FDFD** - metoda różnic skończonych w dziedzinie częstotliwości (ang. Finite Difference Frequency Domain)  
**FLOPS** - liczba operacji zmiennoprzecinkowych wykonanych w czasie jednej sekundy (ang. Floating point Operations Per Second)  
**FSMA** - algorytm szybkiego dodawania macierzy rzadkich (ang. Fast Sparse Matrix Addition)  
**GPGPU** - aplikacje ogólnego zastosowania, dla których wykorzystuje się akceleratorze graficzne celem skrócenia czasu wykonania obliczeń (ang. General-Purpose computing on Graphics Processor Units)  
**GPU** - procesor graficzny (ang. Graphics Processing Unit)  
**HtoD** - przesłanie danych z pamięci CPU do GPU (Device) (ang. Host to Device)  
**MES** - metoda elementów skończonych (ang. Finite Element Method, FEM)  
**MOM** - metoda momentów (ang. Method of Moments)  
**NI** - całkowanie numeryczne (ang. Numerical Integration)  
**PDE** - cząstkowe równania różniczkowe (ang. Partial Differential Equations)  
**SpMV, matvec** - operacja mnożenia macierzy rzadkiej przez wektor (ang. Sparse Matrix Vector multiplication)

# Rozdział 1

## Wprowadzenie

Dużą dokładność w procesie projektowania złożonych układów mikrofalowych wykorzystywanych w komunikacji bezprzewodowej (np. anteny, filtry, sprzęgacze) można uzyskać wykorzystując symulatory elektromagnetyczne do obliczenia odpowiedzi układu. Symulatory elektromagnetyczne to programy komputerowe, które wykorzystują metody elektrodynamiki obliczeniowej (ang. Computational electromagnetics, CEM) do rozwiązania drogą numeryczną równań Maxwella (tzw. symulacja pełnofalowa). Wśród metod CEM do najpopularniejszych zaliczyć można metody różnicowe w dziedzinie czasu (ang. Finite Difference Time Domain, FDTD) i częstotliwości (ang. Finite Difference Frequency Domain, FDFD), metodę momentów (ang. Method of Moments, MOM), metodę elementów skończonych (ang. Finite Element Method, FEM, MES<sup>1</sup>) [1,2].

Metody CEM wykorzystujące bezpośrednią dyskretyzację eliptycznych równań różniczkowych cząstkowych otrzymywanych z równań Maxwella (np. FDFD oraz MES) prowadzą do generacji układów równań liniowych o wielu milionach niewiadomych, których rozwiązanie wymaga dużych zasobów pamięciowych i obliczeniowych. Układ równań liniowych można zapisać w formie macierzowej, z macierzą współczynników, która co prawda jest dużych rozmiarów lecz zawiera niewiele elementów niezerowych<sup>2</sup>. Niestety dla typowej stacji roboczej barierą są obecnie problemy z macierzą rzadką rzędu kilku milionów zmiennych, gdyż bezpośrednie rozwiązanie takich układów wymaga wykonania zbyt kosztownej czasowo i pamięciowo jak na obecny stan techniki faktoryzacji symbolicznej i numerycznej. Zapotrzebowanie na pamięć podczas faktoryzacji jest wielokrotnie większe niż rozmiar pamięci potrzebny na przechowanie rzadkiej macierzy współczynników. Alternatywą dla bezpośrednich metod rozwiązywania układów równań jest iteracyjne rozwiązywanie problemu w oparciu o algorytmy bazujące na podprzestrzeni Kryłowa [3,4]. Podejście iteracyjne jest oszczędne pamięciowo – ilość pamięci potrzebna do rozwiązania problemu jest proporcjonalna do liczby niewiadomych i wielokrotnie mniejsza niż dla metod bezpośrednich. Niestety wadą podejścia iteracyjnego jest słaba zbieżność lub jej brak, dla macierzy źle uwarunkowanych. Polepszenie zbieżności można osiągnąć poprzez zastosowanie w obliczeniach prekondycjonerów – nazywanych w Polsce operatorami ściskającym. Mimo to symulacje są czasochłonne, gdyż do osiągnięcia satysfakcjonującej zbieżności potrzeba wykonać wiele iteracji. W efekcie, analiza zagadnień elektrodynamicznych nawet przy użyciu najnowszych stacji roboczych i symulatorów jest długotrwała oraz może wymagać dużych zasobów pamięciowych i obliczeniowych, dostępnych tylko w klastrach. Czas pojedynczej symulacji

---

<sup>1</sup>W niniejszej rozprawie użyto polskiego akronimu MES.

<sup>2</sup>Macierz, w której większość elementów ma wartość zero nazywa się macierzą rzadką.

układu na jednej częstotliwości sięga wielu minut, a w przypadku obliczeń odpowiedzi w szerokim paśmie częstotliwości konieczne jest wykonanie kilkudziesięciu takich symulacji. Celowe jest zatem poszukiwanie rozwiązań algorytmicznych lub technologicznych pozwalających wydawnie skrócić czas symulacji elektromagnetycznych i dodatkowo, o ile to możliwe, zmniejszyć zapotrzebowanie na pamięć.

Najpopularniejsza metoda przeprowadzenia obliczeń numerycznych w trakcie rozwiązywania zagadnień CEM polega na użyciu procesora centralnego (ang. Central Processing Unit, CPU). W ciągu ostatnich 30 lat jednym z najistotniejszych sposobów zwiększenia mocy obliczeniowej komputerów konsumenckich (a tym samym wydajności obliczeń numerycznych wykonywanych na tych komputerach) było zwiększenie częstotliwości taktowania procesora centralnego. Oryginalnie procesor centralny projektowany był do realizacji instrukcji SISD (ang. Single Instruction Single Data). Poczynając od pierwszych procesorów wykorzystywanych w komputerach konsumenckich na początku lat 80-tych ubiegłego stulecia, do aktualnie dostępnych komputerów, częstotliwość pracy procesora zwiększyła się z 1 MHz do powyżej 4 GHz (co obecnie stanowi barierę technologiczną). Innym ograniczeniem zwiększania mocy obliczeniowej CPU jest duży pobór mocy i wydajność układów chłodzenia procesorów centralnych. W celu zwiększenia wydajności procesorów opracowano procesory wielordzeniowe oraz wprowadzono do CPU instrukcje SIMD (ang. Single Instruction Multiple Data). Celem poprawy wydajności obliczeń numerycznych wprowadzono również technologię wielowątkowości współbieżnej (ang. Hyper-Threading Technology, HT). W technologii HT dla pojedynczego rdzenia system operacyjny przypisuje dwa wirtualne wątki, które dzielą między sobą obliczenia do wykonania. Innym sposobem zwiększenia wydajności obliczeniowej są klastry obliczeniowe składające się z wielu połączonych ze sobą komputerów. Przykładem wdrożenia komercyjnego rozwiązania, które pozwala na wykorzystanie wielu rdzeni CPU oraz umożliwia przeprowadzenie obliczeń w klastrze obliczeniowym złożonym z wielu procesorów jest symulator Ansoft HFSS [5]. Niestety koszt zakupu klastra jest znaczny i dlatego to podejście jest często nieosiągalne dla ośrodków badawczych zajmujących się zagadnieniami elektrodynamiki obliczeniowej (CEM). Najnowszym trendem poprawy wydajności obliczeniowej, zaproponowanym przez firmę Intel jest architektura MIC (ang. Intel Many Integrated Core) [6], w której wiele rdzeni jest umieszczonych w pojedynczym układzie scalonym (max. 72 rdzeni - procesor Knights Landing).

Do wyznaczenia nowego trendu w efektywnym przetwarzaniu danych przyczynił się rozwój systemów operacyjnych opartych na aplikacjach okienkowych (np. Microsoft Windows i Apple Mac OS), który pomógł rozwinąć rynek dla jednostek obliczeniowych dedykowanych do przetwarzania grafiki komputerowej, czyli dla procesora graficznego GPU (ang. Graphics Processing Unit). Wraz ze rozwojem systemów operacyjnych i przemysłu gier komputerowych rosło zapotrzebowanie na coraz wydajniejsze procesory graficzne i w rezultacie firmy takie jak Silicon Graphics, a potem NVIDIA i ATI Technology znacznie rozwinęły technologię wyspecjalizowanych procesorów graficznych [7]. Ponieważ przetwarzanie grafiki można łatwo zrównoleglić, obliczenia na GPU są realizowane za pomocą znacznej liczby rdzeni. Wraz ze wzrostem liczby rdzeni w GPU ich moc numeryczna silnie wzrosła, co w konsekwencji doprowadziło do prób wykorzystania GPU w obliczeniach numerycznych i powstania pierwszych języków programowania, które umożliwiały uruchomienie aplikacji ogólnego zastosowania (ang. General-Purpose computing on Graphics Processor Units, GPGPU [8]) na GPU (BrookGPU [9], ATI CTM [10]).

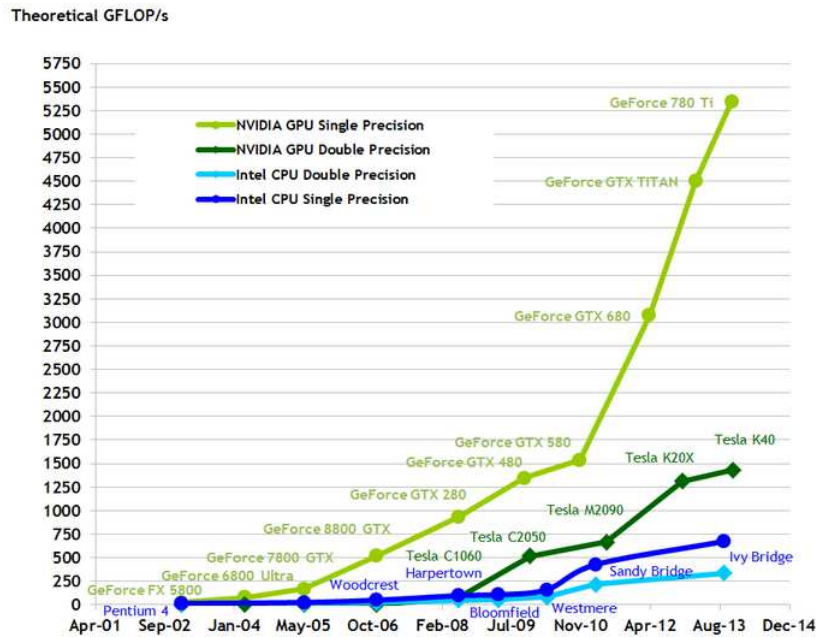


Krokiem milowym w praktycznym zastosowaniu GPU jako koprocatora dla CPU<sup>3</sup>, na którym można wykonać obliczenia numeryczne w aplikacjach ogólnego zastosowania, było wprowadzenie w listopadzie 2006 roku standardu DirectX 10, kart graficznych firmy NVIDIA ósmej generacji (G80) oraz programistycznej i sprzętowej architektury CUDA (ang. Compute Unified Device Architecture) [11]. CUDA jest programistyczną i sprzętową architekturą stworzoną po to, aby wywoływać i zarządzać równoległymi obliczeniami na GPU, bez potrzeby dostosowania implementacji algorytmów do graficznego API (ang. Application Programming Interface). Programowanie GPU w architekturze CUDA odbywa się przy użyciu rozszerzonego języka wysokopoziomowego C/C++, a nie jak to było wcześniej przy użyciu języków niskiego poziomu. Ponadto, ponieważ NVIDIA dedykowała procesory graficzne do użycia w aplikacjach ogólnego zastosowania (GPGPU), jednostki arytmetyczno-logiczne (ang. Arithmetic Logic Unit, ALU) zostały zaprojektowane zgodnie ze standardami IEEE dla przetwarzania danych zmiennoprzecinkowych. Architektura CUDA umożliwiła swobodny zapis i odczyt do pamięci GPU oraz do pamięci podręcznej (ang. cache). Powyższe udogodnienia umożliwiły efektywne wykorzystanie dużej mocy obliczeniowej GPU. W roku 2009, firma Apple Inc.<sup>4</sup> opracowała platformę programistyczną OpenCL (ang. Open Computing Language), która wspomaga pisanie aplikacji działających na platformach składających się z obydwu opisanych powyżej jednostek obliczeniowych czyli CPU i GPU. OpenCL daje możliwość użycia jednego otwartego standardu, w którym można wykorzystać platformy obliczeniowe (CPU, GPU) różnych producentów (np. NVIDIA, AMD, Intel, ARM) i jest alternatywą dla zamkniętego standardu jakim jest CUDA, który jest dedykowany wyłącznie akceleratorom firmy NVIDIA. Wszystkie wymienione tutaj atuty GPU sprawiły, iż procesor graficzny oprócz wykonywania zadań związanych z przetwarzaniem graficznym może być efektywnie użyty do wykonania obliczeń numerycznych wykorzystywanych w programach ogólnego zastosowania (GPGPU) [7, 8, 11].

W ramach krótkiego porównania CPU i GPU, w uproszczeniu można powiedzieć, że dostępne obecnie procesory centralne zawierają kilka-kilkanaście rdzeni, które są optymalizowane do wykonania obliczeń sekwencyjnie, podczas gdy GPU posiada tysiące (mniejszych i mniej skomplikowanych) rdzeni zaprojektowanych tak aby wykonać wiele zadań równocześnie. Powoduje to, że teoretyczna moc obliczeniowa GPU jest znacznie większa od mocy obliczeniowej CPU, co zobrazowano na rys. 1.1, na którym zaprezentowano porównanie wydajności obliczeniowej procesorów centralnych i procesorów graficznych w latach 2001-2013. Z tego powodu wykorzystanie masywnego zrównoleżenia obliczeń oferowanego przez GPU zostało uznane za ważny krok w osiągnięciu superwydajnych systemów komputerowych o wydajności rzędu exaFLOPS ( $10^{18}$  operacji zmiennoprzecinkowych na sekundę, ang. exascale computing [12]). Osiągnięcie wysokiej wydajności wymaga opracowania nowych, dostosowanych do architektury GPU algorytmów obliczeniowych, co nie jest zagadnieniem trywialnym. Dodatkowo, maksymalna teoretyczna moc obliczeniowa często jest niemożliwa do osiągnięcia, gdyż szybkość obliczeń w wielu przypadkach jest ograniczona relatywnie niską w stosunku do szybkości wykonywania operacji zmiennoprzecinkowych przepustowością pamięci.

<sup>3</sup>GPU nie obsługuje urządzeń zewnętrznych innych niż magistrala PCI-E.

<sup>4</sup>Później platforma rozwijana przez Khronos Group w skład którego wchodzi: NVIDIA Co., Intel, 3Dlabs, ATI, Discreet, Evans & Sutherland, SGI, Google i Sun Microsystems.



RYSUNEK 1.1: Porównanie wydajności obliczeniowej procesorów centralnych (CPU) i graficznych (GPU) w latach 2001-13 [11].

## 1.1 Obecny stan wiedzy

Skrócenie czasu symulacji za pomocą akceleratorów graficznych stanowi przedmiot badań wielu naukowców zajmujących się elektrodynamiką obliczeniową. Pionierskie na tym polu były badania grupy z Uniwersytetu w Calgary, która w 2004 opublikowała pierwsze rezultaty dotyczące dwuwymiarowej wersji metody różnic skończonych w dziedzinie czasu<sup>5</sup> [13]. Metoda FDTD jest stosunkowo łatwa do zrównoleglenia tak więc szybko pojawiły się kolejne prace [14–19], w tym dotyczące obliczeń na klastrach akceleratorów graficznych [20]. Na przestrzeni ostatnich dziesięciu lat badania objęły szerszy zakres metod elektrodynamiki obliczeniowej. Oprócz metody FDTD na potrzeby obliczeń z wykorzystaniem akceleratorów graficznych zostały zaadaptowane metody momentów (MOM) [21,22], Alternating Direction Implicit (ADI-FDTD) [23], Transmission Line Modeling (TLM) [24]. Możliwość przeprowadzania wybranych obliczeń przy użyciu akceleratorów graficznych została udostępniona w wielu symulatorach opartych na metodzie różnic skończonych w dziedzinie czasu (FDTD) np. AxFDTD, CST, REMCOM, SPEAG, Agilent ADS, QWED QuickWave i metodzie momentów (MOM) np. FEKO. Należy zauważyć, że jakkolwiek literatura dotycząca zagadnień elektrodynamiki obliczeniowej z wykorzystaniem GPU jest bogata, to relatywnie mało uwagi zostało poświęcone skróceniu czasu symulacji przy użyciu akceleratorów graficznych w metodzie elementów skończonych (MES) zwłaszcza w odniesieniu dla problemów eliptycznych.

Metoda elementów skończonych (MES) w dziedzinie częstotliwości stanowi wydajne i uniwersalne narzędzie analizy układów mikrofalowych [25,26]. MES należy do grupy metod siatkowych, w których rozważa się różniczkową postać problemu brzegowego, zdefiniowanego w pewnym skończonym obszarze nazywanym dziedziną obliczeniową, który dzieli się na małe fragmenty poprzez wykorzystanie dedykowanych generatorów

<sup>5</sup>Badania te szybko doprowadziły do powstania firmy typu spin-off Acceleware i pierwszego wykorzystującego GPU komercyjnego oprogramowania, które pojawiło się na rynku w 2005 roku.

siatki. Jakość i gęstość siatki mocno wpływa na dokładność samej symulacji. Wysoką dokładność numerycznego badanego zagadnienia CEM zwykle osiąga się poprzez zastosowanie siatki elementów skończonych o dużej gęstości (tzw. h-refinement) co skutkuje zwiększeniem rozmiaru macierzy opisujących dane zagadnienie CEM. Drugi sposób to podwyższenie rzędu aproksymacji wewnątrz oczka siatki (tzw. p-refinement), które także zwiększa rozmiar problemu i dodatkowo podnosi liczbę elementów niezerowych w każdym wierszu macierzy. Oba podejścia prowadzą w konsekwencji do znaczącego wzrostu kosztów numerycznych w etapie generacji i rozwiązania układów równań.

Jak wcześniej zauważono, w odniesieniu do elektrodynamiki obliczeniowej najsilniej techniki obliczeniowe wykorzystujące karty graficzne rozwijają się dla schematów otwartych rozwiązywania zagadnień początkowych (tzn. FDTD, DG-TD [27]). Ostatnio pojawiły się w literaturze pierwsze publikacje dotyczące MES. W [28] opisano przyspieszenie MES na GPU, w którym zastosowano liniowe elementy skończone z liniowymi funkcjami bazowymi przez co koszt numeryczny jest mniejszy (mniejszy rozmiar macierzy, mniej liczby elementów niezerowych w wierszu) niż w przypadku hierarchicznych wektorowych funkcji bazowych wysokiego rzędu użytych w niniejszej rozprawie.

Jakkolwiek powyższy krótki opis stanu wiedzy koncentruje się na wykorzystaniu GPU w symulacjach elektromagnetycznych, to należy zauważyć, że w innych obszarach nauk obliczeniowych toczą się intensywne prace nad akceleracją metod numerycznych rozwiązywania problemów brzegowych i początkowych, w tym MES, a także nad rozwojem technik rozwiązywania wielkich układów równań liniowych z macierzą rzadką przy wykorzystaniu procesorów graficznych. W tym ostatnim przypadku wysiłki zmierzają w kierunku znalezienia optymalnego, z punktu widzenia architektury GPU, formatu zapisu macierzy rzadkich, który z jednej strony zachowywałby zwarty zapis elementów niezerowych, a z drugiej umożliwiłaby szybki dostęp do elementów macierzy tysiącom wątków jednocześnie [29–34]. Zagadnienie to zostało omówione dokładniej w rozdziale szóstym tej rozprawy. Kolejnym wątkiem naukowym ściśle związanym z formatem zapisu macierzy są iteracyjne schematy rozwiązywania układów równań liniowych. W literaturze znaleźć można publikacje opisujące implementacje technik iteracyjnych bazujących na podprzestrzeni Kryłowa [35, 36], operatorów ściskających w postaci niekompletnej faktoryzacji LU [37] oraz metod wielosiatkowych [38] dla których uzyskano znaczące skrócenia czasu rozwiązania dzięki zastosowaniu GPU. Dostępne są również biblioteki dedykowane zagadnieniu rozwiązywania układów równań na GPU, np. MAGMA Sparse-Iter Package [39], AmgX [40]. Należy jednak podkreślić, że wiele technik iteracyjnego rozwiązywania układów równań liniowych nie nadaje się dla systemów powstających w elektrodynamice obliczeniowej ze względu na złe uwarunkowanie generowanych macierzy, co przekłada się na słabą zbieżność lub jej brak. Toczą się także prace badawcze dotyczące bezpośredniego rozwiązywania układów równań z wykorzystaniem akceleratorów graficznych - dostępne są dwie biblioteki SPRAL [41] i cuSOLVER [42]. W przeprowadzonych na potrzeby tej rozprawy testach numerycznych wykazano, że w obydwu podejściach faktoryzacja macierzy rzadkiej jest bardzo kosztowna pamięciowo (zapotrzebowanie na pamięć jest większe niż w bibliotece Intel MKL Pardiso [43] przeznaczonej dla CPU). Co prawda w [42] zaprezentowano wyniki, które wskazują, że biblioteka cuSOLVER pozwala uzyskać duże przyspieszenie rozwiązania układu równań względem Intel MKL Pardiso, jednakże macierze testowe są bardzo małych rozmiarów i są dobrze uwarunkowane. W rozdziale szóstym zaprezentowano wyniki, które potwierdzają, że biblioteka Intel MKL Pardiso pozwala na szybsze rozwiązanie układu równań generowanych w niniejszej rozprawie niż biblioteki SPRAL i cuSOLVER, i że

duże zapotrzebowanie na pamięć bibliotek SPRAL i cuSOLVER dyskwalifikuje je do zastosowania dla macierzy generowanych w niniejszej rozprawie.

Wątek szybkiego wyznaczania macierzy współczynników w MES jest istotny dla wielu obszarów nauk obliczeniowych. Metoda elementów skończonych jest jedną z najbardziej popularnych technik symulacyjnych wykorzystywanych w mechanice płynów i mechanice stosowanej i dlatego w literaturze znaleźć można publikacje dotyczące najważniejszych etapów budowania macierzy rzadkich w MES (całkowanie numeryczne w celu wyznaczenia lokalnych macierzy elementów [44–46], konstrukcja macierzy z wykorzystaniem akceleratora graficznego [47–51]).

W niniejszym podrozdziale stan wiedzy w zakresie masywnego zrównoleglenia obliczeń został zaledwie zarysowany. Literatura, jaki i liczba zagadnień jest zbyt szeroka, aby omówić je w krótkim wstępie. Szczegółowe informacje prezentujące wybrane rozwiązania zaproponowane przez innych badaczy przedstawione zostały w kolejnych rozdziałach tej rozprawy. W szczególności w rozdziale czwartym omówiono zagadnienie generacji macierzy rzadkich w MES przy użyciu GPU. Podobnie, w rozdziale szóstym, omówiono problem formatu zapisu macierzy rzadkiej uwzględniający specyfikę wykonania obliczeń na GPU.

## 1.2 Cel i tezy rozprawy

Celem niniejszej rozprawy jest opracowanie numerycznie efektywnego i ekonomicznie atrakcyjnego rozwiązania algorytmicznego i sprzętowego, które umożliwi przyspieszenie analizy problemów elektromagnetycznych o złożonej geometrii z wysoką dokładnością. W szczególności skoncentrowano się na opracowaniu nowego wariantu algorytmu MES opartego na technice masywnego zrównoleglenia obliczeń z wykorzystaniem akceleratorów graficznych oraz implementacji tego algorytmu przystosowanej do działania na akceleratorach graficznych kompatybilnych z architekturą CUDA. Prace badawcze prowadzone w ramach niniejszej rozprawy dotyczyły przede wszystkim nowych sposobów masywnego zrównoleglenia najbardziej kosztownych obliczeniowo etapów rozwiązania problemów elektromagnetycznych metodą elementów skończonych wykorzystującą funkcje bazowe wysokiego rzędu (generacja macierzy i rozwiązanie układu równań liniowych). Dla pozostałych etapów obliczeń, które nie mogły zostać efektywnie zrównoleglone przy użyciu kart graficznych, wykorzystano wielordzeniowy procesor ogólnego przeznaczenia.

Pomimo iż prace nad podobnymi zagadnieniami toczą się w innych obszarach nauk obliczeniowych, specyfika MES w symulacjach elektromagnetycznych i własności powstającej w wyniku jej zastosowania macierzy są na tyle odmienne, że zastosowanie gotowych procedur nie prowadzi do wydajnych obliczeń. Tezy pracy można ująć w sposób następujący:

1. Współczesne karty graficzne zgodne z architekturą CUDA umożliwiają efektywną realizację obliczeń numerycznych kosztownych obliczeniowo etapów rozwiązania problemów elektromagnetycznych metodą elementów skończonych.
2. Uzyskanie dużej wydajności obliczeń przy rozwiązywaniu wielkich rzadkich układów równań liniowych metodami iteracyjnymi za pomocą akceleratorów graficznych jest możliwe dzięki:

- (a) zastosowaniu wielopoziomowego operatora ściskającego z ważoną relaksacją Jacobiego,
  - (b) zastosowaniu nowego formatu zapisu macierzy rzadkiej Sliced ELLR-T,
  - (c) wykorzystaniu CPU do bezpośredniego rozwiązania układu na najniższym poziomie wielopoziomowego operatora ściskającego.
3. Nowy format zapisu macierzy rzadkiej Sliced ELLR-T umożliwia uzyskanie większej szybkości operacji mnożenia macierzy rzadkiej przez wektor niż standardowe procedury biblioteki CUSPARSE wykorzystujące format CRS.

Tezy 1-2a, 2b mają istotne znaczenie dla elektrodynamiki obliczeniowej, lub też MES z funkcjami bazowymi wyższego rzędu (tezy 2a i 2c). Tezy 2b i 3 dotyczące formatu zapisu macierzy rzadkich są ogólniejsze i mają zastosowanie dla szerszej klasy macierzy rzadkich.

## 1.3 Plan rozprawy

Rozprawę podzielona na dziewięć rozdziałów, z których pierwszy stanowi ogólne wprowadzenie i definiuje cel i tezy pracy. W drugim rozdziale podano podstawowe informacje dotyczące rozważanej wersji metody elementów skończonych. Dodatkowo zdefiniowane zostały sformułowania użyte w rozprawie oraz opisano podział algorytmu MES na dwa zasadnicze etapy obliczeń: generację macierzy opisujących problem oraz rozwiązanie problemu (układu równań liniowych).

W rozdziale trzecim przedstawiono charakterystykę architektury CUDA, właściwości architektury, model programowania i podstawowe pojęcia, które są stosowane w opisie opracowanych rozwiązań algorytmicznych i implementacyjnych.

Rozdział czwarty zawiera opis strategii i algorytmów, które umożliwiają maszynowe zrównoleglenie procesu generacji macierzy rzadkich w MES. Najpierw opisane zostały implementacje dotyczące zrównoleglenia generacji macierzy współczynników dla relatywnie niedużych problemów (m.in. całkowanie numeryczne, konwersja między formatami reprezentacji macierzy rzadkich opisujących problem). W rozdziale piątym przedstawiono algorytm pozwalający na budowę dużych macierzy sztywności i bezwładności na pojedynczym i wielu akceleratorach graficznych.

W rozdziale szóstym opisano strategie zastosowane przy maszynowym zrównolegleniu etapu rozwiązywania układu równań liniowych (m.in. procedury wielopoziomowego operatora ściskającego, mnożenie macierzy rzadkiej przez wektor), a w rozdziale siódmym przedstawiono implementacje pozwalające na rozwiązanie układu równań z wykorzystaniem kilku akceleratorów graficznych.

Rozdział ósmy zawiera testy numeryczne, które demonstrują znaczne skrócenie czasu symulacji realizowanych metodą MES dla dużych problemów przy zastosowaniu akceleratorów graficznych w porównaniu do obliczeń przeprowadzonych na wysokowydajnym procesorze Intel Xeon.

Ostatni rozdział zawiera podsumowanie osiągniętych rezultatów przeprowadzonych badań naukowych.

W dodatkach zaprezentowano uzupełnienie dyskusji prowadzonej w rozprawie, w szczególności dotyczące metody elementów skończonych (Dodatek A), architektury CUDA (Dodatek B), kodów implementacji na GPU (Dodatek C) i na CPU (Dodatek D) oraz opisu konwersji między formatami reprezentacji macierzy rzadkich (Dodatek E).





## Rozdział 2

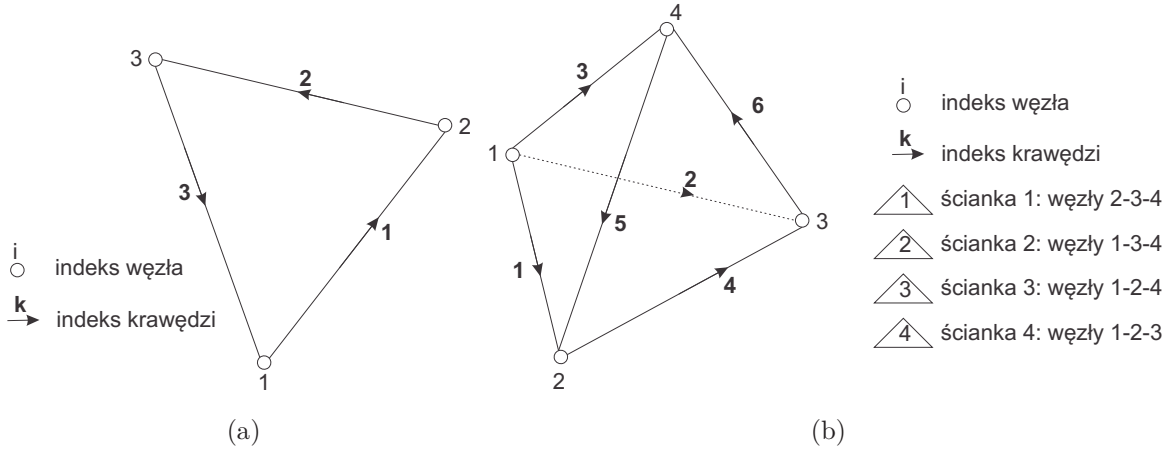
# Metoda elementów skończonych

Metoda elementów skończonych (MES) stanowi jedną z najbardziej popularnych metod numerycznego rozwiązywania równań różniczkowych z określonymi warunkami brzegowymi (2.1):

$$\mathcal{L}\Phi = h \quad (2.1)$$

gdzie  $\mathcal{L}$  reprezentuje operator różniczkowy,  $\Phi$  reprezentuje poszukiwaną wartość,  $h$  to funkcja definiująca pobudzenie. MES znalazła zastosowanie w rozwiązaniu równań różniczkowych w wielu zagadnieniach fizycznych np. obliczeniowa mechanika płynów [52], mechanika stosowana [53], dynamika molekularna [54]. W niniejszej rozprawie przedstawiono zastosowanie metody elementów skończonych w elektrodynamice obliczeniowej do rozwiązywania wektorowego równania falowego, co umożliwia przeprowadzenie pełnofalowej symulacji propagacji fali elektromagnetycznej i wyznaczenie drgań własnych rezonatorów lub symulację odpowiedzi układu mikrofalowego, anteny lub obiektu rozpraszającego na pobudzenie falą elektromagnetyczną na wybranej częstotliwości w harmonicznym stanie ustalonym [1, 25, 26, 55, 56].

Metoda elementów skończonych należy do grupy metod siatkowych, w których rozpatruje się problemy brzegowe i początkowe w postaci różniczkowej, i ograniczoną ciągłą dziedzinę obliczeniową dzieli się na skończone małe fragmenty. W przypadku MES siatka składa się z liniowych lub krzywoliniowych trójkątów lub czworokątów (w dwóch wymiarach) względnie czworościanów lub sześciocianów (w trzech wymiarach). Podział dziedziny obliczeniowej na oczka nazywa się siatkowaniem, a każde oczko odpowiada elementowi skończonemu. Jako element skończony rozumieć się będzie w tej pracy oczko wraz z funkcjami interpolującymi rozkład pola wewnątrz oczka (szczegóły poniżej). W MES często stosuje się siatki nieregularne, w których każdy element może mieć inny rozmiar. Zastosowanie elementów różnej wielkości o krzywoliniowych krawędziach i ścianach daje możliwość przeprowadzenia symulacji dla bardzo złożonych obiektów o niemalże dowolnej geometrii. W przestrzeni dwuwymiarowej trójkąty opisywane są przez współrzędne w przestrzeni i indeksy węzłów (rys. 2.1a). W przestrzeni trójwymiarowej czworościan określony jest przez współrzędne w przestrzeni, węzły, krawędzie i ściany (rys. 2.1b). W praktyce, każdy element jest reprezentowany nie w układzie kartezjańskim  $(x,y,z)$ , ale w układzie simpleksowym  $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$ , co zaprezentowano w Dodatku A.1. Wprowadzenie współrzędnych simpleksowych upraszcza opis, który wykonuje się dla elementu referencyjnego, a następnie na podstawie prostego przekształcenia można zaadaptować go dla każdego elementu z siatki [1, 57]. Do podziału dziedziny obliczeniowej na oczka można wykorzystać generatory siatki np. NETGEN [58], TetGen [59], GMSH [60].



RYSUNEK 2.1: Opis elementów skończonych pierwszego rzędu (a) trójkąt - indeksacja węzłów i krawędzi, (b) czworokąt - indeksacja węzłów, krawędzi i ścianek.

Rozwiązanie metody MES jest konstruowane na podstawie kombinacji liniowej związanych z oczkiem siatki funkcji bazowych<sup>1</sup> (najczęściej wielomianowych), które interpolują rozwiązanie wewnątrz elementu skończonego. W przypadku gdy  $\Phi$  jest polem wektorowym to dokładniejszą aproksymację uzyskuje się stosując wektorowe, a nie skalarne funkcje bazowe<sup>2</sup> [56, 61]. W kwestii nazewnictwa warto doprecyzować, iż w poniższym opisie czworokąt odpowiada geometrycznej interpretacji oczka siatki, na które podzielono domenę obliczeniową.

W metodzie elementów skończonych zamiana zagadnienia brzegowego (2.1) na układ równań liniowych odbywa się przy użyciu jednej z dwóch metod: wariacyjnej lub residuów ważonych, które wychodzą lub prowadzą do słabej postaci zagadnienia brzegowego. W metodzie residuów ważonych równanie różniczkowe (2.1) jest mnożone przez funkcję wagi i następnie całkowane w obszarze elementu skończonego. Szczegóły metody residuów ważonych przedstawiono w Dodatku A.2. W ogólności wymaga się, aby funkcje bazowe były podwójnie różniczkowalne (silna postać zagadnienia brzegowego). Słaba postać wymaga, aby funkcje wagi i funkcje bazowe były tylko raz różniczkowalne [57]. Dodatkowo, jeżeli funkcje wagi wybierane są z zestawu funkcji bazowych to metoda residuów ważonych nazywa się metodą Galerkiną<sup>3</sup>.

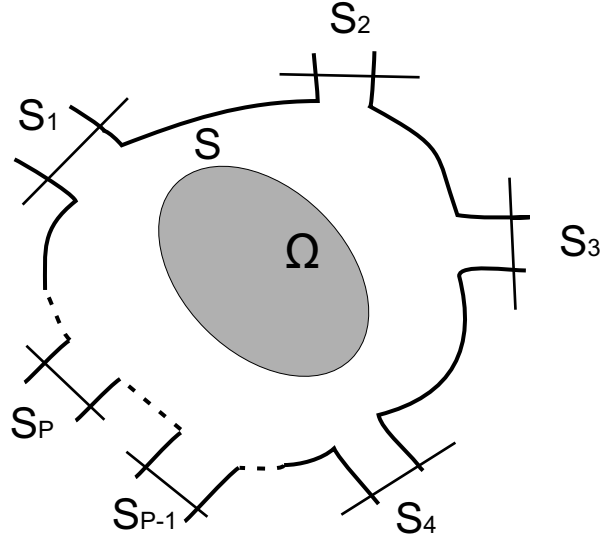
W niniejszej rozprawie MES wykorzystane jest do wyznaczenia parametrów obwodowych układu mikrofalowego. W odniesieniu do opisu obwodowego wykorzystano sformułowania zaproponowane w [62, 63], które bazują na technice segmentacji obwodu. Dobrym sposobem implementacji metody segmentacji jest opis każdego regionu poprzez wielorodzajową macierz układu wielowrotowego, która w połączeniu z innymi macierzami z pozostałych regionów pozwala otrzymać globalną odpowiedź układu. Jednym z

<sup>1</sup>W niniejszej pracy zdecydowano się użyć określenia funkcje bazowe. W literaturze można również spotkać się z określeniami funkcje interpolacyjne i funkcje kształtu.

<sup>2</sup>Wektorowe funkcje bazowe z większą dokładnością wyznaczają fałszywe rodzaje (ang. spurious modes). Kwestia fałszywych rodzajów i ich wpływ na dokładność rozwiązania została szczerzej opisana w p. 2.3.

<sup>3</sup>Należy zaznaczyć, że istnieje wiele dróg prowadzących do tego samego układu równań liniowych, stąd też w literaturze można znaleźć różne równoważne sformułowania związane z określonymi metodami - np. sformułowanie wariacyjne z metodą Rayleigha-Ritza, residuów ważonych, momentów itp. Punktem wyjścia może być postać słaba lub silna. W niniejszej pracy przedstawiono jedynie szkic MES bez zwracania uwagi na niuanse.





RYSUNEK 2.2: Przykład urządzenia wielowrotowego ( $\Omega$  - dziedzina obliczeniowa,  $S_i$  - wrota).

możliwych sposobów obliczenia tej macierzy jest koncepcja uogólnionej macierzy admitancji (ang. generalized admittance matrix, GAM) lub uogólnionej macierzy impedancji (ang. generalized impedance matrix, GIM) [63]. Na podstawie otrzymanych macierzy admitancji lub impedancji następnie wyznacza się macierze parametrów rozproszenia  $\mathbf{S}_C$ <sup>4</sup>. Elementy macierzy rozproszenia są funkcją częstotliwości. Znając wartości elementów macierzy  $\mathbf{S}_C$  w analizowanym paśmie częstotliwości określa się charakterystyki odbicia i transmisji analizowanego układu. W niniejszej rozprawie bazowano na sformułowaniu z [63], które zostało poniżej szczegółowo opisane.

W rozprawie rozwiązywane jest równanie fali Helmholtza:

$$\nabla \times \mu^{-1} \nabla \times \vec{E} - \omega^2 \epsilon \vec{E} = 0 \quad (2.2)$$

gdzie  $\vec{E}$  to wektor natężenia pola elektrycznego,  $\omega$  reprezentuje częstotliwość kątową zależną od częstotliwości  $f$ ,  $\epsilon$  i  $\mu$  opisują, odpowiednio, tensory przenikalność elektrycznej i magnetycznej.

Słabą formę wektorowego równania Helmholtza można wyprowadzić przy użyciu metody Galerkinia poprzez przemnożenie r. (2.2) przez wektorowe funkcje wagi  $\vec{W}$ . W metodzie Galerkinia funkcje wagi  $\vec{W}$  wybiera się spośród zestawu funkcji bazowych  $\vec{N}$  (Dodatek A.2). Tak otrzymane równanie jest całkowane i w efekcie otrzymujemy postać całkową:

$$\int_{\Omega} \vec{W} \cdot (\nabla \times \mu^{-1} \nabla \times \vec{E} - k_0^2 \epsilon_r \vec{E}) d\Omega = 0 \quad (2.3)$$

gdzie:  $k_0 = \frac{2\pi f}{c}$  opisuje liczbę falową, której wartość zależy od częstotliwości  $f$  i prędkości światła  $c$ ,  $\Omega$  to dziedzina obliczeniowa.

Zastosowanie tożsamości wektorowych [61], warunków brzegowych naturalnych (Dirichleta [56]) i zasadniczych (Neumanna [56]) dla  $P$  wrót umieszczonych na brzegu o powierzchni  $S$  otaczającej region  $\Omega$  pozwala przepisać (rys. 2.2) słabą postać równania

<sup>4</sup>Indeks dolny C zastosowano by odróżnić macierz parametrów rozproszenia od macierzy sztywności, zdefiniowanej w dalszej części rozdziału.

wektorowego w formie:

$$\int_{\Omega} (\nabla \times \vec{W} \cdot \mu^{-1} \nabla \times \vec{E} - k_0^2 \vec{W} \cdot \epsilon_r \vec{E}) d\Omega - j\omega\mu_0 \sum_{i=1}^P \int_{S_i} \vec{W} \cdot (\vec{z}_i \times \vec{H}_t^i) dS_i = 0 \quad (2.4)$$

gdzie:  $\vec{z}_i$  to wektor normalny skierowany do wewnątrz regionu na powierzchni  $S_i$  w każdym z wrót układu  $i$ ,  $\vec{H}_t^i$  reprezentuje natężenie pola magnetycznego w  $i$ -tych wrótach.

Styczne pole magnetyczne  $\vec{H}_t^i$  w każdym z  $i$ -tych wrót (w płaszczyźnie XY) można zapisać w postaci r. (2.5), gdzie  $(x_i, y_i, z_i)$  to lokalne współrzędne każdego z wrót,  $I$  prąd,  $\vec{h}_{tk}^i$  określa rozkład poprzeczny pola magnetycznego  $k$ -tego rodzaju występującego w  $i$ -tych wrótach.

$$\vec{H}_t^i = \sum_{k=1}^{\infty} I_k^i(z_i) \vec{h}_{tk}^i(x_i, y_i) \quad (2.5)$$

Włączając r. (2.5) do r. (2.4) otrzymujemy r. (2.6). W praktyce szeregi z r. (2.6) redukowane są dla każdego z wrót do skończonej liczby rodzajów [63].

$$\int_{\Omega} (\nabla \times \vec{W} \cdot \mu^{-1} \nabla \times \vec{E} - k_0^2 \vec{W} \cdot \epsilon_r \vec{E}) d\Omega = j\omega\mu_0 \sum_{i=1}^P \sum_{k=1}^{\infty} I_k^i(z_i) \int_{S_i} \vec{W} \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i \quad (2.6)$$

Wewnątrz analizowanego regionu  $\vec{E}$  i  $\vec{W}$  można zapisać w postaci r. (2.7) i (2.8), gdzie  $N_{dofs}$  to liczba stopni swobody  $E_{cj}$  w siatce, a  $\vec{N}_j$  to wektorowe funkcje bazowe zdefiniowane wewnątrz elementu skończonego.

$$\vec{E} = \sum_{j=1}^{N_{dofs}} E_{cj} \vec{N}_j = \mathbf{N}_j \mathbf{e}_c \quad (2.7)$$

$$\vec{W} = \sum_{j=1}^{N_{dofs}} W_{cj} \vec{N}_j = \mathbf{N}_j \mathbf{w}_c \quad (2.8)$$

Podstawiając r. (2.7) i (2.8) do równania (2.6) oraz przyjmując skończoną liczbę rodzajów we wrótach otrzymujemy układ równań z (2.9), który można zapisać w formie r. (2.11), gdzie  $\mathbf{A}$  to symetryczna macierz rzadka o rozmiarze  $N_{dofs} \times N_{dofs}$  (2.10),  $\mathbf{v}$  to  $Q_r$ -wymiarowy wektor napięć ( $Q_r$  jest sumą liczby rodzajów użytych w każdym z wrót),  $\mathbf{B}$  jest macierzą o rozmiarze  $N_{dofs} \times Q_r$  r. (2.12)-(2.13).

$$\mathbf{w}_c^T \cdot ((\mathbf{S} - k_0^2 \mathbf{T}) \mathbf{e}_c - j\omega\mu_0 \mathbf{B} \mathbf{v}) = 0 \quad \forall \mathbf{w}_c \in C^N \quad (2.9)$$

$$\mathbf{A} = (\mathbf{S} - k_0^2 \mathbf{T}) \quad (2.10)$$

$$\mathbf{A} \mathbf{e}_c = j\omega\mu_0 \mathbf{B} \mathbf{v} \quad (2.11)$$

$$\mathbf{B} = [\mathbf{b}_1^1, \dots, \mathbf{b}_{m1}^1 \dots \mathbf{b}_k^i, \dots, \mathbf{b}_1^P, \dots, \mathbf{b}_{mP}^P] \quad (2.12)$$

$$\mathbf{b}_k^i = \left[ \int_{S_i} \vec{N}_1 \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i, \dots, \int_{S_i} \vec{N}_j \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i, \dots, \int_{S_i} \vec{N}_{N_{dofs}} \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i \right]^T \quad (2.13)$$

Każda z całek  $\mathbf{b}_k^i$  jest rozciągnięta na płaszczyźnie  $i$ -tych wrot gdzie rodzaj  $k$  jest zdefiniowany i ma niezerową wartość jeżeli którykolwiek z komponentów wektorowej funkcji  $\vec{N}_j$  jest styczny do tego wrota. Przy zastosowaniu modalnego rozwinięcia pola elektrycznego można zapisać poniższe równanie:

$$\int_{S_i} \vec{E} \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i = V_k^i(z_i) \int_{S_i} \vec{e}_{tk}^i \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i \quad (2.14)$$

Ponadto lewa strona r. (2.14) może być zapisana z użyciem zdyskretyzowanego r. (2.7):

$$\int_{S_i} \vec{E} \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i = \sum_{j=1}^{N_{dof_s}} E_{cj} b_{jk}^i \quad (2.15)$$

Jeżeli zidentyfikujemy prawe strony r. (2.14) i (2.15) oraz zastosujemy zapis macierzowy to otrzymamy system r. (2.16), w którym  $\mathbf{v}$  jest  $Q_r$  wymiarowym wektorem, a  $\Delta$  jest zdefiniowana w postaci r. (2.17)-(2.18).

$$\mathbf{B}^T \mathbf{e}_c = \Delta \mathbf{v} \quad (2.16)$$

$$\Delta = \text{diag}(\Delta_1^1 \cdots \Delta_{m1}^1 \cdots \Delta_k^i \cdots \Delta_1^P \cdots \Delta_{mP}^P) \quad (2.17)$$

$$\Delta_k^i = \int_{S_i} \vec{e}_{tk}^i \cdot (\vec{z}_i \times \vec{h}_{tk}^i) dS_i \quad (2.18)$$

Stosując normalizację napięć ( $\mathbf{v}_n = \Delta^{1/2} \mathbf{v}$ ) i prądów ( $\mathbf{i}_n = \Delta^{1/2} \mathbf{i}$ ) oraz normalizację macierzy  $\mathbf{B}_n = \mathbf{B} \Delta^{-1/2}$ , uogólnioną macierz impedancji (GIM) można zapisać w postaci r. (2.19). Macierz  $\mathbf{X}$  to rozwiązanie układu równań liniowych (2.20), które interpretuje się jako macierz poszukiwanych amplitud funkcji bazowych opisujących falę elektromagnetyczną.

$$\mathbf{Z}(j\omega) = j\omega\mu_0 \mathbf{B}_n^T \mathbf{A}^{-1} \mathbf{B}_n = j\omega\mu_0 \mathbf{B}_n^T \mathbf{X} \quad (2.19)$$

$$\mathbf{A} \mathbf{X} = \mathbf{B}_n \quad (2.20)$$

Zgodnie z [63] macierz rozproszenia można zapisać w postaci r. (2.21), gdzie macierz admitancji  $\mathbf{Y}$  jest odwrotnością macierzy impedancji ( $\mathbf{Y} = \mathbf{Z}^{-1}$ ). Na podstawie wartości macierzy rozproszenia można określić współczynniki odbicia  $sc_{11}$  i transmisji  $sc_{21}$  układów mikrofalowych w analizowanym paśmie częstotliwości.

$$[\mathbf{S}_C] = 2(\mathbf{I}_d + \mathbf{Y})^{-1} - \mathbf{I}_d \quad (2.21)$$

W dalszej części tego rozdziału scharakteryzowano sformułowania użyte do opisu elementu skończonego (hierarchiczne funkcje bazowe dla krzywoliniowych elementów skończonych (czworościanów)). Następnie opisano etapy tj. tworzenie lokalnych macierzy elementów (p. 2.1), konstrukcja macierzy globalnych z macierzy lokalnych (p. 2.2), które pozwoliły na generację globalnych macierzy rzadkich sztywności  $\mathbf{S}$  i bezwładności  $\mathbf{T}$  z r. (2.10) i rozwiązanie układu równań (2.20).

## 2.1 Lokalne macierze elementów

Po wstępnym opisie głównych założeń metody elementów skończonych oraz sformułowania uogólnionej macierzy impedancji (GIM) użytej do analizy układów mikrofalowych, poniżej przedstawiono szczegóły dotyczące użytych hierarchicznych wektorowych funkcji bazowych wyższego rzędu ( $\mathcal{H}^2(curl)$ , rząd  $p = 3$  [64]<sup>5</sup>) oraz krzywoliniowych elementów skończonych w trzech wymiarach [55].

### Hierarchiczne funkcje bazowe

Wektorowe hierarchiczne funkcje bazowe to zestaw funkcji wielomianowych  $\vec{N}$  o rosnącym rzędzie. Zastosowanie tego typu elementów pozwala na:

- podniesienie dokładności rozwiązania (dzięki lokalnemu lub globalnemu podwyższeniu rzędu interpolacji wewnątrz elementu - adaptacja typu  $p$  [65]),
- zastosowanie adaptacyjnego procesu zagęszczania lub generacji siatki,
- konstrukcję iteracyjnych procedur rozwiązywania układu równań linowych tworzonych w MES wykorzystujących wielopoziomowy operator ściskający (prekondycjoner) [61],
- budowę hierarchicznych estymatorów błędu lokalnego [66].

Przykładami elementu skończonego opisanego przez hierarchiczne funkcje bazowe są elementy: Webba [67], Savage [68], Webb-Forghani [69], Andersena-Volakisa [70]. Przykład elementu Webba, dla którego użyto hierarchicznych funkcji bazowych w sympleksowym układzie współrzędnych opisano w Dodatku A.4. W pracy zastosowano zestaw hierarchicznych funkcji bazowych zdefiniowanych dla krzywoliniowych elementów skończonych (czworościanów) zaproponowanych przez Ingelströma w [64]. Funkcje te należą do przestrzeni  $\mathcal{W}$  r. (2.22), która została zbudowana jako kombinacja funkcji z  $\tilde{\mathcal{V}}_p$  (przestrzeń skalarnych funkcji bazowych) i  $\tilde{\mathcal{A}}_p$  (przestrzeń wektorowych funkcji bazowych) zgodnie z r. (2.23)<sup>6</sup>.

$$\mathcal{W}_p = \tilde{\mathcal{W}}_1 \oplus \dots \oplus \tilde{\mathcal{W}}_p \quad (2.22)$$

$$\tilde{\mathcal{W}}_p = \tilde{\mathcal{A}}_p \oplus \nabla \tilde{\mathcal{V}}_p \quad (2.23)$$

Wszystkie funkcje bazowe rozpięte w przestrzeniach  $\tilde{\mathcal{V}}_1$ - $\tilde{\mathcal{V}}_3$  i  $\tilde{\mathcal{A}}_1$ - $\tilde{\mathcal{A}}_3$  i użyte do opisu elementów skończonych (czworościanów) w niniejszej rozprawie zapisano odpowiednio w Tab. 2.1 i Tab. 2.2. Funkcje bazowe wyrażono w układzie sympleksowym, gdzie  $\lambda_i$  to ciągła funkcja, która jest:

- liniowa w każdym czworościanie,
- jednostkowa w węźle  $i$  oraz zerowa dla pozostałych węzłów,

<sup>5</sup>W rozprawie zastosowano notację z [61], w której  $\mathcal{H}^n$  to przestrzeń wektorowych funkcji bazowych, a  $n$  wskazuje, że funkcje i rotacje funkcji (ang. curl) są wektorowymi wielomianami o maksymalnym rzędzie  $n$ .

<sup>6</sup>Tylda oznacza, iż przestrzeń jest „prawie” ortogonalna. W artykule [64], zamiast tworzyć ortogonalne funkcje bazowe zastosowano funkcje bazowe wyższego rzędu w taki sposób, iż zerują się one podczas projekcji na przestrzeń o niższym rzędzie przy zastosowaniu operatorów zdefiniowanych przez Nédélec [71]. Przewagą tej strategii jest to, że nie jest potrzebne żadne założenie na kształt elementu oraz to, iż implikuje ono pewną ortogonalność w standardowym podejściu.

TABELA 2.1: Kombinacja funkcji bazowych z [64] użytych w niniejszej rozprawie.

Przestrzeń	Skalarne funkcje bazowe	Powiązanie opisane przez węzły $\{i,j,k,l\}$
$\mathcal{V}_1$	$\lambda_i$	węzeł $\{i\}$
$\mathcal{V}_2$	$\lambda_i \lambda_j$	krawędzie $\{ij\}$
$\mathcal{V}_3$	$\lambda_i \lambda_j (\lambda_i - \lambda_j)$ $\lambda_i \lambda_j \lambda_k$	węzły $\{ij\}$ ścianki $\{ijk\}$

TABELA 2.2: Kombinacja rotacyjnych funkcji bazowych z [64] użytych w niniejszej rozprawie.

Przestrzeń	Wektorowe funkcje bazowe	Powiązanie opisane przez węzły $\{i,j,k,l\}$
$\mathcal{A}_1$	$\lambda_i \nabla \lambda_j - \lambda_j \nabla \lambda_i$	krawędzie $\{ij\}$
$\mathcal{A}_2$	$3\lambda_j \lambda_k \nabla \lambda_i - \nabla(\lambda_i \lambda_j \lambda_k)$ $3\lambda_k \lambda_i \nabla \lambda_j - \nabla(\lambda_i \lambda_j \lambda_k)$	ścianki $\{ijk\}$ ścianki $\{ijk\}$
$\mathcal{A}_3$	$4\lambda_j \lambda_k (\lambda_j - \lambda_k) \nabla \lambda_i - \nabla(\lambda_i \lambda_j \lambda_k (\lambda_j - \lambda_k))$ $4\lambda_k \lambda_i (\lambda_k - \lambda_i) \nabla \lambda_j - \nabla(\lambda_j \lambda_k \lambda_i (\lambda_k - \lambda_i))$ $4\lambda_i \lambda_j (\lambda_i - \lambda_j) \nabla \lambda_k - \nabla(\lambda_k \lambda_i \lambda_j (\lambda_i - \lambda_j))$ $4\lambda_j \lambda_k \lambda_l \nabla \lambda_i - \nabla(\lambda_i \lambda_j \lambda_k \lambda_l)$ $4\lambda_k \lambda_l \lambda_i \nabla \lambda_j - \nabla(\lambda_i \lambda_j \lambda_k \lambda_l)$ $4\lambda_l \lambda_i \lambda_j \nabla \lambda_k - \nabla(\lambda_i \lambda_j \lambda_k \lambda_l)$	ścianki $\{ijk\}$ ścianki $\{ijk\}$ ścianki $\{ijk\}$ objętość $\{ijkl\}$ objętość $\{ijkl\}$ objętość $\{ijkl\}$

- skojarzona z węzłami  $\{i\}$ , krawędziami  $\{ij\}$ , ściankami  $\{ijk\}$  i objętościami  $\{ijkl\}$ .

Dla powyżej zdefiniowanych sformułowań liczba funkcji bazowych opisujących element skończony wynosi 45 [61]. W implementacji użytej w rozprawie zdefiniowanych pierwotnie jest jednak 50 funkcji bazowych, z których 5 jest kombinacją liniową pozostałych. Na etapie wyznaczania lokalnych macierzy sztywności i bezwładności wyznaczane są zatem nadmiarowe funkcje bazowe, natomiast w trakcie konstrukcji macierzy globalnych wybierane są z nich odpowiednie funkcje liniowo niezależne - odpowiadające wspólnym krawędziom i ścianom. W zapisie macierzowym hierarchiczne wektorowe funkcje bazowe  $\vec{N}$  użyte w rozprawie zapisano w postaci macierzy  $\mathbf{N}$  o rozmiarze  $K \times D$ , gdzie  $K = 50$  określa liczbę wszystkich funkcji bazowych, a  $D = 3$  wynika z faktu, iż struktura jest trójwymiarowa.

Stopnie swobody (ang. degrees of freedom, dofs), które są ekwiwalentem niewiadomych układu równań (2.20), kojarzone są z funkcjami bazowymi zdefiniowanymi dla krawędzi, ścian czworościanu oraz objętości wewnątrz czworościanu i zapewniają warunek ciągłości składowej stycznej pola na brzegach czworościanu [64].

W przestrzeni trójwymiarowej, dla wektorowych funkcji bazowych wyprowadzenie słabej postaci równania różniczkowego (2.4) i zastosowanie metody Galerkinia prowadzi do wyznaczenia dwóch lokalnych macierzy opisujących pojedynczy element skończony. Pierwsza, macierz sztywności  $\mathbf{S}^{(e)}$ , związana jest ze składnikiem  $\nabla \times \vec{E}$ , a druga, macierz bezwładności  $\mathbf{T}^{(e)}$ , ze składnikiem  $\vec{E}$  równania (2.6). Elementy macierzy można zapisać w postaci:

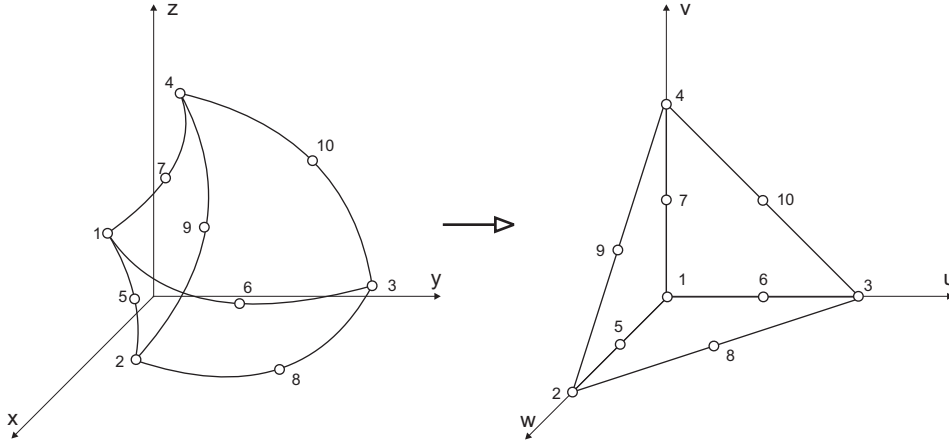
$$\mathbf{S}_{ij}^{(e)} = \iiint_V (\nabla \times \vec{N}_i) \mu^{-1} (\nabla \times \vec{N}_j) dV \quad (2.24)$$

$$\mathbf{T}_{ij}^{(e)} = \iiint_V (\vec{N}_i \epsilon_r \vec{N}_j) dV \quad (2.25)$$

gdzie  $V$  to objętość elementu skończonego  $(e)$ ,  $\vec{N}$  określa zestaw hierarchicznych funkcji bazowych opisujących element skończony,  $\mathbf{S}^{(e)}$  i  $\mathbf{T}^{(e)}$  to odpowiednio lokalne macierze elementów sztywności i bezwładności.

### Krzywoliniowe wektorowe elementy skończone

Kolejnym ze sposobów poprawy dokładności analizy przy użyciu metody elementów skończonych, szczególnie w przypadkach gdy analizowane struktury mają zakrzywione powierzchnie, jest zastosowanie elementów krzywoliniowych (rys. 2.3). Aby obliczyć lokalne macierze elementu (sztywności  $\mathbf{S}^{(e)}$  i bezwładności  $\mathbf{T}^{(e)}$ ) należy wykonać całkowanie wektorowych funkcji bazowych i rotacji wektorowych funkcji bazowych dla referencyjnego elementu skończonego [55]. Element krzywoliniowy (zapisany przy użyciu układu współrzędnych  $(x,y,z)$ ) jest przekształcany do elementu prostoliniowego z lokalnymi współrzędnymi  $(u,v,w)$  przy użyciu macierzy Jacobiego ( $\mathbf{J}$ ) (rys. 2.3):



RYSUNEK 2.3: Odwzorowanie przekształcające element krzywoliniowy w czworościan w układzie współrzędnych  $(u,v,w)$ .

$$\begin{bmatrix} \frac{\partial}{\partial u} \\ \frac{\partial}{\partial v} \\ \frac{\partial}{\partial w} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u} \\ \frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \frac{\partial z}{\partial v} \\ \frac{\partial x}{\partial w} & \frac{\partial y}{\partial w} & \frac{\partial z}{\partial w} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \quad (2.26)$$

Współrzędne  $(u,v,w)$  odpowiadają współrzędnym simpleksowym  $(\lambda_2, \lambda_3, \lambda_4)$ , a czwartą współrzędną można określić na podstawie warunku normalizującego jako  $\lambda_1 = 1 - \lambda_2 - \lambda_3 - \lambda_4$  (Dodatek A.1, r. (A.3)).

W analogiczny sposób między układami współrzędnych transformowane są składowe funkcji bazowych:

$$\begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix} = \mathbf{J} \begin{bmatrix} N_u \\ N_v \\ N_w \end{bmatrix} \quad (2.27)$$

Dla elementów krzywoliniowych lokalne macierze sztywności i bezwładności z r. (2.24)-(2.25) można przepisać przy użyciu równań:

$$\mathbf{S}_{ij}^{(e)} = \iiint_V (\nabla \times \vec{N}_i) \mu^{-1} (\nabla \times \vec{N}_j) dx dy dz = \iiint_V \mathbf{N}_{R_i}^T \mathbf{J} \mu^{-1} \mathbf{J}^T \mathbf{N}_{R_j} \frac{1}{\det(\mathbf{J})} du dv dw \quad (2.28)$$

$$\mathbf{T}_{ij}^{(e)} = \iiint_V \vec{N}_i \epsilon_r \vec{N}_j dx dy dz = \iiint_V \begin{bmatrix} N_{iu} N_{iv} N_{iw} \end{bmatrix} \mathbf{J}^{-T} \epsilon_r \mathbf{J}^{-1} \begin{bmatrix} N_{ju} \\ N_{jv} \\ N_{jw} \end{bmatrix} \det(\mathbf{J}) du dv dw \quad (2.29)$$

gdzie  $(\cdot)^T$  oznacza operację transpozycji macierzy,  $(\cdot)^{-T}$  oznacza transpozycję macierzy odwrotnej, a  $\mathbf{N}_R$  to macierz pomocnicza do wyznaczenia rotacji funkcji bazowych:

$$\mathbf{N}_{Ri}^T = \left[ \left( \frac{\partial N_w}{\partial v} - \frac{\partial N_v}{\partial w} \right), \left( \frac{\partial N_u}{\partial w} - \frac{\partial N_w}{\partial u} \right), \left( \frac{\partial N_v}{\partial u} - \frac{\partial N_u}{\partial v} \right) \right] \quad (2.30)$$

W przypadku, gdy do opisu elementu skończonego użyto hierarchicznych funkcji bazowych, wykonanie całkowania z r. (2.28)-(2.29) metodą analityczną nie jest kwestią oczywistą i wiązałoby się z bardzo dużym kosztem numerycznym. Z tego powodu zdecydowano się wykorzystać kwadratury Gaussa [72] do wykonania całkowania numerycznego. Kwadraturę Gaussa dla pojedynczego czworościanu, można zapisać w postaci r. (2.31):

$$\iiint_V f(x, y, z) dV \approx |V| \cdot \sum_{i=1}^Q w_i \cdot f(p_i) \quad (2.31)$$

gdzie  $Q$  określa liczbę punktów kwadratury  $p_i$ , a  $w_i$  określa wagi kwadratury w układzie simpleksowym. Aby zapewnić satysfakcjonującą dokładność całkowania numerycznego należy wykorzystać kwadratury wysokiego rzędu, co wiąże się ze znaczącym kosztem pamięciowym i obliczeniowym. Z tego powodu wykorzystanie elementów krzywoliniowych jest zasadne w sąsiedztwie zakrzywionych obszarów analizowanej struktury.

Dla powyżej zdefiniowanych sformułowań (hierarchiczne funkcje bazowe wyższego rzędu, elementy krzywoliniowe, kwadratury Gaussa) lokalne macierze elementów obliczyć można zgodnie z r. (2.32)-(2.33):

$$\mathbf{S}^{(e)} \approx \frac{1}{6} \sum_{i=1}^Q w_i \mathbf{N}_{Ri} (\mathbf{J}_i^T \mu^{-1} \mathbf{J}_i) \mathbf{N}_{Ri}^T \frac{1}{\det(\mathbf{J}_i)} \quad (2.32)$$

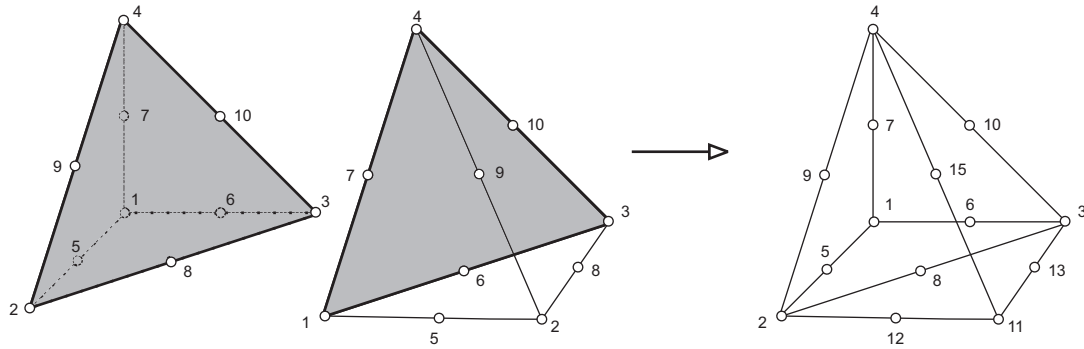
$$\mathbf{T}^{(e)} \approx \frac{1}{6} \sum_{i=1}^Q w_i \mathbf{N}_i (\mathbf{J}_i^{-T} \epsilon \mathbf{J}_i^{-1}) \mathbf{N}_i^T \det(\mathbf{J}_i^{-1}) \quad (2.33)$$

gdzie:  $Q$  oznacza liczbę punktów kwadratury,  $\mathbf{J}_i$  oznacza macierz Jacobiego (2.26),  $(\cdot)^{-T}$  oznacza transpozycję odwrotności macierzy,  $\mathbf{N}_i$  oznacza macierz, która przechowuje wartości hierarchicznych funkcji bazowych  $\tilde{N}_i$ ,  $\mathbf{N}_{Ri}$  oznacza macierz, która przechowuje wartości rotacji hierarchicznych funkcji bazowych  $\nabla \times \tilde{N}_i$ .

## 2.2 Konstrukcja macierzy globalnych

W p. 2.1 pokazano formuły pozwalające na generację lokalnych macierzy elementów sztywności  $\mathbf{S}^{(e)}$  i bezwładności  $\mathbf{T}^{(e)}$ . Następnym krokiem metody elementów skończonych jest konstrukcja rzadkich macierzy globalnych sztywności ( $\mathbf{S}$ ) i bezwładności ( $\mathbf{T}$ ). Wśród formatów reprezentacji macierzy rzadkiej najbardziej popularnymi formatami są COO (ang. Coordinate format), CRS (ang. Compressed Row Storage), CCS (ang. Compressed Column Storage), Ellpack-Itpack [3]. To co wyróżnia formaty to sposób dostępu do elementów niezerowych oraz zapotrzebowanie na pamięć. Wyżej wymienione formaty zostały scharakteryzowane przy opisie implementacji mnożenia macierzy rzadkiej przez wektor (p. 6.2.1) i poniżej przedstawiono tylko ich główne właściwości. Format COO jest bardziej uniwersalny, gdyż trzy wektory przechowują pełną informację nt. wartości elementów niezerowych oraz ich indeksów wierszy i kolumn. Z drugiej strony format CRS cechuje się mniejszym zapotrzebowaniem na pamięć (wektor indeksów wierszy elementów niezerowych jest skompresowany) i dlatego jest on często





RYSUNEK 2.4: Łączenie dwóch elementów skończonych (czworościanów) - odwzorowanie lokalnej indeksacji na indeksację globalną. Na rysunku pogrubiono wspólne krawędzie, a wspólna ściana czworościanów została pokolorowana.

wykorzystywany jako format bazowy w pakietach i bibliotekach dedykowanych obliczeniom na macierzach rzadkich [43, 73].

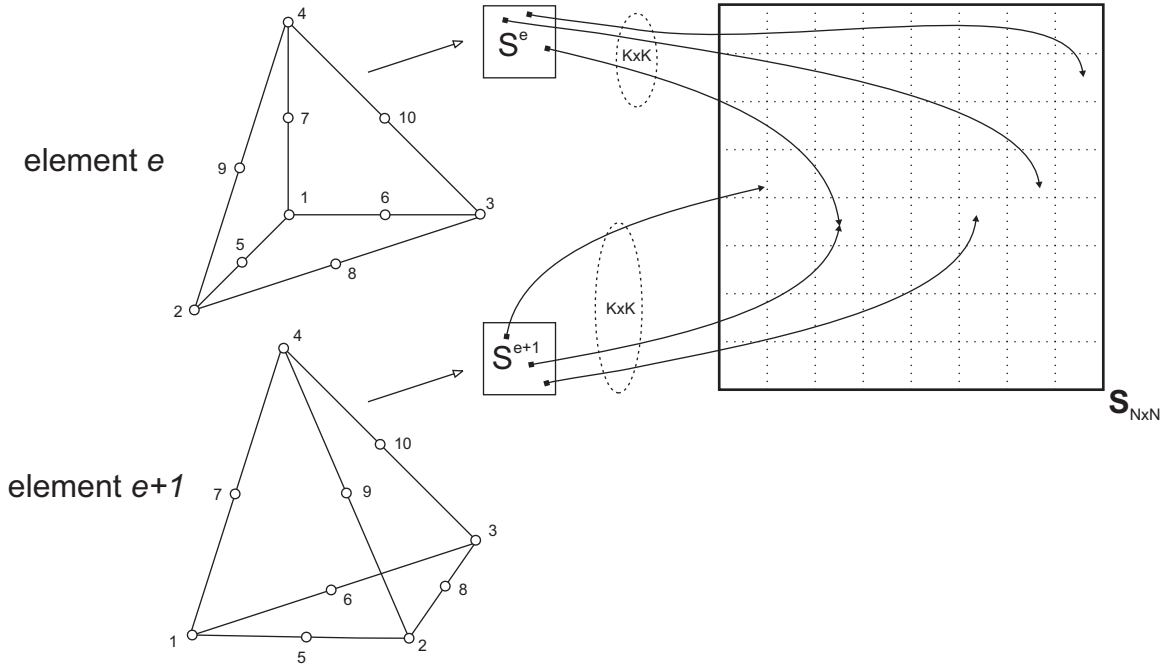
Popularnym sposobem wykonania procesu składania macierzy (ang. matrix assembly) jest przetwarzanie struktury element po elemencie [1, 56]. W siatce elementów, czworościany mają wspólne węzły, krawędzie i ścianki (rys. 2.4). W celu określenia wzajemnego powiązania pomiędzy funkcjami opisującymi wartości pola EM w poszczególnych elementach skończonych wprowadza się globalną numerację węzłów, krawędzi i ścian elementów [1, 56]. Na podstawie globalnej indeksacji definiowane jest odwzorowanie, które określa wartości macierzy globalnych w funkcji wartości lokalnych. W rezultacie poszczególne elementy niezerowe budowanej macierzy są sumą wartości pochodzących z macierzy lokalnych. Dla elementów wektorowych procedura budowy macierzy globalnych musi uwzględniać również fakt, iż funkcje bazowe mają swój kierunek i łączenie elementów wymaga sprawdzenia kierunków. W takim przypadku wprowadza się tablice łączące lokalną i globalną numerację krawędzi (element tablicy, który posiada wartość ujemną ma lokalną krawędź zorientowaną w przeciwnym kierunku co globalna krawędź). Na podstawie globalnej numeracji krawędzi oraz orientacji krawędzi następuje dodanie bądź odejmowanie wartości macierzy lokalnych i wpisanie ich do macierzy globalnych (rys. 2.5).

Dla powyższych sformułowań, w których zastosowano hierarchiczne funkcje bazowe globalne macierze współczynników z r. (2.10) mają charakterystyczną postać, gdyż można w nich wydzielić podmacierze  $\mathbf{M}_{ij}^E$  odpowiadające różnym rzędom funkcji bazowych użytych do opisu elementów skończonych (indeksy 1,2,3 odpowiadają stopniom funkcji bazowych):

$$\mathbf{A} = \mathbf{S} - k_0^2 \mathbf{T} = \begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} & \mathbf{S}_{13} \\ \mathbf{S}_{21} & \mathbf{S}_{22} & \mathbf{S}_{23} \\ \mathbf{S}_{31} & \mathbf{S}_{32} & \mathbf{S}_{33} \end{bmatrix} - k_0^2 \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} & \mathbf{T}_{13} \\ \mathbf{T}_{21} & \mathbf{T}_{22} & \mathbf{T}_{23} \\ \mathbf{T}_{31} & \mathbf{T}_{32} & \mathbf{T}_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_{11}^E & \mathbf{M}_{12}^E & \mathbf{M}_{13}^E \\ \mathbf{M}_{21}^E & \mathbf{M}_{22}^E & \mathbf{M}_{23}^E \\ \mathbf{M}_{31}^E & \mathbf{M}_{32}^E & \mathbf{M}_{33}^E \end{bmatrix} \quad (2.34)$$

Powyższa właściwość macierzy umożliwia zastosowanie wielopoziomowego operatora ściskającego w metodzie iteracyjnego rozwiązywania układów równań liniowych [61], który został przedstawiony w kolejnym punkcie. W ogólności macierz rzadka  $\mathbf{A}$  jest niesymetryczna, jednakże w przypadku gdy  $\epsilon$  i  $\mu$  są tensorami symetrycznymi lub tak jak w przypadku rozprawy skalarami ( $\epsilon = 1$ ,  $\mu = 1$ ), to macierz rzadka jest symetryczna.





RYSUNEK 2.5: Budowa macierzy globalnych sztywności  $\mathbf{S}$ . Po obliczeniu lokalnych macierzy sztywności  $\mathbf{S}^{(e)}$  i  $\mathbf{S}^{(e+1)}$  o rozmiarach  $K \times K$  dla czworościanów (e) i (e+1), w etapie budowy macierzy globalnej elementy zapisywane są w macierzy rzadkiej  $\mathbf{S}$  o rozmiarze  $N \times N$ , gdzie  $N$  to sumaryczna liczba stopni swobody.

## 2.3 Rozwiązanie układu równań

Po wykonaniu dyskretyzacji przestrzeni obliczeniowej, obliczeniu lokalnych macierzy oraz zbudowaniu macierzy globalnych, kolejnym etapem analizy z wykorzystaniem metody elementów skończonych jest rozwiązanie macierzowego układu równań  $\mathbf{A}\mathbf{X} = \mathbf{B}$  (patrz. r. (2.20)), gdzie  $\mathbf{A}$  to globalna macierz współczynników będąca funkcją globalnych macierzy sztywności i bezwładności oraz liczby falowej,  $\mathbf{B}$  to macierz reprezentująca pobudzenie,  $\mathbf{X}$  to macierz poszukiwanych amplitud funkcji bazowych, które opisują falę elektromagnetyczną. Poniżej przeprowadzono dyskusję rozwiązania układu równań liniowych dla przypadku, w którym  $\mathbf{B}=\mathbf{b}$  i  $\mathbf{X}=\mathbf{x}$  są macierzami jednokolumnowymi (rozwiązanie układu w jednych wrotach) i wtedy równanie przyjmuje postać  $\mathbf{A}\mathbf{x} = \mathbf{b}$ .

Wyróżnić można dwa główne typy metod rozwiązywania macierzowego układu równań liniowych (2.20), czyli techniki bezpośrednie i iteracyjne. W podejściu bezpośrednim poszukiwane rozwiązanie układu równań liniowych oparte jest najczęściej na dekompozycji LU [3]. Rezultatem rozkładu macierzy  $\mathbf{A}$  są dwie macierze (faktory) - trójkątna dolna  $\mathbf{L}$  oraz trójkątna górna  $\mathbf{U}$ , które umożliwiają szybkie i dokładne rozwiązywanie układu równań (2.20) w dwóch etapach podstawienia „w przód” (ang. forward substitution) i „wstecz” (ang. back substitution):

$$\mathbf{L}\mathbf{y} = \mathbf{b} \quad (2.35)$$

$$\mathbf{U}\mathbf{x} = \mathbf{y} \quad (2.36)$$

W metodzie MES rozmiar macierzy rzadkiej  $\mathbf{A}$  z r. (6.2) zależy od rozmiaru dziedziny, dyskretyzacji i użytych funkcji bazowych. W praktyce, rozmiar macierzy może osiągnąć miliony, co spowoduje, że problem ten nie może być rozwiązany przy użyciu metod

bezpośrednich, które wymagają wielokrotnie więcej pamięci do wykonania faktoryzacji i przechowywania macierzy (faktorów)  $\mathbf{L}$  i  $\mathbf{U}$ , niż jest to potrzebne do przechowania macierzy współczynników  $\mathbf{A}$ .

Mając na uwadze możliwość rozwiązywania większych układów równań zasadnym jest użycie metod iteracyjnych [3], których zapotrzebowanie na pamięć ogranicza się do macierzy  $\mathbf{A}$  i wektorów pomocniczych. W technikach iteracyjnych poszukiwane rozwiązanie układu równań (2.20) znajduwane jest w procesie iteracyjnym  $\mathbf{x}_{i+1} = \mathbf{K}_{(\mathbf{A}, \mathbf{b})}(\mathbf{x}_i)$ , który zwykle rozpoczyna działanie od rozwiązania początkowego  $\mathbf{x}_0$  (może być bardzo niedokładne) i w kolejnych iteracjach zwiększa dokładność wyznaczanego rozwiązania do zadanej dokładności. W każdej iteracji można określić wektor błędu  $\mathbf{e}_i$  jako:

$$\mathbf{e}_i = \mathbf{x}_i - \mathbf{x} \quad (2.37)$$

gdzie:  $\mathbf{x}_i$  to rozwiązanie w „i”-tej iteracji, a  $\mathbf{x}$  to rozwiązanie dokładne. Metoda iteracyjna zbiega się do rozwiązania dokładnego jeżeli norma  $\|\mathbf{e}_i\|$  dąży do zera, a taka zależność zachodzi, gdy  $\sigma(\mathbf{K}) < 1$ :

$$\sigma(K) = \max_i |\Lambda_i(K)| \quad (2.38)$$

gdzie  $\sigma$  to promień spektralny macierzy,  $\Lambda_i$  to wartość własna.

Jedną z najbardziej popularnych technik iteracyjnych jest metoda gradientów sprzężonych (ang. Conjugate Gradient method, CG) [3, 4, 74]. Metoda gradientów sprzężonych dedykowana jest rozwiązaniu układu równań w przypadku gdy macierz jest symetryczna i dodatnio określona:

$$\forall \mathbf{z} \quad \mathbf{z}^T \mathbf{A} \mathbf{z} > 0 \quad (2.39)$$

Spełnienie warunku (2.39) gwarantuje, że wszystkie wartości własne macierzy  $\mathbf{A}$  są dodatnie.

W przypadku macierzy wygenerowanej przy użyciu sformułowań MES z poprzedniego punktu macierz rzadka jest symetryczna, ale jest nieokreślona (ang. indefinite). Jest to spowodowane faktem, że jej składniki mają odmienny charakter - jeden - macierz sztywności  $\mathbf{S}$  jest półdodatnio określona (ang. positive semidefinite), i ma liczne statyczne (tj. odpowiadające zdegenerowanej zerowej wartości własnej) rozwiązania, które w niektórych przypadkach mogą być źródłem tzw. fałszywych rodzajów<sup>7</sup>, a drugi, czyli macierz  $-k_0^2 \mathbf{T}$  jest ujemnie określona ponieważ ma ujemne wartości własne. Mimo wszystko metodę gradientów sprzężonych stosuje się do rozwiązywania układów równań w MES. Dodatnia określoność macierzy gwarantuje, że podczas obliczeń w procesie iteracyjnym nie wystąpią nieprawidłowe operacje (dzielenie przez zero), jednakże w praktyce taka sytuacja nie występuje mimo, że macierz jest nieokreślona i metoda CG jest wykorzystywana w analizie MES problemów elektromagnetycznych [61, 75].

W metodzie CG dokonuje się ortogonalnej projekcji na bazie podprzestrzeni Kryłowa  $\mathcal{K}_i$ :

$$\mathcal{K}_i(\mathbf{r}_0, \mathbf{A}) = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \dots, \mathbf{A}^{i-1}\mathbf{r}_0\} \quad (2.40)$$

Poszukiwany wektor  $\mathbf{x}$  tworzony jest z residuów  $\mathbf{r}_i$ , gdzie  $\mathbf{r}_0$  jest residuum początkowym:

$$\mathbf{r}_i = \mathbf{b} - \mathbf{A}\mathbf{x}_i \quad (2.41)$$

<sup>7</sup>Rozwiązania statyczne rozpinają jądro operatora różniczkowego ( $\mathcal{L} = \nabla \times \frac{1}{\mu} \nabla \times$ ) - należą do nich wszystkie gradienty funkcji skalarnych gdyż  $\nabla \times (\nabla \Phi) = 0$ . Funkcje bazowe wektorowe używane w MES aproksymują jedynie to jądro, co prowadzi do wartości własnych bliskich zera [61]

Rozwiązanie budowane jest w taki sposób, aby w  $i$ -tej iteracji residuum było ortogonalne do residuum z poprzedniej iteracji  $r$ . (2.42).

$$\mathbf{r}_{i+1}^T \mathbf{r}_i = 0 \quad (2.42)$$

Punktem wyjściowym metody gradientów sprzężonych jest forma kwadratowa [74]:

$$f(x) = 0.5 \cdot \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c \quad (2.43)$$

Przy założeniu, że macierz  $\mathbf{A}$  jest symetryczna i poddając (2.43) operacji gradientu  $f'(\mathbf{x}) = \mathbf{A} \mathbf{x} - \mathbf{b}$ , residuum równe jest  $\mathbf{r} = -f'(\mathbf{x})$  i wskazuje kierunek najszybszego spadku. Przyrównując gradient do zera otrzymujemy rozwiązanie równania  $\mathbf{A} \mathbf{x} = \mathbf{b}$ . Nadając powyższym równaniom postać iteracyjną, poszukiwane rozwiązanie przyjmuje postać:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{r}_i \quad (2.44)$$

Z równań (2.41), (2.42) i (2.44) otrzymujemy współczynnik  $\alpha$ .

$$(\mathbf{b} - \mathbf{A} \mathbf{x}_{i+1})^T \mathbf{r}_i = 0 \quad (2.45)$$

$$(\mathbf{b} - \mathbf{A}(\mathbf{x}_i + \alpha \mathbf{r}_i))^T \mathbf{r}_i = 0 \quad (2.46)$$

$$(\mathbf{b} - \mathbf{A} \mathbf{x}_i)^T \mathbf{r}_i - \alpha (\mathbf{A} \mathbf{r}_i)^T \mathbf{r}_i = 0 \quad (2.47)$$

$$(\mathbf{b} - \mathbf{A} \mathbf{x}_i)^T \mathbf{r}_i = \alpha (\mathbf{A} \mathbf{r}_i)^T \mathbf{r}_i \quad (2.48)$$

$$\mathbf{r}_i^T \mathbf{r}_i = \alpha \mathbf{r}_i^T (\mathbf{A} \mathbf{r}_i) \quad (2.49)$$

$$\alpha = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_i^T \mathbf{A} \mathbf{r}_i} \quad (2.50)$$

Wektor nowego kierunku  $\mathbf{d}_{i+1}$  jest  $\mathbf{A}$ -ortogonalny do poprzedniego wektora kierunku  $\mathbf{d}_i$ :

$$\mathbf{d}_{i+1} = \mathbf{r}_{i+1} + \beta \mathbf{d}_i \quad (2.51)$$

gdzie współczynnik  $\beta = \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}$  wynika z ortogonalizacji Gram-Schmidta. Wszystkie operacje metody CG zostały zebrane w Wydruku 2.1 [74]. Warto dodać, że dla problemów stratnych, w których generowana w MES macierz rzadka  $\mathbf{A}$  jest zespolona należy stosować metodę COCG (ang. Conjugate Orthogonal Gradient Method) [76]. W metodzie COCG podprzestrzeń Kryłowa definiuje się jako  $\mathcal{K}_i(\bar{\mathbf{r}}_0, \bar{\mathbf{A}})$ , gdzie  $(\cdot)$  oznacza sprzężenie.

Algorytm CG występuje także w postaci PCG (ang. Preconditioned Conjugate Gradient) z operatorem ściskającym  $\mathbf{M}$  (Wydruk 2.2), czyli macierzą lub procedurą transformującą problem liniowy do równoważnego problemu o tym samym rozwiązaniu:

$$\mathbf{M}^{-1} \mathbf{A} \mathbf{x} = \mathbf{M}^{-1} \mathbf{b} \quad (2.52)$$

Zastosowanie operatora ściskającego ma na celu poprawę zbieżności algorytmu iteracyjnego, gdyż przy doborze odpowiedniego operatora,  $\mathbf{M}^{-1} \mathbf{A}$  ma lepsze właściwości widmowe niż  $\mathbf{A}$  (mniejszy promień spektralny  $\sigma$ ). Przykładem operatora ściskającego jest prekondycjoner Jacobiego, w którym macierz  $\mathbf{M}$  to macierz diagonalna, która na diagonalu ma elementy z diagonalu macierzy  $\mathbf{A}$  [4]:

$$\mathbf{M} = \begin{bmatrix} \mathbf{a}_{1,1} & 0 & \cdots & 0 \\ 0 & \mathbf{a}_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{a}_{N,N} \end{bmatrix} \quad (2.53)$$

```

1  i = 0
2  r = b - Ax
3  d = r
4  dnew = rTr
5  d0 = dnew
6  while (i < imax && dnew > ε2d0) {
7      q = Ad
8      dq = dTq
9      α =  $\frac{d_{new}}{dq}$ 
10     x = x + αd
11     r = r - αq
12
13     dold = dnew
14     dnew = rTr
15     β =  $\frac{d_{new}}{d_{old}}$ 
16     d = r + βd
17     i = i + 1
18 }
```

WYDRUK 2.1: Metoda gradientów sprzężonych

```

1  i = 0
2  r = b - Ax
3  d = M-1r
4  dnew = rTd
5  d0 = dnew
6  while (i < imax && dnew > ε2d0) {
7      q = Ad
8      dq = dTq
9      α =  $\frac{d_{new}}{dq}$ 
10     x = x + αd
11     r = r - αq
12     s = M-1r
13     dold = dnew
14     dnew = rTs
15     β =  $\frac{d_{new}}{d_{old}}$ 
16     d = s + βd
17     i = i + 1
18 }
```

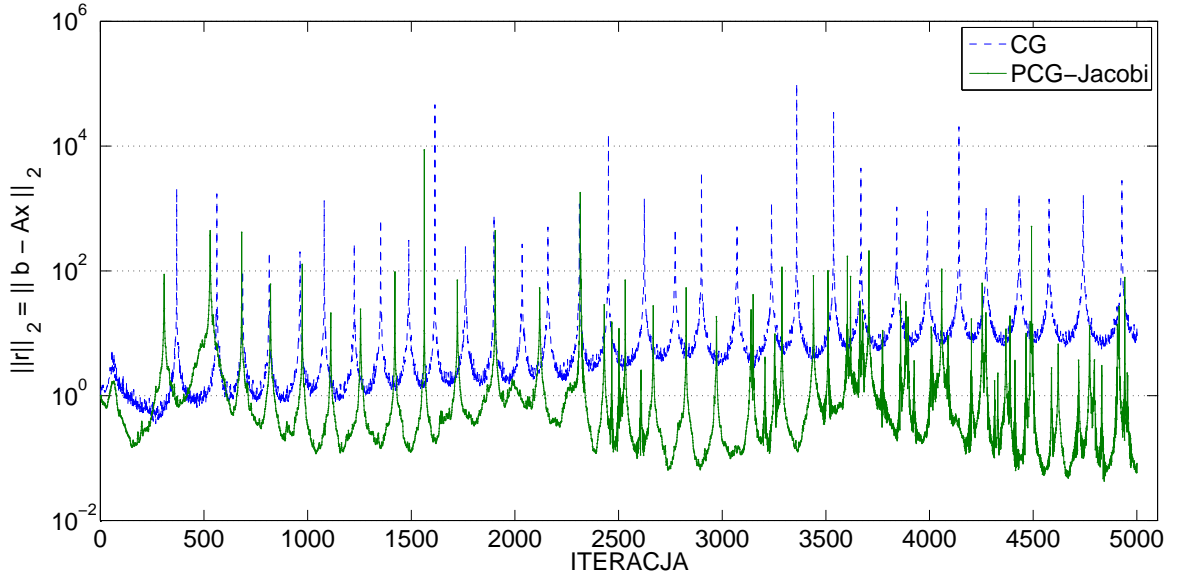
WYDRUK 2.2: Metoda gradientów sprzężonych z operatorem ściskającym  $\mathbf{M}$ .

Niestety operator ściskający Jacobiego nieznacznie poprawia zbieżność rozwiązywania układów równań powstających w wyniku dyskretyzacji równania falowego [77]. Testy wykazały, że uzyskanie satysfakcjonującej zbieżności dla problemów wygenerowanych w MES przy użyciu podstawowych operatorów ściskających i przy zachowaniu dużej wydajności obliczeń było niemożliwe (rys. 2.6).

Przyczyn słabej zbieżności metody iteracyjnego rozwiązywania układu równań generowanych w MES w przypadku wektorowego równania falowego jest kilka. Po pierwsze, macierz  $\mathbf{A}$  jest nieokreślona (ang. indefinite). Po drugie, zwiększenie gęstości siatki (zmniejszenie rozmiarów czworościanów) powoduje wzrost promienia spektralnego. Po trzecie obecność fałszywych, niefizycznych rodzajów [61] powoduje, że w procesie iteracyjnym pojawiają się składniki, których nie można wyeliminować poprzez np. deflację. Po czwarte zastosowanie hierarchicznych funkcji bazowych sprawia, że rośnie wskaźnik uwarunkowania macierzy [78]:

$$\kappa = \frac{\Lambda_{max}(\mathbf{A})}{\Lambda_{min}(\mathbf{A})} \quad (2.54)$$

gdzie  $\Lambda_{max}$  i  $\Lambda_{min}$  to odpowiednio największa i najmniejsza wartość własna macierzy  $\mathbf{A}$ . Wraz ze wzrostem wskaźnika uwarunkowania rośnie liczba iteracji metody gradientów sprzężonych (liczba iteracji jest proporcjonalna do kwadratu wskaźnika uwarunkowania macierzy). Poprawę uwarunkowania macierzy uzyskuje się tworząc ortogonalne funkcje bazowe [79]. Tak jak to zostało opisane wcześniej funkcje bazowe wykorzystane w niniejszej rozprawie [64], są „prawie” ortogonalne, gdyż zastosowano funkcje bazowe wyższego rzędu, które zerują się podczas projekcji na przestrzeń o niższym rzędzie.



RYSUNEK 2.6: Zbieżność (norma euklidesowa wektora residualnego) metody gradientów sprzężonych bez (CG) i z operatorem ściskającym Jacobiego (PCG-Jacobi). Problem testowy: filtr grzebieniowy 9-rzędu, rozmiar macierzy  $N = 806811$  (Tab. 4.1).

W literaturze spotkać można kilka rozwiązań, które pozwalają poprawić zbieżność iteracyjnego rozwiązywania układu równań generowanego w MES. Jednym z rozwiązań jest zastosowanie metod wielosiatkowych [80]. Metody iteracyjne słabo redukują wolnozmiennie składowe przestrzennego rozwinięcia błędu rozwiązania. W metodach wielosiatkowych wolnozmienny błąd po rzutowaniu na rozrzedzoną siatkę staje się szybkozmienny i jest łatwiejszy do usunięcia [80]. Innym sposobem stłumienia rodzajów własnych jest zastosowanie wielopoziomowych operatorów ściskających wykorzystujących fakt, iż do opisu elementów skończonych użyto hierarchiczne funkcje bazowe [61]. Mechanizm ich funkcjonowania jest analogiczny jak dla metod wielosiatkowych.

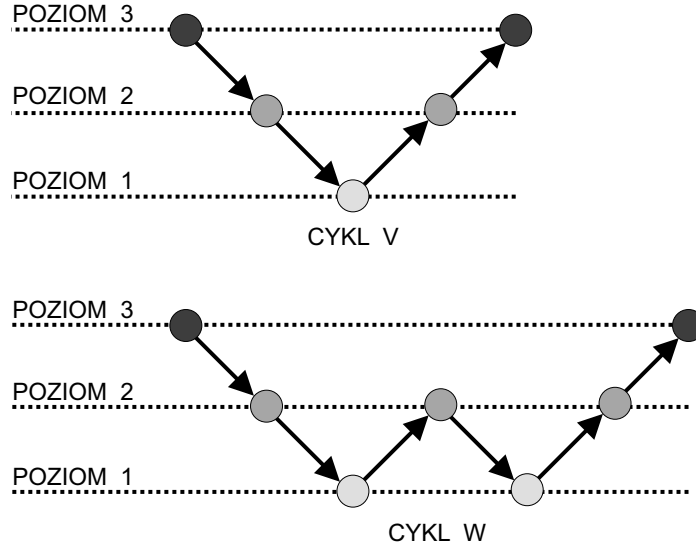
### 2.3.1 Hierarchiczny wielopoziomowy operator ściskający

Hierarchiczny wielopoziomowy operator ściskający jest zestawem operacji, które zostały przedstawione na Wydruku 2.3. Operator ściskający może przyjmować różne schematy (V,W) i posiadać kilka poziomów (rys. 2.7). Poniżej omówiony został operator ściskający o schemacie V. W prekondycjonerze wielopoziomowym macierz  $\mathbf{A}$  podzielona jest na podmacierze  $\mathbf{M}_{ij}^E$ , odpowiadające różnym rzędom funkcji bazowych użytych do opisu elementów skończonych. Macierze w dwupoziomowym ( $\mathbf{A}_{V2}$ ) i trzypoziomowym ( $\mathbf{A}_{V3}$ ) operatorze ściskającym:

$$\mathbf{A} = \mathbf{A}_{V2} = \begin{bmatrix} \mathbf{M}_{11}^E & \mathbf{M}_{12}^E \\ \mathbf{M}_{21}^E & \mathbf{M}_{22}^E \end{bmatrix} \quad (2.55)$$

$$\mathbf{A} = \mathbf{A}_{V3} = \begin{bmatrix} \mathbf{M}_{11}^E & \mathbf{M}_{12}^E & \mathbf{M}_{13}^E \\ \mathbf{M}_{21}^E & \mathbf{M}_{22}^E & \mathbf{M}_{23}^E \\ \mathbf{M}_{31}^E & \mathbf{M}_{32}^E & \mathbf{M}_{33}^E \end{bmatrix} \quad (2.56)$$

gdzie indeksy 1,2,3 odpowiadają stopniom funkcji bazowych.



RYСУNEK 2.7: Schemat cyklu V i W trzypoziomowego operatora ściskającego.

```

1   $\mathbf{z}_E = \text{Hie-ML}(\mathbf{r}_E, i)$ 
2   $\mathbf{z}_E = 0$ 
3  if  $i == 0$  then
4       $\mathbf{M}_{11}^E \mathbf{z}_E = \mathbf{r}_E$     (dolny poziom)
5  else
6      Pre-smoothing ( $\mathbf{z}_E, \mathbf{r}_E$ ) (najwyższy poziom)
7       $\mathbf{r}_E^{i-1} = \mathbf{r}_E - \mathbf{M}_{i-1,i}^E \mathbf{z}_E$ 
8       $\mathbf{z}_E^{i-1} = \text{Hie-ML}(\mathbf{r}_E, i-1)$ 
9       $\mathbf{z}_E^i = \mathbf{r}_E - \mathbf{M}_{i,i-1}^E \mathbf{z}_E^{i-1}$ 
10     Post-smoothing( $\mathbf{z}_E, \mathbf{r}_E$ )    (najwyższy poziom)
```

WYDRUK 2.3: Hierarchiczny wielopoziomowy operator ściskający (ang. Hierarchical Multi-level, *Hie-ML*)

Analizując algorytm wielopoziomowego operatora ściskającego o cyklu V (Wydruk 2.3) wyróżnić można: procedury „wygładzające” (ang. pre-smoothing, post-smoothing) (linie: 6, 10), mnożenie macierzy rzadkich łączące poziomy operatora ściskającego (linie: 7, 9) oraz rozwiązanie układu równań na dolnym poziomie (linia: 4).

Dla etapów „wygładzających” można zastosować relaksacyjne iteracyjne metody rozwiązywania układu  $\mathbf{M}\mathbf{z} = \mathbf{g}$  [3]. Metody relaksacyjne mają taką właściwość, iż schemat iteracji ( $\mathbf{K}$ ) jest niezależny od  $\mathbf{z}_i$  oraz  $i$  w trakcie procesu iteracyjnego:

$$\mathbf{z}_i = \mathbf{K}_{M,g}(\mathbf{z}_{i-1}) \quad (2.57)$$

Do metod relaksacyjnych zaliczyć można metodę Gaussa-Seidela, metody Jacobiego oraz metodę SOR (ang. Successive Over-Relaxation). Na potrzeby tych metod macierz zapisywana jest w postaci:

$$\mathbf{M} = \mathbf{D} - \mathbf{L} - \mathbf{U} \quad (2.58)$$

gdzie macierz  $\mathbf{D}$  jest macierzą diagonalną, macierz  $-\mathbf{L}$  zawiera elementy poniżej diagonal, a macierz  $-\mathbf{U}$  elementy powyżej diagonal. Metoda Gaussa-Seidela wyznacza rozwiązanie w kolejnej iteracji na podstawie rozwiązania otrzymanego w poprzedniej

iteracji wg. wzoru opisanego w równaniu (2.59), w którym wykonuje się tzw. „podstawienie w przód” (ang. forward substitution) lub wg. wzoru opisanego w równaniu (2.60), w którym wykonuje się tzw. „podstawienie wstecz” (ang. back substitution).

$$(\mathbf{D} - \mathbf{L})\mathbf{z}_{i+1} = \mathbf{U}\mathbf{z}_i + \mathbf{g}\mathbf{z}_{i+1} = (\mathbf{D} - \mathbf{L})^{-1}[\mathbf{U}\mathbf{z}_i + \mathbf{g}] \quad (2.59)$$

$$(\mathbf{D} - \mathbf{U})\mathbf{z}_{i+1} = \mathbf{L}\mathbf{z}_i + \mathbf{g}\mathbf{z}_{i+1} = (\mathbf{D} - \mathbf{U})^{-1}[\mathbf{L}\mathbf{z}_i + \mathbf{g}] \quad (2.60)$$

Kolejną metodą relaksacyjną jest metoda SOR, dla której wektor wynikowy obliczany jest na podstawie procedury (2.61). Niestety znalezienie optymalnego dla danego problemu współczynnika relaksacji ( $\omega \in \{0, 2\}$ ) nie jest zagadnieniem trywialnym [3].

$$\mathbf{z}_{i+1} = (\mathbf{D} + \omega\mathbf{L})^{-1}(\omega\mathbf{g} - [\omega\mathbf{U} + (\omega - 1)\mathbf{D}]\mathbf{z}_i) \quad (2.61)$$

Inną metodą relaksacyjną jest metoda Jacobiego, w której rozwiązanie w kolejnej iteracji  $\mathbf{z}_{i+1}$  otrzymywane jest przy użyciu procedury (2.62). Procedurę tę, można zapisać w bardziej zwartej formie (2.63) przy użyciu podstawienia ( $\mathbf{R}_J = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$ ):

$$\mathbf{z}_{i+1} = \mathbf{D}^{-1}(\mathbf{g} + (\mathbf{L} + \mathbf{U})\mathbf{z}_i) \quad (2.62)$$

$$\mathbf{z}_{i+1} = \mathbf{R}_J\mathbf{z}_i + \mathbf{D}^{-1}\mathbf{g}, \quad (2.63)$$

Modyfikacją metody Jacobiego jest ważona metoda Jacobiego, której iterację można zapisać wg. równania (2.64). Odpowiednie dobranie wagi  $\omega$  pozwala uzyskać lepszą zbieżność ważonej metody Jacobiego.

$$\mathbf{R}_{\omega J} = \mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{A}$$

$$\mathbf{z}_{i+1} = \mathbf{R}_{\omega J}\mathbf{z}_i + \omega\mathbf{D}^{-1}\mathbf{g} \quad (2.64)$$

Zhu i Cangellaris [61] zaproponowali użycie metody Gaussa-Seidela w implementacji procedur wygładzania (pre-smoothing, post-smoothing), celem zapewnienia satysfakcjonującej zbieżności. Niestety metoda Gaussa-Seidela nie da się łatwo zrównoleglić (ze względu na operacje podstawiania „w przód” lub „wstecz”), co obniża wydajność obliczeniową operatora ściskającego [77]. W celu lepszego zrównoleglenia, zaproponowano ważoną metodę Jacobiego zamiast metody Gaussa-Seidela [81]. Ważona metoda Jacobiego jest procesem iteracyjnym atrakcyjnym z punktu widzenia zrównoleglenia obliczeń, gdyż oparta jest na operacji mnożenia macierzy rzadkiej przez wektor (Wydruk 2.4), którą można efektywnie zrównoleglić zarówno na CPU jak i na GPU. Na potrzeby wielopoziomowego operatora ściskającego ważoną metodę Jacobiego przepisano, tak aby móc wykorzystać macierz  $\mathbf{M} = \mathbf{M}_{ii}^E$ , i nie było potrzeby obliczania dodatkowych macierzy  $\mathbf{R}_{\omega J}$  (Wydruk 2.4).

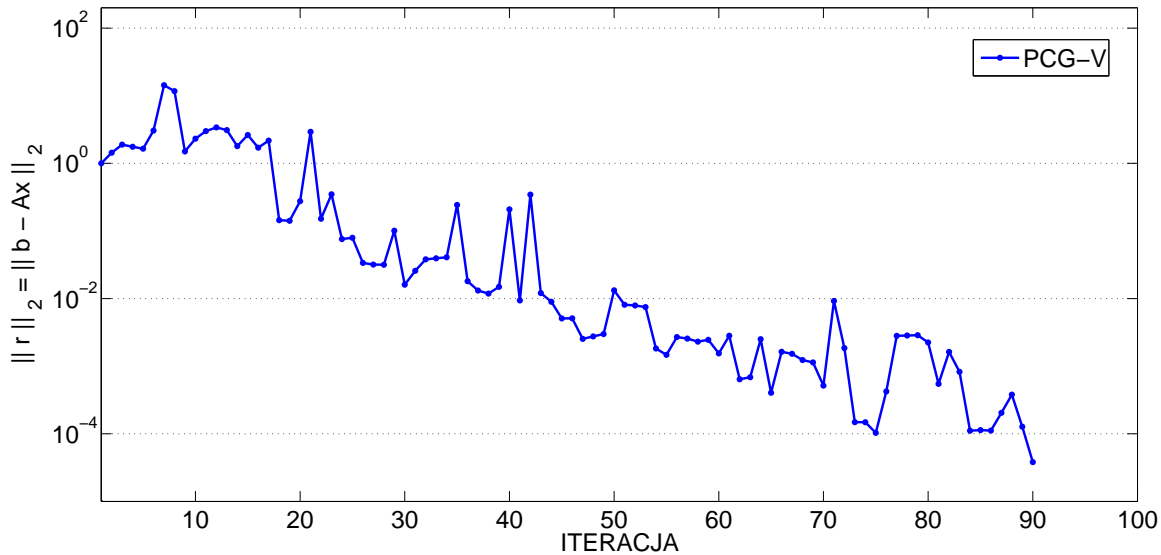
Profil zbieżności przedstawiony na rys. 2.8 potwierdza, iż metoda gradientów sprzężonych z wielopoziomowym operatorem ściskającym (cykl V, ważona metoda Jacobiego) osiąga satysfakcjonującą zbieżność.

```

1 for p=1:iter
2    $\mathbf{r} = \mathbf{g} - \mathbf{M}_{ii}^E \mathbf{z}_E^i$ 
3    $\mathbf{z}_E^i = \mathbf{z}_E^i + w * \mathbf{D}_{ii}^{-1} \mathbf{r}$ 
4 end
```

WYDRUK 2.4: Ważona metoda Jacobiego ( $\mathbf{D}_{ii}$  - diagonalna macierzy  $\mathbf{M}_{ii}^E$ ).





RYSUNEK 2.8: Zbieżność (norma euklidesowa wektora residualnego) metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V (PCG-V). W operacjach wygładzających zastosowano ważoną metodę Jacobiego. Problem testowy: filtr grzebieniowy 9-rzędu, rozmiar macierzy  $N = 806811$  (Tab. 4.1).

## 2.4 Podsumowanie metody elementów skończonych

Podsumowując powyższą dyskusję, w symulacjach numerycznych elektromagnetycznych problemów deterministycznych z zastosowaniem metody elementów skończonych wydzielić można następujące etapy:

**Etap-1 Dyskretyzacja** - podział przestrzeni obliczeniowej na elementy skończone (p. 2)

**Etap-2 Generacja układu równań liniowych**

- a) Wyznaczanie lokalnych macierzy sztywności ( $\mathbf{S}^{(e)}$ ) i bezwładności ( $\mathbf{T}^{(e)}$ ) z zastosowaniem odpowiednich dla danego problemu funkcji bazowych dla każdego elementu skończonego<sup>8</sup> (p. 2.1)
- b) Budowa macierzy globalnych - z macierzy lokalnych konstruowane są globalne macierze rzadkie sztywności ( $\mathbf{S}$ ) i bezwładności ( $\mathbf{T}$ ), (p. 2.2)
- c) Wyznaczenie prawej strony (lub macierzy prawych stron) układu równań poprzez nałożenie odpowiednich dla wybranego problemu warunków brzegowych (p. 2.1)

**Etap-3 Rozwiązanie układu równań liniowych** (p. 2.3)

**Etap-4 Przetwarzanie końcowe** - na podstawie rozwiązania układu równań obliczane są parametry macierzy rozproszenia lub inne wielkości opisujące dany układ mikrofalowy r. (2.21).

<sup>8</sup>W dalszej pracy etap ten nazywany będzie całkowaniem numerycznym, gdyż z punktu widzenia obliczeń, które należy wykonać w ostatecznej formule wyznaczającej macierze lokalnych elementów całkowanie numeryczne jest główną operacją tego etapu.



Najbardziej kosztownym obliczeniowo i czasowo etapem analizy przy użyciu metody elementów skończonych jest rozwiązanie układu równań liniowych (Etap-3). Wynika to z faktu dużej liczby danych, na których wykonywane są obliczenia. Po pierwsze, aby dokładnie analizować zagadnienie, generowane są macierze rzadkie o bardzo dużych rozmiarów - miliony niewiadomych. Po drugie, duże układy równań liniowych (Etap-3) rozwiązywane są dla wielu częstotliwości  $f$ . W przypadku zastosowania metod bezpośrednich, należy być świadomym kosztów pamięciowych i czasowych generowania macierzy (faktorów)  $\mathbf{L}$  i  $\mathbf{U}$ . W przypadku zastosowania metod iteracyjnych, należy zastosować odpowiednią metodę i operator ściskający, które pozwolą osiągnąć satysfakcjonującą zbieżność. Z drugiej strony generacja układów równań (Etap-2), ze względu na kosztowny z punktu widzenia pamięci i czasu wykonania obliczeń etap budowania macierzy lokalnych (całkowanie numeryczne), ma również znaczący udział czasowy w całkowitej symulacji z wykorzystaniem metody elementów skończonych. Efektywne wykonanie tego etapu jest kluczowe zwłaszcza w przypadku zagadnień optymalizacyjnych, w których siatka elementów skończonych ulega modyfikacji co prowadzi do konieczności przebudowy macierzy.

Podsumowując, z punktu widzenia czasu wykonania obliczeń w metodzie MES, najbardziej kosztowne numerycznie etapy to generacja macierzy opisujących problem (Etap-2a,b) oraz rozwiązanie układu równań liniowych (Etap-3). **W niniejszej pracy opracowano strategie i algorytmy, które umożliwiają masywne zrównoleglenia obliczeń w MES z wykorzystaniem akceleratorów graficznych, co pozwoliło na znaczącą redukcję czasu wykonania symulacji elektromagnetycznych.**



## Rozdział 3

# Architektura CUDA

W niniejszym rozdziale przedstawiono charakterystykę i model programowania w architekturze CUDA oraz zdefiniowano podstawowe pojęcia, które są stosowane w opisie opracowanych rozwiązań algorytmicznych i implementacji przedstawionych w kolejnych rozdziałach tej rozprawy.

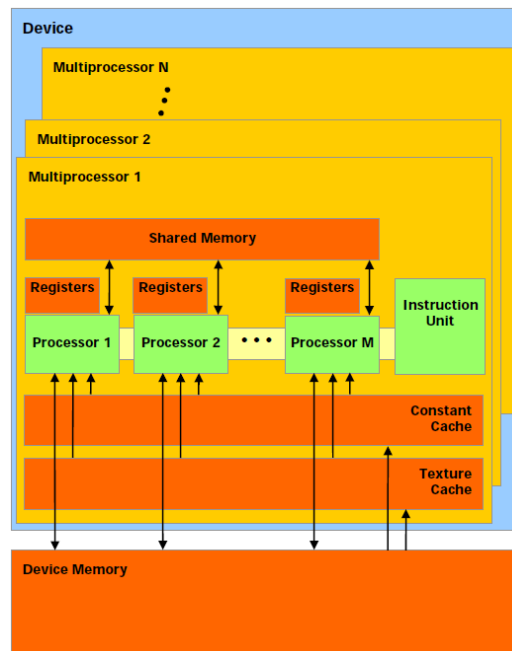
W akceleratorach kompatybilnych z architekturą CUDA występuje zupełnie inna organizacja obliczeń i pamięci niż w CPU [7, 82, 83]. Procesor centralny korzysta z pamięci zewnętrznej (DRAM), podręcznej (cache) i rejestrów. Na GPU procesory obliczeniowe grupowane są w tzw. multiprocesory. Liczba rdzeni w multiprocesorze jest sukcesywnie zwiększona wraz z rozwojem technologii GPU przez firmę NVIDIA. W architekturze GPU G80, Fermi i Kepler jeden multiprocesor zawiera odpowiednio 8, 32 i 192 rdzenie. Multiprocesor może korzystać z pamięci: globalnej (ang. global memory), wspólnej<sup>1</sup> (ang. shared memory), tekstur (ang. texture memory), stałej (ang. constant memory) i rejestrów (rys. 3.1). Największym rozmiarem pamięci charakteryzuje się pamięć globalna, jednakże przy dostępie do tej pamięci występują największe opóźnienia. W zastępstwie pamięci globalnej można wykorzystywać pamięć tekstur, ale istotnym ograniczeniem jest brak możliwości zapisu danych do tej pamięci<sup>2</sup>. Małymi opóźnieniami cechują się pamięć wspólna i rejestry (dane przechowywane w tych obszarach są „blisko” jednostek obliczeniowych ALU). W akceleratorach graficznych kompatybilnych z architekturą CUDA występuje również inny rozkład tranzystorów do poszczególnych bloków funkcjonalnych niż na CPU (rys. 3.2). Na GPU więcej tranzystorów przeznaczonych jest na jednostki arytmetyczno-logiczne niż na sterowanie i pamięci podręczne, co pozwala na uzyskanie dużej mocy obliczeniowej [82, 83].

W dalszej części tego rozdziału przedstawiono podstawowy model programowania w architekturze CUDA. Najpierw dane wejściowe algorytmu alokowane są na CPU, skąd kopiowane są na GPU. Zapisując dane w pamięci GPU do wyboru są pamięć globalna, tekstur i stała. Następnie z poziomu CPU wywoływany jest program wykonywany na GPU (tzw. kernel), w trakcie którego wątki mogą odczytywać dane z powyżej wymienionych trzech rodzajów pamięci. W celu przyspieszenia obliczeń dane, na których wykonywane są obliczenia, warto przenieść do pamięci wspólnej lub rejestrów. Zabieg ten jest szczególnie ważny w przypadku wielokrotnego odwołania się przez wątki do tych samych danych, gdyż dostęp do pamięci wspólnej i rejestrów cechuje się najmniejszymi opóźnieniami. Obliczenia w kernelu wykonywane są przez wątki (ang. threads), które pracują równolegle, tzn. wykonują równolegle te same instrukcje na różnych da-

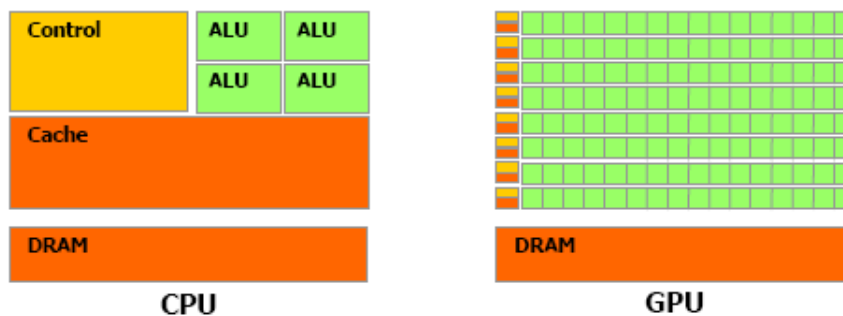
---

<sup>1</sup>Przez niektórych autorów pamięć wspólna jest też tłumaczona jako współdzielona.

<sup>2</sup>Rodzaj pamięci tylko do odczytu (ang. read-only).



RYSUNEK 3.1: Rodzaje i sposób dostępu do pamięci na akceleratorze graficznym GPU [82].



RYSUNEK 3.2: Przeznaczenie tranzystorów na CPU i GPU [82].

nych (ang. Single Instruction Multiple Threads, SIMT). Wątki grupowane są w bloki wątków (ang. blocks), a bloki uporządkowane są w siatki bloków (ang. grids). Wątki z jednego bloku nie mogą komunikować się z wątkami innych bloków i nie mają dostępu do obszarów pamięci wspólnej i rejestrów dostępnych dla innych bloków. Po zakończeniu obliczeń w kernelu należy dane wynikowe zapisać do pamięci globalnej, skąd po wykonaniu kernela mogą zostać skopiowane z GPU na CPU. W Dodatku B.2 (Wydruk B.1) przedstawiono opisany powyżej model programowania na GPU na przykładzie dodawania dwóch wektorów.

Na przestrzeni ostatnich 8 lat architektura CUDA uległa znacznej ewolucji. Wprowadzono nowe architektury GPU: Fermi, Kepler i Maxwell, które są kompatybilne z architekturą CUDA. Postęp w tym zakresie dobrze ilustrują dane w Tab. 3.1. Warto zwrócić uwagę, że względem pierwszych akceleratorów kompatybilnych z architekturą CUDA ponad dziesięciokrotnie zwiększono liczbę tranzystorów. W kolejnych architekturach zwiększono liczbę rdzeni i zmieniono ich grupowanie w multiprocesorach. Potrojono obszar pamięci wspólnej oraz podwojono liczbę rejestrów w multiprocesorach.

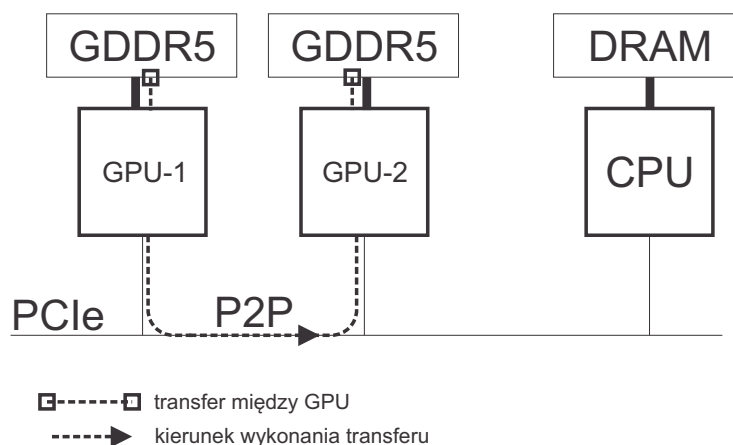
TABELA 3.1: Porównanie właściwości architektur akceleratorów graficznych.

Właściwość	G80	Fermi	Kepler
L. tranzystorów	681 M	3 G	7.1 G
Zapotrzebowanie na moc [W]	400	400	235
TFLOPS (pojedyncza precyzja)	0,5	1,3	4,3
TFLOPS (podwójna precyzja)	-	0,6	1,4
Maksymalna liczba multiprocesorów	16	16	90
Maksymalna liczba rdzeni w jednym multiprocesorze	8	32	192
Liczba rdzeni	128	512	2880
Maksymalna liczba wątków w bloku	512	1024	1024
Maksymalna liczba wątków na multiprocesor	768	1536	2048
Maksymalna liczba warpów na multiprocesor	24	48	64
Maksymalna liczba rejestrów na wątek	124	63	255
Pamięć wspólna na multiprocesor (kB)	16	16\48	15\32\48
Współbieżne kernele	nie	tak	tak
ECC	nie	tak	tak
Hyper-Q	nie	nie	tak
Dynamic Parallelism	nie	nie	tak

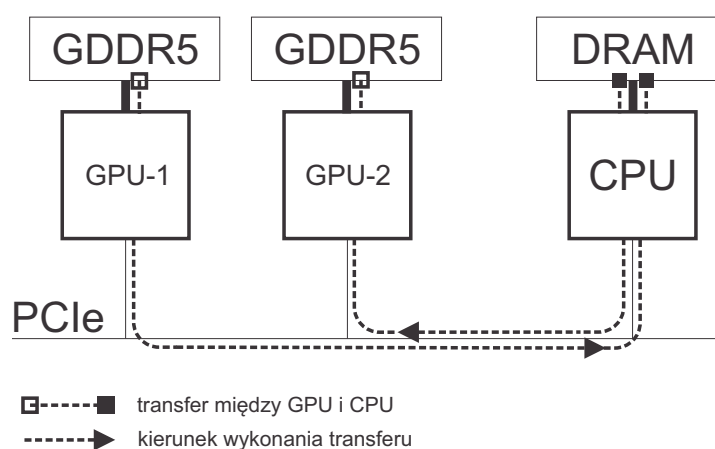
Dodatkowo zadbano o zmniejszenie zapotrzebowania akceleratorów graficznych na energię elektryczną. Modyfikacje doprowadziły do ok. dziewięciokrotnego zwiększenia mocy obliczeniowej, która w obecnych akceleratorach wynosi 4,3 i 1,4 TFLOPS odpowiednio dla pojedynczej i podwójnej precyzji obliczeń.

Architektura CUDA (w obecnej wersji 7.0) oferuje wiele możliwości istotnych z punktu widzenia uzyskania dużej wydajności obliczeniowej. Do najważniejszych zaliczyć można:

- współbieżne wykonanie kilku kerneli (ang. concurrent kernel execution) - właściwość ta pozwala na to by wiele kerneli mogło być wywołanych jednocześnie i korzystać z zasobów pojedynczego GPU - w przypadku, gdy każdy z uruchamianych kerneli nie wykorzystuje w pełni zasobów GPU (pamięciowych i obliczeniowych) to niewykorzystane zasoby mogą być użyte przez inne uruchomione kernele (rys. 3.4(a)-(b)),
- uruchomienie w pojedynczym kernelu (tzw. parent kernel) innych kerneli (tzw. child kernels), co umożliwia synchronizację pracy kerneli, wykonanie operacji na pamięci GPU oraz tworzenie i stosowanie strumieni bez wykorzystania CPU (dynamic parallelism [84]),
- bezpośrednią wymianę danych między akceleratorami graficznymi bez potrzeby komunikacji za pośrednictwem pamięci CPU (GPUDirect, rys. 3.3a), do tej pory przesyłanie danych między akceleratorami odbywało się za pośrednictwem CPU (rys. 3.3b),
- hierarchizację pamięci cache - wprowadzono podział na konfigurowalną pamięć L1 (16kB lub 48kB na multiprocesor) oraz zunifikowaną pamięć L2,
- wielokrotne zwiększenie wydajność operacji atomowych (ang. atomic operations) - operacje atomowe umożliwiają wątkom poprawne wykonanie operacji odczytu

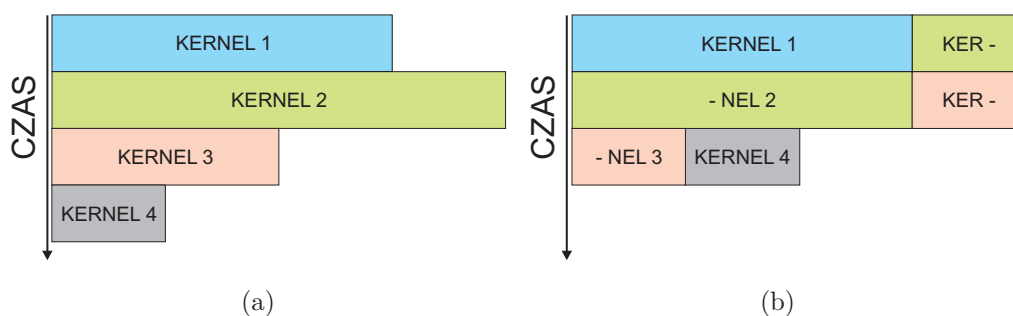


(a)



(b)

RYSUNEK 3.3: Przesyłanie danych między akceleratorami graficznymi (z GPU-1 do GPU-2): (a) komunikacja bezpośrednia (P2P, ang. peer to peer), (b) komunikacja za pośrednictwem CPU.



(a)

(b)

RYSUNEK 3.4: Schemat kolejgowanych (lewy) i współbieżnych (prawy) wykonania kerneli.

i zapisu do pamięci podczas wykonywania na danych, do których dostępu może żądać wiele wątków jednocześnie, operacji: dodawania (atomicAdd), określenie minimum (atomicMin) i maksimum (atomicMax) pomiędzy dwiema wartościami, porównanie i podmianę wartości (atomicCAS).

TABELA 3.2: Wybrane funkcje bibliotek CUBLAS i CUSPARSE realizujące podstawowe operacje macierzowe i wektorowe. Oznaczenia: {S} - pojedyncza precyzja, macierze rzeczywista, {D} - podwójna precyzja, macierze rzeczywista, {C} - pojedyncza precyzja, macierze zespolona, {Z} - podwójna precyzja, macierze zespolona.

Nazwa funkcji	Funkcja
cublas{S,D,C,Z}axcp	$y = \alpha x + y$
cublas{S,D,C,Z}copy	kopiowanie $x$ do $y$
cublas{S,D,C,Z}dot	iloczyn skalarny wektorów, mnożenie poelementowe w wektorów $x$ i $y$
cublas{S,D,C,Z}nrm2	norma euklidesowa wektora $x$
cublas{S,D,C,Z}scal	$x = \alpha x$
cusparse{S,D,C,Z}csrmmv, cusparse{S,D,C,Z}hybmv	$y = \alpha Ax + \beta y$

TABELA 3.3: Parametry GPU użytych w testach numerycznych w rozdziałach 4-8.

GPU	Tesla K20c	Tesla K40c
architektura	Kepler	Kepler
liczba rdzeni	2496	2880
liczba multiprocesorów	13	15
pamięć globalna GDDR5 [GB]	5	12
pamięć wspólna (/multiprocesor) [KB]	16\32\48	16\32\48
częstotliwość zegara [GHz]	0,8	0,9
PCI-E	2.0 x16	3.0
liczba testowanych akceleratorów	2	1

Architektura CUDA pozwala korzystać z dołączonych bibliotek m.in. CUBLAS [85] i CUSPARSE [73], które umożliwiają realizację podstawowych macierzowych i wektorowych operacji algebry liniowej. W Tab. 3.2 zaprezentowano funkcje z biblioteki CUBLAS użyte w implementacji operacji na wektorach w iteracyjnej metodzie rozwiązywania układów równań (p. 6.3) oraz funkcje z biblioteki CUSPARSE, z którymi porównano opracowaną w niniejszej rozprawie implementację mnożenia macierzy rzadkiej przez wektor (p. 6.2.1).

Na zakończenie tego rozdziału trzeba zaznaczyć, że architektura CUDA umożliwia implementację algorytmów z maszynie zrównoleglonymi obliczeniami, jednakże nie każdy rodzaj algorytmu posiada cechy, które pozwalają na jego efektywną implementację na GPU. W niektórych rodzajach algorytmów kod wykonywany na GPU jest wolniejszy niż jego odpowiednik na CPU. Mając to na uwadze, programując karty graficzne należy podjąć trud maksymalnego zrównoleglenia algorytmów, optymalizować użycie pamięci, zwracać uwagę by używać optymalnych instrukcji (Dodatek B). **Często wieloetapowa optymalizacja wybranego algorytmu prowadzi do opracowania zupełnie nowego algorytmu z maszynie zrównoleglonymi obliczeniami, który realizuje tę samą funkcjonalność co algorytm początkowy.**

W Tab. 3.3 zaprezentowano główne parametry GPU, które zostały użyte w testach numerycznych mających na celu potwierdzenie tez badawczych niniejszej rozprawy.





## Rozdział 4

# Budowa macierzy sztywności i bezwładności

Rozdział ten jest poświęcony opisowi strategii i algorytmów jakie zostały zaproponowane celem masywnego zrównoleglenia obliczeń związanych z wyznaczeniem lokalnych macierzy elementów i budową globalnych macierzy sztywności i bezwładności w metodzie elementów skończonych (MES) przy wykorzystaniu akceleratorów graficznych w architekturze CUDA. W rozdziale opisano implementacje etapów generacji macierzy przedstawionych w rozdziale 2. Program komputerowy opracowany na podstawie zaproponowanych w pracy algorytmów składa się z wielu tysięcy linii kodu i z tego powodu algorytmy przedstawiono głównie w postaci schematów blokowych. W nielicznych przypadkach zdecydowano się na prezentację fragmentów kodu, w celu pokazania jak zrównoleglono obliczenia na akceleratorze graficznym i w takich przypadkach kody zostały umieszczone w Dodatku C.

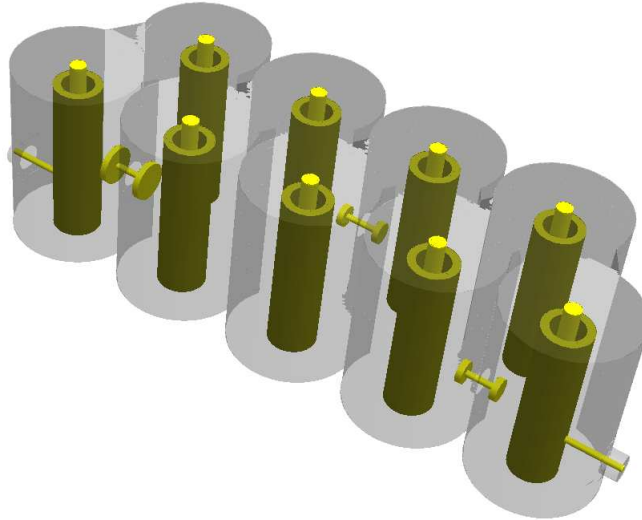
W pierwszej kolejności opisany został podstawowy wariant generacji jaki opublikowano w recenzowanych czasopismach [86–88] (p. 4.1). W punkcie tym szczegółowo omówiono strategię optymalizacyjną jakie zastosowano w implementacji najważniejszych faz generacji układów równań liniowych (całkowanie numeryczne, składanie macierzy globalnych). W kolejnym rozdziale przedstawiono iteracyjny wariant dedykowany obliczeniom na pojedynczym i wielu akceleratorach graficznych zaproponowany w [89], który pozwala na generację dużych macierzy.

Testy numeryczne etapów generacji i rozwiązywania układów równań (omówione w kolejnych rozdziałach) przeprowadzono dla filtru grzebieniowego 9 rzędu pracującego w paśmie GSM (920-980 MHz) (rys. 4.1). Podział struktury trójwymiarowej na czworościany przeprowadzono przy użyciu generatora siatki NETGEN [58]. Czworościany w strukturze wypełnione są izotropowymi ośrodkami bezstratnymi, w których względna przenikalność elektryczna  $\epsilon = 1$  i względna przenikalność magnetyczna  $\mu = 1$ . Parametry generowanych problemów (liczba czworościanów na jakie podzielono domenę obliczeniową, rozmiar i liczba elementów niezerowych generowanych macierzy) zaprezentowano w Tab. 4.1. Jako warunki brzegowe zastosowano PEC (ang. Perfect Electric Conductor) [56], czyli na powierzchni  $S$  obszaru  $\Omega$  spełnione są zależności:

$$\vec{n} \times \vec{E} = 0 \quad (4.1)$$

$$\vec{n} \times \frac{1}{\epsilon_r} \nabla \times \vec{H} = 0 \quad (4.2)$$

Filtr został pobudzony z linii współosiowej falą o rodzaju TEM. Jest to fala poprzeczna elektromagnetyczna (ang. Transverse Electric-Magnetic), w której pole elek-



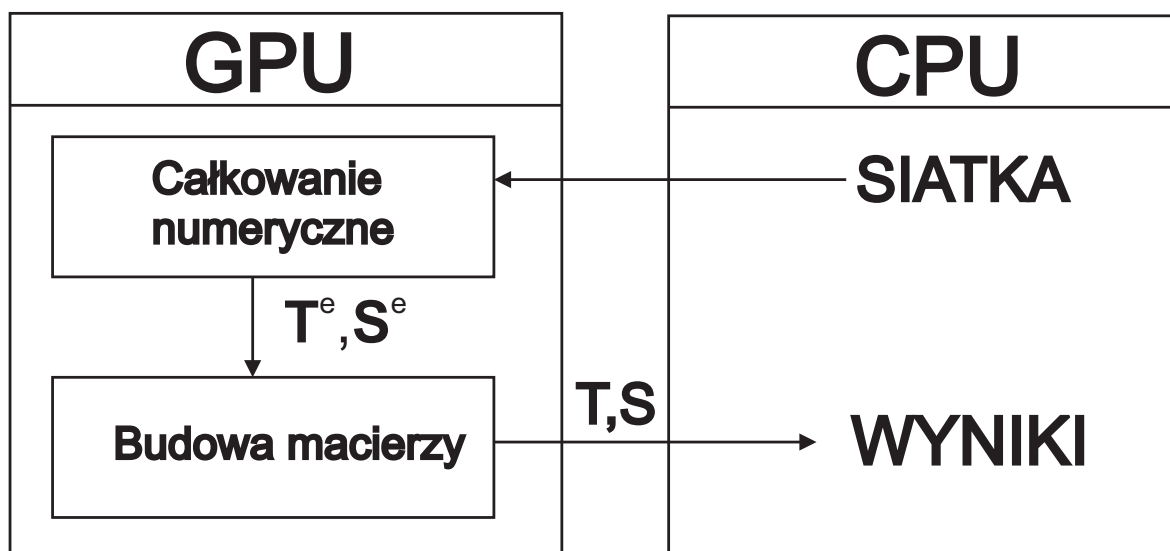
RYSUNEK 4.1: Filtr grzebieniowy 9 rzędu zaprojektowany na pasmo GSM.

tryczne i pole magnetyczne leży w płaszczyźnie prostopadłej do kierunku propagacji fali ( $E_z = 0$ ,  $H_z = 0$ ), a wektory natężenia pola elektrycznego i natężenia pola magnetycznego mają co najwyżej dwie składowe.

Opracowane algorytmy i ich implementacje ze zrównoleglonymi obliczeniami na akceleratorach graficznych przedstawione w niniejszej rozprawie zostały porównane z wynikami otrzymanymi na CPU dla częściowo zoptymalizowanego kodu wykorzystywanego w symulatorze zagadnień elektromagnetycznych *InventSim* [90, 91], który jest opracowywany w zespole prof. dra hab. inż. Michała Mrozowskiego i dra inż. Adama Lamęckiego, i umożliwia analizę właściwości urządzeń stosowanych w komunikacji bezprzewodowej. Obliczenia na GPU i CPU przeprowadzono na zmiennych w podwójnej precyzji.

TABELA 4.1: Problemy testowe wykorzystane do testów numerycznych uzyskane podczas generacji macierzy współczynników w zależności od liczby elementów skończonych na jakie został podzielony filtr grzebieniowy z rys. 4.1 w etapie dyskretyzacji.  $N_i$  to rozmiary podmacierzy  $\mathbf{M}_{ii}^E$  zgodnie z indeksacją z r. (2.34).

Liczba czworościanów	Rozmiar macierzy $\mathbf{A}$	Liczba elementów niezerowych $\mathbf{A}$	Rozmiar podmacierzy		
			$\mathbf{M}_{11}^E$	$\mathbf{M}_{22}^E$	$\mathbf{M}_{33}^E$
M	N	NNZ	$N_1$	$N_2$	$N_3$
13 613	213 984	14 010 426	10 449	57 715	145 820
28 806	455 916	30 447 378	22 606	123 166	310 144
49 854	806 811	56 170 107	41 715	219 083	546 013
116 070	1 998 492	153 850 230	114 252	550 094	1 334 146
195 349	3 370 908	260 934 264	193 405	928 287	2 249 216
287 044	5 079 849	407 716 515	302 357	1 406 239	3 371 253



RYSUNEK 4.2: Podstawowy wariant generacji macierzy sztywności ( $S$ ) i bezwładności ( $T$ ) w metodzie elementów skończonych [86].

## 4.1 Podstawowy wariant generacji macierzy na pojedynczym akceleratorze

Według formuł przedstawionych w punkcie 2.4, podstawowa implementacja generacji macierzy bezwładności i sztywności składa się z następujących po sobie etapów: dyskretyzacja, całkowanie numeryczne i konstrukcja macierzy sztywności i bezwładności. Z punktu widzenia wykorzystania akceleratora graficznego powyższe etapy mogą być zaimplementowane tak jak to zostało zaprezentowane na rys. 4.2 [86]. Strategia zakłada wykonanie jak największej liczby obliczeń na GPU. W pierwszej kolejności informacje na temat siatki elementów skończonych przesyłane są z pamięci hosta (CPU) do pamięci urządzenia (GPU). Etap ten (wraz z dodatkową alokacją danych na GPU) można nazwać przetwarzaniem wstępnym (ang. preprocessing) generacji macierzy. Następnie na akceleratorze graficznym (GPU) wykonywane jest całkowanie numeryczne (ang. numerical integration) i składanie rzadkich macierzy globalnych macierzy sztywności i bezwładności (szczegóły tych etapów zostały opisane w punktach 4.1.1-4.1.2). Po zakończeniu obliczeń na GPU, macierze rzadkie są przesłane z pamięci akceleratora do pamięci CPU.

### 4.1.1 Całkowanie numeryczne

W punkcie tym szczegółowo opisano implementację całkowania numerycznego na akceleratorze graficznym jaką zaproponowano w niniejszej rozprawie. Etap ten skrótowo określano jako **NI** co wynika z angielskich akronimów (ang. numerical integration).

W literaturze znaleźć można propozycje innych autorów, którzy zajmowali się z zagadnieniem całkowania numerycznego na GPU [28, 44, 45]. Sformułowania, dla których implementowano całkowanie numeryczne były mniej skomplikowane, gdyż użyto mniejszej liczby funkcji bazowych lub kwadratury niższego rzędu. W [28] zastosowano liniowe elementy skończone z liniowymi funkcjami bazowymi, a macierze lokalne zostały wyznaczone z zastosowaniem całkowania numerycznego z kwadraturą maksymal-

nie ośmiopunktową. W artykule [44] zastosowano czworościenne elementy skończone z zakrzywionymi krawędziami i do całkowania numerycznego wykorzystano środowisko OpenCL (ang. Open Computing Language). W implementacji na GPU jeden wątek (ang. work-item<sup>1</sup>) był odpowiedzialny za obliczenie co najmniej jednego elementu lokalnej macierzy, a jedna porcja wątków (ang. work-group) była odpowiedzialna za przetwarzanie pojedynczego elementu skończonego. W artykule do reprezentacji pojedynczego elementu skończonego użyto 32 zmiennych typu float (23 zmienne opisujące geometrię, orientację krawędzi i informację o ośrodku oraz 9 zer dodanych aby zapewnić dostęp łączny w trakcie dostępu do pamięci globalnej). Także w artykule [45] zastosowano strategię, w której jeden blok wątków został przypisany do obliczeń potrzebnych do wykonania dla pojedynczego elementu skończonego (graniastosłupy), zastosowano uzupełnienie reprezentacji danych dodatkowymi zerami w celu uzyskania efektywnego łącznego dostępu do pamięci globalnej. Przed realizacją obliczeń całkowania numerycznego dane kopiowane są z pamięci globalnej do pamięci wspólnej. Kwadratura została zrównoleglona i pojedynczy wątek wykonuje obliczenia dla pojedynczego punktu kwadratury. Po wyznaczeniu wartości macierzy sztywności przechowywane w pamięci wspólnej są kopiowane do pamięci globalnej. W artykule [46] autorzy prowadzą ciekawą dyskusję w celu znalezienia kompromisów pomiędzy wykorzystaniem zasobów obliczeniowych (liczba wątków i bloków wątków wykonujących obliczenia), a zasobami pamięciowymi (np. wzrost liczby wątków zwiększa zapotrzebowanie na rejestry, zwiększenie liczby bloków zwiększa zapotrzebowanie na pamięć wspólną) w trakcie wykonania operacji całkowania numerycznego. W implementacji operacji całkowania numerycznego autorzy stosują dwupoziomowe zrównoleglenie obliczeń (wyższy poziom - zrównoleglona pętla po elementach skończonych, niższy poziom - zrównoleglone obliczenia dla elementu skończonego). Na niższym poziomie autorzy wskazują dwa możliwe podejścia, pierwsze to zrównoleglenie kwadratury, drugie to zrównoleglenie obliczeń w trakcie wyznaczania macierzy sztywności. Autorzy decydują się na drugi wariant wskazując, że ograniczeniem zrównoleglenia kwadratury są wyścigi przy aktualizacji danych z różnych punktów kwadratury. Opracowana strategia zakłada, że jeden element skończony jest przypisany do pojedynczego bloku wątków (z tym, że jeden blok może pracować na kilku elementach) i jeden wątek wykonuje obliczenia, które prowadzą do wyznaczenia kilku elementów macierzy sztywności. Ponadto autorzy wskazują, żeby macierze Jacobianów i macierze funkcji bazowych liczyć na CPU i przysyłać na GPU. Wyniki uzyskane w artykule [46] potwierdzają duże skrócenie czasu całkowania na GPU względem CPU, jednakże czas transferu danych z CPU na GPU jest duży i dla elementów niższego rzędu porównywalny z czasem całkowania numerycznego na GPU.

Dla sformułowań MES przyjętych w niniejszej rozprawie przetwarzanie pojedynczego elementu wymaga przeprowadzenia zdecydowanie większej liczby obliczeń niż w przypadku publikacji [28, 44, 45]. Tak jak to zostało opisane w punkcie 2.1, całkowanie numeryczne wymaga użycia kwadratur Gaussa wysokiego rzędu dla hierarchicznych funkcji bazowych użytych do opisu elementu skończonego.

Lokalne macierze sztywności i bezwładności oblicza się zgodnie z wzorami:

$$\mathbf{S}^{(e)} \approx \frac{1}{6} \sum_{i=1}^Q w_i \mathbf{N}_{Ri} (\mathbf{J}_i^T \boldsymbol{\mu}^{-1} \mathbf{J}_i) \mathbf{N}_{Ri}^T \frac{1}{\det(\mathbf{J}_i)} \quad (4.3)$$

<sup>1</sup>W OpenCL work-item i work-group odpowiadają wątkowi i blokowi wątków w architekturze CUDA.

$$\mathbf{T}^{(e)} \approx \frac{1}{6} \sum_{i=1}^Q w_i \mathbf{N}_i (\mathbf{J}_i^{-T} \epsilon \mathbf{J}_i^{-1}) \mathbf{N}_i^T \det(\mathbf{J}_i) \quad (4.4)$$

gdzie:  $Q$  oznacza liczbę punktów kwadratury,  $(\cdot)^{-T}$  oznacza transpozycję odwrotności macierzy,  $\mathbf{J}_i$  oznacza macierz Jacobiego,  $\mathbf{N}_i$  oznacza macierz, która przechowuje wartości hierarchicznych funkcji bazowych,  $\mathbf{N}_{Ri}$  oznacza macierz, która przechowuje wartości rotacji hierarchicznych funkcji bazowych,  $\mathbf{S}^{(e)}$  oznacza lokalną macierz sztywności elementu skończonego,  $\mathbf{T}^{(e)}$  oznacza lokalną macierz bezwładności elementu skończonego.

Warto zaobserwować, że niemal wszystkie czynniki występujące pod sumą są macierzami. Z tego względu najprostszym sposobem wyznaczenie składników sumy byłoby zastosowanie operacji macierzowych, co dla GPU łatwo uczynić posługując się biblioteką CUBLAS, zaprojektowaną i zoptymalizowaną do podstawowych obliczeń numerycznych z macierzami gęstymi [85]. To samonasuwające się rozwiązanie nie jest jednak dostatecznie wydajnie. Operacje macierzowe w środowisku GPU są wolne jeśli macierze są rozmiarów niebędących wielokrotnością liczby 32. Niestety w interesującym nas przypadku taka sytuacja ma miejsce. Macierze Jacobiego, macierze przechowujące wartości funkcji bazowych i ich rotacji z równań (4.3)-(4.4) w niniejszej pracy mają odpowiednio rozmiary  $3 \times 3$ ,  $50 \times 3$  i  $50 \times 3$ . Z tego względu do wykonania operacji na macierzach gęstych zdecydowano się opracować własne procedury (kernele) [86].

Analizując zależności (4.3)-(4.4) można zauważyć, że poszczególne składniki sumy w kwadraturach nie są od siebie zależne, a więc obliczenie każdego z nich może odbywać się równocześnie z pozostałymi. Z tego względu dedykowane kernele zostały zaprojektowane tak, aby można było wykonać równocześnie wiele mnożeń macierzy gęstych dla różnych punktów kwadratury. Co więcej, zaproponowano zastosowanie kwadratur Gaussa różnych rzędów, celem redukcji obliczeń potrzebnych do wykonania i redukcji zapotrzebowania na pamięć w trakcie wykonania operacji całkowania numerycznego. Poniżej przedstawiono szczegóły zaproponowanej w niniejszej pracy implementacji całkowania numerycznego na GPU. Dla uproszczenia opisu najpierw przedstawiono przypadek, gdy materiał opisany jest przez skalarną przenikalność elektryczną  $\epsilon$  i magnetyczną  $\mu$  jest bezstratny, czyli że oba skalary są liczbami rzeczywistymi. Jest to sytuacja, która dotyczy rozważanego filtra mikrofalowego przedstawianego na rys. 4.1, w którym  $\epsilon = 1$  i  $\mu = 1$ . Tym niemniej metoda całkowania jest ogólna i z tego względu w dalszej kolejności rozszerzono dyskusję o warianty, w których ośrodek był reprezentowany przez skalary zespolone, tensory rzeczywiste i zespolone.

#### 4.1.1.1 Całkowanie numeryczne z wykorzystaniem mieszanych kwadratur Gaussa

Koncepcję kwadratury mieszanej, czyli takiej, której rząd zależy od właściwości czworościanu, w kontekście obliczeń MES na GPU zaproponowano w pracy [87]. Jak wiadomo, kwadratury Gaussa rzędu  $n$  prowadzą do dokładnego wyniku jeśli funkcja podcałkowa jest wielomianem rzędu  $2n - 1$ . Funkcje bazowe są wielomianami, a więc o tym czy trzeba zastosować wyższy rząd kwadratury decyduje macierz Jacobiego, która zależy od krzywizny czworościanu. W związku z tym opracowano metodę selekcji czworościanów, w której wszystkie czworościany podzielono na dwa podzbiory: „słabo” i „silnie” krzywoliniowe. W rezultacie, całkowanie numeryczne (które ma na celu obliczenie macierzy lokalnych elementów skończonych) odbywa się z użyciem kwadratury Gaussa o odpowiednio niższym i wyższym rzędzie (Tab. 4.2). Dla elementów krzywoliniowych, przy zastosowaniu koncepcji kwadratury mieszanej formuły do obliczenia lokalnych macierzy

elementów można zapisać w notacji macierzowej:

$$\mathbf{S}_W^{(e)} \approx \frac{1}{6} \sum_{i=1}^{Q_W} w_i \mathbf{N}_{WRi} (\mathbf{J}_{Wi}^T \mu^{-1} \mathbf{J}_{Wi}) \mathbf{N}_{WRi}^T \frac{1}{\det(\mathbf{J}_{Wi})} \quad (4.5)$$

$$\mathbf{S}_S^{(e)} \approx \frac{1}{6} \sum_{i=1}^{Q_S} w_i \mathbf{N}_{SRi} (\mathbf{J}_{Si}^T \mu^{-1} \mathbf{J}_{Si}) \mathbf{N}_{SRi}^T \frac{1}{\det(\mathbf{J}_{Si})} \quad (4.6)$$

$$\mathbf{T}_W^{(e)} \approx \frac{1}{6} \sum_{i=1}^{Q_W} w_i \mathbf{N}_{Wi} (\mathbf{J}_{Wi}^{-T} \epsilon \mathbf{J}_{Wi}^{-1}) \mathbf{N}_{Wi}^T \det(\mathbf{J}_{Wi}) \quad (4.7)$$

$$\mathbf{T}_S^{(e)} \approx \frac{1}{6} \sum_{i=1}^{Q_S} w_i \mathbf{N}_{Si} (\mathbf{J}_{Si}^{-T} \epsilon \mathbf{J}_{Si}^{-1}) \mathbf{N}_{Si}^T \det(\mathbf{J}_{Si}) \quad (4.8)$$

gdzie:  $W, S$  określają liczbę punktów kwadratury dla „słabo” i „silnie” krzywoliniowych czworościanów oraz indeksują poniższe macierze Jacobiego, funkcji bazowych, lokalnych macierzy elementów,

$(\cdot)^{-T}$  to operacja transpozycji macierzy odwrotnej,

$\mathbf{J}_{Wi}, \mathbf{J}_{Si}$  - macierze Jacobiego o rozmiarze  $3 \times 3$ ,

$\mathbf{N}_{WDi}, \mathbf{N}_{Si}$  - macierze o rozmiarze  $50 \times 3$ , które przechowują wartości funkcji bazowych  $\mathbf{N}_{Wi}$  i  $\mathbf{N}_{Si}$ ,

$\mathbf{N}_{WRi}, \mathbf{N}_{SRi}$  - macierze o rozmiarze  $50 \times 3$ , które przechowują wartości rotacji funkcji bazowych,

$\mathbf{S}_W^{(e)}, \mathbf{S}_S^{(e)}$  - macierze sztywności o rozmiarze  $50 \times 50$ ,

$\mathbf{T}_W^{(e)}, \mathbf{T}_S^{(e)}$  - macierze bezwładności o rozmiarze  $50 \times 50$ .

Koncepcja kwadratur mieszanych została zaproponowana, aby zredukować liczbę operacji do wykonania i zapotrzebowanie na pamięć w trakcie wykonywania operacji całkowania numerycznego celem obliczania macierzy lokalnych sztywności  $\mathbf{S}^{(e)}$  i bezwładności  $\mathbf{T}^{(e)}$  (Tab. 4.3). Obydwa te parametry są istotne z punktu widzenia wykorzystania zasobów akceleratorów graficznych. Z tego powodu w artykule [87] podjęto skuteczną próbę określenia minimalnego rzędu kwadratur Gaussa, która pozwoliła osiągnąć dużą dokładność całkowania numerycznego oraz całej symulacji MES. Znalezienie optymalnego rzędu kwadratury wymaga określenia dokładności całkowania numerycznego dla kwadratur Gaussa z Tab. 4.2. W tym celu obliczono błędy względne (4.9) pomiędzy wartościami lokalnych macierzy sztywności  $\mathbf{T}^{(e)}$  otrzymanych dla kwadratur o rzędzie  $R = \{5, 6, 9, 10, 11\}$  i lokalną macierzą otrzymaną z najwyższym rzędem kwadratury  $R = 14$  r. (4.9).

$$\text{błąd względny} = \frac{\|\mathbf{T}_{R=\{5,6,9,10,11\}}^{(e)} - \mathbf{T}_{R=14}^{(e)}\|_2}{\|\mathbf{T}_{R=14}^{(e)}\|_2} \quad (4.9)$$

Minimalny rząd kwadratury dla prostoliniowych elementów może być określony na podstawie rzędu funkcji bazowych (4.10). Dla funkcji bazowych trzeciego rzędu, kwadratura powinna być rzędu co najmniej  $R = 6$  [72].

$$R = 2 \cdot p - 1 \quad (4.10)$$

TABELA 4.2: Rząd ( $R$ ) i liczba punktów ( $Q$ ) kwadratury dla reguł zaproponowanych dla czworościanów w artykule [72].

$(R, Q)$	(5,14)	(6,24)	(9,61)	(10,81)	(11,109)	(14,236)
----------	--------	--------	--------	---------	----------	----------



TABELA 4.3: Ilość pamięci i liczba operacji zmiennoprzecinkowych (ang. floating point operation, flop) potrzebnych do wyznaczenia pojedynczej macierzy sztywności w zależności od rzędu punktu ( $R \backslash Q$ ) kwadratury.

$R \backslash Q$	Pamięć [kB]	Mflop
6 (24)	79,5	0,4
9 (61)	171,3	1,0
10 (81)	220,9	1,3
11 (109)	290,3	1,8
14 (236)	605,3	3,9

TABELA 4.4: Udział procentowy elementów określonych jako „słabo” i „silnie” krzywoliniowe na podstawie weryfikacji liniowości każdego elementu skończonego dla przykładowej siatki z 49854 czworoscianami.

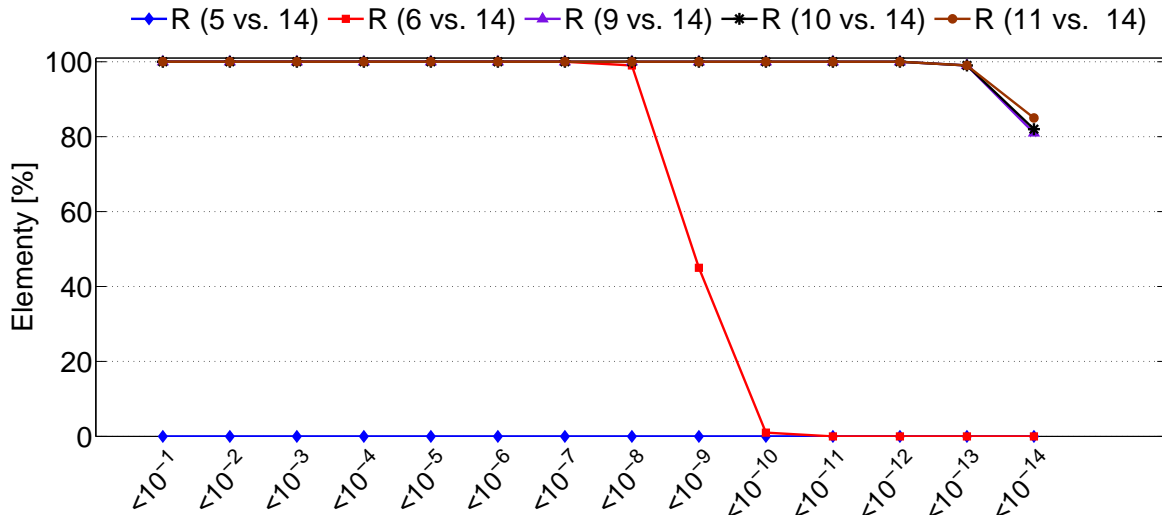
Tolerancja	$10^{-1}$	$10^{-2}$	$10^{-3}$	$10^{-4}$	$10^{-5}$	$10^{-6}$	$10^{-7}$
„słabo” krzywoliniowe [%]	76	43	41	41	<b>41</b>	40	1
„silnie” krzywoliniowe [%]	24	57	59	59	<b>59</b>	60	99

gdzie:  $R$  określa rząd kwadratury,  $p$  określa rząd funkcji bazowych.

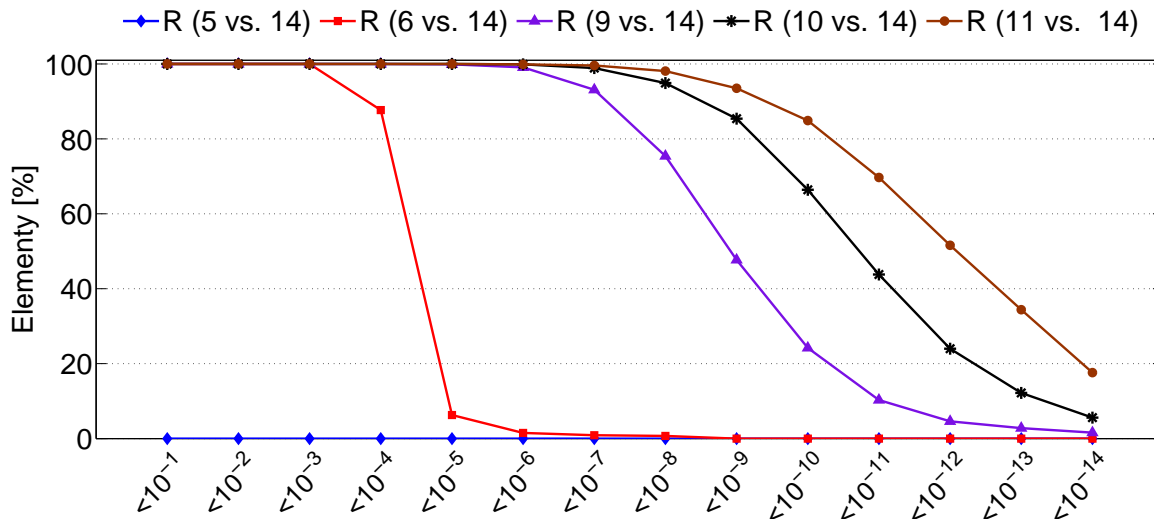
Jeśli czworoscian nie jest krzywoliniowy, to stosowanie kwadratury 6 rzędu jest wystarczające. Ponieważ, na etapie generowania siatki dopuszcza się zakrzywienie krawędzi i ścian czworoscianów, należy w pierwszej kolejności zbadać jak silne jest zakrzywienie. W celu określenia kandydatów dla kwadratury niskiego rzędu, przed wykonaniem całkowania przeprowadzana jest weryfikacja liniowości każdego elementu skończonego. W ogólności czworosciany wyższego rzędu opisane są jako zbiór 10 punktów w przestrzeni: 4 punkty na końcu krawędzi oraz 6 punktów w środkach krawędzi (rys. 2.3). Współrzędne punktów środkowych krawędzi czworoscianów porównywane są z punktami środkowymi linii łączących końce krawędzi. Jeżeli wszystkie punkty leżą, z założoną dokładnością, dostatecznie blisko punktów referencyjnych, to czworoscian klasyfikowany jest jako „słabo” krzywoliniowy, w przeciwnym razie, jako „silnie” krzywoliniowy. Na podstawie testów zdecydowano się przyjąć tolerancję na poziomie  $10^{-5}$ , gdyż zapewnia ona zarówno niski poziom błędu względnego jak i dużą liczbę czworoscianów „słabo” krzywoliniowych (Tab. 4.4).

Na rys. 4.3-4.4 przedstawiono udział procentowy czworoscianów określonych jako „słabo” i „silnie” krzywoliniowych, dla których błąd względny z r. (4.9) jest mniejszy niż  $10^{-N}$  dla kwadratur o rzędzie  $R = \{5, 6, 9, 10, 11\}$ . Tak jak oczekiwano dla wszystkich czworoscianów „słabo” i „silnie” krzywoliniowych kwadratura  $R = 5$  rzędu ( $Q = 14$  punktów) nie jest wystarczająca. Ponadto dla czworoscianów „słabo” krzywoliniowych błąd względny mniejszy niż  $10^{-8}$  został otrzymany dla wszystkich kwadratur o rzędzie większym niż  $R = 5$ . Z racji tego, iż kwadratura o rzędzie  $R = 6$  jest wystarczająca do osiągnięcia wysokiej dokładności całkowania numerycznego, ma mniejsze zapotrzebowanie na pamięć i wymaga przeprowadzenia mniejszej liczby obliczeń niż kwadratura wyższego rzędu zdecydowano się uznać ją za optymalną dla elementów „słabo” krzywoliniowych. Jak pokazuje rysunek 4.4, kwadratura o rzędzie  $R = 6$  nie gwarantuje jednak poprawności wykonania operacji całkowania dla elementów „silnie” krzywoliniowych (bardzo mała jest liczba elementów, dla których błąd względny jest mniejszy niż  $10^{-8}$ ). Analizując rysunek 4.4 można zauważyć, iż dla kwadratur rzędu





RYSUNEK 4.3: Udział procentowy „słabo” krzywoliniowych elementów dla błędów względnych (r. (4.9)) o wartości mniejszej niż  $10^{-N}$ .



RYSUNEK 4.4: Udział procentowy „silnie” krzywoliniowych elementów dla błędów względnych (r. (4.9)) o wartości mniejszej niż  $10^{-N}$ .

$R = \{9, 10, 11\}$  otrzymano błędy względne odpowiednio na poziomie  $10^{-7}$ ,  $10^{-8}$ ,  $10^{-9}$  dla ponad 90% elementów. Jako kompromis między kosztem a dokładnością obliczeń wybrano kwadraturę rzędu  $R = 10$  ( $Q = 81$  punktów). Jest to najmniejsza kwadratura pozwalająca osiągnąć błąd względny mniejszy niż  $10^{-8}$  dla ponad 90% czworoscianów oraz błąd względny mniejszy niż  $10^{-5}$  dla wszystkich krzywoliniowych elementów.

Implementacja całkowania numerycznego z zastosowaniem kwadratur mieszanych wymaga separacji czworoscianów „słabo” i „silnie” krzywoliniowych. Z tego powodu, przed wykonaniem operacji całkowania numerycznego podzbiór czworoscianów jest skanowany i każdy czworoscian klasyfikowany jest jako „słabo” albo „silnie” krzywoliniowy. Indeksy czworoscianów danej kategorii zapisywane są w dwóch oddzielnych tablicach. Następnie na podstawie w\w indeksów wykonywane są sekwencyjnie **dwie** pętle najpierw dla czworoscianów „silnie”, a potem dla „słabo” krzywoliniowych. Podział na dwie

TABELA 4.5: Czasy (w sekundach) wykonania całkowania numerycznego (NI) w zależności od liczby równoległe przetwarzanych czworościanów w porcji oraz zastosowanej kwadratury ( $R$ -rzęd, $Q$ -liczba punktów). Problem testowy liczby 13613 czworościanów. GPU: Tesla K40c.

l. czworościanów w porcji	$R=6$ $Q=24$	$R=9$ $Q=61$	$R=10$ $Q=81$	$R=11$ $Q=109$	$R=14$ $Q=236$
1	1,244	2,117	2,517	3,100	4,070
2	0,662	1,130	1,359	1,676	4,089
4	0,382	0,912	0,777	1,455	3,146
8	0,274	0,691	0,73	1,357	2,880
16	0,201	0,776	0,693	1,259	2,663
32	0,183	0,698	0,608	1,144	2,583
64	0,163	0,682	0,573	1,121	2,542
128	0,148	0,666	0,553	1,087	2,521
256	0,140	0,659	0,542	1,073	2,506
512	0,133	0,657	0,537	1,066	2,500

pętla został zaproponowany, aby zagwarantować zrównoważenie obliczeń w trakcie obliczania lokalnych macierzy elementów sztywności  $\mathbf{S}^{(e)}$  i bezwładności  $\mathbf{T}^{(e)}$  (najpierw przetwarzane są czworościany z kwadraturą 10 rzędu (81 punktów), potem czworościany z kwadraturą niższego 6 rzędu (24 punktów)).

Podzbiory „słabo” i „silnie” krzywoliniowych czworościanów są następnie dzielone na mniejsze porcje. W obrębie jednej porcji czworościany przetwarzane są równoległe. W tym przypadku jeden blok wątków CUDA (wymiar Y) został przypisany do jednego czworościanu w porcji, a kolejne porcje przetwarzane były sekwencyjnie (psedokod został umieszczony w Dodatku C, Wydruk C.1). Optymalna liczba czworościanów w porcji zależy od architektury. Testy wykazały, iż dla architektury Kepler (Tesla K40c) optymalne<sup>2</sup> są porcje zawierające po 256 czworościanów (Tab. 4.5). Kolejnym poziomem zrównoleglenia obliczeń było wykonanie niezależnie obliczeń dla wszystkich punktów kwadratury Gaussa dla pojedynczego czworościanu - pojedynczy blok wątków (wymiar Z) został przypisany do jednego punktu kwadratury.

Na najniższym poziomie wykonywane są mnożenia macierzy gęstych o rozmiarze wynikającym z liczby funkcji bazowych ( $B = 50$ ) i rozmiarów macierzy Jacobiego ( $D = 3$ ). Jak wcześniej wyjaśniono, z uwagi na rozmiar macierzy do tych operacji nie powinno stosować się biblioteki CUBLAS. W Dodatku C na Wydruku C.2 przedstawiono kod kernela dedykowanego operacji mnożenia rzeczywistych macierzy gęstych wartości funkcji bazowych  $\mathbf{N}$  i macierzy  $\mathbf{J}\mathbf{J}$  będącej iloczynem transpozycji odwrotności macierzy Jacobiego, ośrodka reprezentowanego przez  $\epsilon$  oraz odwrotności macierzy Jacobiego ( $\mathbf{J}\mathbf{J} = \mathbf{J}^{-T}\epsilon\mathbf{J}^{-1}$ , patrz. r. (4.8)) dla  $Q = 81$  punktów kwadratury (rzęd kwadratury  $R = 10$ ). Zamieszczony w dodatku kernel oprócz mnożenia macierzy realizuje pozostałe etapy strategii zrównoleglenia opisane powyżej. Inaczej mówiąc w rozważanym kernelu zrównoleglenie obliczeń odbywa się na kilku poziomach:

<sup>2</sup>Dla porcji liczącej 256 czworościanów uzyskano znaczną redukcję czasu wykonania całkowania numerycznego. Dwukrotne zwiększenie liczby czworościanów (z 256 do 512) nie zmniejsza znacząco czasu wykonania, a dwukrotnie zwiększa zapotrzebowanie na pamięć.

TABELA 4.6: Czas i przyspieszenie wykonania operacji całkowania na GPU i CPU w zależności od zastosowanej kwadratury niższego rzędu, mieszanej lub wyższego rzędu:  $Q=\{24,24\backslash 81,81\}$ . GPU: Tesla K40c, CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków, Hyper-Threading wyłączony). Problem testowy liczy 49854 czworościanów, z których 41% jest „słabo” krzywoliniowych.

Implementacja	$Q=24$	$Q=\{24\backslash 81\}$	$Q=\{81\}$
GPU [s]	0,503	1,392	1,980
CPU [s]	3,065	10,353	14,755
GPU vs. CPU	6,1	7,4	7,5

- każdy wiersz macierzy wynikowej jest obliczany przez niezależny wątek (w bloku wątków o wymiarze  $X$ ),
- równolegle wykonywane są obliczenia dla każdego czworościanu w porcji (*parallelTets*) czworościanów (wymiar bloku wątków -  $Y$ ),
- zrównoleglenie obliczeń dla  $Q = 81$  punktów kwadratury odbywa się na poziomie bloku wątków (wymiar bloku wątków -  $Z$ ).

Czasy całkowania numerycznego z zastosowaniem zaproponowanych procedur i strategii umieszczono w Tab. 4.6. W referencyjnej implementacji na CPU użyto funkcji z biblioteki Intel MKL do wykonania operacji na macierzach gęstych i, sugerując się zaleceniami, wyłączono opcję Hyper-Threading celem uzyskania lepszej wydajności na CPU. Zauważyć można, iż zarówno dla GPU jak i CPU uzyskuje się znaczące skrócenie czasu wykonania operacji całkowania, gdy zamiast kwadratury wyższego rzędu  $R = 10$  ( $Q = 81$ ) stosuje się kwadraturę mieszaną. Zaproponowane strategie pozwalają na 6,1-7,5 krotne skrócenie czasu wykonania operacji całkowania numerycznego na GPU względem CPU. Ponadto przewaga GPU rośnie wraz ze wzrostem liczby wykonywanych operacji (rzędem kwadratury).

#### 4.1.1.2 Całkowanie w przypadku innych typów ośrodków

Przedstawione powyżej rozważania dotyczyły najprostszego (i odpowiedniego dla filtru mikrofalowego przedstawionego na rys 4.1) przypadku, w którym parametry ośrodka (przenikalność elektryczna i magnetyczne) opisane są przez rzeczywiste skalary. Dla innych struktur  $\epsilon$  i  $\mu$  z równań (4.5)-(4.8) mogą być tensorami rzeczywistymi, skalarami zespolonymi lub tensorami zespolonymi. W ogólności implementując metodę elementów skończonych można było założyć, że wszystkie rodzaje ośrodków  $\epsilon$  i  $\mu$  reprezentowane są jako tensor zespolony. Jednakże takie podejście byłoby nieefektywne z punktu widzenia wykorzystania zasobów pamięciowych i obliczeniowych. Skala nadmiarowości tych dwóch parametrów została zaprezentowana w Tab. 4.7. Zapotrzebowanie na pamięć i liczba operacji zmiennoprzecinkowych niezbędnych do wyznaczenia 256 lokalnych macierzy elementów  $\mathbf{T}^{(e)}$  z równania (4.8) przy wykorzystaniu kwadratury Gaussa 10 rzędu zostały zestawione w Tab. 4.7 niezależnie dla czterech wariantów, tzn. parametry fizyczne ośrodka dla każdego czworościanu są reprezentowane przez pojedynczy typ (np. skalar rzeczywisty - rząd 5, tensor zespolony - rząd 2). Liczby przedstawione w Tab. 4.7 wskazują bezspornie, iż dla ośrodka, którego parametry fizyczne są liczbami

TABELA 4.7: Zapotrzebowanie na pamięć (MB) i liczba operacji zmiennoprzecinkowych (Mflop) jaka musi zostać wykonana aby obliczyć 256 lokalnych macierzy sztywności  $\mathbf{T}^{(e)}$  przy wykorzystaniu kwadratury Gaussa 10 rzędu dla każdego ośrodka niezależnie. W nawiasach przedstawiono procent zaoszczędzonej pamięci i redukcję liczby operacji zmiennoprzecinkowych względem wariantu, gdy ośrodek jest reprezentowany przez tensor zespolony.

Ośrodek	MB	Mflop
Tensor zespolony	115,7 ( 0%)	720,3 ( 0%)
Skalar zespolony	113,1 (-2%)	684,3 (-5%)
Tensor rzeczywisty	57,9 (-50%)	360,2 (-50%)
Skalar rzeczywisty	56,5 (-51%)	342,1 (-53%)

zespolonymi potrzeba jest alokacja większej ilości pamięci oraz trzeba wykonać więcej obliczeń.

Zgodnie z tym co opisano wcześniej, zdecydowano by w trakcie całkowania numerycznego porcja 256 czworościanów była przetwarzana równolegle. Celem maksymalizacji efektywności procesu całkowania numerycznego zapewniono zrównoważenie obliczeń (ang. load balancing), tak aby przetwarzanie pojedynczego czworościanu potrzebowało tej samej liczby obliczeń i operacji na pamięci. Zrównoważenie obliczeń dla wariantu całkowania numerycznego, który wykorzystuje koncepcję kwadratur mieszanych oraz podział na różne typy ośrodka jest możliwe w przypadku, gdy czworościany podzielono na podzbiory nie tylko ze względu na typ kwadratury, ale także ze względu na rodzaje ośrodka. Taki podział sprawił, iż przy generacji macierzy może wystąpić aż 16 różnych przypadków w zależności od:

- rodzaju macierzy (2x -  $\mathbf{S}$  i  $\mathbf{T}$ )
- rodzaju zastosowanej kwadratury (2x - „słabo” lub „silnie” krzywoliniowe)
- rodzaju ośrodka (4x - skalar rzeczywisty, tensor rzeczywisty, skalar zespolony, tensor zespolony)

Dla każdego z 16 wariantów należałoby przygotować 16 wersji dedykowanych kodów, aby dla każdego z przypadku otrzymać wysoką wydajność. Zauważono jednak, że koszt numeryczny dla niektórych typów ośrodków jest podobny (np. skalary rzeczywiste i tensory rzeczywiste). Dla skalarów zespolonych koszt obliczeniowy jest około dwukrotnie większy w przypadku gdy rzeczywiste i urojone części macierzy  $\mathbf{T}^{(e)}$  są obliczane oddzielnie (Tab. 4.7). Jednakże przy odpowiednim uporządkowaniu obliczeń, liczba operacji może zostać dwukrotnie zredukowana w sytuacji gdy skalarny parametr ośrodka zostanie wyciągnięty przed pętlę kwadratury z r. (4.7)-(4.8). Liczba operacji jest wówczas praktycznie taka sama jak w przypadku, gdy ośrodek reprezentuje skalar rzeczywisty. Analogicznego podejścia nie można zastosować dla ośrodków reprezentowanych przez tensor zespolony (zespolona macierz o rozmiarze  $3 \times 3$ ) ponieważ mnożenie macierzy nie jest przemienne. Identyczne wnioski dotyczą obliczeń lokalnych macierzy sztywności  $\mathbf{S}^{(e)}$  i przenikalności magnetycznej  $\mu$ .

W efekcie powyżej opracowanych strategii optymalizacyjnych i dzięki odpowiedniej organizacji obliczeń, uzyskano podobną wydajność w trakcie całkowania numerycznego dla skalarów rzeczywistych, tensorów rzeczywistych i skalarów zespolonych. Z tego powodu zdecydowano się na opracowanie dwóch oddzielnych wariantów całkowania numerycznego:

TABELA 4.8: Możliwe tablice indeksów po podziale na podzbiory czworoscianów w zależności od macierzy, kwadratury i rodzaju ośrodka w zaproponowanej implementacji generacji macierzy. typ1 - indeksy czworoscianów dla sytuacji, w której parametry fizyczne ośrodka są skalarą rzeczywistym, tensorem rzeczywistym lub skalarą zespolonym, typ2 - indeksy czworoscianów wypełnionych ośrodkiem, którego parametry fizyczne opisane są przez tensor zespolony.

Macierz	Kwadratura	Indeksy czworoscianów	Strumień
$\mathbf{T}^{(e)}$	Q=81	indeks_T_Q81_typ1	0
$\mathbf{T}^{(e)}$	Q=81	indeks_T_Q81_typ2	1
$\mathbf{S}^{(e)}$	Q=81	indeks_S_Q81_typ1	2
$\mathbf{S}^{(e)}$	Q=81	indeks_S_Q81_typ2	3
$\mathbf{T}^{(e)}$	Q=24	indeks_T_Q24_typ1	0
$\mathbf{T}^{(e)}$	Q=24	indeks_T_Q24_typ2	1
$\mathbf{S}^{(e)}$	Q=24	indeks_S_Q24_typ1	2
$\mathbf{S}^{(e)}$	Q=24	indeks_S_Q24_typ2	3

			krok (porcja)			
NI - warianty:	M	strumień	0 (0-3)	1 (4-6)	2 (7-8)	3 (9)
indeks_T_Q81_typ1	256	0	0	-	-	-
indeks_T_Q81_typ2	1024	1	1	4	7	9
indeks_S_Q81_typ1	786	2	2	5	8	-
indeks_S_Q81_typ2	512	3	3	6	-	-

RYSUNEK 4.5: Strumienie (streams) przypisane do wariantów całkowania numerycznego dla kwadratury 10 rzędu (81 punktów) dla podzbioru 1280 czworoscianów przetwarzanych sekwencyjnie w porcjach 256 czworoscianów.

- dla skalarów rzeczywistych, tensorów rzeczywistych oraz skalarów zespolonych (typ-1),
- dla tensorów zespolonych (typ-2).

Mniejsza liczba wariantów pozwoliła na otrzymanie większej wydajności. Ponadto zaproponowana optymalizacja zredukowała liczbę tablic indeksów czworoscianów do 8, w zależności od:

- macierzy (2x -  $\mathbf{S}^{(e)}$  i  $\mathbf{T}^{(e)}$ ),
- rzędu zastosowanej kwadratury (2x - „słabo” lub „silnie” krzywoliniowe),
- rodzaju ośrodka (2x - skalar rzeczywisty\tensor rzeczywisty\skalar zespolony lub tensor zespolony).

Ponadto aby wykorzystać możliwość współbieżnego wykonania kerneli (ang. concurrent kernel execution p. 3) oddzielne strumienie zostały przyporządkowane do opracowanych wariantów całkowania numerycznego (Tab. 4.8).

TABELA 4.9: Czasy (w sekundach) wykonania operacji całkowania numerycznego (NI) w zależności od procentowego udziału różnych typów ośrodków reprezentowanych przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony, użytej kwadratury oraz liczby strumieni. Problem testowy liczy 13613 czworościanów. GPU: Tesla K40c.

SR\TR\SZ\TZ %	Q=81 1 strumień	Q=81 2 strumienie	Q={24\81} 1 strumień	Q={24\81} 2 strumienie	Q={24\81} 4 strumienie
0\0\0\100	0,579	0,555	0,509	0,493	0,490
0\0\100\0	0,562	0,554	0,450	0,442	0,440
0\100\0\0	0,545	0,537	0,434	0,427	0,425
100\0\0\0	0,544	0,535	0,434	0,427	0,425
25\25\25\25	0,558	0,545	0,459	0,450	0,449

W celu podsumowania zastosowanych strategii poniżej przeanalizowano hipotetyczny scenariusz, w którym należy wykonać całkowanie numeryczne dla 1280 „silnie” krzywoliniowych czworościanów, z których każdy wypełniony jest ośrodkiem który można zakwalifikować do jednego z czterech typów (rys. 4.5). W 256 czworościanach występuje ośrodek typu-1 (bezstratne i stratne izotropowe dielektryki lub bezstratne anizotropowe dielektryki), oraz w 1024 czworościanach występuje ośrodek typu-2 w postaci stratnych anizotropowych dielektryków. Ponadto przenikalność magnetyczna w 768 czworościanach jest typu-1 i w 512 czworościanach typu-2 (tensor zespolony). Dla tych czworościanów należy obliczyć 1280 lokalnych macierzy sztywności i bezwładności. Z racji tego, że w jednej porcji przetwarzanych jest równolegle 256 czworościanów, w pętli wykonanych jest 10 kroków z użyciem 10 strumieni lub 4 kroki z użyciem 4 strumieni, które wykonują operacje związane z całkowaniem niezależnie dla różnych typów ośrodków zaprezentowanych na rys. 4.5. Warto zauważyć, iż w każdym kroku aktywna jest różna liczba strumieni. Rysunek 4.5 ma charakter poglądowy i ilustruje jak czworościany są podzielone zgodnie z typem macierzy, rodzajem ośrodka oraz jak strumienie są przydzielone do różnych wariantów całkowania numerycznego. Na omawianym rysunku szerokość każdego z bloków (odpowiadającego porcji 256 czworościanów) jest taka sama, bo dla każdego wariantu przetwarzana jest ta sama liczba czworościanów. W prawdziwej symulacji czas potrzebny na wykonanie obliczeń w porcji 256 czworościanów jest różny z powodu różnej liczby obliczeń jakie muszą być wykonane dla każdego typu ośrodka [88].

Potwierdzeniem zasadności przyjętych strategii implementacji operacji całkowania numerycznego na GPU jest Tab. 4.9. W każdym wierszu zaprezentowano czasy wykonania dla różnych typów ośrodków. W każdej kolejnej kolumnie podano czas wykonania całkowania numerycznego w zależności od przyjętej kwadratury oraz liczby strumieni. Analizując tabelę w kierunku pionowym można stwierdzić, że opracowanie osobnych kerneli dla każdego z typu ośrodka pozwala otrzymać duży zysk czasowy. Ponadto odpowiednie uporządkowanie obliczeń prowadzi do podobnego czasu dla wariantów: SR-skalar rzeczywisty, SZ-skalar zespolony, TR-tensor rzeczywisty, co uzasadnia połączenie tych wariantów w jeden (patrz. Tab. 4.8). Analizując tabelę w kierunku poziomym, zauważyć można że zaproponowane strategia użycia kwadratury mieszanej i przypisanie strumieni do obliczeń macierzy lokalnych także pozwoliło na redukcję czasu wykonania



TABELA 4.10: Przyspieszenie operacji całkowania numerycznego (NI) w zależności od procentowego udziału ośrodka reprezentowanego przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. Kwadratura mieszana  $Q=\{24\backslash 81\}$ . Problem testowy liczy 13613 czworościanów. GPU: Tesla K40c (4 strumienie), CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków, Hyper-Threading wyłączony).

SR\TR\SZ\TZ %	$Q=\{24\backslash 81\}$ GPU vs. CPU
0\0\0\100	11,9
0\0\100\0	7,6
0\100\0\0	7,5
100\0\0\0	7,5
25\25\25\25	8,8

całkowania numerycznego. Przyspieszenie implementacji na GPU (kwadratura mieszana, 4 strumienie) względem CPU zostało zaprezentowane w Tab. 4.10. W przypadku gdy parametry fizyczne ośrodka są tensorami zespolonymi zaproponowana w niniejszej pracy technika masywnego zrównoleglenia wykorzystująca możliwości GPU pozwoliła na prawie 12-krotne skrócenie czasu obliczeń całkowania numerycznego.

Podsumowując ten etap, rezultatem całkowania numerycznego są lokalne macierze elementów. W zależności od rodzaju ośrodka po wykonaniu obliczeń otrzymywanych jest  $M$  (=liczba przetwarzanych czworościanów) macierzy o rozmiarze  $50 \times 50$  wynikającym z liczby funkcji bazowych  $K$  przyjętych w p. 2.1:

- **SE:**  $M$  macierzy sztywności  $\mathbf{S}^{(e)}$
- **TE:**  $M$  macierzy bezwładności  $\mathbf{T}^{(e)}$

#### 4.1.2 Strategie masywnego zrównoleglenia wątków w składaniu globalnych macierzy sztywności i bezwładności na GPU

Kolejnym etapem MES, w którym można zastosować masywne zrównoleglenie wątków jest zebranie lokalnych macierzy utworzonych dla poszczególnych elementów i konstrukcja globalnych macierzy sztywności i bezwładności. W literaturze znaleźć można publikacje dotyczące budowy macierzy w metodzie elementów skończonych przy wykorzystaniu CPU i GPU [28, 47, 48]. W publikacji [47], autorzy dokonali porównania dwóch metod *Addto* i *LMA - Local Matrix Approach*. W pierwszej metodzie najpierw tworzona jest macierz, która odwzorowuje lokalny węzeł elementu na globalny numer węzła. Następnie wartości lokalnych macierzy elementów są sumowane przy użyciu operacji *Addto*. Globalna macierz jest przechowywana w formacie CRS (ang. Compressed Row Storage [3]). Autorzy zauważyli, iż zaproponowany algorytm jest efektywniejszy dla obliczeń wykonywanych na procesorze centralnym. Algorytm ten nie pozwoliłby uzyskać wysokiej wydajności na GPU, gdyż celem zapewnienia poprawności wyników należałoby wykonać wiele operacji atomowych<sup>3</sup>, które wprowadzają duże opóźnienia na

<sup>3</sup>Operacje atomowe zastosowane w celu uniknięcia „wyścigów” przy próbie zapisu danych w sam adres pamięci przez niezależne wątki, patrz. rozdział 3.



GPU. Ponadto, ponieważ macierz globalna jest przechowywana w formacie CRS, znalezienie pozycji, w którą należałoby wpisać element niezerowy wymagałoby wykonania poszukiwania (ang. bisection search) w strukturze macierzy rzadkiej<sup>4</sup>. Z tego powodu, autorzy zaproponowali algorytm *LMA* (ang. *Local Matrix Approach*). W algorytmie tym nie wykonuje się całościowej budowy macierzy globalnej i przez to nie ma potrzeby wykonania kosztownych operacji atomowych. Algorytm LMA bazuje na tym, że wyznaczenie wektora wynikowego operacji *matvec*  $\mathbf{q} = \mathbf{A}\mathbf{d}$  (która jest główną operacją metod iteracyjnego rozwiązywania układów równań) odbywa się wg. wzoru:

$$\mathbf{q} = (\mathcal{M}^T(\mathbf{A}^E(\mathcal{M}\mathbf{d}))) \quad (4.11)$$

gdzie  $\mathcal{M}$  określa macierz odwzorowującą lokalną indeksację węzłów na globalną indeksację węzłów w siatce,  $\mathbf{A}^E$  jest blokowo-diagonalną macierzą, w której blok „e” odpowiada lokalnej macierzy elementu. Algorytm LMA na GPU został również wykorzystany w [50].

Kolejne strategie implementacji budowy macierzy globalnych w MES zostały zaproponowane przez Darve, Cecka i Lew [48]. W pierwszym zaproponowanym w pracy [48] algorytmie (*GlobalNZ*) wykorzystuje się pamięć globalną i rozбивa się proces budowy macierzy na dwie fazy (dwa kernele). W pierwszym kernelu, pojedynczy wątek oblicza element niezerowy macierzy globalnej i aby uniknąć „wyścigów” (przy zapisie do pamięci) zapisuje kolejne elementy do pamięci globalnej. W drugim kernelu, wykorzystuje się wiele wątków i każdy z nich pobiera element z pamięci globalnej i zapisuje go w odpowiednie miejsce macierzy rzadkiej. Ograniczeniem tej strategii są opóźnienia przy dostępie do pamięci globalnej. Druga strategia zaproponowana w publikacji [48] oznaczona jako *SharedNZ*, redukuje opóźnienia przy dostępie do pamięci globalnej przez wykorzystanie pamięci wspólnej (ang. shared memory) do przechowywania elementów. W implementacji na GPU jeden blok wątków odpowiedzialny jest za wyznaczenie grupy elementów i umieszczenie ich w globalnej macierzy rzadkiej.

W artykule [28] zaproponowano dwie strategie budowy macierzy rzadkich w metodzie elementów skończonych na GPU. W pierwszej strategii, po tym jak wyznaczono lokalne macierze wszystkich elementów skończonych (przechowywane w pomocniczej tablicy), każdy wątek ma za zadanie wpisanie do macierzy globalnej wartości z macierzy lokalnej. Wpisanie do macierzy globalnej odbywa się przy użyciu kolejnej tablicy pomocniczej, którą uprzednio wyznaczono na CPU. Wadą powyższego algorytmu jest - analogicznie jak strategii z [47] - duża liczba operacji atomowych podczas tworzenia globalnych macierzy, co obniża wydajność algorytmu wykonanego na GPU. W drugiej strategii zaproponowanej w [28] nie ma etapu, w którym budowane są macierze lokalne dla wszystkich elementów skończonych (co znacząco optymalizuje wykorzystanie pamięci). Zamiast tablicy macierzy lokalnych na CPU stworzono strukturę danych *Edge*, która przechowuje w tablicy *eleGloNum* indeksy elementów skończonych, które zawierają daną krawędź. Na podstawie *Edge* pojedynczy blok wątków buduje jeden wiersz macierzy lokalnej jednego elementu skończonego. Następnie blok wątków wyznacza w pamięci wspólnej jeden wiersz macierzy globalnej, który następnie kopiowany jest z pamięci wspólnej do pamięci globalnej.

Powyższa analiza literatury wykazała, iż z punktu widzenia zrównoleglenia obliczeń budowa macierzy bezpośrednio w formacie CRS nie jest optymalna, gdyż wymaga

<sup>4</sup>W artykule [51] zaproponowano by poszukiwanie pozycji, w którą należałoby wpisać element niezerowy w strukturze macierzy rzadkiej wykonać przy użyciu algorytmu wyszukiwania binarnego (ang. binary search), w którym indeksy kolumn przechowywane są w pamięci wspólnej.

kosztownego preprocessingu lub operacji atomowych co znacząco wydłużyłoby czas wykonania budowy macierzy [28, 47, 48]. Z tego powodu zasadnym było rozdzielenie tego procesu na dwa etapy: łączenie elementów i budowa macierzy w formacie COO, a następnie konwersja formatu COO do formatu CRS [86]. W pierwszym kroku, w pętli przetwarzane są wszystkie macierze lokalne i generowane są trójki  $(i, j, v)$  przechowujące informacje o docelowym indeksie wiersza, indeksie kolumny i wartości niezerowej, które w naturalny sposób tworzą wektory reprezentacji macierzy w formacie COO (wektory  $I, J, V$ ). Z punktu widzenia zrównoleglenia obliczeń trójki te mogą być generowane przez niezależne wątki i nie pojawia się problem „wyścigów”, który wymusza stosowanie operacji atomowych. Niestety wektory reprezentujące macierz rzadką w formacie COO nie są posortowane (według rosnących indeksów wierszy i kolumn - co jest wymagane w pakietach tj. Intel MKL [43], CUSPARSE [92]) i zawierają duplikaty (wielokrotne występowanie elementów niezerowych o tych samych indeksach wierszy i kolumn, które pochodzą ze stopni swobody wspólnych dla przylegających elementów). Z tego powodu, drugi krok składania macierzy - konwersja z formatu COO do CRS - musi zagwarantować sortowanie elementów oraz eliminację duplikatów. Po wykonaniu powyższych kroków globalne macierze sztywności ( $\mathbf{S}$ ) i bezwładności ( $\mathbf{T}$ ) przechowywane były w formacie CRS.

#### 4.1.2.1 Składanie macierzy globalnych w formacie COO

Pierwszym krokiem proponowanego w niniejszej pracy podejścia do składania macierzy globalnych, korzystnego z punktu widzenia właściwości GPU, jest utworzenie rzadkich macierzy sztywności i bezwładności reprezentowanych w formacie COO (ang. Coordinate format [3]), czyli w postaci wektorów indeksów wierszy i kolumn ( $\mathbf{JI}_{\text{coo}}$ )<sup>5</sup> oraz wartości niezerowych ( $\mathbf{S}_{\text{coo}}, \mathbf{T}_{\text{coo}}$ ). Na samym początku warto zaznaczyć, iż w celu optymalizacji procesu budowania macierzy zdecydowano by macierze  $\mathbf{S}$  i  $\mathbf{T}$  miały ten sam układ elementów niezerowych (ang. pattern), tzn. te same indeksy wierszy i kolumn oraz różne wartości niezerowe (rzeczywiste i urojone). Dzięki powyższemu założeniu operacje na indeksach zostały wykonywane tylko dla części rzeczywistej macierzy  $\mathbf{T}^{\text{Re}}$  i są takie same dla części rzeczywistej macierzy  $\mathbf{T}^{\text{Im}}, \mathbf{S}^{\text{Re}}, \mathbf{S}^{\text{Im}}$ . W konsekwencji zredukowano liczbę operacji na pamięci i zmniejszono zapotrzebowanie na pamięć (nie było potrzeby przechowywania indeksów wierszy i kolumn trzech macierzy na GPU:  $\mathbf{T}^{\text{Im}}, \mathbf{S}_e^{\text{Re}}, \mathbf{S}_e^{\text{Im}}$ )<sup>6</sup>.

Unikalną cechą proponowanego podejścia do składania macierzy jest umieszczenie indeksów wierszy i kolumn macierzy rzadkich w formacie COO w jednym wektorze  $\mathbf{JI}_{\text{coo}}$ . Indeksy kolumn  $\mathbf{J}$  oraz wierszy  $\mathbf{I}$  są przechowywane w wektorze  $\mathbf{JI}_{\text{coo}}$  naprzemiennie ( $\mathbf{JI}_{\text{coo}} = \{J[0], I[0], J[1], I[1], \dots\}$ ). Rozwiązanie to zostało zastosowane ze względu na szybsze sortowanie indeksów macierzy COO zapisanych w takiej formie w następnym etapie generacji macierzy (p. 4.1.2.2)<sup>7</sup>.

<sup>5</sup>W ogólności, w formacie COO wektory wierszy ( $\mathbf{I}_{\text{coo}}$ ) i kolumn ( $\mathbf{J}_{\text{coo}}$ ) są rozdzielone - w dalszej części rozdziału wyjaśni się, czemu w niniejszej pracy wektory te zostały połączone.

<sup>6</sup>Założenie, że macierze mają ten sam układ elementów niezerowych („pattern”) jest także korzystne z punktu widzenia redukcji liczby operacji i zapotrzebowania na pamięć w kolejnych krokach, tzn. konwersji z formatu COO do formatu CRS oraz składanie macierzy na CPU.

<sup>7</sup>Sortowanie indeksów macierzy stanowi istotną część opracowanego algorytmu konwersji macierzy z formatu COO do formatu CRS z eliminacją duplikatów (p. 4.1.2.2), opracowany zapis reprezentacji indeksów wierszy i kolumn umożliwia osiągnięcie przyspieszenia obliczeń, gdyż wystarczy wykonać sortowanie indeksów jednokrotnie, co jest efektywniejsze w porównaniu do sortowania przeprowadzonego dwuetapowo: najpierw po wierszach, a następnie po kolumnach.

Budowę macierzy w formacie COO zrealizowano w dwóch etapach opisanych poniżej.

### Pierwszy etap składania macierzy cząstkowych COO

W pierwszym etapie budowania macierzy w formacie COO wykonuje się obliczenia wstępne, których zasadnicze cele są następujące:

1. wyznaczenie liczby elementów niezerowych docelowych macierzy COO z duplikatami (w tej części kernelu brane są pod uwagę warunki brzegowe danego problemu MES),
2. wyznaczenie wartości indeksów pomocniczych `dofs`, które w drugim etapie budowania macierzy w formacie COO stosuje się do wyznaczenia indeksów wierszy i kolumn (`JIcoo`) macierzy,
3. wyznaczenie wartości indeksów pomocniczych `doflocs`, które w drugim etapie budowania macierzy w formacie COO stosuje się do określenia wartości elementów niezerowych macierzy. `doflocs` określa to gdzie zapisane zostały wartości z lokalnych macierzy elementów sztywności  $\mathbf{S}^{(e)}$  i bezwładności  $\mathbf{T}^{(e)}$  w macierzach globalnych.

Indeksy `dofs` i `doflocs` zostały tak skonstruowane, by docelowe macierze reprezentowane w formacie COO utworzyły hierarchiczną postać macierzy (p. 2.1), w której wydzielić można podmacierze związane z poszczególnymi rzędami funkcji bazowych.

Wyznaczenie liczby elementów niezerowych docelowych macierzy cząstkowych jest bardzo korzystne, gdyż umożliwia przydzielenie tablicom `JIcoo` i `Scoo`, `Tcoo` precyzyjnie określonej liczby bajtów. W takiej sytuacji przed faktycznym składaniem macierzy COO, można wyznaczyć ile należy przydzielić pamięci GPU RAM oraz czy jest wystarczająco dużo miejsca w pamięci GPU RAM dla budowanych globalnych macierzy sztywności i bezwładności.

### Drugi etap składania macierzy cząstkowych COO

W drugim etapie budowane są macierze rzadkie  $\mathbf{S}$  i  $\mathbf{T}$  reprezentowane w formacie COO. W wariantcie generacji macierzy całościowej na podstawie indeksów `dofs`, najpierw określone są wektory indeksów wierszy i kolumn elementów niezerowych macierzy w formacie COO (`JIcoo`), a na podstawie indeksów `doflocs` i wartości macierzy lokalnych wyznaczane są wektory wartości elementów niezerowych (`Scoo`, `Tcoo`).

Na tym etapie przyjęto konwencję dwupoziomowego zrównoleglenia obliczeń. Po pierwsze lokalne macierze elementów przetwarzano niezależnie (zrównoleglenie na poziomie bloków CUDA). Po drugie, wewnątrz bloków każdy z wątków równolegle wpisywał elementy do globalnych macierzach rzadkich. Innymi słowy, „i”-ty wątek wygenerował i zapisywał do pamięci globalnej czwórkę: `JIcoo[i]`, `JIcoo[i+1]` (na podstawie `dofs`), `Scoo[i]`, `Tcoo[i]` (na podstawie `doflocs`). Warto zaznaczyć, iż z racji maszynowego zrównoleglenia obliczeń elementy wektorów `JIcoo`, `Scoo`, `Tcoo` nie są posortowane oraz zawierają duplikaty (wielokrotne występowanie elementów niezerowych o tych samych indeksach wierszy i kolumn, które pochodzą ze stopni swobody wspólnych dla przylegających elementów).

W tym miejscu warto wspomnieć o jeszcze jednej strategii optymalizacyjnej zastosowanej w tej rozprawie. Ponieważ rozmiar pamięci GPU RAM jest silnie ograniczony, to w obliczeniach zastosowano mechanizmy wielokrotnego użycia tego samego obszaru pamięci (ang. data reuse). Wprowadzenie tego mechanizmu umożliwiło znaczące zwiększenie maksymalnego rozmiaru macierzy cząstkowych generowanych na pojedynczym

	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>
NI	1	1	1	1	1
COO	0	0	1	1	1
CRS k.1	0	0	1	1	1
CRS k.2	1	1	1	1	1
CRS k.3	1	1	1	1	1
CRS k.4	1	1	0	0	0
CRS k.5	1	1	1	0	0
CRS k.6	1	1	1	1	1
CRS k.7	1	1	1	1	1
CRS k.8	0	0	1	1	1
CRS k.9	1	0	0	1	1

RYSUNEK 4.6: Stan buforów ( $|B_1|B_2|B_3|B_4|B_5|$ ) w trakcie generacji macierzy (NI-całkowanie numeryczne, COO-utworzenie macierzy w formacie COO, CRS-9 kroków konwersji z formatu COO do formatu CRS z eliminacją duplikatów (p. 4.1.2.2)).  $B_i = 0$  - oznacza, że  $i$ -ty bufor nie został zarezerwowany do obliczeń i nie zawiera istotnych danych.  $B_i = 1$  oznacza, że bufor został zarezerwowany do obliczeń i może zawierać istotne dane.

GPU, co skutkuje poprawą efektywności generacji macierzy **S** i **T**, w szczególności w kolejnej fazie budowy macierzy rzadkich, jakim jest konwersja z formatu COO do formatu CRS z eliminacją duplikatów<sup>8</sup>. Wielokrotne użycie tych samych obszarów pamięci polega na utworzeniu pięciu buforów w GPU RAM oraz ich odpowiednim zarządzaniu. Bufory te w różnych etapach generacji macierzy zawierają różne informacje. Użycie buforów zrealizowano poprzez rzutowanie wskaźników do tych buforów (bufor1, ..., bufor5) na wskaźniki stosowane w danym etapie obliczeń. Stan buforów został oznaczony jako  $(|B_1|B_2|B_3|B_4|B_5|)$ , gdzie:

$B_i = 0$  oznacza, że  $i$ -ty bufor nie został zarezerwowany do obliczeń i nie zawiera istotnych danych,

$B_i = 1$  oznacza, że bufor został zarezerwowany do obliczeń i może zawierać istotne dane.

W pierwszym etapie generacji macierzy wszystkie bufor ( $|1|1|1|1|1|$ ) są zarezerwowane na potrzeby operacji całkowania numerycznego i budowy macierzy w formacie COO:

- bufor1 i bufor2 są przeznaczone do przechowywania macierzy sztywności **SE** i bezwładności **TE** (zbiór lokalnych macierzy generowany na etapie całkowania numerycznego).
- bufor3, bufor4, i bufor5 są zarezerwowane do przechowywania macierzy rzadkich w formacie COO (wektory: **S**coo, **T**coo, **J**Icoo).

Przed wykonaniem etapu budowy macierzy w formacie COO stan buforów wynosił  $|1|1|1|1|1|$ , a po wykonaniu zmienia się na  $|0|0|1|1|1|$  (rys. 4.6). Bufory 1-2 oznaczono jako puste, gdyż dane wyznaczone na etapie całkowania numerycznego (macierze sztywności **SE** i bezwładności **TE**), nie są potrzebne w dalszym etapie i obszar może być

<sup>8</sup>Więcej czworoscianów przetwarzanych w pojedynczej iteracji generacji macierzy, oznacza mniej duplikatów do sumowania (eliminacji).

wykorzystany w etapie konwersji do formatu CRS. Po wykonaniu budowy macierzy w formacie COO, bufor 3-5 przechowują odpowiednio wartości elementów niezerowych macierzy cząstkowych w formacie COO z duplikatami ( $S_{\text{coo}}$ ,  $T_{\text{coo}}$ ) oraz połączonego wektora indeksów kolumn i wierszy elementów niezerowych ( $JI_{\text{coo}}$ ).

W ramach podsumowania tej części w Tab. 4.11 porównano czasy wykonania etapu budowy macierzy w formacie COO na GPU i CPU. Zaproponowane strategie zrównoleglenia obliczeń pozwoliły uzyskać znaczne skrócenie czasu względem CPU (15,5-krotna i 8,5-krotna redukcja czasu wykonania dla problemów bezstratnych i stratnych). W implementacji na CPU wykorzystano zrównoleglenie przy użyciu interfejsu OpenMP, w taki sposób, że lokalne macierze elementów przetwarzano niezależnie (zrównoleglenie na poziomie wątków). Na CPU wykorzystano strukturę danych, w której wartości niezerowe macierzy przechowywane są naprzemiennie (część rzeczywista, część zespolona). Na GPU wartości przechowywane są w dwóch oddzielnych wektorach, co ma swoje konsekwencje w redukcji czasu zapisu danych. Z tego powodu przyspieszenie na GPU dla problemów zespolonych jest ok. dwa razy mniejsze niż dla problemów rzeczywistych.

TABELA 4.11: Czasy i przyspieszenie operacji budowy macierzy w formacie COO dla problemów rzeczywistych (parametry fizyczne ośrodka reprezentowanego przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty) i zespolonych (ośrodek reprezentowany przez: SZ - skalar zespolony, TZ - tensor zespolony). GPU: Tesla K40c, CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków, OpenMP). Rzadkie macierze sztywności i bezwładności o rozmiarze 213984 powstały z 13613 lokalnych macierzy.

Implementacja \ Ośrodek	SR, TR	SZ, TZ
GPU [s]	0,028	0,057
CPU [s]	0,435	0,502
GPU vs. CPU	<b>15,5</b>	<b>8,8</b>

#### 4.1.2.2 Konwersja z formatu COO do CRS z eliminacją duplikatów

Po zbudowaniu na GPU macierzy sztywności i bezwładności w formacie COO wykonana jest konwersja tych macierzy do formatu CRS (ang. Compressed Row Storage, [3]) wraz z eliminacją duplikatów. W ogólności konwersja z formatu COO do formatu CRS lub CCS (ang. Compressed Column Storage, [3]) jest jedną z fundamentalnych operacji na macierzach rzadkich i jest włączona w wiele bibliotek. W literaturze i w dostępnych bibliotekach UMFPACK<sup>9</sup> [94], SPARSKIT [95], Intel MKL [43], CUSPARSE [92] (Tab. 4.12) występuje kilka wariantów tej konwersji w zależności od różnych właściwości wektorów reprezentujących macierz w formacie COO ( $I$ - wektor indeksów wierszy,  $J$ - wektor indeksów kolumn,  $V$ - wektor wartości niezerowych). Podstawowa konwersja reprezentacji macierzy zakłada, że:

- nie występują duplikaty (elementy o różnych wartościach niezerowych a tych samych indeksach wierszy i kolumn  $(i, j)$ ),
- wektory indeksów są posortowane w rosnącym porządku (od najmniejszego na największego indeksu).

<sup>9</sup>UMFPACK jest jednym z modułów SUITESPARSE [93]



TABELA 4.12: Biblioteki pozwalające na konwersje macierzy z formatu COO do formatów CRS i CCS.

Funkcja	wyście: posortowane elementy	eliminacja duplikatów	zrównoleglenie	CPU	GPU
UMFPACK <i>UMFPACK_triplet_to_col</i>	Tak	Tak	Nie	Tak	Nie
SPARSKIT <i>coocsr</i>	Nie	Nie	Nie	Tak	Nie
SPARSKIT <i>coocsr, clncsr(job = 1)</i>	Nie	Tak	Nie	Tak	Nie
SPARSKIT <i>coocsr, clncsr(job = 3)</i>	Tak	Tak	Nie	Tak	Nie
Intel MKL <i>mkl_coo2csr</i>	Nie	Nie	Tak	Tak	Nie
CUSPARSE <i>cusparses &lt; t &gt; coo2csr</i>	Nie	Nie	Tak	Nie	Tak
Zaproponowany algorytm	Tak	Tak	Tak	Tak	Tak

W przypadku opisanym powyżej konwersja z formatu COO do formatu CRS ogranicza się do kompresji wektora indeksów wierszy (z  $I$  do  $I_{ptr}$ ). Powyższa funkcjonalności jest dostępna na CPU w bibliotece Intel MKL (funkcja `mkl_<t>coo2csr`), UMFPACK (`UMFPACK_triplet_to_col`), SPARSKIT (funkcja `coocsr`) i na GPU w bibliotece CUSPARSE (funkcja `cusparses<t>coo2csr`) (Tab. 4.12).

W przypadku gdy macierz reprezentowana w formacie COO zawiera duplikaty, należy je wyeliminować (zsumować) celem redukcji zapotrzebowania na pamięć. Biblioteka UMFPACK gwarantuje eliminację duplikatów. Biblioteka SPARSKIT pozwala na eliminację duplikatów, ale w takim przypadku najpierw należy wykonać podstawową konwersję `coocsr`, a następnie dodatkowa funkcja `clncsr` (wywołana z argumentem `job=1`) usuwa duplikaty.

Kolejnym przypadkiem jest sytuacja, w której wektory indeksów wierszy i/lub kolumn nie są posortowane. Z punktu widzenia wydajności kolejnych procedur wykonywanych na macierzy (np. mnożenie macierzy rzadkiej przez wektor z wykorzystaniem bibliotek Intel MKL [43] lub NVIDIA CUSPARSE [92]) należy zagwarantować, by po wykonaniu konwersji wektory indeksów kolumn (CRS) lub wierszy (CCS) były posortowane w porządku rosnącym. Taka funkcjonalność jest dostępna w bibliotece UMFPACK. W bibliotece SPARSKIT funkcja `coocsr` gwarantuje, że macierz rzadka będzie posortowana tylko względem rosnących indeksów wierszy. Aby uzyskać porządek rosnących indeksów kolumn w każdym wierszu macierzy rzadkiej, należy wykonać dodatkową funkcję `clncsr` (z argumentem `job=3`). Podsumowując charakterystykę konwersji z formatu COO do formatu CRS lub CCS można zauważyć, iż koszt podstawowej konwersji jest znacznie mniejszy niż w przypadku, gdy macierz wejściowa zawiera duplikaty oraz gdy indeksy wierszy i kolumn są nieposortowane. Ponadto wszystkie z powyższych implementacji na CPU dedykowane są obliczeniom na jednym wątku, a jedyna implementacja na GPU (CUSPARSE, funkcja `cusparses<t>coo2csr`) pozwala wyłącznie na wykonanie podstawowej konwersji (Tab. 4.12).

Najbardziej uniwersalną konwersję na CPU umożliwia biblioteka UMFPACK (funkcja

UMFPACK\_triplet\_to\_col, której etapy zostały opisane w Dodatku E.1). Powody dla czego bezpośrednie zrównoleglenie poszczególnych etapów konwersji z biblioteki UMFPACK nie pozwala otrzymać satysfakcjonujących wyników zostały szczegółowo opisane w Dodatku E.2. W tym miejscu przedstawiono wyłącznie otrzymane wnioski:

- zrównoleglenie niektórych etapów wymaga wykonania dużej liczby ( $NNZ$  = długość wektorów  $S_{\text{coo}}$ ,  $T_{\text{coo}}$ ) operacji atomowych,
- eliminacja duplikatów wymaga sekwencyjnego wyszukania duplikatów w każdym wierszu,
- zagwarantowanie posortowanych indeksów wierszy\kolumn wymaga wykonania dodatkowego sortowania na koniec konwersji.

Mając na względzie powyższe powody zdecydowano się stworzyć nowy równoległy algorytm konwersji reprezentacji macierzy rzadkich.

Argumentami wejściowymi algorytmu konwersji są macierze  $\mathbf{S}$  i  $\mathbf{T}$  zapisane w formacie COO z nieposortowanymi indeksami kolumn i wierszy umieszczonymi w jednej tablicy  $J_{\text{coo}}$  oraz z wartościami elementów niezerowych macierzy umieszczonymi w tablicach  $S_{\text{coo}}$  i  $T_{\text{coo}}$ . Macierze w formacie COO zawierają duplikaty (wielokrotne występowanie elementów niezerowych o tych samych indeksach wierszy i kolumn).

Opracowany algorytm konwersji składa się z dziewięciu głównych działań, które zostały pogrupowane na dwa etapy: *faza wstępna* oraz *docelowa konwersja*. W fazie wstępnej kluczową operacją jest jednokrotne podwójne sortowanie po indeksach wektorów  $J_{\text{coo}}$ . Jak się później okaże, wydajność konwersji zależy od efektywności funkcji sortującej. Sortowanie pozwoliło na ograniczenie liczby operacji atomowych i wydajniejsze zrównoleglenie działań w etapie docelowej konwersji. Opis wszystkich niuansów wymaga szczegółowej dyskusji na temat optymalizacji konkretnych etapów algorytmu konwersji. Poniżej zamieszczono tylko skrócony opis najważniejszych etapów konwersji z komentarzem odnoszącym się do zrównoleglenia:

#### Faza wstępna:

1. **Alokacja pamięci** - alokacja wektorów pomocniczych potrzebnych w późniejszych krokach konwersji,
2. **Jednoczesne sortowanie po indeksach kolumn i wierszy** - wektor  $J_{\text{coo}}$  (zawierający naprzemiennie indeksy kolumn i wierszy) został potraktowany jako tablica typu unsigned long (Wydruk 4.1) i dzięki temu w jednym wywołaniu **zrównoleglonej** funkcji sortującej elementy wektora posortowano względem kolumn i wierszy.  
[zrównoleglona funkcja sortująca (GPU: biblioteka Thrust)],

#### Konwersja:

3. **Wyznaczenie liczby elementów niezerowych w każdym wierszu (również duplikatów)** - dzięki fazie wstępnej liczba operacji atomowych została zredukowana do  $2 \times N$  (dwukrotność liczby wierszy macierzy rzadkich). Gdyby nie sortowanie w fazie wstępnej liczba operacji atomowych wynosiłaby  $NNZ$  (liczba elementów niezerowych)  
[zrównoleglenie:  $NNZ$  wątków zaangażowanych w obliczenia],



```
// d_Key = {0,1,...,Nz-1}
// d_JI = {J[0], I[0], J[1], I[1], ..., J[Nz-1], I[Nz-1]}
// THRUST pointers:
thrust::device_ptr<int> d_ptrKey ( d_Key );
thrust::device_ptr<unsigned long> d_ptrJI ( (unsigned long*) d_JI );
// sortowanie z kluczem (THRUST)
thrust::sort_by_key(d_ptrJI, d_ptrJI+Nz, d_ptrKey);
```

WYDRUK 4.1: Jednoczesne sortowanie po indeksach kolumn i wierszy przy użyciu funkcji z biblioteki Thrust.

4. **Wyznaczenie nowych wektorów z wartościami elementów niezerowych** - ułożenie zawartości tablic z wartościami elementów niezerowych zgodnie z permutacją zastosowaną wobec wektora  $JI_{coo}$ . Posortowane wektory wartości są zapisywane do buforów 1-2 i zwalniane są bufor 4-5, w których dotychczas przechowywane były nieposortowane wektory wartości niezerowych. Po wykonaniu obliczeń tej operacji stan buforów wynosił  $|1|1|0|0|$ .

*[zrównoleglenie: jeden wątek pracuje na jednym wierszu macierzy],*

5. **Kompresja wektora indeksów wierszy macierzy z duplikatami** - posortowany wektor indeksów wierszy jest wyodrębniony z wektora  $JI_{coo}$ . Wektor ten zostaje umieszczony w tablicy pomocniczej (bufor3) i stan buforów zmienia się na  $|1|1|1|0|0|$ . Następnie wektor indeksów poddawany jest kompresji, co zrealizowano przy użyciu algorytm *prefix sum*<sup>10</sup>.

*[zrównoleglony algorytm prefix sum],*

6. **Sumowanie wartości duplikatów** - eliminacja duplikatów polegała na sumowaniu wartości elementów niezerowych o takich samych indeksach wierszy i kolumn, a następnie odpowiednim zmniejszeniu rozmiaru wektorów indeksów kolumn i wartości elementów niezerowych w połączeniu z modyfikacją liczności elementów niezerowych zapisanych w skompresowanym wektorze wierszy. Ponieważ wektory indeksów kolumn i wartości elementów niezerowych zostały posortowane (patrz. faza wstępna, krok-2) to eliminacja duplikatów może zostać wykonana efektywniej, niż jej odpowiednik w pakiecie UMFPACK. Zwiększoną efektywność osiągnięto ponieważ sumowanie wartości duplikatów sprowadzało się do sumowania sąsiednich elementów wektora wartości niezerowych.

*[zrównoleglenie: jeden wątek pracuje na jednym wierszu macierzy],*

7. **Kompresja wektora indeksów wierszy macierzy bez duplikatów** - kompresja wykonana analogicznie do kroku. *[zrównoleglony prefix sum]*

8. **Budowanie wektorów wartości elementów niezerowych** - w etapie tym tworzy się wektory wartości elementów niezerowych macierzy  $S$  i  $T$  w formacie CRS. Stan buforów po wykonaniu tego kroku wynosi:  $|0|0|1|1|1|$  (zwolnione zostały bufor 1-2 przechowujące wartości elementów niezerowych z duplikatami, w buforach 4-5 umieszczono finalne wektory wartości niezerowe elementów niezerowych macierzy cząstkowych  $S$  i  $T$ ).

---

<sup>10</sup>Dla ciągu liczb  $x_n$  wyznaczono ciąg liczb  $y_n$ , taki iż  $y_n = \sum_{i=0}^n x_i$

*[zrównoleglenie: jeden wątek pracuje na jednym wierszu macierzy rzadkiej; dzięki sortowaniu w fazie wstępnej nie było potrzeby sortowania wartości względem rosnących indeksów kolumn],*

9. **Budowanie wektora indeksów kolumn** - z wektorów indeksów (bufor3) wyznacza się indeksy kolumn i wynik wpisuje się do bufor1. Stan buforów po wykonaniu tego kroku to: |1|0|0|1|1| (bufor3 został zwolniony).

*[zrównoleglenie: jeden wątek pracuje na jednym wierszu macierzy rzadkiej; dzięki sortowaniu w fazie wstępnej nie było potrzeby sortowania wartości względem rosnących indeksów kolumn]*

Po wykonaniu konwersji macierze sztywności i bezwładności przechowywane są w formacie CRS, mają odpowiednio posortowane elementy w każdym wierszu i nie zawierają duplikatów. W Tab. 4.13 przedstawiono czasy poszczególnych kroków konwersji macierzy rzeczywistych (przenikalność elektryczna i magnetyczna ośrodka reprezentowane przez skalary rzeczywiste i/lub tensory rzeczywiste) i macierzy zespolonych (ośrodek opisany przez skalary zespolone i/lub tensory zespolone). W fazie wstępnej alokacja wektorów nie ma znaczącego wpływu, z racji tego iż wielokrotnie użyto tych samych danych i alokowane są wyłączone zmienne pomocnicze o małym rozmiarze. W całej procedurze najwięcej czasu zajmuje sortowanie (ok. 67% całej konwersji), które na GPU zostało wykonane przy użyciu funkcji z biblioteki THRUST. Faza ta trwa tyle samo dla obydwu typów macierzy, bo sortowane są tylko indeksy kolumn i wierszy (JI<sub>COO</sub>). Tak jak to zostało zauważone wcześniej, sortowanie w fazie wstępnej pozwala na maszynowe zrównoleglenie operacji docelowej konwersji i ograniczenie liczby operacji atomowych. W docelowej konwersji, wzrost czasu wykonania kroków 3 i 8 spowodowany jest koniecznością wykonania operacji na wektorze przechowującym części urojone wartości elementów niezerowych. Porównanie czasów wykonania zaproponowanej konwersji z funkcją z biblioteki UMFPACK przedstawiono w Tabeli 4.14. Uzyskane 30 i 40-krotne skrócenie czasu obliczeń odpowiednio dla problemów rzeczywistych i zespolonych, potwierdza zasadność wykonania tego etapu generacji macierzy rzadkich na GPU.

TABELA 4.13: Czas (w milisekundach) wykonania opracowanej w ramach rozprawy konwersji z formatu COO do formatu CRS z eliminacją duplikatów na GPU (Tesla K40c). Kroki 1-2 to faza wstępna, a kroki 3-9 to docelowa konwersja z eliminacją duplikatów. Rozmiar macierzy testowej wynosi 213984 (Tab. 4.1).

Etapy konwersji	macierz rzeczywista	macierz zespolona	m. zespolona vs. rzeczywista
1. Alokacja pamięci	0,01	0,01	<b>1,0</b>
2. Jednoczesne sortowanie po indeksach JI <sub>COO</sub>	46,96	46,78	<b>1,0</b>
3. Wyznaczenie l. elementów niezerowych w wierszu	3,68	7,22	<b>2,0</b>
4. Wyznaczenie wektorów z wart. el. niezerowych	1,02	1,02	<b>1,0</b>
5. Kompresja indeksów wierszy z duplikatami	0,06	0,06	<b>1,0</b>
6. Sumowanie wartości duplikatów	13,91	13,93	<b>1,0</b>
7. Kompresja indeksów wierszy bez duplikatów	0,06	0,06	<b>1,0</b>
8. Budowanie wektorów wartości el. niezerowych	4,24	6,93	<b>1,6</b>
9. Budowanie wektora indeksów kolumn	0,03	0,03	<b>1,0</b>
Suma	69,97	76,04	<b>1,1</b>

TABELA 4.14: Czas wykonania i przyspieszenie zaproponowanej konwersji z eliminacją duplikatów na GPU względem implementacji dostępnej w bibliotece UMFPACK. GPU: Tesla K40c, CPU: Intel Xeon Sandy Bridge E5-2687W. Rozmiar macierzy testowej wynosi 213984 (Tab. 4.1).

	UMFPACK (CPU) [s]	GPU [s]	GPU vs. CPU
macierz rzeczywista	2,150	0,070	<b>30,7</b>
macierz zespolona	3,156	0,076	<b>41,5</b>

### 4.1.3 Podsumowanie podstawowego wariantu generacji macierzy na GPU.

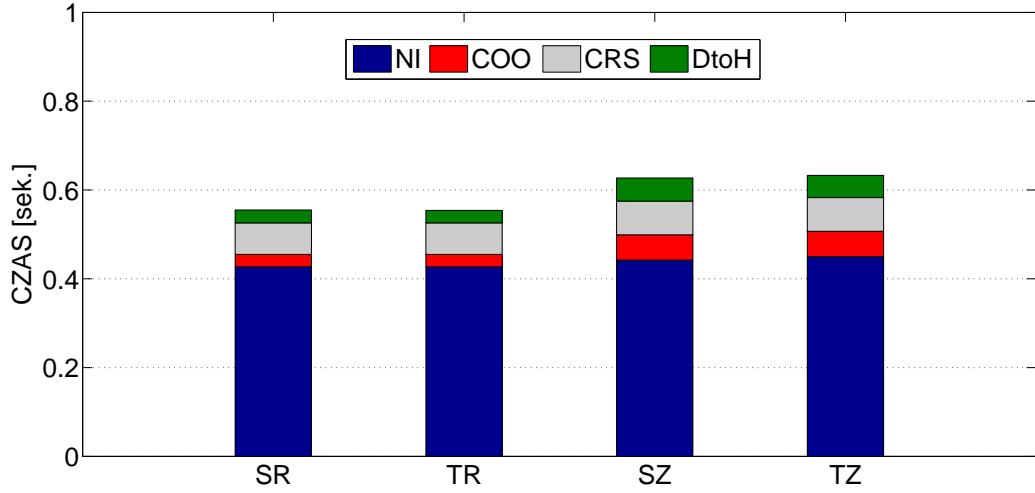
W punkcie tym podsumowano opisane wcześniej etapy generacji macierzy sztywności i bezwładności. W pierwszej kolejności dokonano porównania czasów wykonania poszczególnych etapów generacji na GPU (rys. 4.7a) i na CPU (rys. 4.7b). Zarówno na GPU i CPU czas całkowania numerycznego jest czynnikiem dominującym. Poprzez zaproponowane strategie masywnego zrównoleglenia i ich implementacji oraz odpowiedni porządek wykonania obliczeń czas tego etapu dla problemów, w których ośrodki reprezentowane są przez skalary lub tensory zespolone jest niemal identyczny (przyp. Tab. 4.7). Istotny udział w skróceniu czasu ma też wykorzystanie koncepcji kwadratury mieszanej. Dane zgromadzone w Tab. 4.15 potwierdzają, iż dla wszystkich etapów generacji uzyskano znaczące przyspieszenie na GPU względem CPU. Wykorzystanie akceleratorów graficznych w obliczeniach pozwoliło na kilkunastokrotną redukcję czasu generacji macierzy. Jako miarę szybkości generacji macierzy MES, można przyjąć liczbę czworościanów jakie analizowane są w trakcie jednej sekundy w trakcie obliczeń na GPU i na CPU. Dane z Tab. 4.16 wskazują, że w generacji macierzy przeprowadzonej na GPU - w zależności od typu ośrodka - w trakcie jednej sekundy można przeanalizować ponad 20 000 czworościanów. Dla tych samych sformułowań MES, na CPU przeanalizować można około od półtora do dwóch tysięcy czworościanów.

Analizując czasy poszczególnych etapów w zależności od zastosowanego ośrodka wprowadzono wzór pozwalający oszacować czas generacji macierzy na GPU i CPU:

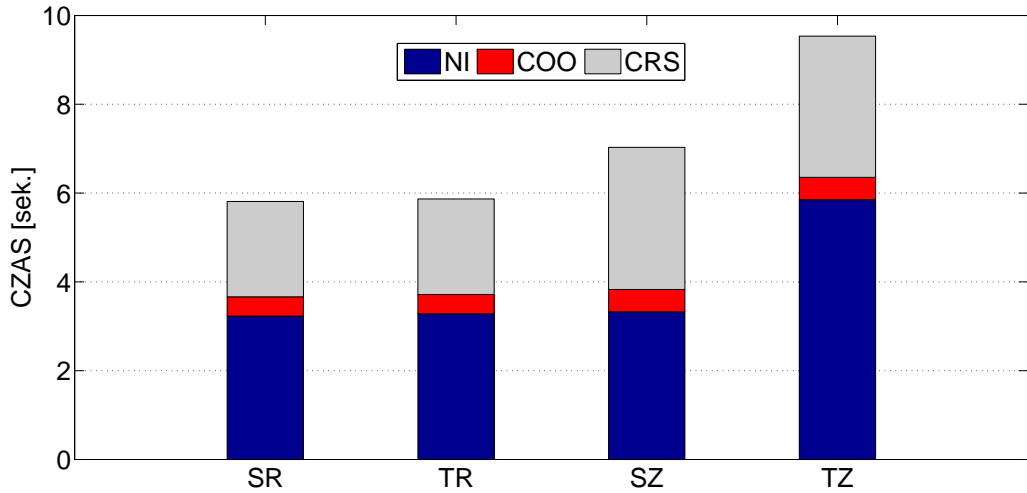
$$MatGen = NI \cdot (K_R \cdot X + K_{SZ} \cdot Y + K_{TZ} \cdot Z) + COO \cdot K_1 + CRS \cdot K_2 \quad (4.12)$$

TABELA 4.15: Przyspieszenie wykonania faz generacji macierzy sztywności i bezwładności na GPU (Tesla K40c) względem CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków) dla ośrodka reprezentowanego przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. W operacji całkowania numerycznego zastosowano kwadraturę mieszaną  $Q=\{24\backslash 81\}$ . Rozmiar macierzy sztywności i bezwładności  $N = 213984$ . GPU (Tesla K40c), CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków).

Faza	SR	TR	SZ	TZ
NI	7,6	7,7	7,5	11,9
COO	15,5	15,5	8,8	8,9
CRS	30,3	30,4	42,0	41,8
<b>MatGen</b>	<b>11,1</b>	<b>11,1</b>	<b>12,2</b>	<b>15,2</b>



(a) GPU (Tesla K40c)



(b) CPU (Intel Xeon Sandy Bridge E5-2687W)

RYSUNEK 4.7: Czas wykonania podstawowych faz budowy macierzy globalnych sztywności i bezwładności (NI-całkowanie numeryczne, COO-składanie macierzy globalnych w formacie COO, CRS-konwersja z formatu COO do formatu CRS z eliminacją duplikatów, DtoH (ang. Device to Host)-przesłanie macierzy z pamięci GPU (Device) do pamięci CPU (Host)). Dla wszystkich elementów skończonych ośrodki są reprezentowane przez: SR - skalary rzeczywiste, TR - tensory rzeczywiste, SZ - skalary zespolone, TR - tensory zespolone. W operacji całkowania numerycznego zastosowano kwadraturę mieszaną  $Q=\{24\backslash 81\}$ ; Rzadkie macierze sztywności i bezwładności o rozmiarze 213984 powstały z 13613 lokalnych macierzy.

gdzie: NI, COO, CRS reprezentują czas wykonania podstawowych faz generacji dla skalarnego ośrodka bezstratnego,  $K_i$  to współczynniki reprezentujące wzrost czasu etapów COO i CRS w zależności od zastosowanego wariantu;  $K_R$  współczynnik dla skalarów i/lub tensorów rzeczywistych;  $K_{TR}$  współczynnik dla tensorów rzeczywistych;  $K_{SZ}$  współczynnik dla skalarów zespolonych;  $K_{TZ}$  współczynnik dla tensorów zespolonych

TABELA 4.16: Liczba czworościanów przetwarzanych w trakcie pojedynczej sekundy w procesie generacji macierzy sztywności i bezwładności. Ośrodek reprezentowany przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. GPU (Tesla K40c), CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków). Problem testowy liczy 13613 czworościanów.

	SR	TR	SZ	TZ
GPU (M \ sek.)	25902	25863	23669	21763
CPU (M \ sek.)	2343	2320	1937	1428

TABELA 4.17: Współczynniki do wyznaczenia czasu generacji macierzy sztywności i bezwładności na GPU (Tesla K40c) dla wariantów w których ośrodek reprezentowany jest przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony, i kwadratura mieszana  $Q=\{24\backslash 81\}$ .

Współczynniki	SR	TR	SZ	TZ
$K_{SR}$	1	0	0	0
$K_{TR}$	0	1	0	0
$K_{SZ}$	0	0	1,04	0
$K_{TZ}$	0	0	0	1,15
$K_1$	1	1	2,03	2,02
$K_2$	1	1	1,07	1,07

(Tab. 4.17); X określa odsetek elementów reprezentowanych przez skalary i\lub tensorzy rzeczywiste; Y określa udział procentowy elementów reprezentowanych przez skalar zespolony; Z określa odsetek elementów reprezentowanych przez tensor zespolony. Powyższe rezultaty testów numerycznych potwierdzają, iż zaproponowany podstawowy algorytm generacji macierzy w metodzie elementów skończonych, wykorzystujący masywne zrównoleglenie obliczeń na GPU, pozwala na znaczącą redukcję czasu obliczeń tego etapu MES względem implementacji na procesorze centralnym CPU. Należy jednak zauważyć, że algorytm ten może być wykorzystany dla relatywnie małych macierzy z racji tego, że wszystkie dane (informacje nt. siatki, macierze wykorzystywane i obliczane w etapie całkowania numerycznego i budowy macierzy globalnych) muszą być przechowywane na pojedynczym akceleratorze. Innymi słowy, rozmiar globalnych macierzy sztywności (**S**) i bezwładności (**T**) zależy od rozmiaru pamięci akceleratora i podstawowy algorytm pozwala na analizę problemów o rozmiarze, który mieści się w pamięci akceleratora graficznego. Na akceleratorze K20, który ma do dyspozycji 5 GB pamięci, dla sformułowań MES wykorzystanych w niniejszej pracy można przeprowadzić symulację maksymalnie 40 000 elementów skończonych, co prowadzi do generacji macierzy o rozmiarze ok. 600 000. Niestety w symulacjach elektromagnetycznych potrzeba zwykle znacznie gęstszej siatki, aby przeprowadzić dokładną analizę komponentów urządzeń mikrofalowych. Z tego powodu w kolejnych punktach zaprezentowano warianty przewyższające te ograniczenie i pozwalające na generację znacznie większych macierzy MES.

## Rozdział 5

# Konstrukcja dużych globalnych macierzy sztywności i bezwładności

Sposób konstrukcji macierzy w MES przedstawiony w poprzednim rozdziale nadaje się do problemów małych i średnich, a maksymalny rozmiar macierzy ograniczony jest przez ilość szybkiej pamięci RAM akceleratora. Celem przezwyciężenia ograniczenia związanego z rozmiarem pamięci akceleratora graficznego można zastosować algorytm **iteracyjnej** generacji dużych macierzy bezwładności i sztywności, który pozwala zachować dużą efektywność przejawiającą się redukcją czasu generacji macierzy względem procesora centralnego (rys. 5.1) [89].

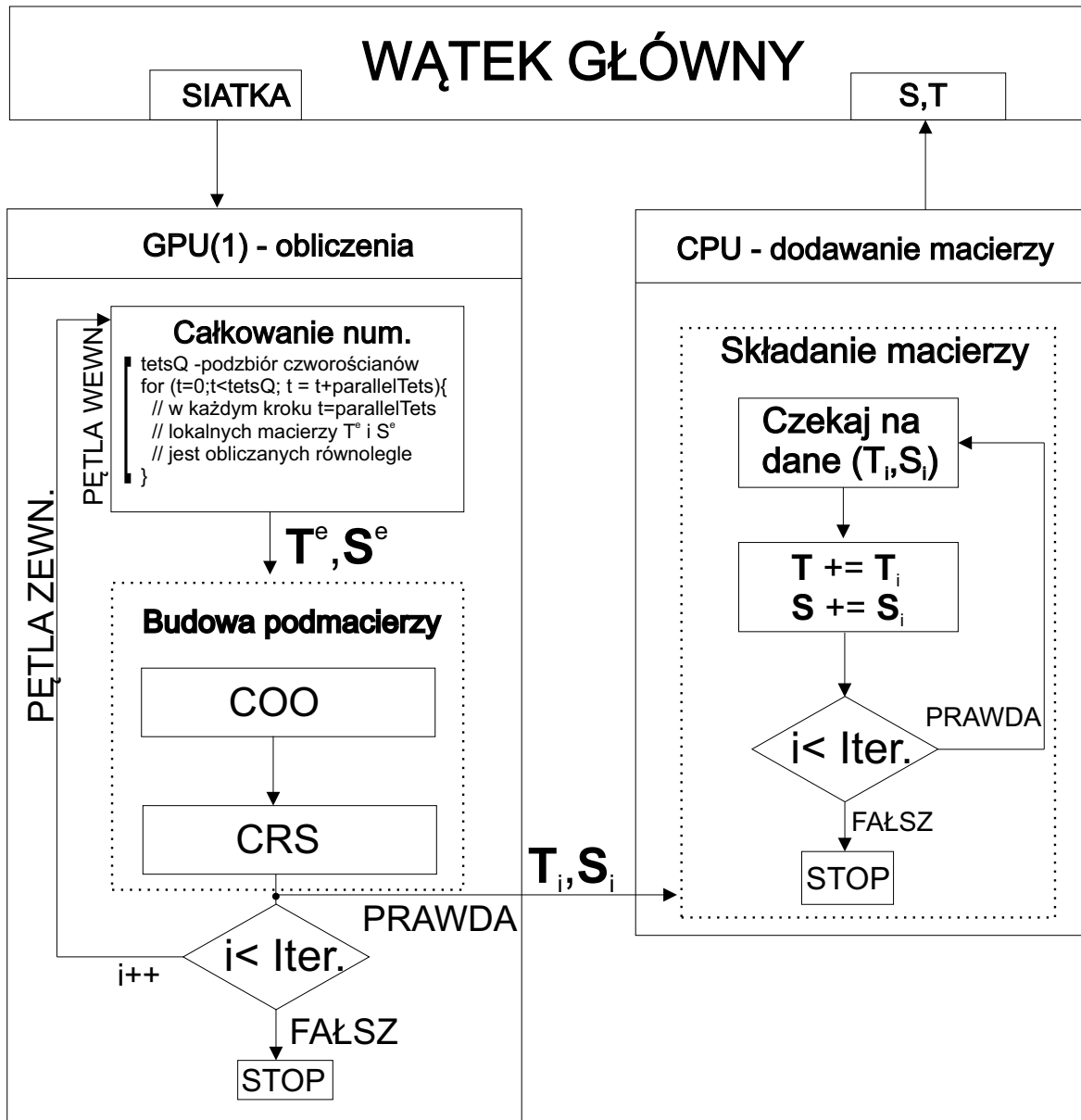
W wariancie podstawowym (p. 4.1) przetwarzane są wszystkie elementy skończone (czworościany). W wariancie iteracyjnym (rys. 5.1), zbiór wszystkich czworościanów dzieli się na mniejsze podzbiory. Rozmiar podzbioru czworościanów jest dobrany tak, aby w pojedynczej iteracji maksymalnie wykorzystać rozmiar pamięci dostępnej na GPU. Każdy podzbiór czworościanów przetwarza się tak jak w wariancie podstawowym, tzn. najpierw wykonuje się całkowanie numeryczne, następnie konstruuje fragmenty globalnych macierzy sztywności i bezwładności w formacie COO i przeprowadza się konwersję z eliminacją duplikatów z formatu COO do formatu CRS. Rezultatem obliczeń wykonanych w każdej iteracji są podmacierze sztywności  $\mathbf{S}_i$  i bezwładności  $\mathbf{T}_i$ , które na koniec każdej iteracji przesyła się z pamięci urządzenia (GPU) do pamięci hosta (CPU) w celu otrzymania macierzy globalnych. Aby otrzymać macierze globalne  $\mathbf{T}$  i  $\mathbf{S}$  po stronie CPU sumuje się macierze rzadkie  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ,  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ . **Ponieważ tylko porcja czworościanów jest przetwarzana na akceleratorze graficznym w pojedynczej iteracji, ilość pamięci na GPU nie determinuje rozmiaru generowanych macierzy. Z tego powodu, w iteracyjnym algorytmie generacji macierzy rozmiar macierzy zależy jedynie od ilości pamięci RAM jaka jest dostępna w stacji roboczej.**

Liczba wszystkich  $T$  wątków na CPU, które wykorzystuje się w procesie generacji macierzy została przedstawiona w r. (5.1):

$$T = T_{Master} + T_{GPU} + T_{HtoD} + T_{Mt} \quad (5.1)$$

gdzie  $T_{Master}$  to wątek główny odpowiedzialny za kontrolę całej generacji macierzy (odpowiada za przetwarzanie danych wejściowych wymaganych w generacji (siatka, ośrodek, wagi kwadratury), tworzy i wywołuje wątki, które przeprowadzają obliczenia na CPU i GPU, czeka na zakończenie obliczeń na GPU i CPU;  $T_{GPU}$  to wątek odpowiedzialny za wywołanie obliczeń na GPU (etapy: NI - całkowanie numeryczne, COO





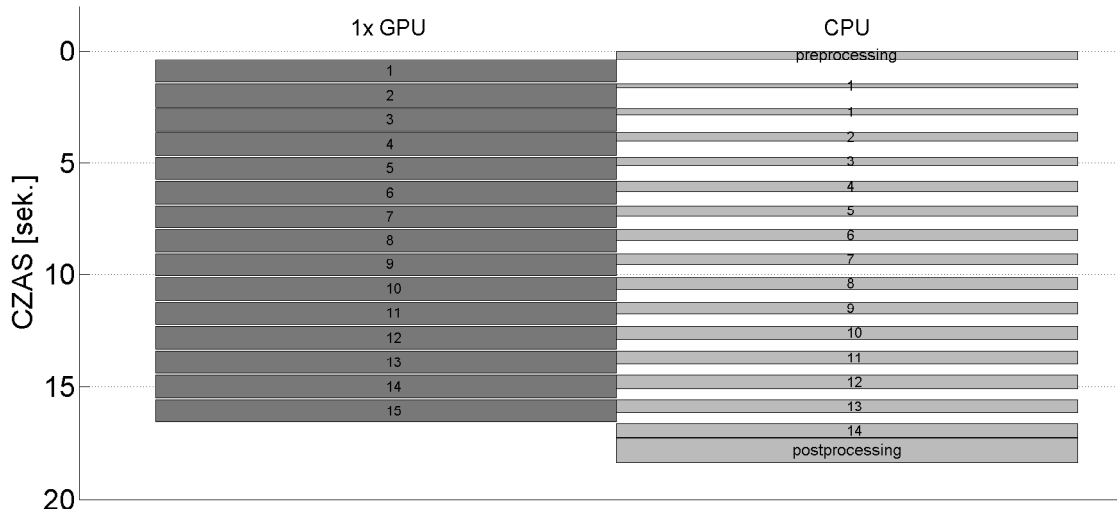
RYSUNEK 5.1: Iteracyjny wariant generacji macierzy w metodzie elementów skończonych [89]. Na GPU wykonywana jest generacja podmacierzy sztywności  $S_i$  i bezwładności  $T_i$ , a na CPU wykonywane jest składanie macierzy globalnych sztywności ( $S = S + S_i$ ) i bezwładności ( $T = T + T_i$ ).

- budowa macierzy w formacie COO, CRS - konwersja macierzy z formatu COO do formatu CRS z eliminacją duplikatów);  $T_{HtoD}$  to wątek odpowiedzialny za pobranie macierzy z pamięci GPU do pamięci CPU;  $T_{Mt}$  reprezentuje  $Mt$  wątków wykonujących dodawanie macierzy rzadkich na CPU ( $S = S + S_i, T = T + T_i$ ).

Kolejną własnością zaproponowanego iteracyjnego algorytmu konstrukcji macierzy w metodzie elementów skończonych jest możliwość częściowego „ukrycia” obliczeń poprzez wykonanie dodawania macierzy rzadkich na CPU równolegle do generacji podmacierzy  $S_i$  i  $T_i$  na GPU (rys. 5.2) [89]. Właściwość ta jest możliwa do osiągnięcia tylko w przypadku, gdy obliczenia na CPU trwają krócej niż obliczenia na GPU:

$$t_{CPU}[s] < t_{GPU}[s] \quad (5.2)$$



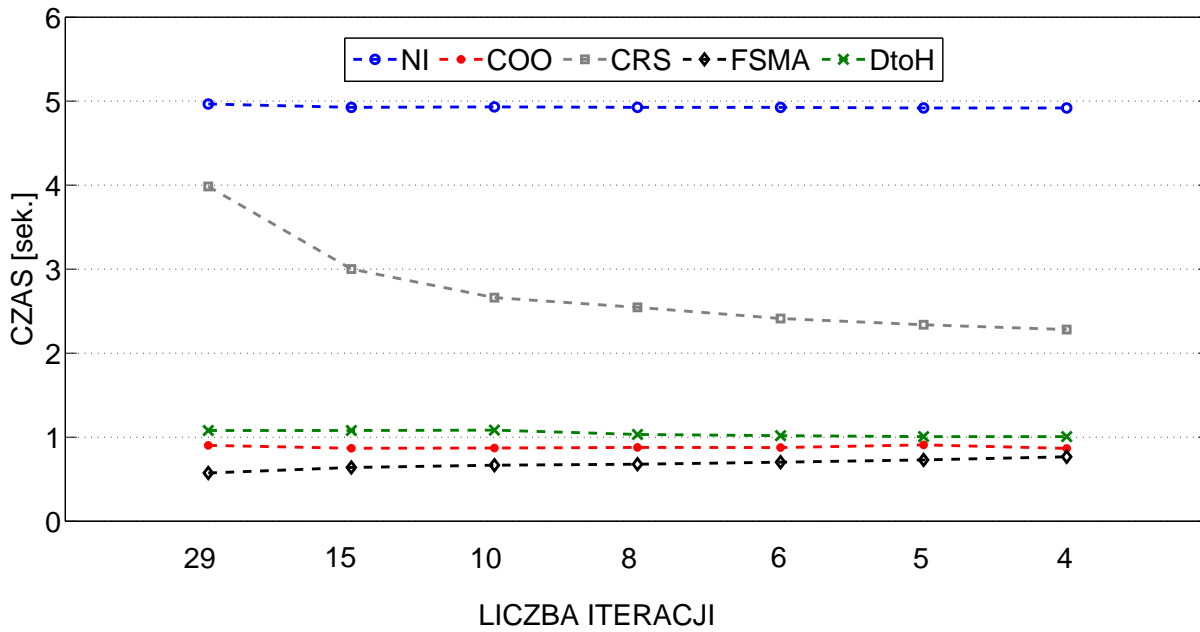


RYSUNEK 5.2: Oś czasu iteracyjnego algorytmu generacji macierzy. Po lewej stronie bloki reprezentują czas wykonania operacji całkowania numerycznego, budowy macierzy w formacie COO i konwersji do formatu CRS w pojedynczej iteracji (GPU: Tesla K40c), przerwy między blokami po stronie GPU reprezentują czas przesłania podmacierzy  $\mathbf{S}_i$  i  $\mathbf{T}_i$  z GPU na CPU. Po prawej stronie bloki reprezentują czas wykonania algorytmu szybkiego sumowania macierzy (FSMA) na CPU (Intel Xeon Sandy Bridge E5-2687W). W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1,  $\epsilon = 1$ ,  $\mu = 1$ ).

gdzie:  $t_{CPU}$  oznacza czas wykonania sumowania macierzy rzadkich na CPU w pojedynczej iteracji  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ ,  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ,  $t_{GPU}$  oznacza czas wykonania obliczeń na GPU związanych z generacją podmacierzy w pojedynczej iteracji (rys. 5.1).

W procesie iteracyjnym akcelerator rozpoczyna wykonywać operacje w  $i+1$  iteracji, podczas w tym samym czasie CPU wykonuje sumowanie w  $i$ -ej iteracji ( $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ,  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ ). Innymi słowy, akcelerator graficzny nie musi czekać na to by CPU zakończyło obliczenia związane z sumowaniem macierzy. Warunek (5.2) wskazuje, iż czas wykonania obliczeń na CPU jest krytyczny z punktu widzenia „ukrywania” obliczeń. Z tego powodu dodanie macierzy rzadkich  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ,  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$  musi być wykonane bardzo efektywnie. W pierwszej kolejności wykorzystano funkcję *mkl\_dcsradd* z biblioteki Intel Math Kernel Library (MKL) [43]. Zgodnie z opisem biblioteki, procedury użyte do wykonania tej operacji zostały zoptymalizowane pod kątem wykonania obliczeń na wielordzeniowym procesorze centralnym. Niestety przeprowadzone testy wykazały, iż funkcja *mkl\_dcsradd* wykonuje obliczenia na pojedynczym wątku ( $Mt = 1$ ), co jest poniżej możliwości wykorzystania zasobów wielordzeniowych CPU oraz nie gwarantuje „ukrywania” obliczeń gdyż przy jej użyciu nie jest spełniony warunek (5.2). Z tego powodu, w algorytmie generacji macierzy wykorzystano algorytm FSMA (ang. Fast Sparse Matrix Addition) opracowany przez dra inż. Piotra Sypka, który pozwala na wykorzystanie wielu wątków w trakcie wykonywania obliczeń związanych z dodawaniem macierzy rzadkich, dzięki czemu okazał się wielokrotnie wydajniejszy niż funkcja z biblioteki Intel MKL<sup>1</sup>. Założenia algorytmu FSMA zostały opisane w Dodatku D.

<sup>1</sup>Ponadto algorytm FSMA pozwala na jednoczesne dodawanie macierzy z kilku iteracji generacji



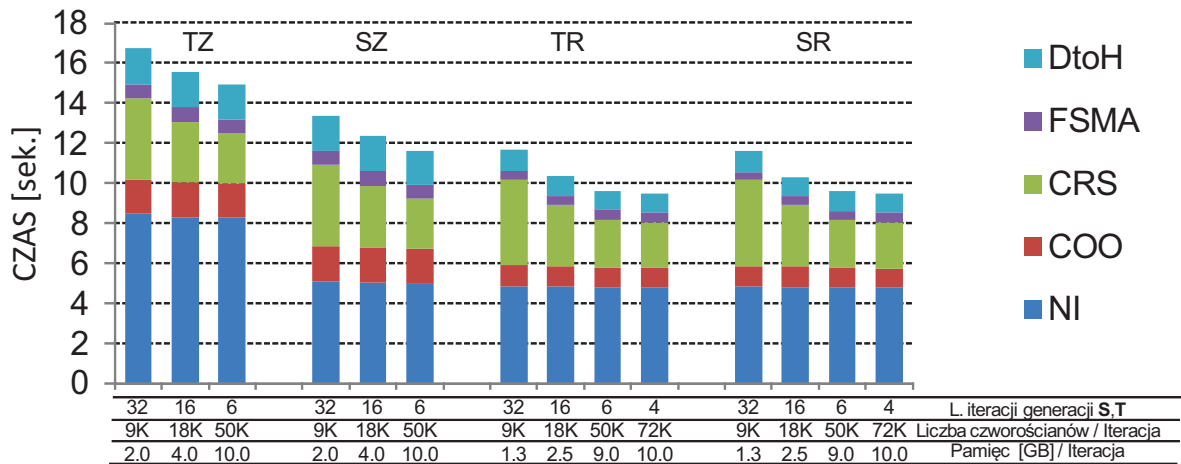
RYSUNEK 5.3: Wpływ liczby iteracji na proces generacji macierzy. NI - całkowanie numeryczne, COO - budowa macierzy w formacie COO, CRS - konwersja macierzy z formatu COO do formatu CRS z eliminacją duplikatów, FSMA - szybki algorytm sumowania macierzy, DtoH (ang. Device to Host) - przesłanie macierzy w formacie CRS z pamięci GPU (Device) do pamięci CPU (Host). W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1,  $\epsilon = 1$ ,  $\mu = 1$ ).

Charakteryzując algorytm iteracyjnej generacji macierzy na GPU należy opisać dwie fazy, które muszą być dodatkowo wykonane: przetwarzanie wstępne (ang. preprocessing) i przetwarzanie końcowe (ang. postprocessing) generacji macierzy sztywności i bezwładności. Pierwsza faza wykonana jest jednokrotnie przed rozpoczęciem procesu iteracyjnego zawiera alokację danych w pamięci CPU (potrzebnych dla algorytmu FSMA), alokację danych na GPU (potrzebnych do wykonania generacji podmacierzy sztywności  $\mathbf{S}_i$  i bezwładności  $\mathbf{T}_i$ ). Faza przetwarzania końcowego zawiera kopiowanie macierzy  $(\mathbf{T}, \mathbf{S})$  z pamięci niestronicowanej (ang. page-locked memory) do pamięci stronicowanej (ang. pageable memory). Pamięć stronicowana gwarantuje krótszy czas przesłania danych pomiędzy GPU i CPU [7], gdyż system operacyjny zapewnia przechowywanie jej w fizycznej pamięci RAM komputera.

Kolejną właściwością wariantu iteracyjnego jest wpływ liczby iteracji na czas generacji macierzy. O ile czas całkowania numerycznego i budowy macierzy w formacie COO nie zmienia się przy zmianie liczby iteracji, to czas konwersji (CRS) rośnie wraz ze wzrostem liczby iteracji (rys. 5.3). Przyczyną tej zależności jest wzrost liczby duplikatów, które należy wyeliminować w etapie konwersji. Z tego powodu należy zawsze obciążać GPU jak największą liczbą obliczeń. Nie tylko zapewni to optymalny czas konwersji CRS, ale także stworzy możliwość ukrycia obliczeń na CPU związanych z wykonaniem algorytmu FSMA.

---

macierzy, co jest szczególnie istotne z punktu widzenia kolejnego wariantu generacji macierzy układów równań z wykorzystaniem kilku akceleratorów graficznych.



RYSUNEK 5.4: Wpływ liczby iteracji na konstrukcję macierzy sztywności i bezwładności. NI - całkowanie numeryczne, COO - budowa macierzy w formacie COO, CRS - konwersja macierzy z formatu COO do formatu CRS z eliminacją duplikatów, FSMA - szybki algorytm sumowania macierzy, DtoH (ang. Device to Host) - przesłanie macierzy w formacie CRS z GPU (Device) do CPU (Host). Parametry fizyczne ośrodka opisane reprezentowane przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1).

Wpływ rozmiaru pamięci dostępnej na GPU na czas wykonania iteracyjnego procesu generacji macierzy przedstawiono na rys. 5.4. W poprzednich punktach położono duży nacisk na redukcję zapotrzebowania na pamięć (całkowanie numeryczne - zastosowanie kwadratur mieszanych, podział na warianty generacji ze względu na parametry fizyczne ośrodka; budowa macierzy w formacie COO - zastosowanie tego samego ułożenia elementów niezerowych zmniejsza liczbę macierzy przechowywanych w pamięci, CRS - wielokrotne użycie wcześniej zaalokowanych danych). Wyżej wymienione zabiegi optymalizacyjne prowadzą do zmniejszenia zapotrzebowania na pamięć, co ma duży wpływ na całościowe wykonanie generacji. Dane z rys. 5.4 potwierdzają, że gdy więcej pamięci jest dostępne na GPU, wtedy więcej czworościanów może być przetwarzanych wewnątrz pojedynczej iteracji generacji macierzy. Po drugie, w trakcie konwersji z formatu COO do formatu CRS eliminuje się więcej duplikatów. Konsekwencją tego jest mniejsza liczba elementów, które trzeba zsumować na CPU w algorytmie FSMA ( $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ ,  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ). Po trzecie, redukcja liczby iteracji w procesie generacji obniża ilość zmiennych przesyłanych pomiędzy GPU i CPU. W rezultacie porównując czas generacji macierzy dla tensorów zespolonych na akceleratorze Tesla K40c w procesie składającym się z 32 iteracji (2 GB użytych w pojedynczej iteracji) i w procesie składającym się z 6 iteracji (10 GB użytych w pojedynczej iteracji) można zauważyć, że uzyskano ok. 10% zysk. Dla problemów bezstratnych (skalary rzeczywiste, tensory rzeczywiste) zysk ten był nawet większy i wyniósł 18%.

W Tab. 5.1 przedstawiono miarę wydajności iteracyjnego procesu generacji macierzy sztywności i bezwładności, którą zdefiniowano jako liczbę czworościanów przetwarzanych w trakcie sekundy. Zauważyć można, iż względem wariantu podstawowego liczba przetwarzanych czworościanów na GPU zmniejszyła się o 6%, 7%, 22%, 36%, odpowiednio dla ośrodka reprezentowanego przez skalary rzeczywiste, tensory rzeczywiste, skalary zespolone i tensory zespolone. Jest to spowodowane tym, iż w wariacie itera-

TABELA 5.1: Liczba czworościanów przetwarzanych w trakcie pojedynczej sekundy w iteracyjnym procesie generacji macierzy sztywności i bezwładności (MatGen). Ośrodek reprezentowane przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. GPU (Tesla K40c), CPU (Intel Xeon Sandy Bridge E5-2687W). W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1).

	SR	TR	SZ	TZ
GPU (M \ sek.)	24455	24038	19411	16000
CPU (M \ sek.)	2499	2475	1834	1588

TABELA 5.2: Przyspieszenie iteracyjnego wariantu generacji macierzy sztywności i bezwładności (MatGen) na GPU względem implementacji na CPU. Parametry fizyczne ośrodka opisane przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. GPU (Tesla K40c), CPU (Intel Xeon Sandy Bridge E5-2687W). W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1)

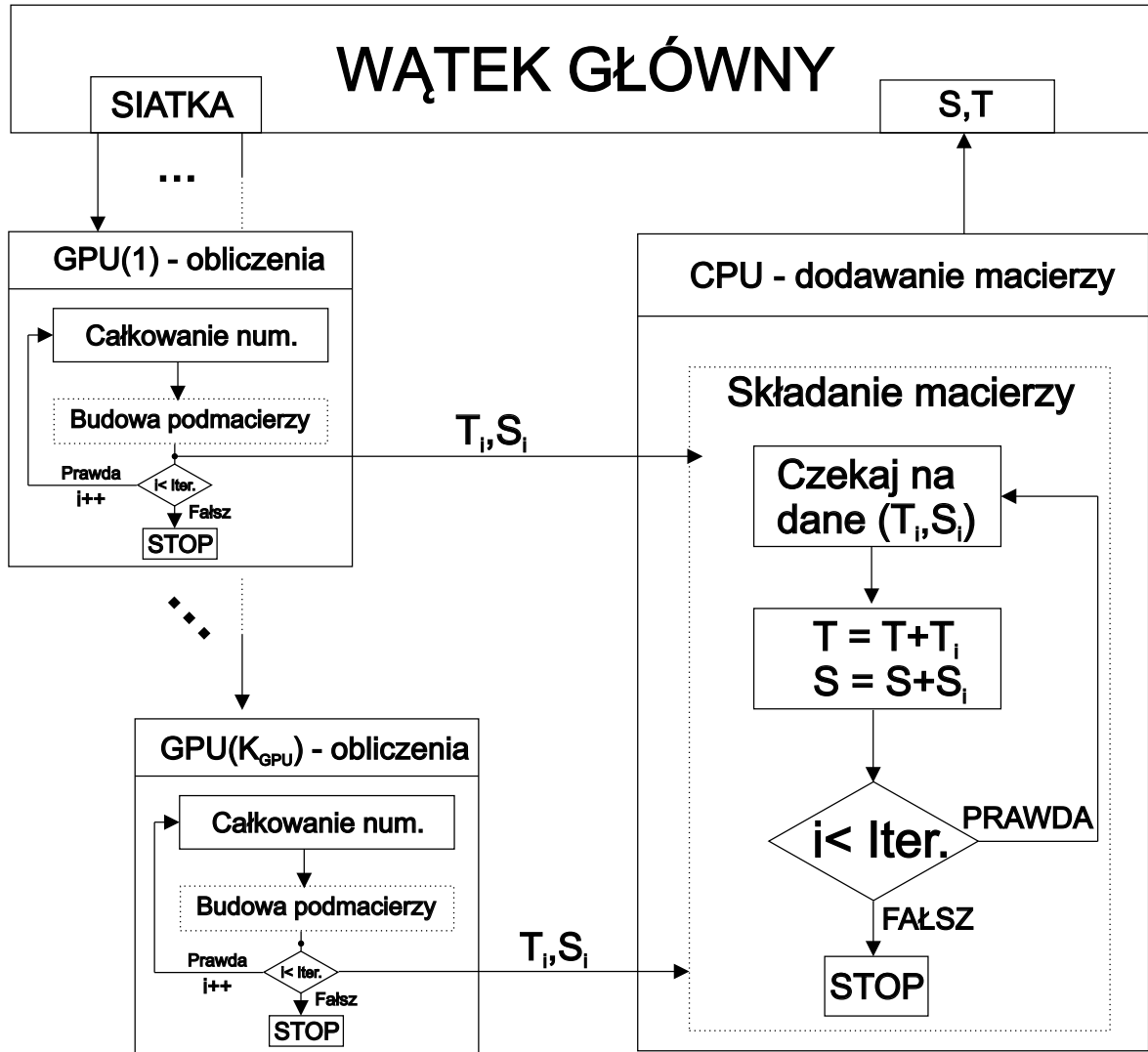
MatGen	SR	TR	SZ	TZ
GPU vs. CPU	9,8	9,7	10,6	10,1

cyjnym występują dodatkowe operacje przed i po docelowej generacji (preprocessing i postprocessing). Ponadto w trakcie generacji po każdej iteracji wymagane jest przesłanie macierzy  $\mathbf{S}_i$  i  $\mathbf{T}_i$  z GPU na CPU (ang. Device to Host, DtoH), który jest szczególnie kosztowny dla macierzy zespolonych. Ostatnim czynnikiem jest czas FSMA po ostatniej iteracji generacji, którego nie da się „ukryć”, gdyż wtedy GPU nie wykonuje już żadnych obliczeń (rys. 5.2). Wszystkie opisane powyżej zależności mają wpływ na redukcję przyspieszenia (zwłaszcza dla macierzy zespolonych) wariantu iteracyjnej generacji macierzy względem wariantu podstawowego (Tab. 5.2). Podsumowując, wariant iteracyjny pozwala na ok. 10-krotne skrócenie czasu generacji macierzy sztywności i bezwładności o rozmiarze ograniczonym wyłącznie pamięcią RAM stacji roboczej.

## 5.1 Iteracyjny wariant konstrukcji macierzy globalnych dla kilku akceleratorów graficznych

Iteracyjny wariant generacji macierzy zakładał użycie jednego akceleratora graficznego. Kolejną możliwą strategią zrównoleglenia jest wykorzystanie kilku akceleratorów graficznych w procesie generacji macierzy sztywności  $\mathbf{S}_i$  i bezwładności  $\mathbf{T}_i$ , celem poprawy wydajności obliczeniowej (rys. 5.5) [89].

Przeanalizowano dwie koncepcje rozdziału obliczeń na wiele akceleratorów graficznych. Pierwsza, z nich zakładała wykorzystanie kilku akceleratorów do wykonania poszczególnych faz (całkowanie numeryczne, składanie macierzy). Uznano, iż taki wariant wymagałby komunikacji między akceleratorami, co wprowadziłoby dodatkowe opóźnienia. Druga koncepcja wykorzystuje podejście przyjęte w iteracyjnym wariantcie generacji macierzy, w którym przetwarzanie podzbiorów elementów skończonych (czworościanów) w różnych iteracjach jest od siebie niezależne. Z tego powodu, porcje czworościanów



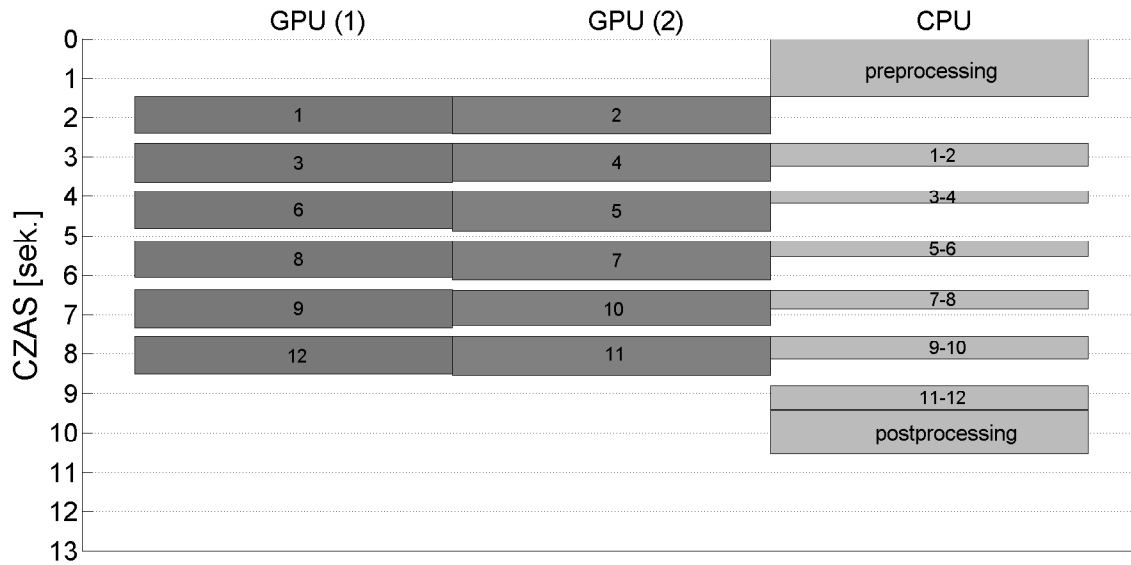
RYSUNEK 5.5: Iteracyjny wariant generacji macierzy bezwładności i sztywności w metodzie elementów skończonych, w którym obliczenia wykonywane są przez  $K_{GPU}$  akceleratorów graficznych. W procesie generacji pracuje  $T$  wątków na CPU (r. (5.1),  $T_{Master} = 1$ ,  $T_{GPU} = K_{GPU}$ ,  $T_{add} = 1$ ,  $T_{Mt}$  - zależy od implementacji dodawania macierzy rzadkich na CPU).

mogą być przetwarzane przez różne akceleratory graficzne niezależnie (=równolegle) (rys. 5.5). W tym podejściu, liczba wątków, które wywołują obliczenia na  $K_{GPU}$  różnych akceleratorach równa się liczbie dostępnych w danej stacji roboczej procesorów graficznych.

Warto zauważyć, iż „ukrycie” obliczeń na CPU (związanych z sumowaniem macierzy rzadkich) jest również możliwe w wariantcie generacji macierzy na wielu akceleratorach graficznych. Jednakże warunek, który gwarantuje, że obliczenia wykonane na CPU wykonywane są równolegle z obliczeniami na  $K_{GPU}$  kartach graficznych jest dużo bardziej restrykcyjny r. (5.3). Jest to spowodowane faktem, iż w pojedynczej iteracji CPU otrzymuje  $K_{GPU}$  (równe liczbie akceleratorów graficznych) macierzy  $T_i$  i  $S_i$ :

$$K_{GPU} \cdot t_{CPU}[s] < t_{GPU}[s] \quad (5.3)$$

gdzie:



RYSUNEK 5.6: Oś czasu iteracyjnego algorytmu generacji macierzy, w którym obliczenia wykonywane są przez dwa akceleratory graficzne. Po lewej stronie bloki reprezentują czas wykonania operacji całkowania numerycznego, budowy macierzy w formacie COO i konwersji do formatu CRS w pojedynczej iteracji (GPU: 2x Tesla K20c), przerwy między blokami po stronie GPU reprezentują czas przesłania podmacierzy  $\mathbf{S}_i$  i  $\mathbf{T}_i$  z GPU na CPU. Po prawej stronie bloki reprezentują czas wykonania algorytmu szybkiego sumowania macierzy (FSMA) na CPU (Intel Xeon E5-2620). W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1).

$K_{GPU}$  oznacza liczbę akceleratorów graficznych (oraz liczbę macierzy  $\mathbf{T}_i$  i  $\mathbf{S}_i$ , które muszą być pobrane i zsumowane na CPU),

$t_{CPU}$  oznacza czas potrzebny na wykonanie dodawania macierzy rzadkich  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ ,  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ ,

$t_{GPU}$  oznacza czas potrzebny do generacji podmacierzy  $\mathbf{T}_i$  i  $\mathbf{S}_i$  przez jeden akcelerator graficzny w jednej iteracji (założenie na potrzeby warunku: każdy z akceleratorów jest taki sam, tzn. ma te same zasoby pamięciowe i obliczeniowe, oraz każdy z akceleratorów przetwarza tę samą liczbę czworościanów).

Właściwość algorytmu FSMA pozwalająca na dodawanie macierzy z kilku iteracji jest kluczowa z punktu widzenia „ukrywania” obliczeń CPU, gdyż GPU mogą wykonywać obliczenia związane z generacją podmacierzy w kolejnych iteracjach bez przestojów wynikających z oczekiwania na skończenie obliczeń przez CPU (rys. 5.6).

W tabelach 5.3-5.4 przedstawiono porównanie czasów i przyspieszenie wykonania generacji macierzy w wariancie iteracyjnym na pojedynczym akceleratorze oraz z wykorzystaniem dwóch akceleratorów Tesla K20c<sup>2</sup>. Zastosowanie dwóch akceleratorów pozwoliło na ok. 1,6-1,7-krotne skrócenie czasu obliczeń w pętli głównej (całkowanie numeryczne - NI, budowa macierzy w formacie COO, konwersja do formatu CRS z eliminacją duplikatów, przesłanie macierzy z GPU do CPU - DtoH (ang. Device to Host),

<sup>2</sup>Stacja robocza udostępniona autorowi w trakcie stażu pod kierunkiem profesora M. Okoniewskiego w ramach projektu NCN Etiuda (Uniwersytet w Calgary, Kanada)



TABELA 5.3: Iteracyjny wariant generacji macierzy na jednym i dwóch akceleratorach graficznych (Tesla K20c) - czas w sekundach i udział procentowy głównych etapów. MatGen - preprocessing: alokacja danych na CPU i GPU, MatGen - pętla: to sumaryczny czas wykonania obliczeń całkowania numerycznego, budowania macierzy w formacie COO i konwersji do formatu CRS. MatGen - postprocessing: kopiowanie macierzy z pamięci niestronicowanej (ang. page-locked memory) do pamięci stronicowanej (ang. pageable memory), zwalnianie zasobów pamięciowych. Parametry fizyczne ośrodka opisane przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1).

1x K20c	SR	TR	SZ	TZ
MatGen - preprocessing	0,8 (5%)	0,8 (5%)	1,0 (5%)	0,9 (4%)
MatGen - pętla	13,6 (87%)	13,7 (87%)	16,8 (86%)	19,3 (89%)
MatGen - postprocessing	1,1 (7%)	1,1 (7%)	1,7 (9%)	1,6 (7%)
MatGen - całość	15,5 (100%)	15,6 (100%)	19,4 (100%)	21,8 (100%)
2x K20c	SR	TR	SZ	TZ
MatGen - preprocessing	1,5 (14%)	1,5 (14%)	1,9 (14%)	1,9 (12%)
MatGen - pętla	8,0 (76%)	8,1 (76%)	10,5 (75%)	11,7 (77%)
MatGen - postprocessing	1,1 (10%)	1,1 (10%)	1,6 (11%)	1,6 (10%)
MatGen - całość	10,5 (100%)	10,6 (100%)	14,0 (100%)	15,2 (100%)

TABELA 5.4: Skrócenie czasu wykonania iteracyjnego wariantu generacji macierzy dzięki zastosowaniu dwóch akceleratorów graficznych Tesla K20c. Parametry fizyczne ośrodka opisane przez: SR - skalar rzeczywisty, TR - tensor rzeczywisty, SZ - skalar zespolony, TZ - tensor zespolony. W procesie iteracyjnym generowane są macierze sztywności i bezwładności o rozmiarze  $N = 5079849$  i liczbie elementów niezerowych  $NNZ = 407716515$  (Tab. 4.1).

2x GPU vs. 1x GPU	SR	TR	SZ	TZ
MatGen - pętla	1,70	1,71	1,59	1,64
MatGen - całość	1,47	1,48	1,39	1,43

szybkie sumowanie macierzy rzadkich - FSMA). Całościowe skrócenie czasu generacji jest mniejsze (ok. 1,4-1,5 krotne), gdyż czas przetwarzania wstępnego (preprocessingu) w wariancie na dwóch akceleratorach jest ok. dwukrotnie dłuższy. Powyższe rezultaty dowodzą, że obliczenia numeryczne wykonywane na GPU trwają już ta tyle krótko, że ważną rolę odgrywają operacje wspomagające generację (przetwarzanie wstępne (preprocessing) i końcowe (postprocessing)).

## 5.2 Podsumowanie zrównoleglenia generacji globalnych macierzy sztywności i bezwładności

W tej sekcji podsumowane zostały właściwości iteracyjnego algorytmu generacji macierzy bezwładności i sztywności z punktu widzenia zrównoleglenia obliczeń. Po pierwsze



generacja podmacierzy ( $\mathbf{S}_i$ ,  $\mathbf{T}_i$ ) może odbywać się z wykorzystaniem wielu akceleratorów graficznych, a w każdej iteracji operacje w poszczególnych etapach zostały zrównoleglone w następujący sposób:

(a) Całkowanie numeryczne (p. 4.1.1):

- w każdej iteracji generacji macierzy podzbiór czworościanów jest przetwarzany w taki sposób, iż porcja czworościanów (256) jest przetwarzana **zrównoległe** (na poziomie bloków wątków),
- dla każdego czworościanu obliczenia związane z kwadraturą Gaussa są **zrównoleglone** (na poziomie bloków wątków),
- operacje na macierzach gęstych (mnożenie macierzy, dodawanie macierzy) w każdym punkcie kwadratury są **zrównoleglone** (na poziomie wątków),
- współbieżne strumienie (ang. concurrent streams, Hyper-Q) są w zależności od ośrodka przyporządkowane do oddzielnych wariantów całkowania numerycznego

(b) Konstrukcja macierzy globalnych (p. 4.1.2):

- COO - **zrównoleglona** budowa (składanie) macierzy w formacie COO,
- CRS - zrównoleglona konwersja między formatami COO i CRS z eliminacją duplikatów.

## Rozdział 6

# Metody rozwiązywania układów równań

W symulacjach zagadnień elektromagnetycznych w niniejszej rozprawie, w których parametry ośrodków nie są funkcjami częstotliwości (co ma na przykład miejsce gdy występują absorpcyjne warunki brzegowe w postaci PML (ang. perfect matched layer)) i siatka elementów nie ulega modyfikacji (jak to ma miejsce w przypadku zagadnienia optymalizacyjnego z modyfikacją kształtu obwodu), generacja macierzy sztywności ( $\mathbf{S}$ ) i bezwładności ( $\mathbf{T}$ ) wykonywana jest raz. Następnie, dla każdej z częstotliwości  $f$  z pasma w jakim analizowany jest problem elektromagnetyczny wyznaczana jest globalna macierz współczynników  $\mathbf{A}$ :

$$\mathbf{A} = \mathbf{S} - k_0^2 \mathbf{T} \quad (6.1)$$

i rozwiązywany jest układ równań, który można przedstawić jako:

$$\mathbf{A}\mathbf{X} = \mathbf{B} \quad (6.2)$$

gdzie:

$k_0 = \frac{2\pi \cdot f}{c}$  - liczba falowa, której wartość zależy od częstotliwości ( $f$ ) i prędkości światła ( $c$ ),

$\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_{Q_r}]$  - macierz reprezentująca pobudzenie,

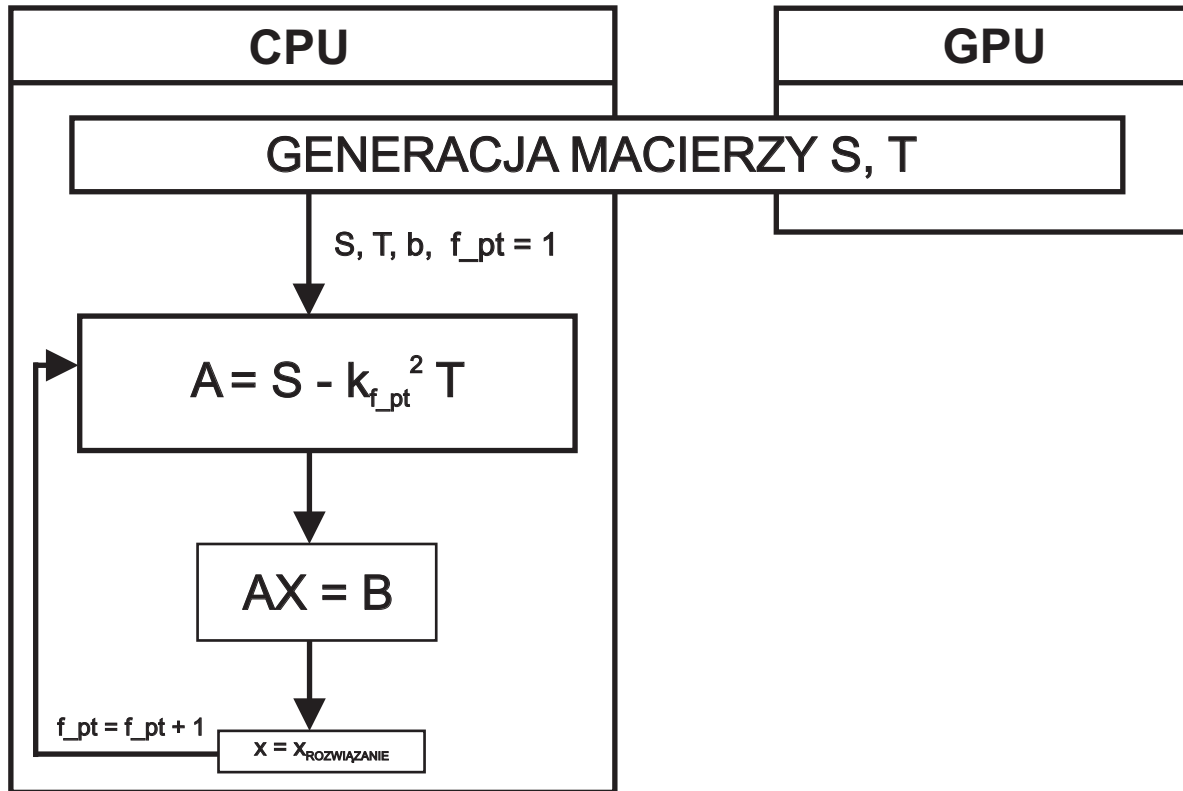
$\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_{Q_r}]$  - macierz poszukiwanych amplitud funkcji bazowych, które opisują falę elektromagnetyczną,

$Q_r$  jest sumą liczby rodzajów użytych w każdym z wrót (patrz. p. 2).

W poniższych podpunktach przeanalizowano koszt czasowy i pamięciowy dwóch wariantów rozwiązania układu równań z wykorzystaniem metod bezpośrednich i iteracyjnych.

### 6.1 Metoda bezpośredniego rozwiązania układu równań

Na rysunku 6.1 przedstawiono schemat analizy zagadnień elektromagnetycznych w paśmie częstotliwości, w których generacja układu równań odbywa się przy użyciu implementacji przedstawionych w rozdziałach 4-5, a układ równań rozwiązywany jest na CPU. Dla zadanej częstotliwości obliczana jest globalna macierz współczynników  $\mathbf{A}$  i następnie uruchamiana jest procedura rozwiązywania układów równań. W literaturze znaleźć można biblioteki, które pozwalają na bezpośrednie rozwiązanie układu równań



RYSUNEK 6.1: Schemat symulacji zagadnienia elektromagnetycznego z wykorzystaniem metody elementów skończonych w paśmie dla  $f_m$  częstotliwości ( $f_{pt}$  to indeks częstotliwości  $f$  i  $f_{pt}=\{1, \dots, f_m\}$ ). Generacja układu równań wykonana na GPU i CPU wg. implementacji z rozdziałów 4-5, układ równań liniowych rozwiązywany na CPU.

liniowych na CPU (Intel MKL Pardiso [43], SuperLU [96], TAUCS [97], MUMPS [98]) i na GPU (SPRAL [41], cuSOLVER [42]). Rozwiązanie układu równań składa się z dwóch etapów: faktoryzacji macierzy oraz docelowego rozwiązania układu równań. W rozprawie zdecydowano się użyć bibliotekę Intel MKL Pardiso, gdyż pozwala ona na zrównoleżenie obliczeń we wszystkich etapach rozwiązania układu równań. W przypadku wykorzystania metody bezpośredniej z wykorzystaniem biblioteki Intel MKL Pardiso macierz  $\mathbf{A}$  poddawana jest faktoryzacji symbolicznej i numerycznej, a następnie wykonywane jest docelowe rozwiązanie układu  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_{Q_r}]$ , odpowiednio dla pobudzeń  $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_{Q_r}]$ .

W Tab. 6.1 przedstawiono czas wykonania faktoryzacji (symbolicznej i numerycznej) oraz docelowego rozwiązania układu równań metodą bezpośrednią (Intel MKL Composer XE 2013, v. 11.0.3, pakiet 2013.3.163). Zauważyć można, iż czas faktoryzacji jest o dwa rzędy większy niż czas docelowego rozwiązania. Wraz ze wzrostem rozmiaru problemu czas faktoryzacji numerycznej jest dominujący. Ma to duże znaczenie w przypadku symulacji szerokopasmowej urządzenia mikrofalowego z wykorzystaniem MES, gdyż dla każdej częstotliwości zmieniają się wartości macierzy  $\mathbf{A}$  i przed rozwiązaniem układu równań należy wykonać faktoryzację numeryczną.

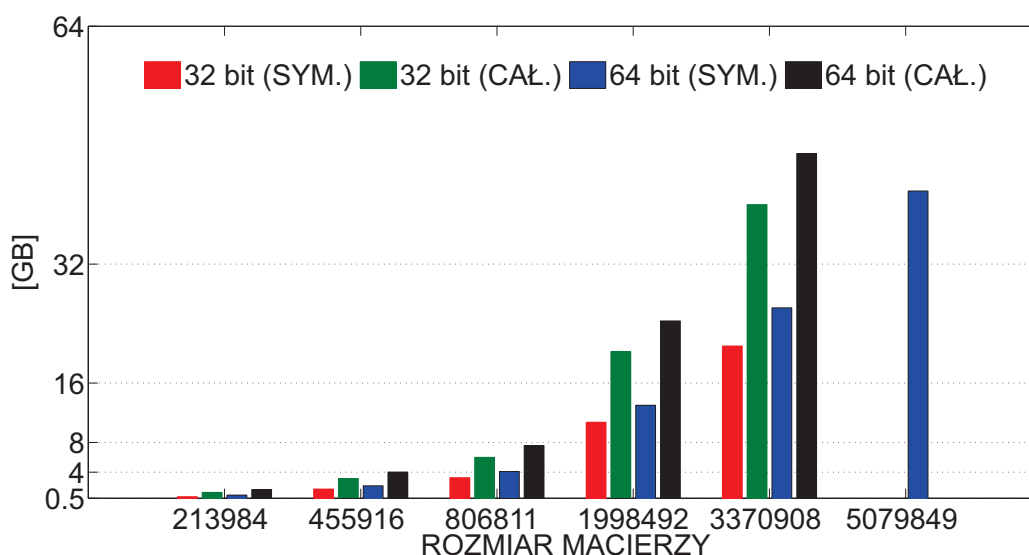
Czas faktoryzacji jest jednym z dwóch czynników ograniczających stosowanie metody bezpośredniej. Drugim - i przy obecnym stanie techniki istotniejszym czynnikiem jest zapotrzebowanie na pamięć. Liczba elementów niezerowych w czynnikach powstających w wyniku faktoryzacji jest wielokrotnie większa niż w oryginalnej macierzy (Tab. 6.2, rys. 6.2). Zastosowany format przechowywania czynników na CPU przez firmę Intel dla

TABELA 6.1: Czas wykonania etapów bezpośredniego rozwiązania układu równań w bibliotece Intel MKL Pardiso (faktoryzacja symboliczna, faktoryzacja numeryczna i docelowe rozwiązanie układu równań) [43]. W testach użyto funkcji PARDISO\_64 - indeksy przechowywane w pamięci jako zmienne 64 bitowe. (CAŁ.) - macierz  $A$  przechowywana jako całościowa, (SYM.) - w pamięci przechowywana jest górna połówka macierzy  $A$ . CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków).

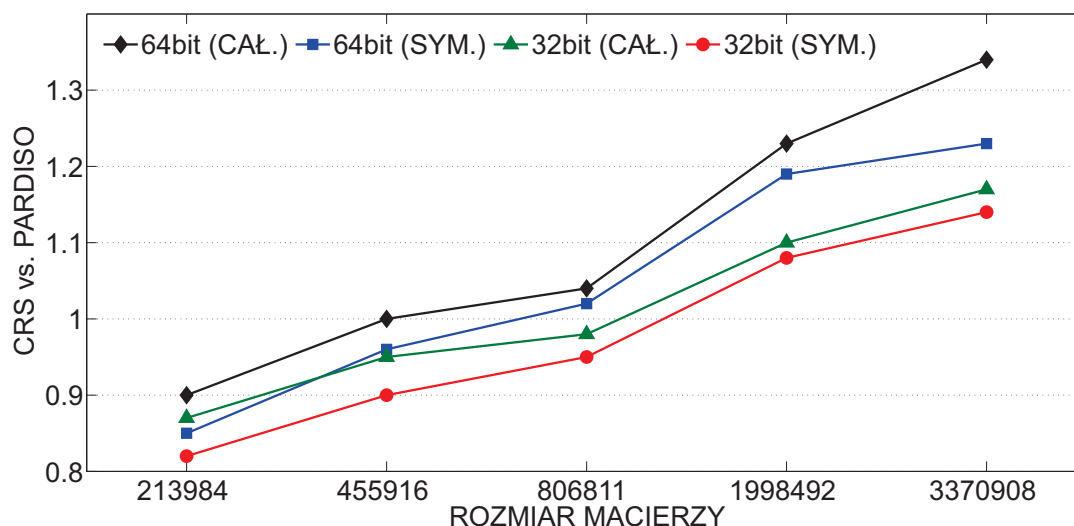
N	Fakt. symb. (CAŁ.) [s]	Fakt. num. (CAŁ.) [s]	Rozwiązanie (CAŁ.) [s]	Fakt. symb. (SYM.) [s]	Fakt. num. (SYM.) [s]	Rozwiązanie (SYM.) [s]
213 984	2,0	0,5	0,029	1,6	0,3	0,025
455 916	4,6	2,1	0,073	3,6	1,0	0,067
806 811	8,8	4,3	0,147	6,7	2,1	0,149
1 998 492	25,2	30,6	0,513	19,1	13,7	0,493
3 370 908	50,5	136,1	1,885	34,2	52,9	1,043
5 079 849	-	-	-	60,9	166,8	3,554

TABELA 6.2: Zestawienie zapotrzebowania na pamięć macierzy  $A$  i faktoryzacji macierzy  $A$  w celu otrzymania macierzy  $L$  i  $U$  potrzebnych do bezpośredniego rozwiązywania układu równań z wykorzystaniem pakietu PARDISO\_64 z biblioteki Intel MKL [43] (PARDISO\_64 - indeksy przechowywane jako zmienne 64 bitowe).

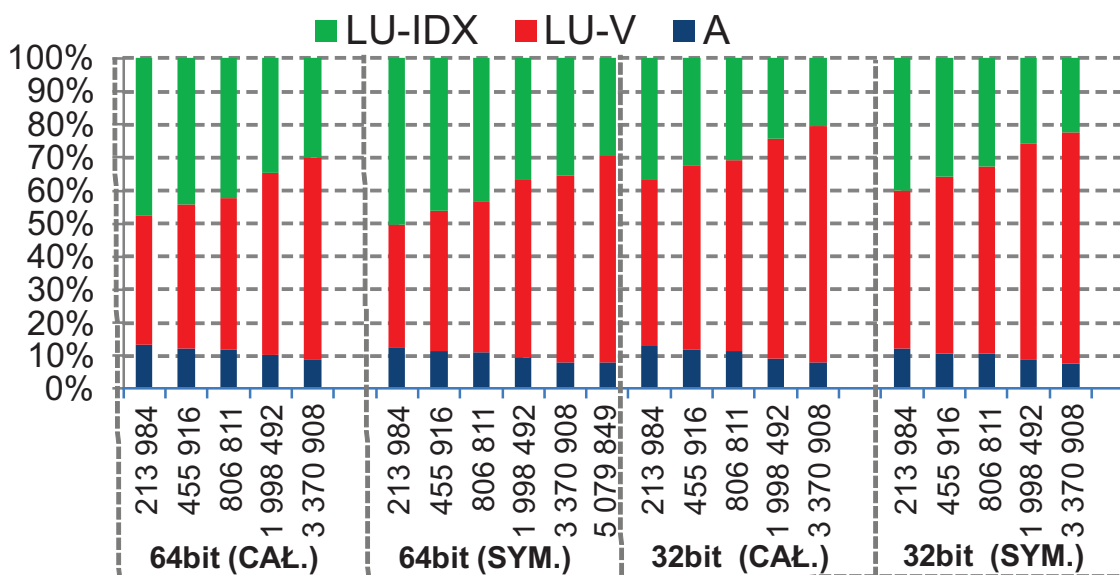
Macierz N	PARDISO_64 (CAŁ.)			PARDISO_64 (SYM.)		
	A [GB]	Fakt. (L,U) [GB]	Fakt. /A	A [GB]	Fakt. (L,U) [GB]	Fakt. /A
213 984	0,2	1,4	6,4	0,1	0,8	7,0
455 916	0,4	3,5	9,6	0,2	1,9	7,7
806 811	0,7	6,7	9,9	0,5	3,6	7,9
1 998 492	1,9	21,9	11,8	1,3	11,8	9,4
3 370 908	3,1	42,7	13,6	2,1	24,0	11,3
5 079 849	4,9	-	-	3,3	38,5	11,6



RYSUNEK 6.2: Zapotrzebowanie na pamięć Intel MKL Pardiso. Macierz  $A$  i faktory  $L$  oraz  $U$  przechowywane w dedykowanym formacie pakietu Intel MKL Pardiso.



RYSUNEK 6.3: Porównanie zapotrzebowania na pamięć formatu przechowywania danych w Intel MKL Pardiso z hipotetycznym zapotrzebowaniem na pamięć gdyby faktory przechowywać w formacie CRS.



RYSUNEK 6.4: Procentowy udział zapotrzebowania na pamięć: **A** - macierz główna, **LU-V** - udział w pamięci wartości zmiennoprzecinkowych faktorów **L** i **U**, **LU-IDX** - udział w pamięci indeksów. Oznaczenia: **64bit** - indeksy przechowywane jako zmienne 64-bitowe (PARDISO\_64), **32bit** - indeksy przechowywane jako zmienne 32-bitowe (PARDISO); **(CAŁ.)** - macierz **A** przechowywana jako całościowa, **(SYM.)** - w pamięci przechowywana jest górna połówka macierzy **A**.

większych macierzy jest wydajniejszy pamięciowo niż gdyby faktory były przechowywane w formacie CRS (rys. 6.3). Dodatkowo na rys. 6.4 przedstawiono procentowy udział zajętości pamięci przez macierz główną **A**, wartości niezerowe faktorów (**LU-V**) i indeksy (**LU-IDX**). Wraz ze wzrostem rozmiaru problemu widać znaczący wzrost liczby elementów niezerowych faktorów, a udział indeksów maleje. Wskazuje to, że wartości niezerowe są przechowywane w postaci gęstych bloków. Na osi rzędnych rys. 6.2

dane zaprezentowano w skali logarytmicznej, w celu pokazania jakie problemy można rozwiązać mając do dyspozycji stację roboczą o rozmiarze pamięci RAM: 4GB, 8GB, 16GB, 32GB i 64GB. Dla przykładu: faktoryzacji macierzy o rozmiarze  $N = 3370908$  nie można wykonać na stacji roboczej o rozmiarze pamięci RAM 16GB<sup>1</sup>. Biblioteka Intel MKL Pardiso dostarcza dedykowane procedury pozwalające na rozwiązanie problemów, w których macierz współczynników jest symetryczna. W takim przypadku w pamięci przechowywana jest macierz trójkątna górna w formacie CRS. Wykorzystanie symetrii pozwala na rozwiązanie większych rozmiarów problemu gdyż zapotrzebowanie na pamięć w etapie faktoryzacji w przypadku macierzy symetrycznej jest ok. 1,8 razy mniejsze niż w faktoryzacji macierzy całościowej. Ponadto czas faktoryzacji macierzy trójkątnej górnej jest krótszy (dla macierzy o rozmiarze  $N = 3370908$  faktoryzacja symboliczna i numeryczna jest odpowiednio 1,5 i 2,6 krotnie szybsza w przypadku zastosowania procedur uwzględniających symetrię macierzy).

Funkcja PARDISO, w której indeksy przechowywane są jako zmienne 32 bitowe, może być użyta dla układów których liczba elementów niezerowych faktorów  $\mathbf{L}$  i  $\mathbf{U}$  nie przekracza wartości  $2^{31} - 1$ . W przypadku problemów testowych z Tab. 4.1, dla układu równań z macierzą o rozmiarze 5 079 849 powyższy warunek nie jest spełniony i układ ten musi być rozwiązany przy użyciu funkcji PARDISO\_64 (indeksy przechowywane jako zmienne 64 bitowe). Testy numeryczne wykazały, że faktoryzacja symboliczna dla funkcji PARDISO jest, w zależności od rozmiaru problemu, od 23% do 36% krótsza, niż w przypadku PARDISO\_64. Faktoryzacja numeryczna i docelowe rozwiązanie układu równań - które są wykonywane dla każdej częstotliwości w paśmie - są realizowane w podobnym czasie dla PARDISO i PARDISO\_64 i dlatego dla ujednolicenia dyskusji, redukcji liczby tabel i rysunków w poniższych rozważaniach przedstawiono wyniki wyłącznie dla wywołania funkcji PARDISO\_64.

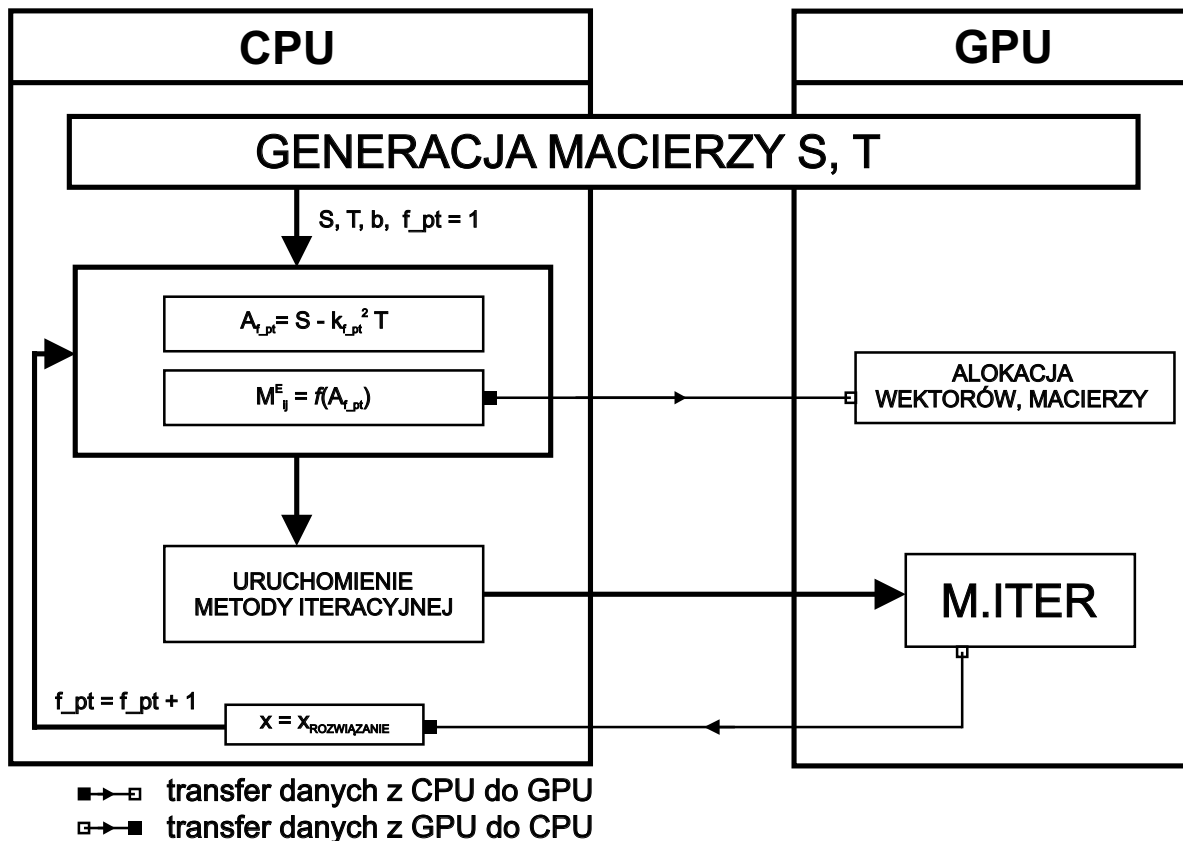
Dostępne implementacje bezpośredniego rozwiązywania układów równań na GPU [41, 73] charakteryzują się bardzo dużym zapotrzebowaniem na pamięć co dyskwalifikuje ich użycie dla macierzy MES generowanych w niniejszej pracy. W p. 6.2.3 przedstawiono testy wydajności obliczeniowej i pamięciowej małych macierzy, które potwierdzają tę konkluzję.

Podsumowując, rozwiązanie układu równań metodą bezpośrednią jest obciążone dużym kosztem (szczególnie pamięciowym) na etapie faktoryzacji macierzy, co przy obecnym stanie techniki jest barierą nie pozwalającą na rozwiązanie dużych macierzy generowanych w MES. Alternatywą dla metod bezpośrednich są metody iteracyjnego rozwiązywania układów równań, które wymagają przechowywania w pamięci macierzy rzadkiej i wektorów pomocniczych.

## 6.2 Metoda iteracyjnego rozwiązania układu równań

W punkcie tym przeprowadzono dyskusję nt. implementacji i wydajności metod iteracyjnego rozwiązywania układów równań w MES w oparciu o algorytmy bazujące na

<sup>1</sup>Dane w tabelach i rysunkach pochodzą z testów wykonanych dla opcji, w której Intel MKL Pardiso korzysta wyłącznie z pamięci RAM (*In-Core*). Biblioteka Intel MKL Pardiso umożliwia użycie pamięci dysku twardego (*Out-of-core*), ale jest to bardzo kosztowne czasowo. Dla przykładu sumaryczny czas faktoryzacji symbolicznej i numerycznej macierzy o rozmiarze  $N = 213984$  wynosi ok. 2,5 i 11,5 sekund, gdy odpowiednio korzystano wyłącznie z pamięci RAM (*In-core*) i korzystano z pamięci dysku twardego (*Out-of-core*).



RYSUNEK 6.5: Schemat symulacji zagadnienia elektromagnetycznego z wykorzystaniem metody elementów skończonych w paśmie dla  $f_m$  częstotliwości ( $f_{pt}$  to indeks częstotliwości  $f$  i  $f_{pt} = \{1, \dots, f_m\}$ ). Generacja układu równań wykonana na GPU i CPU wg. implementacji z rozdziałów 4-5, układ równań liniowych rozwiązywany na GPU (blok **M.ITER**).

podprzestrzeni Kryłowa, które charakteryzują się znacznie mniejszym kosztem pamięciowym niż metody bezpośrednie. W p. 2.3.1 wskazano, że satysfakcjonującą zbieżność rozwiązania można uzyskać dzięki zastosowaniu wielopoziomowego operatora ściskającego. Dodatkowe skrócenie czasu rozwiązania można uzyskać przy użyciu akceleratora graficznego do wykonania operacji na macierzach i wektorach, po warunkiem, że są to operacje, które można efektywnie zrównoleglić.

Na rysunku 6.5 przedstawiono schemat analizy zagadnień elektromagnetycznych w paśmie częstotliwości, w którym generacja układu równań odbywa się przy użyciu algorytmów przedstawionych w rozdziałach 4-5, a układ równań rozwiązywany jest przy użyciu metody iteracyjnej z wykorzystaniem akceleratora graficznego. Dla zadanej częstotliwości obliczana jest globalna macierz współczynników  $\mathbf{A}$ . Następnie macierz jest przesyłana z pamięci CPU do pamięci GPU. Z hosta (CPU) kilkakrotnie (dla kolejnych prawych stron) uruchamiane są procedury mające na celu rozwiązanie układu równań (blok **M.ITER** na rys. 6.5). W procesie iteracyjnym aproksymowane jest rozwiązanie układu równań  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_{Q_r}]$ , odpowiednio dla pobudzeń  $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_{Q_r}]$ . Po uzyskaniu zakładanej zbieżności wektory rozwiązania  $\mathbf{x}_i$  przesyłane są z pamięci GPU do pamięci CPU. Następnie wykonywane jest przetwarzanie końcowe, którego celem jest wyznaczenie parametrów rozproszenia na podstawie r. (2.21).

W kolejnych podpunktach opisano strategie jakich użyto do implementacji bloku **M.ITER** z rys. 6.5. Analizując cechy metody gradientów sprzężonych z wielopoziomo-



wym operatorem ściskającym (p. 2.3) zauważono, iż kluczową operacją jest mnożenie macierzy rzadkiej przez wektor (ang. Sparse Matrix Vector Product, SpMV, *matvec*). Z tego powodu, implementacji tej operacji na akceleratorze graficznym poświęcono oddzielny p. 6.2.1, a następnie opisano format reprezentacji macierzy rzadkiej zaproponowany w niniejszej rozprawie (p. 6.2.2). Z racji tego, że operator ściskający jaki wykorzystano w rozprawie na najniższym poziomie zawiera rozwiązanie układu równań liniowych to w p. 6.2.3 przeprowadzono szczegółowe rozważania nt. efektywnej implementacji tego etapu.

### 6.2.1 Mnożenie macierzy rzadkiej przez wektor na akceleratorze graficznym

Operacja mnożenia macierzy przez wektor *matvec* ( $y = Ax$ ) jest podstawową operacją wykonywaną w iteracyjnych technikach rozwiązywania układów liniowych. Sposób implementacji tej operacji zależy w głównej mierze od własności macierzy, tzn. tego czy jest ona gęsta czy rzadka, symetryczna czy niesymetryczna. Na akceleratorze graficznym w implementacji mnożenia macierzy gęstej przez wektor najłatwiej wykorzystać funkcję *cublasSgemv* z biblioteki CUBLAS. To podejście jest jednak nieefektywne, gdy mamy do czynienia z macierzami rzadkimi, gdyż wykonywalibyśmy dużo nadmiarowych operacji mnożenia elementów zerowych macierzy  $\mathbf{A}$  z elementami wektora  $\mathbf{x}$  - co wydłuża czas obliczeń. Ponadto musimy liczyć się z faktem, iż mamy do dyspozycji ograniczony zasób pamięci. W pamięci o rozmiarze 5GB możemy w najlepszym przypadku zaalokować macierz gęstą o 25000 wierszy. Z tych powodów macierz rzadką poddaje się kompresji. Poniżej scharakteryzowane zostały najczęściej używane kompresje macierzy rzadkich. W kontekście użyteczności kompresji na GPU pod uwagę brane są dwa czynniki:

- efektywność (wydajność) obliczeniowa - mierzona w GFLOPS-ach (6.3). Każdy z formatów determinuje specyficzny sposób adresowania pamięci, co ma kluczowe znaczenie dla wykonania obliczeń na GPU. W wielu przypadkach z powodu niebezpośredniego dostępu do wartości niezerowych macierzy, GPU wykonuje więcej operacji na pamięci niż operacji zmiennoprzecinkowych, co wprowadza dodatkowe opóźnienia,

$$GFLOPS = \frac{\text{liczba operacji zmiennoprzecinkowych}}{\text{czas} \cdot 10^9} \quad (6.3)$$

- efektywność (wydajność) pamięciowa - określa ilość pamięci w bajtach potrzebnej do przechowania macierzy rzadkiej. Im mniej danych potrzebnych jest do opisu macierzy rzadkich tym efektywność jest większa.

**Najczęściej stosowane przy wykonywaniu operacji *matvec* na GPU formaty zapisu macierzy rzadkich to:**

**CRS** - kompresja CRS (ang. Compressed Row Storage) polega na przechowaniu informacji o macierzy rzadkiej  $\mathbf{A}$  w postaci trzech wektorów:  $\mathbf{V}$  - wartości niezerowe,  $\mathbf{J}$  - indeksy kolumn wartości niezerowych,  $\mathbf{Iptr}$  - indeks  $\mathbf{J}$  pierwszego elementu niezerowego w każdym wierszu (rys. 6.6) [4]. Format ten jest używany w bibliotece Intel MKL dedykowanej obliczeniom na wielordzeniowych procesorach CPU oraz na GPU w bibliotece CUSPARSE [92]. Najprostsze zrównoleglenie obliczeń dla tego formatu to aranżacja obliczeń tak, aby jeden wątek wykonywał obliczenia w jednym wierszu i obliczenia w

A	B						
	C	D	E				
F			G		H	I	J
		K					
	L		M	N			
		O	P				
				Q	R	S	T
	U	Y					Z

RYSUNEK 6.6: Reprezentacja macierzy rzadkiej w formacie CRS.

wierszach były wykonywane niezależnie. Takie przyporządkowanie wątków do wiersza przedstawiono na rys. 6.7. Z punktu widzenia efektywności pamięciowej format CRS jest najbardziej efektywny ze wszystkich formatów i zapotrzebowanie na pamięć wynosi:

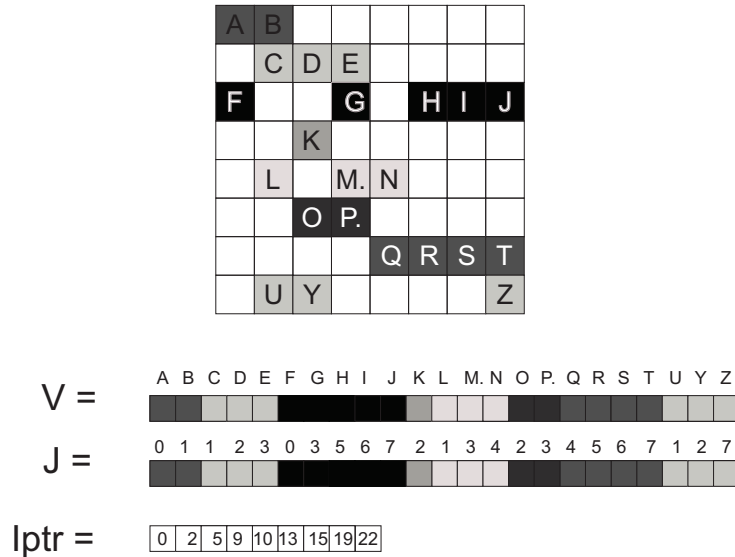
$$CRS [MB] = (NNZ \cdot 8 + (NNZ + N) \cdot 4) \cdot 2^{-20} \quad (6.4)$$

gdzie:  $NNZ$  oznacza liczbę elementów niezerowych,  $N$  oznacza liczbę wierszy macierzy rzadkiej, wektor elementów niezerowych przechowywane jako zmienne 64 bitowe, wektory indeksów kolumny i skompresowany wiersz przechowywane w pamięci jako zmienne 32 bitowe.

Niestety dla większości macierzy rzadkich w trakcie wykonywania operacji *matvec* nie jest spełniony warunek dostępu łącznego do pamięci GPU (Dodatek B.1), co ogranicza wykorzystanie tego formatu na GPU. W literaturze znaleźć można artykuły, w których autorzy proponują implementacje operacji mnożenia macierzy rzadkiej przez wektor na GPU z wykorzystaniem formatu CRS lub modyfikacją tego formatu [29, 34, 99–102]. Są to często procedury dedykowane właściwościom macierzy (np. dla macierzy generowanych w metodzie różnic skończonych można wydzielić podmacierze, w których liczba elementów niezerowych w wierszu jest stała i obliczenia w podmacierzach realizują oddzielne kernele, w których zapewniony jest dostęp łączny do elementów niezerowych i wektora [103]).

**CCS** - W kompresji macierzy przy użyciu formatu CCS (ang. Compressed Column Storage) - podobnie jak w przypadku formatu CRS - macierz rzadka przechowywana jest w trzech wektorach. Występuje jednak znacząca różnica, gdyż wektor  $V$  tworzą wartości niezerowe pobierane z kolejnych kolumn macierzy  $A$ . Pozostałe dwa wektory to:  $I$  - indeksy wierszy wartości niezerowych,  $Jptr$  - wskazuje indeks  $I$  pierwszego elementu niezerowego każdej kolumny [4]. Gdy analogicznie do formatu CRS, w formacie CCS jeden wątek jest przypisany do jednego wiersza to występują konflikty przy zapisie danych do tego samego obszaru pamięci, które znacząco zmniejszają efektywność obliczeń na GPU. Obsługa tych konfliktów wymaga użycia operacji atomowych i z tego powodu format ten nie jest wykorzystywany w implementacjach na GPU.

**ELL, ELL-R** - celem zapewnienia dostępu łącznego do pamięci GPU wykorzystuje się format Ellpack (ELL) [31]. W formacie tym, informacja o elementach niezerowych przechowywana jest w dwóch wektorach:  $e11\_A$  - wektor elementów niezerowych i  $e11\_J$  - wektor indeksów kolumn wartości niezerowych. Z punktu widzenia wykonania operacji *matvec* na GPU, format ELL jest efektywniejszy niż CRS, gdyż gwarantuje łączny dostęp do pamięci. Niestety format ELL wprowadza znaczącą nadmiarowość w postaci obowiązku przechowywania w pamięci dodatkowych zer. Liczba nadmiarowych



RYSUNEK 6.7: Strategia przypisania wątków w operacji *matvec* dla macierzy w formacie CRS (jeden wątek przypisany do jednego wiersza).

zer zależy od różnicy między najdłuższym i najkrótszym wierszem w macierzy. Ponadto celem zagwarantowania dostępu łącznego liczba wierszy macierzy powinna być podzielna przez 16 (połowa *warpa*, czyli grupy wątków wykonywanych równolegle na GPU), co generuje dodatkowe wiersze z wartościami zerowymi. Ostatecznie, nadmiarowość można oszacować jako  $N' \cdot (N_{max} - N_{min}) - NNZ$ . Zapotrzebowanie na pamięć dla podwójnej precyzji:

$$ELL - R [MB] = ((N_{max} \cdot N') \cdot 8 + (N_{max} \cdot N' + N) \cdot 4) \cdot 2^{-20} \quad (6.5)$$

gdzie  $N_{max}$  oznacza liczbę elementów niezerowych w najdłuższym wierszu,  $N'$  oznacza nową liczbę wierszy która jest liczbą podzielną przez 16, wektor elementów niezerowych przechowywane jako zmienne 64 bitowe, wektory indeksów kolumny i skompresowany wiersz przechowywane w pamięci jako zmienne 32 bitowe.

Podobnie jak w standardowej implementacji operacji *matvec* opartej na CRS, również w implementacji *matvec* opartej na formacie ELL jeden wątek odpowiada za obliczenia wykonywane w pojedynczym wierszu. Celem wykonywania obliczeń wyłącznie na elementach niezerowych wprowadzono format ELL-R [30]. Modyfikacja polega na wprowadzeniu dodatkowego wektora `e11_r1`, przechowującego informacje o liczbie elementów niezerowych w każdym wierszu (rys. 6.8). Zabieg ten pozwolił otrzymać poprawę wydajności obliczeniowej z racji tego, że ograniczono nadmiarowe mnożenia przez wartości zerowe.

**Sliced ELLPACK** - celem redukcji nadmiarowych obliczeń i zer przechowywanych w pamięci wprowadzanej przez format ELL-R, Monakov w [104] zaproponował „pocięcie” (ang. *slicing*) macierzy i wykonanie kompresji ELL-R na otrzymanych w skutek podziału podmacierzach (rys. 6.9). W rezultacie otrzymano redukcję nadmiarowości, która w formacie Sliced ELL-R zależy od różnicy między najdłuższym i najkrótszym wierszem w podmacierzy, a nie jak to miało miejsce dla ELL-R - w całej macierzy.

**ELLR-T** - kolejnym formatem kompresji macierzy rzadkiej dedykowanym obliczeniom na GPU jest format ELLR-T [105]. Format ten wymaga przetworzenia wstępnego macierzy (ang. *preprocessing*), polegającego na permutacji elementów niezerowych (i odpowiadającym im indeksom kolumn), tak aby w trakcie wykonania operacji *matvec*

ell_A =		A	B	0	0	0
		C	D	E	0	0
		F	G	H	I	J
		K	0	0	0	0
		L	M	N	0	0
		O	P	0	0	0
		Q	R	S	T	0
		U	Y	Z	0	0
ell_J =		0	1	0	0	0
		1	2	3	0	0
		0	3	5	6	7
		2	0	0	0	0
		1	3	4	0	0
		2	3	0	0	0
		4	5	6	7	0
		1	2	7	0	0
		ell_rl =				
		2				
		3				
		5				
		1				
		3				
		2				
		4				
		3				

RYSUNEK 6.8: Reprezentacja macierzy rzadkiej w formacie ELL-R [3].

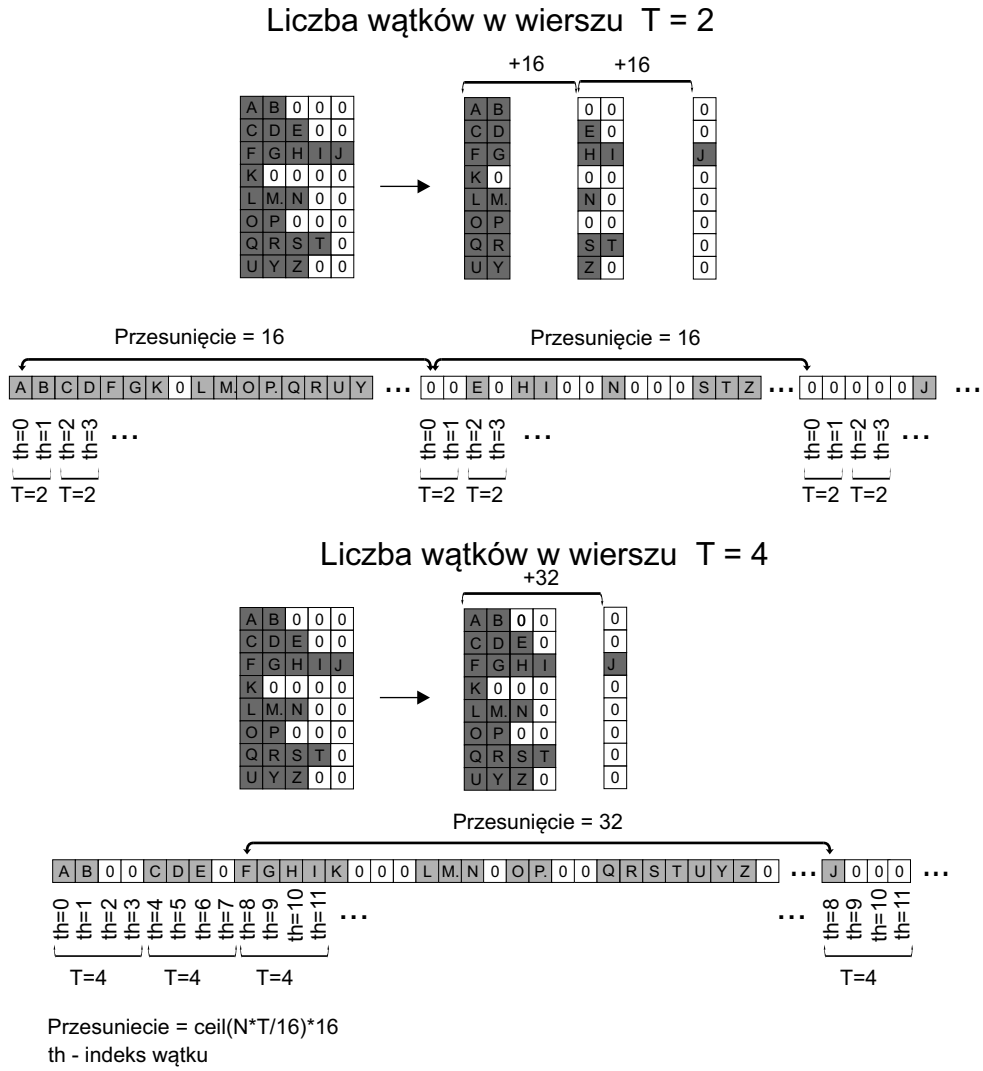
sell_A1 =	ELL-R	sell_rl1 = 2 sell_rl2 = 5 sell_rl3 = 3 sell_rl4 = 4
sell_A2 =	ELL-R	
sell_A3 =	ELL-R	
sell_A4 =	ELL-R	
sell_J1 =	ELL-R	
sell_J2 =	ELL-R	
sell_J3 =	ELL-R	
sell_J4 =	ELL-R	

RYSUNEK 6.9: Podział na podmacierze w formacie Sliced ELL-R.

zagwarantować dostęp łączny do wartości niezerowych przechowywanych w pamięci globalnej (rys. 6.10). Dodatkowo, preprocessing musi zapewnić, by każdy wiersz miał długość będącą wielokrotnością 16. Aby spełnić ten warunek, do wierszy dodawane są nadmiarowe zera, co sprawia, że podobnie jak ELL-R, format ELLR-T nie jest atrakcyjny z punktu widzenia efektywności pamięciowej. Zaletami formatu ELLR-T, jest zagwarantowany dostęp łączny oraz możliwość przypisania kilku wątków ( $T = 1, 2, 4, 8, 16, 32$ ) do jednego wiersza w trakcie wykonywania operacji *matvec*. Ostatecznie, dzięki specyficznemu adresowaniu, użyciu pamięci wspólnej i kilku wątkach operujących na wierszu, format ten pozwala na poprawę wydajności obliczeniowej względem ELL-R i Sliced ELL-R.

### 6.2.2 Sliced ELLR-T - nowy formatu zapisu macierzy rzadkiej

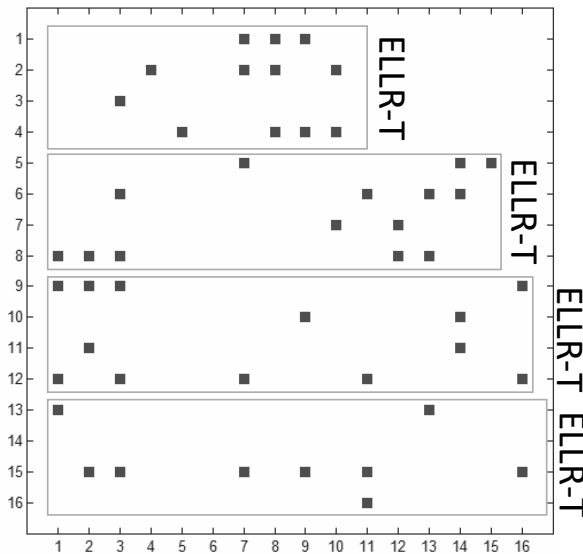
Na podstawie analizy dostępnych formatów, w niniejszej rozprawie zaproponowano format przechowywania macierzy rzadkiej (**Sliced ELLR-T**), który charakteryzuje się bardzo dobrą wydajnością obliczeniową, przy zapewnieniu dużej wydajności pamięciowej [106]. Po pierwsze, podobnie jak w formacie Sliced ELL-R, macierz została podzielona („pocięta”) na podmacierze celem redukcji liczby nadmiarowych zer (rys. 6.11), a

RYSUNEK 6.10: Dwa warianty zapisu macierzy rzadkiej w formacie ELLR-T ( $T = 2$  i  $T = 4$ ).

następnie na otrzymanych podmacierzach stosuje się preprocessing podobny do formatu ELLR-T. W efekcie kilka wątków jest zaangażowanych w obliczenia w pojedynczym wierszu macierzy ( $T = 1, 2, 4, 8, 16, 32$ ), występuje dostęp łączny do pamięci, co gwarantuje dobrą wydajność obliczeniową.

Z punktu widzenia wydajności obliczeniowej parametr  $T$ , liczba wątków przypisanych do pojedynczego wiersza, odgrywa kluczową rolę. Zauważono, iż dobór tego parametru uzależniony jest od struktury macierzy, a dokładniej średniej liczby elementu w wierszu. Na podstawie licznych testów wyznaczono zależność parametru  $T$  od średniej liczby elementów w wierszu co zostało przedstawione w [106]. W ogólności można powiedzieć, że wraz ze wzrostem średniej liczby elementów przypadającej na pojedynczy wiersz, rośnie liczba wątków  $T$  zaangażowanych w obliczenia.

Analogicznie jak dla formatów ELL, ELL-R, Sliced ELL-T, również na potrzeby formatu Sliced ELLR-T macierz musiała zostać poddana odpowiedniemu przetwarzaniu wstępnemu (ang. preprocessing). W przypadku formatu Sliced ELLR-T preprocessing ma na celu podział macierzy na „plastry”, zapewnić późniejszy dostęp łączny do pamięci na GPU oraz odpowiednie przypisanie wątków ( $T$ ) do poszczególnych grup elementów niezerowych w wierszu. Opracowano dwa warianty przygotowania macierzy, tzn. może być on wykonany na GPU lub na CPU. Pierwszy można wykorzystać dla



RYSUNEK 6.11: Realizacja formatu Sliced ELLR-T - macierz rzadka „pocięta” na podmacierze o  $S = 4$  liczbie wierszy, a każda podmacierz przechowywana w formacie ELLR-T.

macierzy o małym rozmiarze, gdy obydwie macierze oryginalna i spermutowana muszą zmieścić się w pamięci pojedynczego GPU. Dla dużych problemów, pamięć GPU jest maksymalnie obciążona i na potrzeby takiego przypadku preprocessing wykonywany jest na CPU z wykorzystaniem interfejsu OpenMP. Ponadto, aby poprawić wydajność pamięciową w trakcie przetwarzania wstępnego, wiersze macierzy mogą być dodatkowo uporządkowane w porządku od najmniej licznego do najbardziej licznego wiersza.

Kod dedykowany operacji mnożenia macierzy rzadkiej przez wektor na GPU z użyciem formatu Sliced ELLR-T, gdy liczba wątków przypisanych do wiersza  $T = 4$  i jeden blok wątków wykonuje obliczenia w jednym plastrze (slice) macierzy, został zaprezentowany na Wydruku C.3. W pierwszej kolejności określone są indeksy wątków w bloku ( $\mathbf{tx}$ ), globalny indeks wątku ( $\mathbf{txg}$ ) oraz indeks wiersza ( $\mathbf{n}$ ), na którym pracują wątki. Następnie wykonywana jest pętla, w której dla każdego wiersza cztery wątki obliczają niezależnie składniki wyniku ( $\mathbf{sub}$ ). Dzięki odpowiedniemu preprocessingowi danych dostęp do wektorów  $\mathbf{d\_ella}$  i  $\mathbf{d\_ja}$  jest łączny. Odczyt danych z wektora  $\mathbf{d\_x}$  nie jest łączny (wątki „skaczą” po wektorze  $\mathbf{d\_x}$ ) (WARIANT-I). Aby poprawić wydajność odczytu z wektora  $\mathbf{d\_x}$  przed wykonaniem operacji SpMV wektor  $\mathbf{d\_x}$  może być związany z szybką podręczną pamięcią tekstur (WARIANT-II), co zmniejsza opóźnienia przy dostępie do danych. W dalszej kolejności składniki wyniku są zapisywane do tablicy w pamięci wspólnej ( $\mathbf{array}$ ) i na podstawie operacji redukcji obliczane są elementy wektora wynikowego ( $\mathbf{d\_y}$ ).

Podsumowując format Sliced ELLR-T cechuje:

- dostęp łączny przy odczycie elementów niezerowych macierzy na GPU,
- zredukowana - względem innych formatów typu ELLPACK - liczba nadmiarowych elementów zerowych, które zapewniają łączny dostęp do elementów niezerowych macierzy,
- zrównoleglenie obliczeń przez przypisanie  $T$  wątków do wykonania obliczeń w wierszu macierzy,



- możliwość zapisu wektora prawej strony do pamięci tekstur celem skrócenia czasu dostępu do wartości wektora,
- zoptymalizowany preprocessing macierzy.

### 6.2.2.1 Wydajność formatu Sliced ELLR-T

Poniżej porównane zostały rezultaty otrzymane dla formatu Sliced ELLR-T w porównaniu do funkcji z biblioteki CUSPARSE (`cusparsedcsmv`, `cusparsedhybm`) dla wartości niezerowych macierzy przechowywanych w podwójnej precyzji. Funkcja `cusparsedcsmv` wykorzystuje format CRS. Funkcja `cusparsedhybm` wykorzystuje kombinację dwóch formatów COO i ELL. Wykonano testy dla dwóch wariantów uruchomienia: **AUTO** - automatyczny podział macierzy na podmacierze przechowywane w formacie COO i ELL, **MAX** - macierz przechowywana w formacie ELL. Na potrzeby formatu hybrydowego należy wykonać konwersję z formatu CRS do formatu hybrydowego (`cusparsedcsr2hyb`). W Tab. 6.3 przedstawiono wydajność obliczeniową operacji *matvec* wyrażoną w GFLOPS. Dla niektórych macierzy funkcja `cusparsedhybm` jest wydajniejsza niż `cusparsedcsmv`, ale wykonanie konwersji `cusparsedcsr2hyb` wymaga dla wariantu (AUTO) i (MAX) od 2 do 4 razy więcej pamięci niż przechowywanie macierzy podstawowej w formacie CRS, co dyskwalifikuje ten format dla wielu macierzy. Dla przykładu, macierz testowa o rozmiarze 1998492 zajmuje w pamięci ok. 1,8 GB. Konwersja do formatu hybrydowego wymagała dodatkowych 2 GB i 6 GB danych dla wariantów (AUTO) i (MAX). Dla porównania format Sliced ELLR-T wprowadza średnio 10% dodatkowych elementów niezerowych (przyp. gdy macierz została poddana permutacji w celu przegrupowania wierszy od najmniej do najbardziej licznych to liczba dodatkowych elementów jest mniejsza). Warto zaznaczyć, że wydajność obliczeniowa biblioteki CUSPARSE z każdą kolejną wersją znacząco się poprawiła. Obecnie Sliced ELLR-T pozwala na ok. 25% skrócenie czasu obliczeń operacji *matvec* względem CUSPARSE dla macierzy MES generowanych w niniejszej rozprawie.

Z racji tego, że wydajność obliczeniowa operacji *matvec* jest ograniczona przez transfer danych, dodatkowo w Tab. 6.4 otrzymane wartości GFLOPS zostały skonfrontowane z parametrem wyrażającym efektywną wydajności jaką można osiągnąć ze względu na przepustowość pamięci (ang. effective bandwidth bound performance). Użycie tej dodatkowej miary wydajności wpisuje się w postulaty z artykułu [34], by wydajność obliczeniową mierzoną w GFLOPS skonfrontować z maksymalną możliwą do uzyskania

TABELA 6.3: Wydajność obliczeniowa (GFLOPS) operacji *matvec* dla różnych formatów reprezentacji macierzy rzadkiej. Problem testowy to macierz podstawowa  $\mathbf{A}$  o rozmiarze 1998492 (Tab. 4.1) i jej symetryczne podmacierze  $\mathbf{M}_{ii}^E$ .

GFLOPS	<code>cusparsedcsmv</code>	<code>cusparsedhybm</code> (AUTO)	<code>cusparsedhybm</code> (MAX)	Sliced ELLR-T
$\mathbf{M}_{11}^E$	13,3	13,3	18,7	15,6
$\mathbf{M}_{22}^E$	14,4	16,1	17,2	14,0
$\mathbf{M}_{33}^E$	9,8	8,5	8,7	12,2
$\mathbf{A}$	11,6	10,5	10,6	12,4



TABELA 6.4: Wydajność obliczeniowa operacji *matvec* ( $\frac{GFLOPS}{P_{ebw}} \cdot 100\%$ ) dla różnych formatów reprezentacji macierzy rzadkiej. Problem testowy to macierz podstawowa **A** o rozmiarze 1998492 (Tab. 4.1) i jej symetryczne podmacierze  $M_{ii}^E$ .

GFLOPS / $P_{ebw}$	<code>cusparsedcsmv</code>	<code>cusparsedhybm</code> (AUTO)	<code>cusparsedhybm</code> (MAX)	Sliced ELLR-T
$M_{11}^E$	41%	41%	57%	48%
$M_{22}^E$	44%	49%	52%	42%
$M_{33}^E$	30%	26%	26%	37%
<b>A</b>	35%	31%	32%	37%

TABELA 6.5: Wydajność obliczeniowa (wyrażona w GFLOPS) operacji *matvec* uzyskana dla implementacji `cusparsedcsmv` i Sliced ELLR-T na GPU dla problemów testowych z kolekcji Williamsa [110].

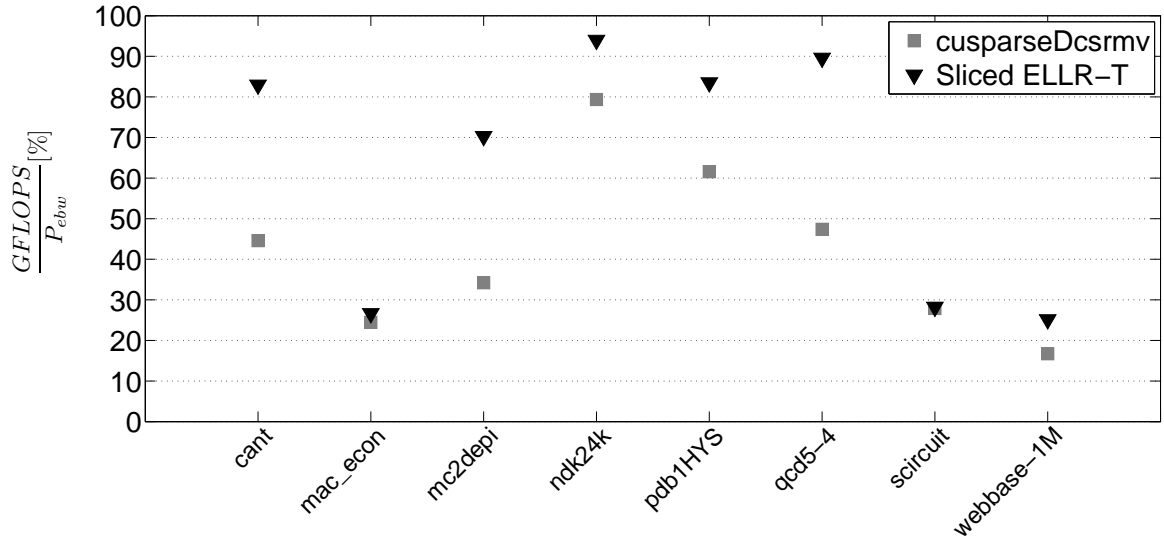
Macierz	N	NNZ	NNZ / N	<code>cusparsedcsmv</code>	Sliced ELLR-T	Przyspieszenie
cant	62 451	4 007 383	64	16,0	29,7	1,9
mac_econ_fwd500	206 500	1 273 389	6	8,3	9,1	1,1
mc2depi	525 825	2 100 225	4	11,4	23,3	2,1
nd24k	72 000	28 715 634	399	28,5	33,8	1,2
pdb1HYS	36 417	4 344 765	119	22,1	30,0	1,4
qcd5_4	49 152	1 916 928	39	16,9	31,9	1,9
scircuit	170 998	958 936	6	9,5	9,6	1,0
webbase-1M	1 000 005	3 105 536	3	5,4	8,2	1,5

wydajnością obliczeniową (ang. roofline model) [107–109]:

$$P_{ebw} = \frac{2 \cdot NNZ}{\frac{M_A}{B_{GPU}}} \quad (6.6)$$

gdzie:  $NNZ$  - liczba elementów niezerowych,  $M_A$  to rozmiar pamięci zaalokowanej na potrzeby kernela,  $B_{GPU}$  przepustowość pamięci GPU (określona dla akceleratora na podstawie programu `bandwidthTest` udostępnionego w przykładach CUDA. Dla akceleratora K40c  $B_{GPU} = 201$  GB/s). Dane przedstawione w Tab. 6.4 wskazują, iż format Sliced ELLR-T w zależności od macierzy osiąga od 37% do 48% teoretycznej wydajności obliczeniowej akceleratora graficznego.

Lepszą wydajność można uzyskać dla innych macierzy. W Tab. 6.5 porównano wydajność obliczeniową operacji *matvec* uzyskaną dla dwóch implementacji (`cusparsedcsmv` i Sliced ELLR-T). W testach wykorzystano macierze z kolekcji Williamsa [110]. Oprócz jednej macierzy (scircuit), wykorzystanie formatu Sliced ELLR-T skraca czas obliczeń mnożenia macierzy rzadkiej przez wektor względem `cusparsedcsmv`. Największe przyspieszenie względem `cusparsedcsmv` uzyskano dla macierzy qcd5\_4 i mc2depi, w których występuje stała liczba elementów niezerowych w wierszu (w macierzy mc2depi 99,4% wierszy zawiera tylko 4 elementy niezerowe w wierszu, w macierzy qcd5\_4 wszystkie wiersze mają 39 elementów niezerowych w wierszu). Na rys. 6.12 skonfrontowano wydajność obliczeniową mierzoną w GFLOPS z maksymalną możliwą do uzyskania wydajnością obliczeniową ze względu na przepustowość pamięci ( $P_{ebw}$ ). Dla formatu Sliced ELLR-T wydajność powyżej 80% uzyskano dla macierzy (nd24k, qcd5\_4, cant,



RYSUNEK 6.12: Wydajność obliczeniowa (wyrażona jako  $\frac{GFLOPS}{P_{eww}} \cdot 100\%$ ) operacji *matvec* uzyskana dla implementacji *cusparseDcsrmmv* i *Sliced ELLR-T* na GPU dla problemów testowych z kolekcji Williamsa (Tab. 6.5) [110].

pdb1HYS), w których średnia liczba elementów niezerowych w wierszu wynosi powyżej 39 i obliczenia w wierszu wykonuje odpowiednio  $T = 4$  (cant, qcd5-4) i  $T = 8$  (ndk24k, pdb1HYS) wątków.

### 6.2.2.2 Rozwój formatu Sliced ELLR-T

Artykuł z formatem *Sliced ELLR-T* został opublikowany w roku 2011 [106] i jego kod został udostępniony na stronie internetowej autora rozprawy [111]. Od tego czasu inni badacze rozwijali ten format czego efektem są poniższe publikacje, w których nazwa bazowa „*Sliced ELLR-T*” była zmieniana w zależności od zastosowanej modyfikacji. W publikacji [32] autorzy wykorzystali *Sliced ELLR-T* i w swoim formacie *SELL-C-σ* również dzielą macierz na podmacierze (w tym przypadku o rozmiarze  $C$ ). W formacie *Sliced ELLR-T* globalna macierz może być posortowana (np. od najmniej do najbardziej licznej wiersza). W formacie *SELL-C-σ* parametr  $σ$  określa zakres wierszy, które są sortowane. Modyfikacja w sposobie sortowania optymalizuje odczyt danych z pamięci akceleratora. W artykule [32] autorzy zastosowali kombinację parametrów, w której  $σ$  jest wielokrotnością  $C$  (np.  $C = 16$ ,  $σ = 256$ ). W publikacji [33] autorzy proponują format *SELL-P*, w którym dokonali modyfikacji *SELL-C-σ* o uzupełnienie wierszy macierzy o dodatkowe zera. Poza tym podobnie jak w formacie *Sliced ELLR-T* obliczenia w pojedynczym wierszu wykonywane są przez kilka wątków.

### 6.2.3 Rozwiązanie problemu liniowego w operatorze ściskającym

Stosując wielopoziomowy operator ściskający na najniższym poziomie należy rozwiązać układ równań ( $\mathbf{M}_{11}^E z_E = r_E$ ) (patrz. Wydruk 2.3). W tym celu można skorzystać z metod bezpośrednich lub iteracyjnych. Testy numeryczne wykazały, iż aby zapewnić zbieżność metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym

o cyklu  $V$  należy na tym etapie zapewnić wysoką dokładność rozwiązania. Na przykładzie problemu testowego (rozmiar macierzy  $\mathbf{A}$ :  $N = 806811$ , rozmiar macierzy  $\mathbf{M}_{11}^E$ :  $N_1 = 41715$ , Tab. 4.1) zweryfikowano skuteczność metody bezpośredniej rozwiązywania układu równań oraz metod iteracyjnych z operatorami ściskającymi mającymi za zadanie poprawę zbieżności [3].

Na wstępie wykonano testy numeryczne metod bezpośrednich z wykorzystaniem środowiska MATLAB. Uzyskano bardzo dokładne rozwiązanie (dla podwójnej precyzji:  $\|\mathbf{r}_E - \mathbf{M}_{11}^E \mathbf{z}_E\|_2 < 10^{-12}$ ), które pozwoliło uzyskać bardzo dobrą zbieżność całej metody iteracyjnej z wielopoziomowym operatorem ściskającym.

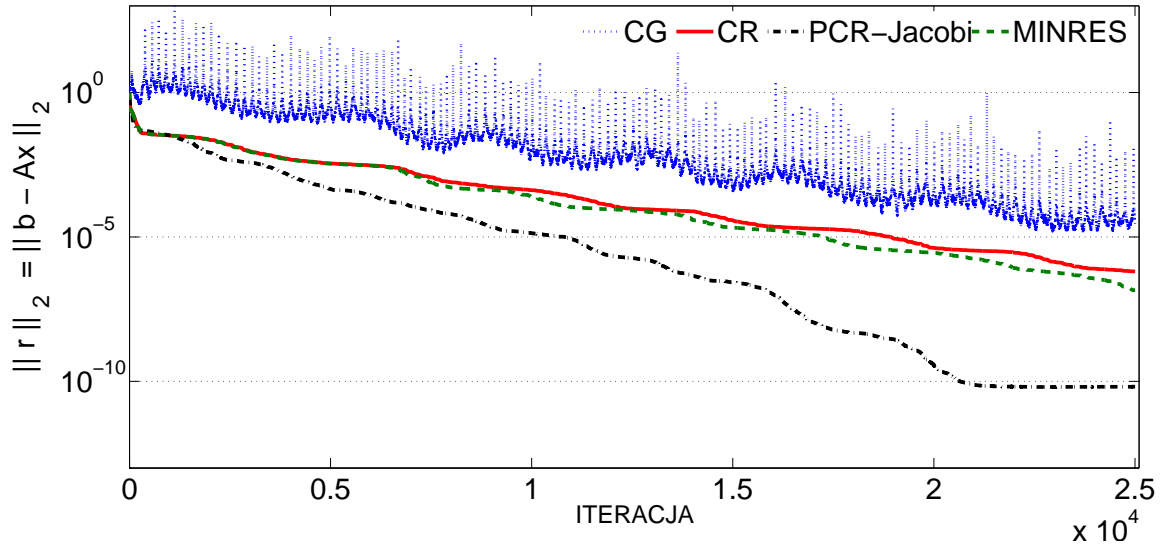
W dalszej kolejności zastosowano metody iteracyjne (gradientów sprzężonych (CG), residuów sprzężonych (CR), MINRES (ang. minimal residual method) [3]), które pozwoliły osiągnąć zbieżność na poziomie  $\|\mathbf{r}_E - \mathbf{M}_{11}^E \mathbf{z}_E\|_2 < 10^{-5}$ , ale potrzebowały ok. piętnastu tysięcy iteracji (rys. 6.13), co jest kilkaset razy kosztowniejsze niż rozwiązanie bezpośrednie. Zastosowanie operatora ściskającego Jacobiego w metodzie sprzężonych residuów poprawiło zbieżność, jednakże potrzebnych było ok. dwudziestu pięciu tysięcy iteracji by uzyskać zbieżność na poziomie metody bezpośredniej (rys. 6.13). Zweryfikowano również przydatność innych operatorów ściskających:

- (a) wielomianowe operatory ściskające (Neumanna, Czebyszewa [3]):  
prekondycjoner Neumana pozwolił na zmniejszenie liczby iteracji potrzebnych od osiągnięcia zbieżności na poziomie  $10^{-9}$ , jednakże koszt czasowy pojedynczej iteracji jest większy niż iteracji w metodach PCR-Jacobi (rys. 6.14),
- (b) operator ściskający w postaci niekompletnej faktoryzacji LU z progiem (ILUT, ang. incomplete LU factorization with threshold<sup>2</sup>, [3]):  
przeanalizowano przydatności takiego operatora ściskającego w środowisku MATLAB i zaobserwowano, iż dla macierzy testowej niekompletna faktoryzacja ILUT trwała bardzo długo. Jeżeli macierz została najpierw poddana permutacji wierszy i kolumn RCM (ang. Reverse Cuthill-McKee [112, 113]) to czas uzyskania czynników  $\mathbf{L}$  i  $\mathbf{U}$  skrócił się znacząco, a zastosowanie operatora ściskającego ILUT dla metod GMRES, PCG pozwoliło osiągnąć zbieżność na poziomie  $10^{-12}$  po zaledwie 10 iteracjach (rys. 6.15),
- (c) operator ściskający w postaci niekompletnej faktoryzacji ILU(0) [3]:  
testy wykazały, iż prekondycjoner ten dla testowanej macierzy pozwala uzyskać niezadowalającą zbieżność na poziomie  $\|\mathbf{r}_E - \mathbf{M}_{11}^E \mathbf{z}_E\|_2 < 10^{-2}$ .

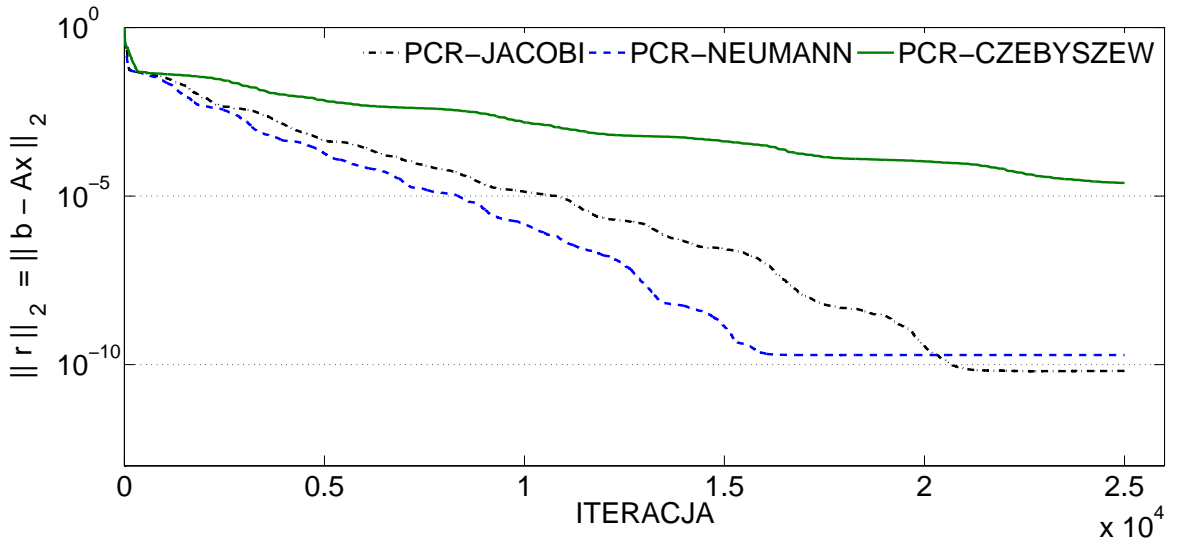
Powyżej oceniono, iż dużą dokładność rozwiązania układu równań uzyskuje się dla metod iteracyjnych (PCG, GMRES) z operatorem ściskającym ILUT i dla metody bezpośredniej. Faktoryzację ILUT można wykonać na CPU, następnie faktory przesłać na GPU i obliczenia operatora ściskającego polegające na wykonaniu operacji podstawiania „w przód” i podstawiania „wstecz” (ang. forward substitution, back substitution) wykonać przy użyciu biblioteki CUSPARSE [92]. Niestety testy wykazały, iż czynnik  $\mathbf{L}$  wygenerowany w środowisku MATLAB ma wartości niezerowe powyżej diagonalii co sprawia, że funkcja realizująca podstawienie „w przód” z biblioteki CUSPARSE [92]

---

<sup>2</sup>Faktoryzację w środowisku MATLAB wykonano przy użyciu komendy: `[L,U] = ilu(A,struct('type','ilutp','droptol',lu_threshold))` gdzie `lu_threshold=10-6`.



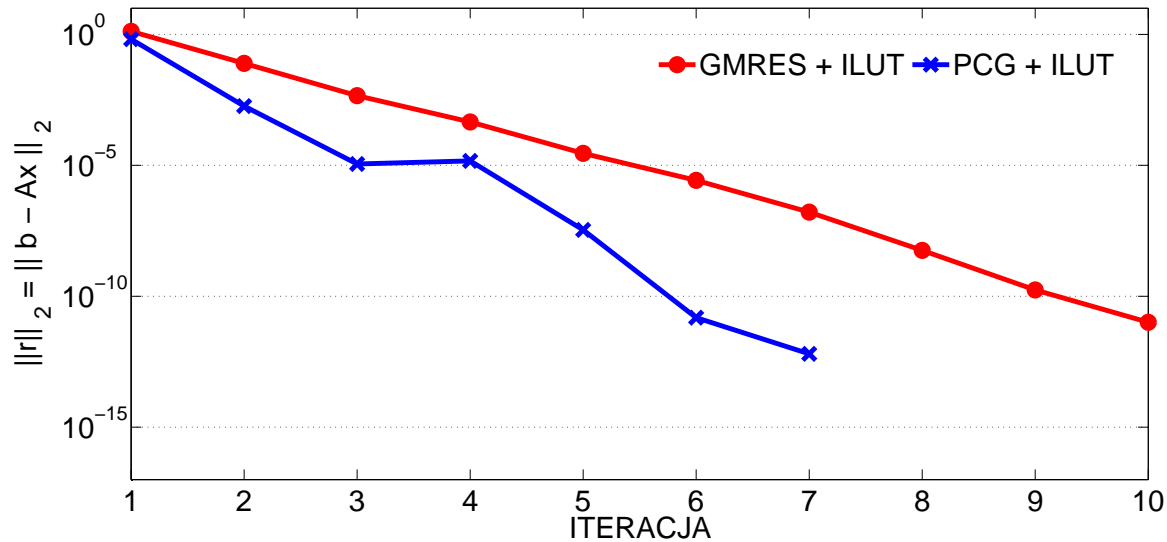
RYSUNEK 6.13: Zbieżność (norma euklidesowa wektora residualnego) metod iteracyjnych: CG - metoda gradientów sprzężonych, CR - metoda residuów sprzężonych, PCR - metoda residuów sprzężonych z operatorem ściskającym Jacobiego, MINRES - metoda minimalnych residuów). Problem testowy: filtr grzebieniowy 9-rzędu, macierz  $\mathbf{M}_{11}^E$  z dolnego poziomu operatora ściskającego (rozmiar macierzy głównej  $N = 806811$ , rozmiar macierzy  $\mathbf{M}_{11}^E$ :  $N_1 = 41715$ ).



RYSUNEK 6.14: Zbieżność (norma euklidesowa wektora residualnego) metod iteracyjnych z wielomianowymi operatorami ściskającymi Neumanna i Czebyszewa w porównaniu z metodą residuów sprzężonych z operatorem ściskającym Jacobiego. Problem testowy: filtr grzebieniowy 9-rzędu, macierz  $\mathbf{M}_{11}^E$  z dolnego poziomu operatora ściskającego (rozmiar macierzy głównej  $N = 806811$ , rozmiar macierzy  $\mathbf{M}_{11}^E$ :  $N_1 = 41715$ ).

uzyskuje błędne wyniki i ostatecznie metody PCG oraz GMRES z operatorem ściskającym ILUT na GPU nie uzyskują poprawnego rozwiązania.

Zweryfikowano implementację rozwiązywania układów równań metodą bezpośrednią dostępną w CUDA 7.0 (biblioteka cuSOLVER [42]) i okazało się, że zapotrzebowanie faktoryzacji na pamięć jest tak duże (faktoryzacja macierzy o rozmiarze 41715



RYSUNEK 6.15: Zbieżności metod iteracyjnych (PCG, GMRES) z operatorem ściskającym ILUT. Problem testowy: filtr grzebieniowy 9-rzędu, macierz  $\mathbf{M}_{11}^E$  z dolnego poziomu operatora ściskającego (rozmiar macierzy głównej  $N = 806811$ , rozmiar macierzy  $\mathbf{M}_{11}^E$ :  $N_1 = 41715$ ).

TABELA 6.6: Porównanie wydajności biblioteki Intel (CPU) [43] i Spral (GPU) [41]. Macierz  $\mathbf{M}_{11}^E$  o rozmiarze  $N = 41715$  i liczbie elementów niezerowych  $NNZ = 511599$ . Przy zastosowaniu typu double (64-bit) i integer (32-bit) w formacie CRS górna połówka macierzy zajmuje w pamięci 6 MB. Etap analizy w przypadku biblioteki Intel MKL Pardiso obejmuje faktoryzację symboliczną, a dla biblioteki SPRAL zarówno alokację danych na GPU oraz faktoryzację symboliczną.

Etap	SPRAL (GPU)	Intel MKL Pardiso (CPU)
zapotrzebowanie na pamięć [MB]	92,3	35,2
analiza (faktoryzacja symboliczna)[s]	0,545	0,247
faktoryzacja numeryczna [s]	0,133	0,045
rozwiązanie ukł. równań [s]	0,008	0,004

wymagała aż 7 GB pamięci), że dyskwalifikuje to użycie tej biblioteki dla macierzy generowanych w niniejszej rozprawie. Testy wydajnościowe (Tab. 6.6) uruchomione na CPU (Intel MKL Pardiso) i na GPU (SPRAL) wykazały, iż zarówno czasy faktoryzacji, jak i rozwiązania małych układu równań są dłuższe dla biblioteki SPRAL niż Intel MKL Pardiso. Podsumowując, metodę bezpośredniego rozwiązania układu równań na CPU (Intel MKL Pardiso<sup>3</sup>) uznano za najbardziej optymalną z punktu widzenia czasu wykonania obliczeń i dokładności otrzymanego rozwiązania na dolnym poziomie operatora ściskającego. Ponadto testy wykazały, że operacje przesłania wektorów prawej strony z GPU do pamięci CPU (przed rozwiązaniem układu równań) i wektora wynikowego z CPU i GPU (po rozwiązaniu układu równań) nie wprowadzają dużych opóźnień, gdyż są wielokrotnie krótsze niż etap rozwiązania układu równań na CPU.

Na zakończenie tego punktu warto doprecyzować kwestię przyjętego nazewnictwa.

<sup>3</sup>W artykule [114] z roku 2011 wykorzystano pakiet z Uniwersytetu z Bazylei [115, 116]. W roku 2012, wydajności pakietu Intel MKL Pardiso [43] osiągnęła podobny poziom i od tej pory korzystano z pakietu Intel MKL Pardiso [43].

Implementację w dalszej części będzie określana jako *hybrydowa*, gdyż rozwiązanie na dolnym poziomie operatora ściskającego wykonane jest na CPU, a reszta obliczeń wykonywana jest na GPU [114].

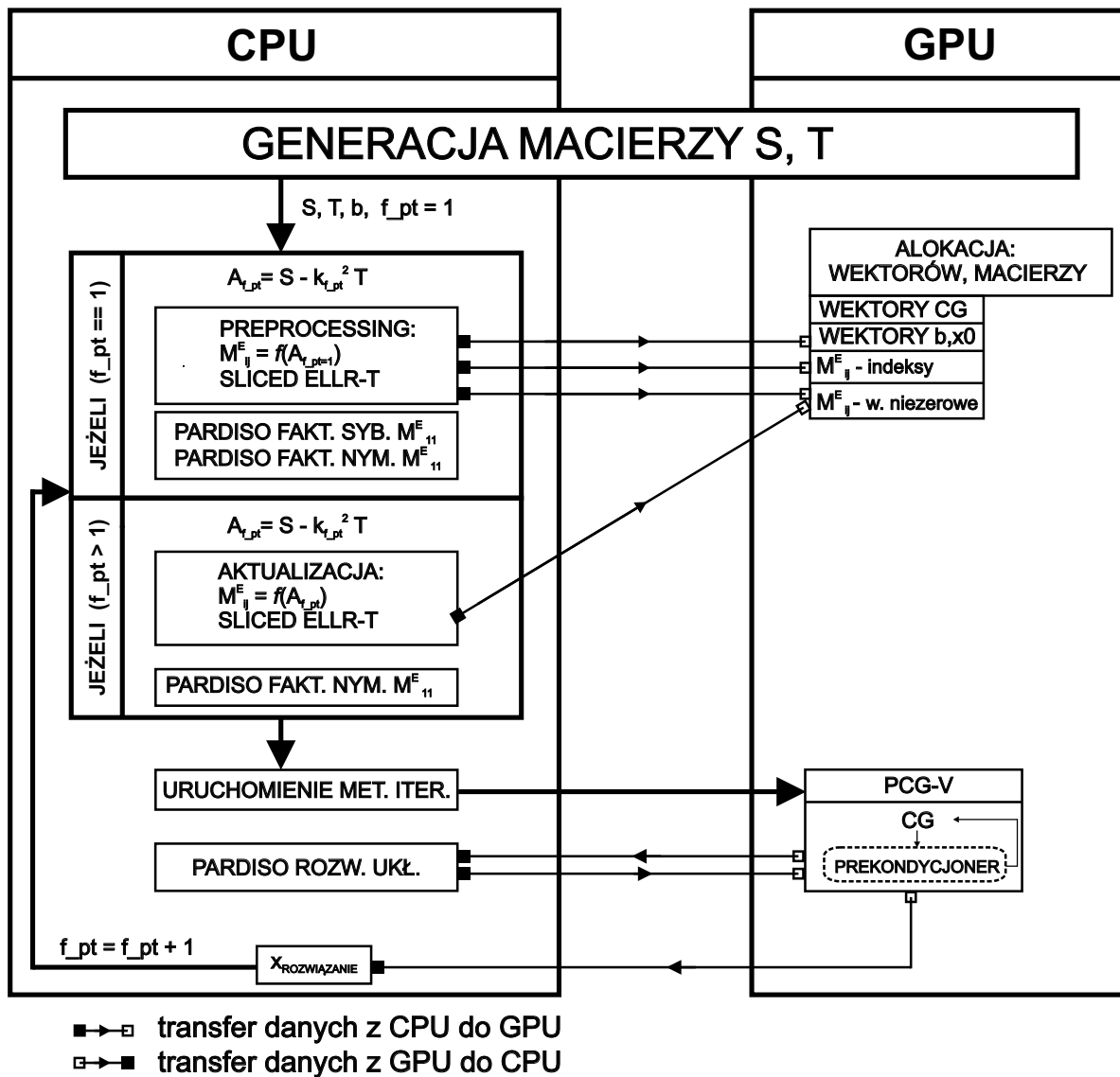
### 6.3 Implementacja metody PCG z wielopoziomym operatorem ściskającym dla jednego akceleratora graficznego

Zastosowanie strategii implementacji mnożenia macierzy rzadkiej przez wektor (p. 6.2.2) i rozwiązania układu równań na najniższym poziomie operatora ściskającego (p. 6.2.3) wymusiło wprowadzenie dodatkowych etapów w schemacie symulacji zagadnień elektromagnetycznych z metodzie elementów skończonych z rysunku 6.5.

Po pierwsze, z racji tego, że w docelowej implementacji zastosowano nowy format zapisu macierzy rzadkiej Sliced ELLR-T do wykonania operacji *matvec* na akceleratorze graficznym, to przed uruchomieniem metody iteracyjnego rozwiązania, macierze rzadkie  $\mathbf{M}_{ij}^E$  poddane są odpowiedniemu przetwarzaniu wstępnemu (p. 6.2.2, w etapie konwersji macierzy z formatu CRS do formatu Sliced ELLR-T permutowane są wektory wartości niezerowych i indeksów tak, by uzyskać efektywny dostęp do danych na akceleratorze graficznym w trakcie wykonywania operacji *matvec*). Po drugie, w p. 6.2.3 podjęto decyzję o rozwiązaniu układu równań na najniższym poziomie operatora ściskającego na CPU. Z tego powodu, przed rozpoczęciem procesu iteracyjnego należy wykonać faktoryzację symboliczną i numeryczną macierzy  $\mathbf{M}_{11}^E$ . Gdyby powyższe procedury (przetwarzanie wstępne macierzy, faktoryzacja) trzeba było wykonać dla każdej częstotliwości, to proces rozwiązywania układów równań obciążony byłby znaczącym narzutem. Na szczęście macierze  $\mathbf{A}$  mają tę samą strukturę (ang. pattern), tzn. dla każdej częstotliwości położenie elementów niezerowych macierzy  $\mathbf{A}$  jest takie samo. Innymi słowy, jeśli macierze  $\mathbf{A}$  przechowywane są w formacie CRS to macierze reprezentowane są przez różne wektory wartości ( $\mathbf{v}$ ) i niezależne od częstotliwości takie same wektory indeksów kolumn ( $\mathbf{J}$ ) oraz skompresowanego wiersza ( $\mathbf{Iptr}$ ). Powyższa właściwość niesie za sobą wiele korzyści dla procesu rozwiązywania układów równań liniowych. Po pierwsze, wyłuskanie macierzy  $\mathbf{M}_{ij}^E$  z macierzy  $\mathbf{A}$  odbywa się w pełni tylko dla pierwszej częstotliwości, a dla następnych częstotliwości podmieniane są wyłącznie wektory wartości niezerowych. Przez analogię, przetwarzanie wstępne macierzy wymagane w formacie Sliced ELLR-T wykonane jest w całości dla pierwszej częstotliwości, a dla kolejnych częstotliwości ma miejsce wyłącznie aktualizacja wektorów niezerowych macierzy  $\mathbf{M}_{ij}^E$  przechowywanych w formacie Sliced ELLR-T. Po drugie, z punktu widzenia wykorzystania akceleratora graficznego, powyższa właściwość przynosi korzyści w postaci redukcji ilości danych przesyłanych pomiędzy CPU i GPU. Przesłanie wszystkich wektorów (wartości niezerowych, indeksów) opisujących macierze  $\mathbf{M}_{ij}^E$  wykonywany jest wyłącznie dla pierwszej częstotliwości. Dla kolejnych częstotliwości konieczne jest już tylko przesłanie zaktualizowanych wektorów przechowujących wartości niezerowe macierzy<sup>4</sup>. Można również zoptymalizować liczbę procedur związanych z rozwiązaniem układu równań na najniższym poziomie operatora ściskającego. W tym celu dla pierwszej częstotliwości należy wykonać faktoryzację symboliczną i numeryczną macierzy

<sup>4</sup>Warto zaznaczyć, iż z CPU na GPU przesyłane są wektory reprezentujące macierze  $\mathbf{M}_{ij}^E$  i nie ma potrzeby przesłania całej macierzy  $\mathbf{A}$ .





RYSUNEK 6.16: Efektywny schemat symulacji zagadnienia elektromagnetycznego z wykorzystaniem metody elementów skończonych w paśmie dla  $f_m$  częstotliwości ( $f_{pt}$  to indeks częstotliwości  $f$  i  $f_{pt} = \{1, \dots, f_m\}$ ). Obliczenia w etapach generacji macierzy i rozwiązywania układu wykonane na CPU i GPU.

$M_{11}^E$ . Z racji tego, że macierze mają ten sam rozkład elementów niezerowych, dla kolejnych częstotliwości nie ma potrzeby wykonywania faktoryzacji symbolicznej i można wykonać wyłącznie faktoryzację numeryczną.

Wszystkie powyższe modyfikacje pozwoliły uzyskać efektywny - z punktu widzenia komunikacji i przesyłu danych między GPU i CPU oraz wykonywanych operacji po stronie CPU (faktoryzacja, preprocessing macierzy) - schemat pozwalający na rozwiązanie układu równań z wykorzystaniem pojedynczego akceleratora graficznego (rys. 6.16).

W dalszej części rozdziału przedstawiono schemat uruchomienia i wykonania obliczeń metody iteracyjnej, która na rys. 6.16 została oznaczona jako **PCG-V**. Pseudokod, który pozwolił na masowo zrównoleglenie metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V zaprezentowano na Wydrukach C.4-C.5. W prawej części wydruków (w komentarzach) opisano wszystkie operacje z uwzględnieniem klasyfikacji operacji wg. BLAS (ang. Basic Linear Algebra Subprograms). Ope-



racjom na wektorach tj. skalowanie, kopiowanie, iloczyn skalarny wektorów przypisano kategorię - *I*. Operacjom na macierzach rzadkich: mnożenie macierzy rzadkiej przez wektor, rozwiązywanie układu równań liniowych przypisano kategorię - *II*. Wszystkie etapy metody iteracyjnej (z wyjątkiem przesyłania danych między CPU i GPU) zostały **zrównoleglone**: *matvec* i operacje na wektorach na GPU, rozwiązanie na dolnym poziomie - funkcje z biblioteki Intel MKL (Pardiso). Dodatkowo na rysunkach 6.17-6.18, które korespondują z Wydrukami C.4-C.5 pokazano kolejność wykonywanych obliczeń, wskazano które operacje wykonywane są na GPU, a które na CPU oraz w jakich kierunkach odbywa się przesyłanie danych między GPU i CPU. Ponadto w każdym z bloków reprezentującym etapy metody iteracyjnej (MATVEC, DOT - iloczyn skalarny, AXPY - sumowanie wektora i dodawanie wektorów) umieszczono informację o liczbie wykonywanych operacji zmiennoprzecinkowych r. (6.7)-(6.9). Ta informacja jest również bardzo istotna z punktu widzenia implementacji metody iteracyjnego rozwiązania układu równań z wykorzystaniem kilku akceleratorów graficznych (rozdział 7), gdyż umożliwia porównanie liczby wykonywanych operacji w poszczególnych wariantach i ocenę czy obliczenia zostały efektywnie rozdzielone na akceleratorach graficznych. Dla przykładu liczbę operacji dla procedur *matvec*, *dot*, *axpy* metody gradientów sprzężonych przedstawiono w r. (6.7)-(6.9):

$$\begin{aligned} \text{matvec} &= \sum_{i=1}^N \text{l. iloczynów w wierszu} + \text{l. sum w wierszu} = \sum_{i=1}^N (NNZ_i) + (NNZ_i + 1) = \\ &= \sum_{i=1}^N (2 \cdot NNZ_i + 1) = (2 \cdot NNZ + N) \approx 2 \cdot NNZ \end{aligned} \quad (6.7)$$

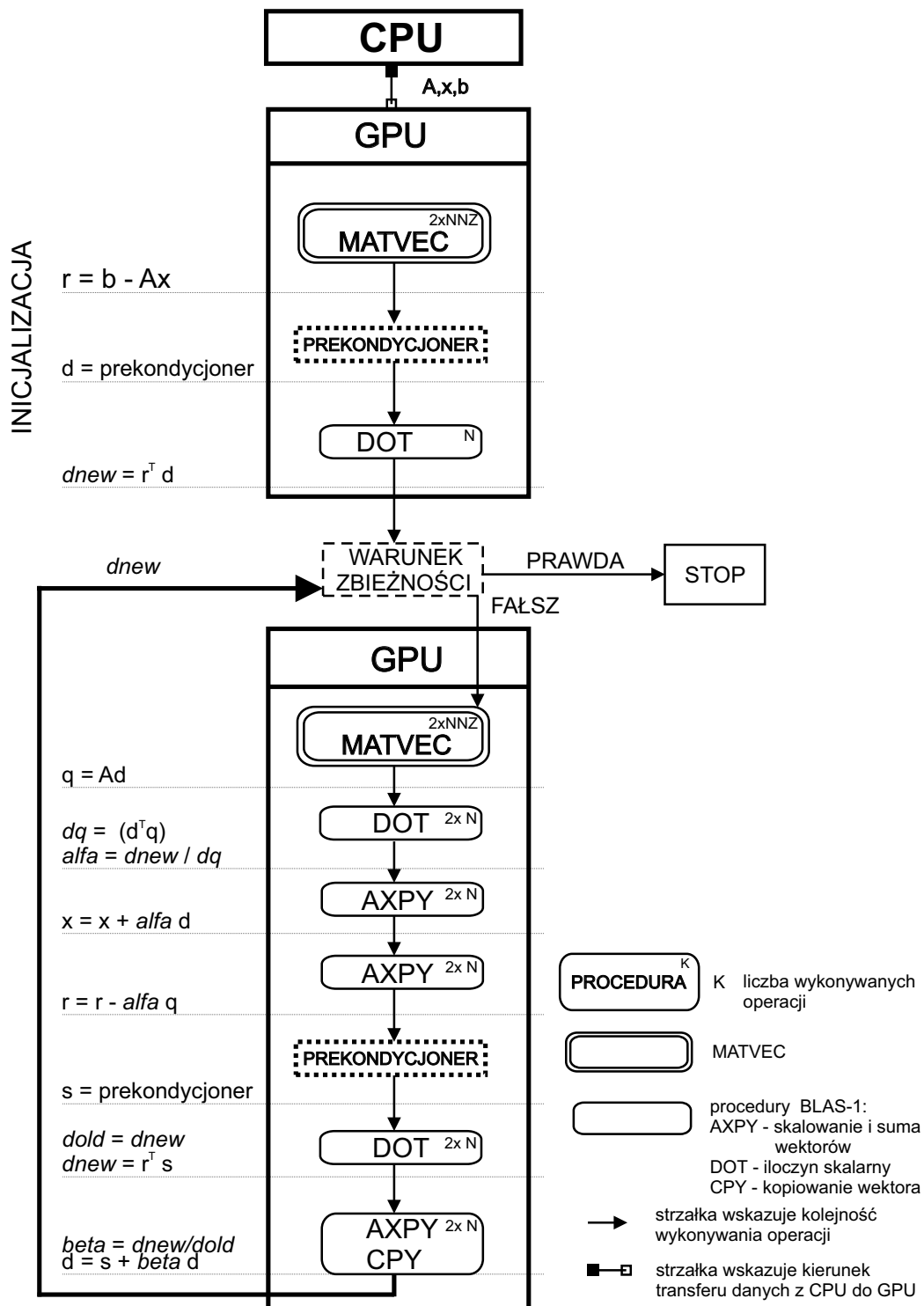
$$\text{dot} : [\mathbf{d}^T \mathbf{r}] = \sum_{i=1}^N \mathbf{d}_i \cdot \mathbf{r}_i = \text{l. iloczynów} + \text{l. sum} = N + (N + 1) = 2 \cdot N + 1 \approx 2 \cdot N \quad (6.8)$$

$$\text{axpy} : [\mathbf{x} = \mathbf{x} + \alpha \mathbf{d}] = \sum_{i=1}^N \mathbf{x}_i + \alpha \mathbf{d}_i = \text{l. sum} + \text{l. iloczynów} = N + N = 2 \cdot N \quad (6.9)$$

gdzie  $NNZ_i$  oznacza liczbę elementów niezerowych w „i”-tym wierszu macierzy  $\mathbf{A}$ ,  $NNZ$  oznacza sumaryczną liczbę elementów niezerowych w macierzy  $\mathbf{A}$ ,  $N$  oznacza liczbę wierszy macierzy  $\mathbf{A}$ .

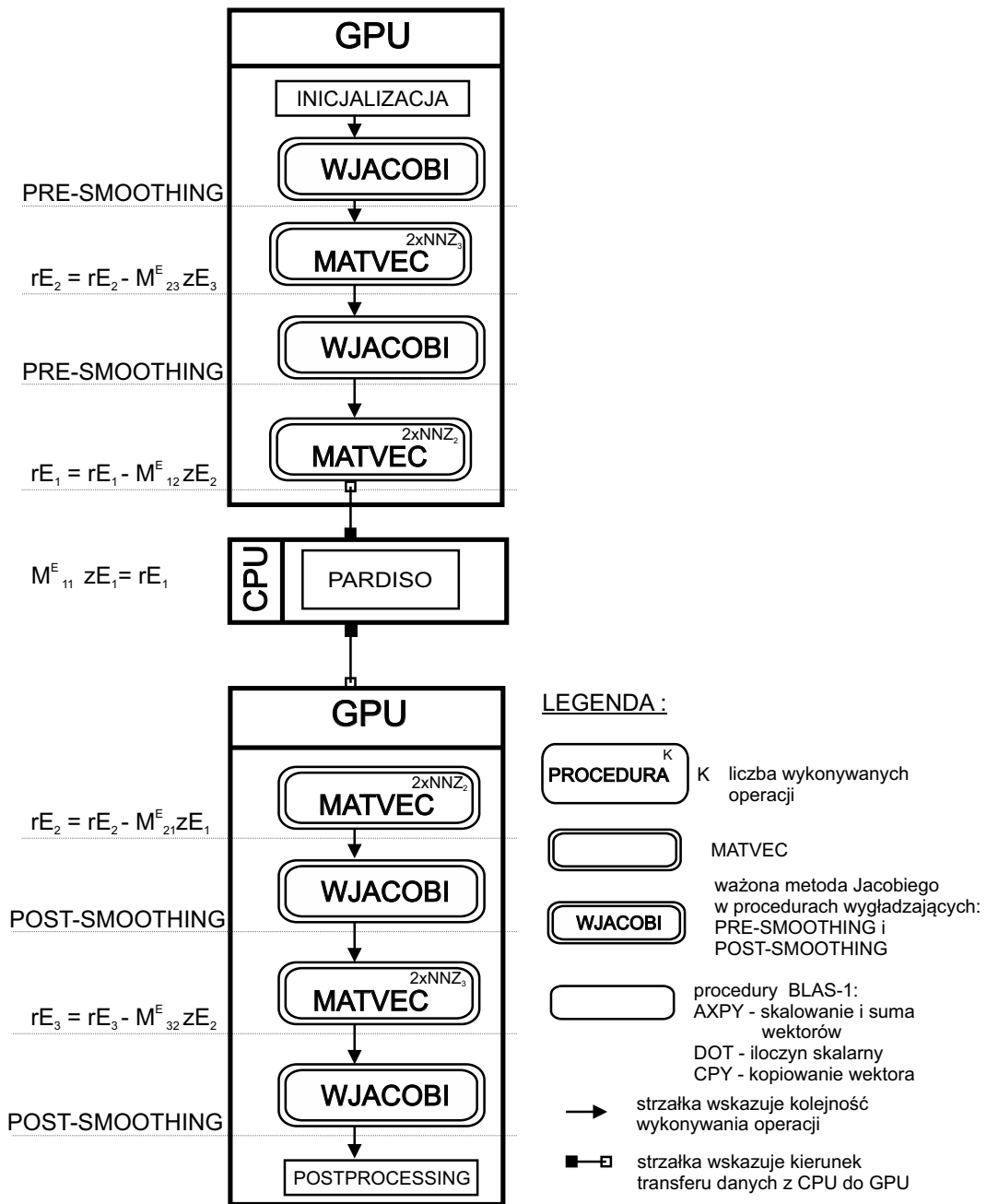
Poniżej opisano operacje hybrydowej implementacji metody iteracyjnej z wielopoziomowym operatorem ściskającym o cyklu V, której operacje umieszczono na Wydrukach C.4-C.6 oraz przedstawiono na rys. 6.17-6.18. Najpierw wykonywana jest inicjalizacja, tzw. iteracja zerowa (Wydruk C.4, linie: 22-26), w której na GPU inicjalizowany jest wektor residualny  $\mathbf{r}$  oraz przy użyciu wielopoziomowego operatora ściskającego określany jest wektor kierunku  $\mathbf{d}$  poszukiwań rozwiązania  $\mathbf{x}$ . Inicjalizacja operatora ściskającego polega na wyłuskaniu wektorów  $\mathbf{r}_{E_L}$  z wektora residualnego  $\mathbf{r}_E$  oraz na wyzerowaniu wartości wektorów  $\mathbf{z}_{E_L}$  (Wydruk C.5, linie: 20-31)<sup>5</sup>. Następnie wykonywane są operacje wielopoziomowego operatora ściskającego. Na najwyższym poziomie wykonywana jest ważona metoda Jacobiego celem uzyskania zgrubnego rozwiązania układu równań  $\mathbf{M}_{33}^E \mathbf{z}_{E3} = \mathbf{r}_{E3}$  (Wydruk C.5, linia: 35; Wydruk C.6). Następnie wykonywana jest operacja *matvec* będąca łącznikiem pomiędzy III a II poziomem. Na drugim poziomie ponownie wykonywana jest ważona relaksacja Jacobiego ( $\mathbf{M}_{22}^E \mathbf{z}_{E2} = \mathbf{r}_{E2}$ ). Kolejną

<sup>5</sup> $L$  oznacza poziom operatora ściskającego: na poziomie  $L = 1$  wektor ma rozmiar  $N_1$ , na poziomie  $L = 2$  wektor ma rozmiar  $N_2$ , na poziomie  $L = 3$  wektor ma rozmiar  $N_3$ .  $N = N_1 + N_2 + N_3$  - rozmiar macierzy współczynników  $\mathbf{A}$ .



RYSUNEK 6.17: Schemat wykonania obliczeń w metodzie gradientów sprzężonych z operatorem ściskającym (prekondycjoner) na pojedynczym akceleratorze graficznym.

operacją jest *matvec*, która łączy poziom II i I. Z racji tego, że operacja na najniższym poziomie wykonywana jest na CPU (p. 6.2.3) wektor  $r_{E_1}$  musi zostać przesłany z GPU na CPU, a po rozwiązaniu układu na CPU, do pamięci GPU przesyłany jest wektor  $z_{E_1}$  (rys. 6.18). Dalej operacje operatora ściskającego wykonywane są na GPU w kierunku najwyższego poziomu. Po wykonaniu iteracji inicjalizującej rozpoczyna się proces iteracyjny (Wydruk C.4, linie: 27-38). W każdej iteracji uruchamiane są procedury



RYSUNEK 6.18: Schemat wykonania obliczeń w wielopoziomowym operatorze ściskającym (prekondycjoner) o cyklu V na pojedynczym akceleratorze.

operatora ściskającego, operacja mnożenia macierzy rzadkiej przez wektor i operacje na wektorach, które mają na celu uaktualnić wektor poszukiwanego rozwiązania ( $\mathbf{x}$ ), określić nowy wektor residualny ( $\mathbf{r}$ ) i wyznaczyć nowy wektor poszukiwań rozwiązania ( $\mathbf{d}$ ).

W Tab. 6.7 przedstawiono uśrednione czasy procedur wykonywanych w implementacjach metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym:

- PCG- $V_{(GPU+CPU)}$  - implementacja hybrydowa (GPU: *matvec* - Sliced ELLR-T, operacje na wektorach - CUBLAS, CPU: rozwiązanie układu równań na najniższym poziomie - Intel MKL Pardiso),

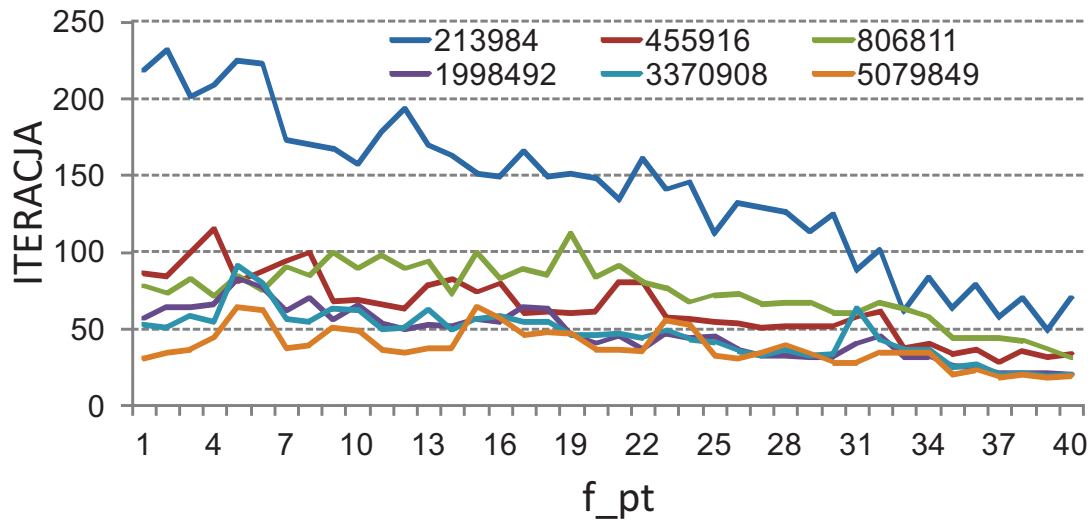
TABELA 6.7: Czas wykonania procedur PCG-V dla implementacji hybrydowej PCG-V<sub>(GPU+CPU)</sub> i implementacji referencyjnej na CPU PCG-V<sub>(CPU)</sub>. Uśredniony czas z 50 iteracji potrzebnych do rozwiązania układu równań na częstotliwości  $f = 920$  MHz. Liczba iteracji ważonej metody Jacobiego w operacjach wygładzających na III poziomie  $it\_L3 = 3$  i II poziomie  $it\_L2 = 6$  z rys. 2.7.

L.p.	PCG-V pojedyncza iteracja	(GPU+CPU) [ms]	[%]	(CPU) [ms]	[%]	Przyp.
1	$\mathbf{q} = \mathbf{A}\mathbf{d}$	25,1	19%	90,0	19%	3,6
2	$dq = \mathbf{d}^T \mathbf{q}$	0,2	0%	1,0	0%	5,8
3	$\alpha = \frac{d_{new}}{dq}$	0,0	0%	0,0	0%	-
4	$\mathbf{x} = \mathbf{x} + \alpha \mathbf{d}$	0,3	0%	1,0	0%	4,0
5	$\mathbf{r} = \mathbf{r} - \alpha \mathbf{q}$	0,2	0%	1,0	0%	4,5
6	$prekondycjoner\_V(\mathbf{M}_{11}^E, \dots, \mathbf{M}_{K,K}^E, \mathbf{r}, \mathbf{s})$	107,7	80%	390,0	80%	3,6
6.1	Inicjalizacja ( $\mathbf{r}_{Ei} = \mathbf{f}(\mathbf{r})$ , $\mathbf{z}_{Ei} = \mathbf{0}$ )	0,8	1%	1,0	0%	1,3
6.2	$\mathbf{z}_{E3} = wJacobi(\mathbf{M}_{33}^E, \mathbf{r}_{E3}, \mathbf{z}_{E3}, it\_L3)$	27,1	20%	110,0	23%	4,1
6.3	$\mathbf{r}_{E2} = \mathbf{r}_{E2} - \mathbf{M}_{23}^E \mathbf{z}_{E3}$	5,0	4%	20,0	4%	4,0
6.4	$\mathbf{z}_{E2} = wJacobi(\mathbf{M}_{22}^E, \mathbf{r}_{E2}, \mathbf{z}_{E2}, it\_L2)$	13,9	10%	59,0	12%	4,2
6.5	$\mathbf{r}_{E2} = \mathbf{r}_{E2} - \mathbf{M}_{21}^E \mathbf{z}_{E1}$	1,0	1%	2,0	0%	2,0
6.6	$\mathbf{r}_{E1}$ (GPU $\rightarrow$ CPU)	0,1	0%	0,0	0%	0,0
6.7	$\mathbf{M}_{11}^E \mathbf{z}_{E1} = \mathbf{r}_{E1}$	11,9	9%	12,0	2%	1,0
6.8	$\mathbf{z}_{E1}$ (CPU $\rightarrow$ GPU)	0,1	0%	0,0	0%	0,0
6.9	$\mathbf{r}_{E2} = \mathbf{r}_{E2} - \mathbf{M}_{21}^E \mathbf{z}_{E1}$	1,1	1%	3,0	1%	2,7
6.10	$\mathbf{z}_{E2} = wJacobi(\mathbf{M}_{22}^E, \mathbf{r}_{E2}, \mathbf{z}_{E2}, it\_L2)$	14,1	11%	59,0	12%	4,2
6.11	$\mathbf{r}_{E3} = \mathbf{r}_{E3} - \mathbf{M}_{32}^E \mathbf{z}_{E2}$	4,6	3%	14,0	3%	3,0
6.12	$\mathbf{z}_{E3} = wJacobi(\mathbf{M}_{33}^E, \mathbf{r}_{E3}, \mathbf{z}_{E3}, it\_L3)$	27,5	21%	109,0	22%	4,0
6.13	$\mathbf{s} = [\mathbf{z}_{E1}; \mathbf{z}_{E2}; \mathbf{z}_{E3}]$	0,4	0%	1,0	0%	2,8
7	$d_{old} = d_{new}$	0,0	0%	0,0	0%	-
8	$d_{new} = \mathbf{r}^T \mathbf{s}$	0,2	0%	1,0	0%	4,5
9	$\beta = \frac{d_{new}}{d_{old}}$	0,0	0%	0,0	0%	-
10	$\mathbf{d} = \mathbf{s} + \beta \mathbf{d}$	0,4	0%	1,0	0%	2,4
{1-10}	PCG-V [s]	134,0	100%	485,0	100%	3,6

- PCG-V<sub>(CPU)</sub> - implementacja w całości na CPU: *matvec* -Intel MKL, operacje na wektorach Intel MKL, rozwiązanie układu równań na najniższym poziomie - Intel MKL Pardiso).

W obydwu implementacjach metody iteracyjnej czas wykonania operacji *matvec* jest dominujący. Procentowy udział tej operacji (Tab. 6.7, procedury: 1, 6.2-6.5, 6.9-6.12) dla PCG-V<sub>(GPU+CPU)</sub> i PCG-V<sub>(CPU)</sub> wyniósł odpowiednio 89% i 96%. Drugą pod względem kosztu czasowego jest operacja rozwiązywania układu równań na najniższym poziomie operatora ściskającego. Procentowy udział tej operacji (Tab. 6.7, procedura: 6.7) dla PCG-V<sub>(GPU+CPU)</sub> i PCG-V<sub>(CPU)</sub> wyniósł odpowiednio 9% i 2%. Warto zauważyć, że w implementacji hybrydowej PCG-V<sub>(GPU+CPU)</sub> czasy przesyłania wektorów przed i po rozwiązaniu układu równań jest bardzo mały. Operacje na wektorach mają marginalny udział procentowy w pojedynczej iteracji. Analizując wyniki z Tab. 6.7, można zauważyć, iż przyspieszenie uzyskane dla operacji *matvec* (GPU: Sliced ELLR-T, względem CPU: Intel MKL), przekłada się na ok. 3,6-krotne skrócenie czasu obliczeń implementacji hybrydowej.

W Tab. 6.8 przedstawiono czasy wykonania metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym dla implementacji hybrydowej PCG-V<sub>(GPU+CPU)</sub>.



RYSUNEK 6.19: Liczba iteracji potrzebnych do uzyskania zbieżności rozwiązania na poziomie  $10^{-5}$ .  $f_{pt}$  - indeks częstotliwości w paśmie  $f_{bw}=920-980$  MHz.

TABELA 6.8: Czas iteracyjnego rozwiązania układu równań (metoda gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V). GPU: Tesla K40c (*matvec* - Sliced ELLR-T, operacje na wektorach - CUBLAS), CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków, rozwiązanie na dolnym poziomie operatora ściskającego - Intel MKL Pardiso). KON - konwersja z formatu CRS do formatu Sliced ELLR-T, HtoD - przesłanie macierzy z CPU do GPU. Problemy testowe Tab. 4.1.

L. wierszy N	KON, HtoD [s]	Iteracje $\ r\ _2 < 10^{-4}$	Dolny Poziom [s] (%)	PCG-V <sub>(GPU+CPU)</sub> [s]	PCG-V <sub>(GPU+CPU)</sub> (+KON,HtoD) [s]
213 984	0,3	141	0,06 (4%)	1,6	1,9
455 916	0,6	65	0,08 (5%)	1,5	2,1
806 811	1,0	74	0,19 (6%)	3,5	4,5
1 998 492	3,1	47	0,53 (9%)	6,3	9,4
3 370 908	4,7	47	0,97 (9%)	11,1	15,8
5 079 849	8,2	39	1,50 (10%)	14,4	22,6

W drugiej i trzeciej kolumnie umieszczono również czas konwersji z formatu CRS do formatu Sliced ELLR-T (KON) oraz czas przesłania danych z CPU na GPU (HtoD). Zauważono, że liczba iteracji potrzebnych do uzyskania zadanego progu zbieżności<sup>6</sup> waha się w zależności gęstości siatki i od częstotliwości od 19 do 232 (rys. 6.19)<sup>7</sup>. Z tego powodu liczba iteracji w Tab. 6.8 została uśredniona. W zależności od rozmiaru problemu implementacja PCG-V<sub>(GPU+CPU)</sub> potrzebowała od kilku sekund do kilku minut do uzyskania zadanego poziomu zbieżności. Warto zauważyć, że procentowy udział rozwiązania na dolnym poziomie rośnie wraz ze wzrostem rozmiaru problemu i wynosi od 4% do 10% czasu rozwiązania. W tab. 6.9 przedstawiono czas uzyskany dla implementacji, w której obliczenia wykonywane są wyłącznie na CPU (funkcje z biblioteki

<sup>6</sup>Przyjęto satysfakcjonujący poziom zbieżności normy euklidesowej wektora residualnego ( $\|r\|_2 < 10^{-4}$ ). Uzasadnienie w dalszej części rozprawy.

<sup>7</sup>Najwięcej iteracji potrzebnych do uzyskania satysfakcjonującej zbieżności potrzeba dla problemu o najmniejszej liczbie czworoscianów.

TABELA 6.9: Czas iteracyjnego rozwiązania układu równań (metoda gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V). CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków, *matvec*, operacje na wektorach - Intel MKL, rozwiązanie na dolnym poziomie operatora ściskającego - Intel MKL Pardiso). Ostatnie dwie kolumny to przyspieszenie wersji hybrydowej (bez i z uwzględnieniem narzutów wynikających z konwersji (KON) i przesyłania danych z CPU na GPU (HtoD)) względem implementacji na CPU. Zbieżność (norma euklidesowa wektora residualnego)  $\|r\|_2 < 10^{-4}$ .

L. wierszy N	PCG-V <sub>(CPU)</sub> [s]	PCG-V <sub>(CPU)</sub> vs. PCG-V <sub>(GPU+CPU)</sub>	PCG-V <sub>(CPU)</sub> vs. PCG-V <sub>(GPU+CPU)</sub> (+KON,HtD)
213 984	5,1	<b>3,3</b>	<b>2,7</b>
455 916	5,5	<b>3,6</b>	<b>2,6</b>
806 811	12,2	<b>3,5</b>	<b>2,7</b>
1 998 492	22,8	<b>3,6</b>	<b>2,4</b>
3 370 908	43,1	<b>3,9</b>	<b>2,7</b>
5 079 849	59,0	<b>4,1</b>	<b>2,6</b>

TABELA 6.10: Czasy (w sekundach) rozwiązania układu równań przy użyciu metod iteracyjnych z wielopoziomowym operatorem ściskającym (PCG-V) i metody bezpośredniej (PARDISO\_64: (CAŁ.) - macierz całościowa, (SYM.) - macierz symetryczna). W nawiasach przyspieszenie wariantu hybrydowego PCG-V<sub>(GPU+CPU)</sub> względem innych implementacji. KON - konwersja z formatu CRS do formatu Sliced ELLR-T, HtoD (ang. Host to Device) - przesłanie wektorów opisujących macierz rzadką z CPU (Host) do GPU (Device). GPU: Tesla K40c, CPU: CPU: Intel Xeon Sandy Bridge E5-2687W (8 wątków).

L. wierszy N	PCG-V <sub>(GPU+CPU)</sub> (KON, HtoD, rozwiązanie)	PCG-V <sub>(CPU)</sub> Intel MKL	PARDISO_64 (CAŁ.)	PARDISO_64 (SYM.)
213 984	1,8	5,1 (x2,7)	2,5 (x1,4)	1,9 (x1,0)
455 916	2,1	5,5 (x2,6)	6,8 (x3,2)	4,7 (x2,2)
806 811	4,5	12,2 (x2,7)	13,3 (x2,9)	9,0 (x2,0)
1 998 492	9,4	22,8 (x2,4)	56,3 (x6,0)	33,3 (x3,5)
3 370 908	15,8	43,1 (x2,7)	188,4 (x11,9)	88,1 (x5,6)
5 079 849	22,5	59,0 (x2,6)	- (-)	231,3 (x10,3)

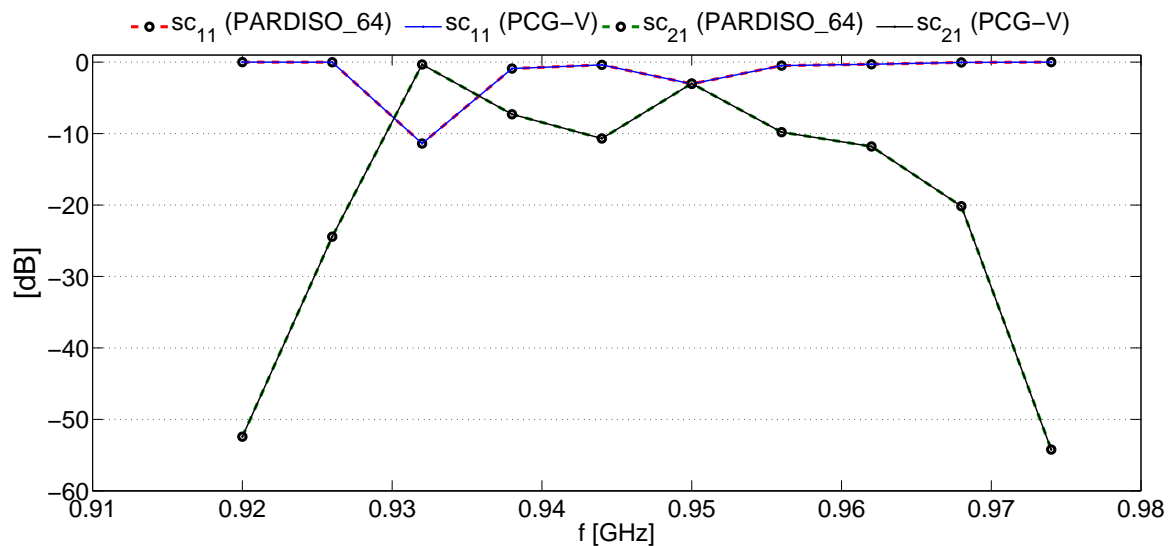
Intel MKL, obliczenia były wykonane przez 8 wątków<sup>8</sup>). W ostatnich dwóch kolumnach Tab. 6.9 umieszczono przyspieszenia implementacji hybrydowej (bez i z uwzględnieniem narzutu w postaci konwersji i przesyłania danych) w której obliczenia wykonywane są wyłącznie na CPU. Zauważyć można, iż zastosowanie akceleratora graficznego do wykonania obliczeń w operacji *matvec* i operacji na wektorach pozwoliło na ok. 2,5-4,5 krotną redukcję czasu rozwiązania układu równań.

W Tab. 6.10 zaprezentowano porównanie czasów rozwiązania układów równań przy użyciu metod iteracyjnych i bezpośrednich. Hybrydowa implementacja PCG-V<sub>(GPU+CPU)</sub> została uzupełniona o konwersję do formatu Sliced ELLR-T oraz przesyłanie wekto-

<sup>8</sup>Zgodnie z sugestią z dokumentacji wyłączono technologię Hyper-Threading.

TABELA 6.11: Parametry rozproszenia  $sc_{11}$  i  $sc_{21}$  uzyskane gdy układ równań rozwiązano przy użyciu metody bezpośredniej PARDISO\_64 i metody iteracyjnej (PCG-V). Błąd względny uzyskany dla metody iteracyjnej (PCG-V) względem metody bezpośredniej (PARDISO\_64) obliczono zgodnie z r. (6.10)-(6.11).

$f_{pt}$	$f$ [GHz]	$sc_{11}^{PARDISO}$ [dB]	$sc_{11}^{PCGV}$ [dB]	Err_ $sc_{11}$	$sc_{21}^{PARDISO}$ [dB]	$sc_{21}^{PCGV}$ [dB]	Err_ $sc_{21}$
1	0,92	-0,000025	-0,000025	-1,6E-04	-52,418047	-52,419457	2,7E-05
2	0,926	-0,015705	-0,015705	-3,0E-07	-24,425374	-24,425375	5,3E-08
3	0,932	-11,388997	-11,389011	1,2E-06	-0,327457	-0,327460	9,5E-06
4	0,938	-0,895778	-0,895745	-3,6E-05	-7,296033	-7,296228	2,7E-05
5	0,944	-0,388238	-0,388238	-4,6E-07	-10,679534	-10,679535	1,6E-07
6	0,95	-3,041446	-3,041446	-5,5E-08	-2,979376	-2,979382	2,2E-06
7	0,956	-0,480548	-0,480548	-6,9E-07	-9,798530	-9,798541	1,1E-06
8	0,962	-0,296654	-0,296647	-2,4E-05	-11,802820	-11,802940	1,0E-05
9	0,968	-0,042001	-0,042000	-4,3E-06	-20,166269	-20,166232	-1,8E-06
10	0,974	-0,000016	-0,000016	-3,6E-03	-54,219683	-54,223729	7,5E-05



RYSUNEK 6.20: Wartości  $sc_{11}$  i  $sc_{21}$  dla 10 częstotliwości. Dla metody iteracyjnej PCG-V zastosowano próg zbieżności na poziomie  $10^{-5}$ . Rozmiar macierzy  $N = 1998492$ .

rów reprezentujących macierz z CPU do GPU. Czas rozwiązania metodą bezpośrednią (PARDISO\_64) uwzględnia czas faktoryzacji (symbolicznej i numerycznej). Uzyskane rezultaty potwierdzają, iż hybrydowa implementacja PCG-V<sub>(GPU+CPU)</sub> pozwala uzyskać dużą redukcję czasu rozwiązania układu równań w metodzie elementów skończonych. Przyspieszenie obliczeń jest szczególnie widoczne dla problemów, w których macierze mają kilka milionów niewiadomych i czas faktoryzacji jest bardzo duży.

Na tym etapie należy wyjaśnić dlaczego wybrano poziom zbieżności metody iteracyjnej na poziomie  $10^{-5}$ . Przeanalizowano dokładność otrzymanych wyników parametrów rozproszenia otrzymanych w przypadku rozwiązania układu równań metodą bezpośrednią i iteracyjną. Na podstawie analizy błędów względnych r. (6.10)-(6.11) uznano, iż nie ma potrzeby wymuszania większej zbieżności, gdyż dla  $\|\mathbf{r}\|_2 = 10^{-5}$  błąd względny uzyskany dla metody iteracyjnej jest na poziomie  $10^{-5}$  -  $10^{-8}$  co sprawia, że charakte-



rystyki filtru są niemal identyczne (Tab. 6.11, rys. 6.20).

$$Err_{sc_{11}} = \frac{sc_{11}^{PARDISO\_64} - sc_{11}^{PCG-V}}{sc_{11}^{PARDISO\_64}} \quad (6.10)$$

$$Err_{sc_{21}} = \frac{sc_{21}^{PARDISO\_64} - sc_{21}^{PCG-V}}{sc_{21}^{PARDISO\_64}} \quad (6.11)$$

### 6.3.1 Analiza układu w paśmie częstotliwości

Na rysunku 6.22 przedstawiono przyspieszenie ( $PCG-V_{(GPU+CPU)}$  względem referencyjnych implementacji na CPU) rozwiązania układu równań dla dwóch prawych stron (pobudzenie w dwóch wrotach) dla symulacji przeprowadzonej dla wielu częstotliwości  $f_m = \{1, 2, 3, 4, 10, 20, 50, 100\}$ . Charakterystyki testowanego filtru GSM dla symulacji przeprowadzonej dla stu częstotliwości ( $f_m = 100$ ), dla problemu testowego, w którym rozmiar macierzy wynosi  $N = 1998492$  wierszy, przedstawiono na rys. 6.21.

Z punktu widzenia rozwiązania układu równań dla kilku prawych stron i dla wielu częstotliwości w paśmie, dla każdej z implementacji należy uwzględnić dodatkowe czynniki wynikające z faktu, iż macierz  $\mathbf{A}$  z r. (6.1)-(6.2) ma ten sam układ elementów niezerowych, ale dla każdej częstotliwości posiada inne wartości niezerowe:

#### 1. Implementacja hybrydowa $PCG-V_{(GPU+CPU)}$

- dla pierwszej częstotliwości w paśmie z macierzy współczynników należy wydzielić macierze rzadkie  $\mathbf{M}_{ij}^E$ , a dla kolejnych częstotliwości należy podmienić wektory wartości niezerowych macierzy  $\mathbf{M}_{ij}^E$ ,
- stosunkowo kosztowny etap konwersji macierzy do formatu Sliced ELLR-T należy wykonać dla pierwszej częstotliwości, a dla kolejnych częstotliwości należy podmienić wektor niezerowych wartości macierzy  $\mathbf{M}_{ij}^E$  w formacie Sliced ELLR-T,
- przesłanie wszystkich wektorów reprezentowanych macierzy odbywa się wyłącznie dla pierwszej częstotliwości, dla kolejnych przesyłane są wyłącznie wektory wartości niezerowych.

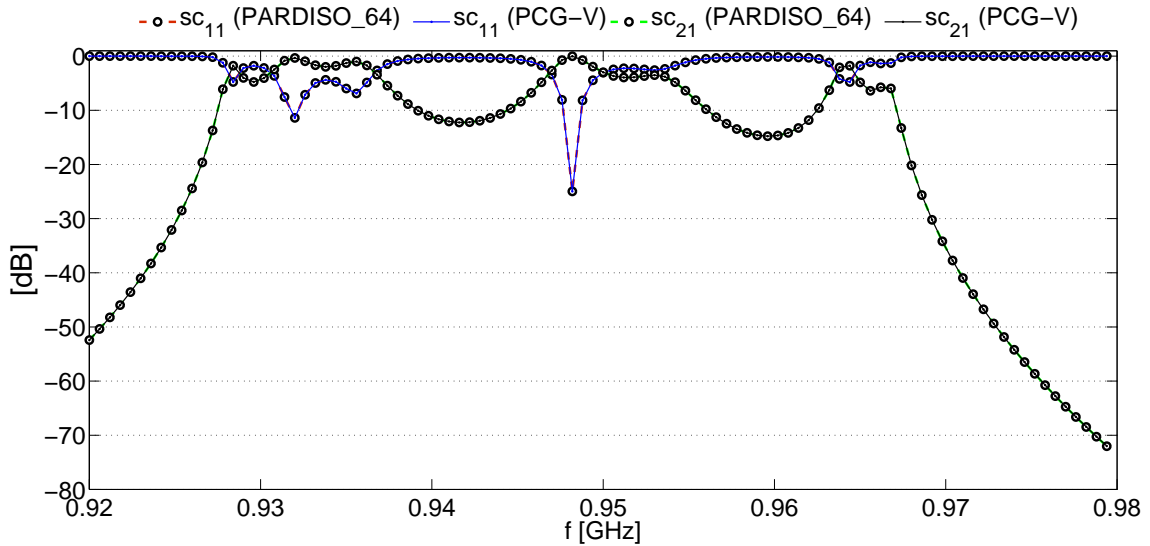
#### 2. Implementacja na CPU $PCG-V_{(CPU)}$

- dla pierwszej częstotliwości w paśmie z macierzy współczynników należy wydzielić macierze rzadkie  $\mathbf{M}_{ij}^E$ , dla kolejnych częstotliwości należy podmienić wektory wartości niezerowych macierzy  $\mathbf{M}_{ij}^E$ .

#### 3. Metoda bezpośrednia (PARDISO\_64)

- dla pierwszej częstotliwości należy wykonać faktoryzację symboliczną i numeryczną, a dla kolejnych częstotliwości należy wykonać wyłącznie faktoryzację numeryczną.

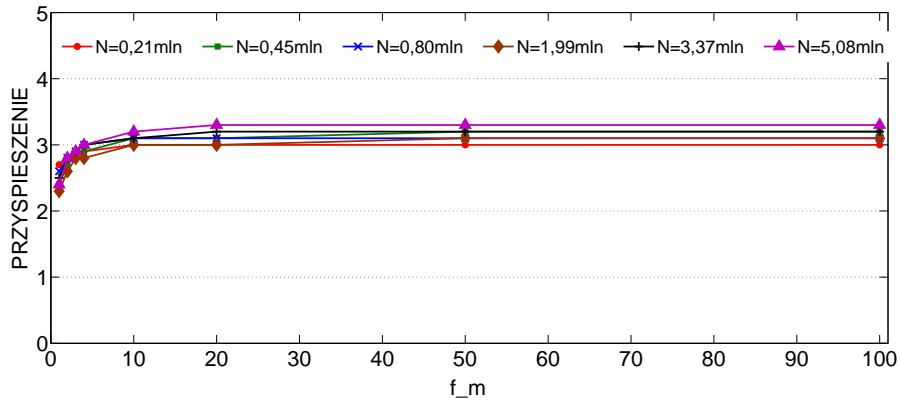
Dane zaprezentowane na rysunku 6.22a potwierdzają, że hybrydowa implementacja, wykorzystująca masywne zrównoleglenie wątków na GPU, ( $PCG-V_{(GPU+CPU)}$ ) pozwala uzyskać skrócenie czasu rozwiązania względem swojego odpowiednika ( $PCG-V_{(CPU)}$ ) na CPU bez względu na liczbę częstotliwości. Wraz ze wzrostem liczby częstotliwości narzut związany z dodatkowymi operacjami (konwersja macierzy z formatu CRS



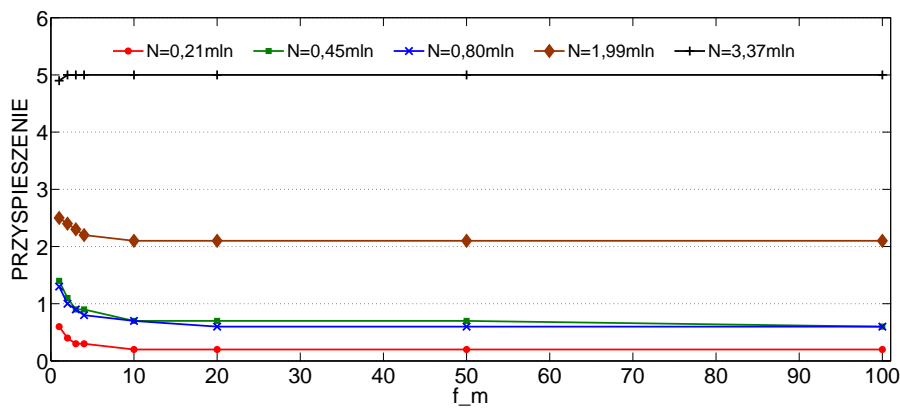
RYSUNEK 6.21: Charakterystyki (odbicia  $sc_{11}$  i transmisji  $sc_{21}$ ) filtra grzebieniowego w paśmie częstotliwości  $f_{bw}=920\text{--}980\text{ MHz}$ , liczba częstotliwości  $f_m=100$ .

do formatu Sliced ELLR-T, przesyłanie danych z CPU do GPU) ma coraz mniejszy udział w całkowitym czasie rozwiązania układów równań i przyspieszenia implementacji hybrydowej rosną (tendencja wzrostowa od wartości z czwartej kolumny do wartości z trzeciej kolumny Tab. 6.9).

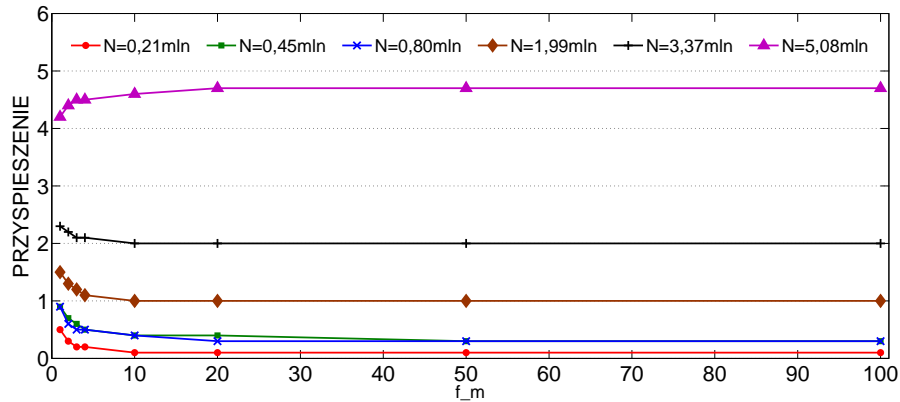
Dla platformy testowej przewaga hybrydowej implementacji  $PCG-V_{(GPU+CPU)}$  względem implementacji metody bezpośredniej (rys. 6.22a) topnieje wraz ze wzrostem liczby częstotliwości w paśmie dla problemów o rozmiarze mniejszym niż dwa miliony niewiadomych. Jest to spowodowane dwoma czynnikami. Po pierwsze czas rozwiązania układu równań liniowych (bez faktoryzacji) w metodzie bezpośredniej (Tab. 6.1) jest dużo mniejszy niż czas iteracyjnego rozwiązania układu równań  $PCG-V_{(GPU+CPU)}$  (Tab. 6.8). Dodatkowo w przypadku, jeżeli czas faktoryzacji numerycznej jest mniejszy niż czas rozwiązania układu w metodzie iteracyjnej (taka zależność występuje dla macierzy całościowych i trójkątnych górnych o rozmiarze  $N=\{213984, 455916, 806811\}$ ), wraz ze wzrostem liczby częstotliwości całkowity czas rozwiązania układów równań przy użyciu metody bezpośredniej będzie krótszy niż dla metody iteracyjnej. Zależność tę bardzo dobrze oddaje rys. 6.23a-6.23b, na którym zestawiono czasy przygotowania macierzy i czasy rozwiązania układu równań o rozmiarze  $N = 806811$ . W hybrydowej implementacji  $PCG-V_{(GPU+CPU)}$  na czas przygotowania składa się: wyznaczenie macierzy  $\mathbf{M}_{ij}^E$ , konwersja macierzy do formatu Sliced ELLR-T, przesłanie macierzy z CPU do GPU (przyp. dla  $f_{pt}>1$  powyższe przygotowanie sprowadza się do wyznaczenia, konwersji i przesłania elementów niezerowych macierzy). W implementacji bezpośredniego rozwiązania układu równań na CPU dla pierwszej częstotliwości należy wykonać faktoryzację symboliczną i numeryczną, a dla kolejnych częstotliwości ( $f_{pt}>1$ ) tylko faktoryzację numeryczną. Dla problemu  $N = 806811$  czas faktoryzacji numerycznej jest mniejszy niż czas rozwiązania układu równań  $PCG-V_{(GPU+CPU)}$  i z tego powodu sumaryczny czas rozwiązania układów równań dla wielu częstotliwości przy użyciu metody bezpośredniej jest krótszy. Dla problemów większych niż  $N = 806811$  faktoryzacja numeryczna jest bardzo kosztowna (rys. 6.23c-6.23d) i trwa dłużej niż rozwiązanie układu równań przy użyciu implementacji hybrydowej  $PCG-V_{(GPU+CPU)}$  i dlatego całkowity czas rozwiązania układów równań na wielu częstotliwościach przy użyciu metody iteracyjnej



(a)

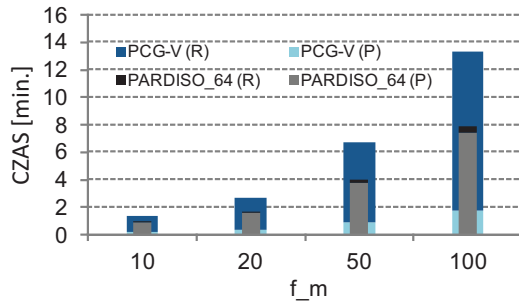
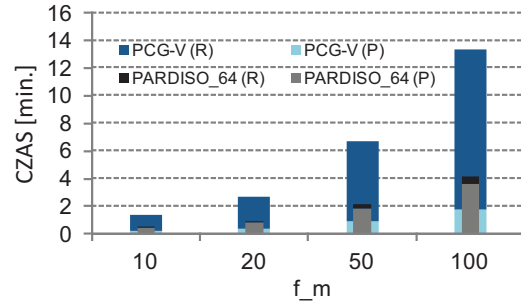
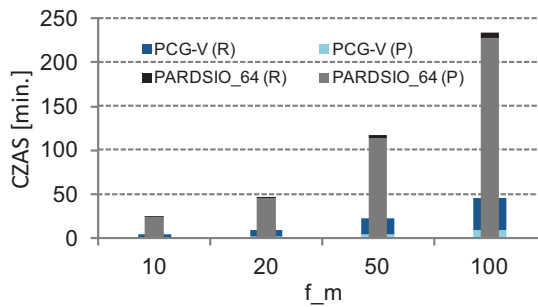
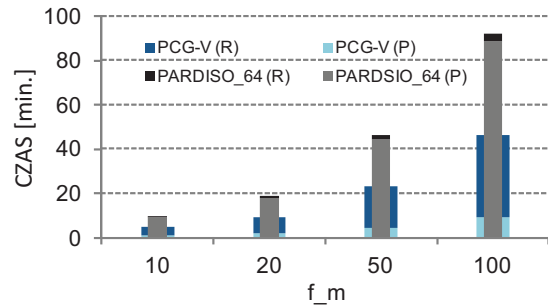


(b)



(c)

RYSUNEK 6.22: Skrócenie czasu dwukrotnego rozwiązania układu równań (wraz z uwzględnieniem czasu przygotowania macierzy do rozwiązania układu równań) uzyskane dla hybrydowej implementacji PCG- $V_{(\text{GPU}+\text{CPU})}$  względem (a) implementacji PCG- $V_{(\text{CPU})}$  na CPU (Intel MKL), (b) metody bezpośredniej na CPU (*PARDISO\_64*), w pamięci przechowywana cała macierz  $\mathbf{A}$ , (c) metody bezpośredniej na CPU (*PARDISO\_64*), w pamięci przechowywana górna połówka macierzy  $\mathbf{A}$ .

(a)  $N = 806\,811$  (CAŁ.)(b)  $N = 806\,811$  (SYM.)(c)  $N = 3\,370\,908$  (CAŁ.)(d)  $N = 3\,370\,908$  (SYM.)

RYSUNEK 6.23: Porównane czasu przygotowania danych (P) i dwukrotnego rozwiązania układu równań (R) dla  $f_m = \{10, 20, 50, 100\}$  częstotliwości. PCG-V to hybrydowa implementacja metody iteracyjnej z wielopoziomowym operatorem ściskającym o cyklu V (PCG-V<sub>(GPU+CPU)</sub>), Pardiso\_64 to bezpośrednia metoda rozwiązywania układu równań. GPU (Tesla K40c), CPU (Intel Xeon Sandy Bridge E5-2687W).

jest krótszy niż dla implementacji metody bezpośredniej.



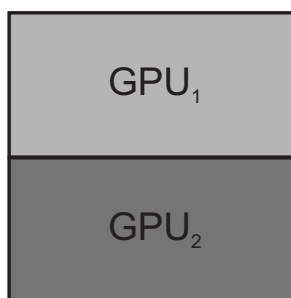
## Rozdział 7

# Strategie iteracyjnego rozwiązywania układów równań dla kilku GPU

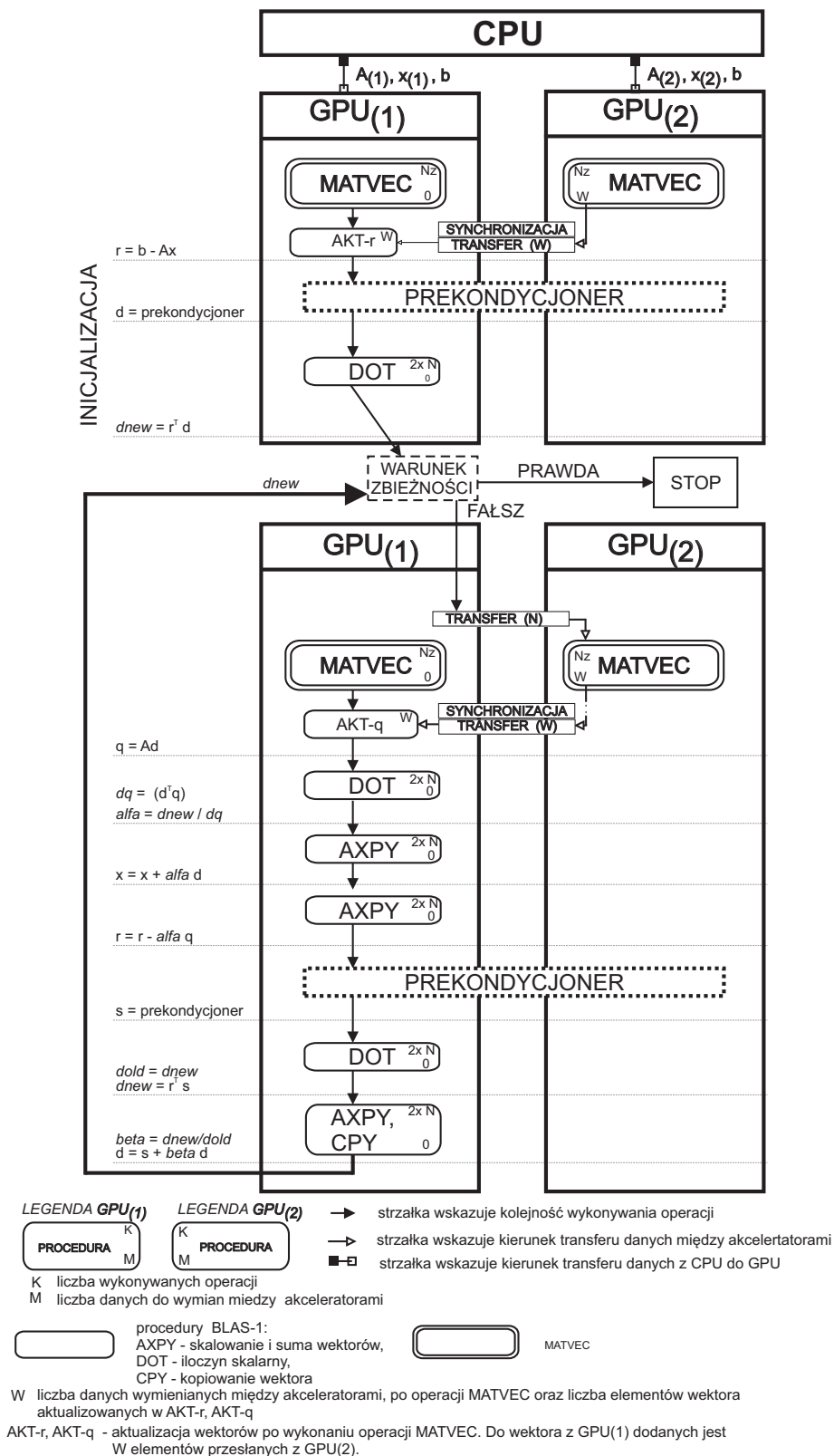
W rozdziale tym opisano implementacje metody gradientów sprzężonych dla kilku akceleratorów graficznych. Zwiększenie zasobów pamięciowych umożliwia rozwiązanie układów o większym rozmiarze. Dodatkowo efektywny podział obliczeń na akceleratorach i zapewnienie efektywnej komunikacji między akceleratorami może przyczynić się do skrócenia czasu rozwiązania układu równań. Po to aby uprościć dyskusję poniżej omówiono przypadki, gdy liczba akceleratorów wynosi  $K_{GPU} = 2$ .

### 7.1 Wariant z podziałem macierzy ze względu na wiersze

W pierwszej kolejności przedstawiono bardzo proste podejście do zagadnienia wykorzystania zwiększonych zasobów obliczeniowych, w którym tylko operacja mnożenia macierzy przez wektor jest wykonywana na dwóch akceleratorach graficznych, a pozostałe operacje metody iteracyjnej (tj. iloczyn skalarny, skalowanie wektorów) wykonywane są na jednym akceleratorze ( $GPU_1$ ). Taki algorytm może być zastosowany gdy macierz oryginalna  $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2]$  została podzielona na dwie macierze w porządku wierszowym, tzn. macierz  $\mathbf{A}_1$  przechowuje  $1, \dots, \frac{N}{2}$  wierszy, a macierz  $\mathbf{A}_2$  kolejne wiersze  $\frac{N}{2} + 1, \dots, N$  (rys. 7.1). Powyższy podział na podmacierze i rozdział obliczeń na kilka akceleratorów został wykorzystany w artykule [28, 36].



RYSunEK 7.1: Podział macierzy  $\mathbf{A}$  na podmacierze, które przechowywane są w pamięci  $GPU_1$  i  $GPU_2$ .



RYSUNEK 7.2: Schemat wykonania obliczeń na dwóch akceleratorach graficznych w metodzie gradientów sprzężonych z operatorem ściskającym (prekondycjoner). Wariant, w którym tylko operacja *matvec* jest wykonywana na dwóch akceleratorach, a pozostałe operacje wykonywane na akceleratorze  $GPU_1$ .



Przy powyższym rozdzieleniu macierzy na akceleratory graficzne należy zapewnić poprawność obliczeń operacji *matvec* w głównej pętli metody iteracyjnej:  $\mathbf{q} = \mathbf{A}\mathbf{d}$  (rys. 7.2). Po pierwsze wartości wektora  $\mathbf{d}$  muszą być dostępne dla obydwu akceleratorów przed rozpoczęciem obliczeń. Jeżeli dwa akceleratory graficzne umożliwiają komunikację P2P (ang. Direct per to peer (P2P) transfer), wtedy wektor  $\mathbf{d}$  jest przesyłany bezpośrednio z pamięci  $GPU_1$  do pamięci  $GPU_2$  (rozdział 3, rys. 3.3a). Inny wariant wymiany danych - kosztowniejszy z punktu widzenia opóźnień w dostępie do danych - przewiduje, że  $GPU_2$  odczytuje wartości wektora  $\mathbf{d}$  bezpośrednio z pamięci  $GPU_1$  (ang. Direct per to peer (P2P) access). Jeżeli dostęp P2P nie jest możliwy<sup>1</sup> to wektor przesyłany jest za pośrednictwem CPU (rozdział 3, rys. 3.3b), co wiąże się z większymi opóźnieniami niż komunikacja P2P. Po wykonaniu operacji mnożenia macierzy przez wektor, z akceleratora  $GPU_2$  połowa wektora wynikowego jest przesłana na akcelerator  $GPU_1$  i potem następuje aktualizacja wektora wynikowego (suma wektorów wynikowych obliczonych na dwóch akceleratorach graficznych). Po wykonaniu wyżej opisanych procedur na  $GPU_1$  przechowywany jest poprawny wynik operacji  $\mathbf{q} = \mathbf{A}\mathbf{d}$ .

Kod realizujący obliczenia w metodzie gradientów sprzężonych z wyżej opisanym wariantem operacji *matvec* zaprezentowano na Wydruku C.7 i rys. 7.2. Analizując schemat zaprezentowany na rys. 7.2 zauważyć można, iż z punktu widzenia wydajności obliczeniowej, oczekiwany zysk wynikający z zastosowania dwóch akceleratorów to redukcja o połowę liczby operacji zmiennoprzecinkowych w etapie mnożenia macierzy rzadkiej przez wektor (*matvec*). Z drugiej strony, opisana powyżej implementacja z wykorzystaniem dwóch akceleratorów w porównaniu z implementacją z poprzedniego rozdziału wymaga dodatkowych synchronizacji i kosztownej wymiany danych między akceleratorami przed ( $N$  elementów) i po ( $\frac{N}{2}$ ) wykonaniu operacji (*matvec*). Operacje na wektorach (iloczyn skalarny, skalowanie wektorów) wykonywane są analogicznie jak w wariancie na pojedynczym akceleratorze i wymagają tej samej liczby obliczeń zmiennoprzecinkowych.

## 7.2 Wariant z podziałem macierzy ze względu na dziedzinę obliczeniową

Mając na uwadze ograniczenia przedstawione w powyższym podejściu wynikające z kosztownej wymiany danych między akceleratorami, w niniejszej rozprawie opracowano algorytm, w którym w trakcie wykonania operacji *matvec*: wymiana danych między akceleratorami występuje wyłącznie po wykonaniu operacji *matvec*, porcja wymienianych danych między akceleratorami jest znacznie mniejsza niż w poprzednim algorytmie i operacje zmiennoprzecinkowe na wektorach rozdzielone są równomiernie na akceleratory graficzne. Uzyskanie wyżej opisanych właściwości możliwe było dzięki zastosowaniu jednej z dwóch poniższych procedur podziału macierzy  $\mathbf{A}$ :

- podział macierzy oparty na metodzie podziału grafu przeprowadzony **po** wykonaniu generacji macierzy  $\mathbf{A}$  (macierz reprezentowana jako graf) lub
- przeprowadzenie dekompozycji dziedziny obliczeniowej  $\Omega$  (siatka czworościanów reprezentowana jest jako graf) **przed** wykonaniem generacji macierzy.

<sup>1</sup>GPU z różnych architektur np.  $GPU_1$  - Fermi,  $GPU_2$  - Kepler, lub gdy akceleratory o tej samej architekturze nie są przypisane do jednego IOH (ang. Input Output Hub).

Pierwszy wariant jest uważany za nieefektywny. Czas podziału dużych macierzy rzadkich opartego na metodzie podziału grafu połączeń macierzy w stosunku do czasu realizacji operacji *matvec* jest bardzo długi [117]. Założenie to nie jest aktualne dla koncepcji dekompozycji dziedziny obliczeniowej  $\Omega$ , gdyż wówczas nie jest dzielona sama macierz, ale dziedzina obliczeniowa, czyli graf reprezentujący znacznie mniejszy zbiór czworościanów. W rozprawie zastosowano dekompozycję czworościanów, która jak dotąd nie była przedstawiana w literaturze w kontekście użycia w metodzie elementów skończonych i wykorzystania GPU. Przed generacją układów równań, siatka została podzielona przy użyciu metody podziału grafu (siatka oczek reprezentowana jako graf) przez funkcje z pakietu Metis [118], na tyle poddziedzin  $\Omega_i$ , ile jest GPU użytych w obliczeniach:

$$\Omega = \sum_{i=1}^{K_{GPU}} \Omega_i \quad (7.1)$$

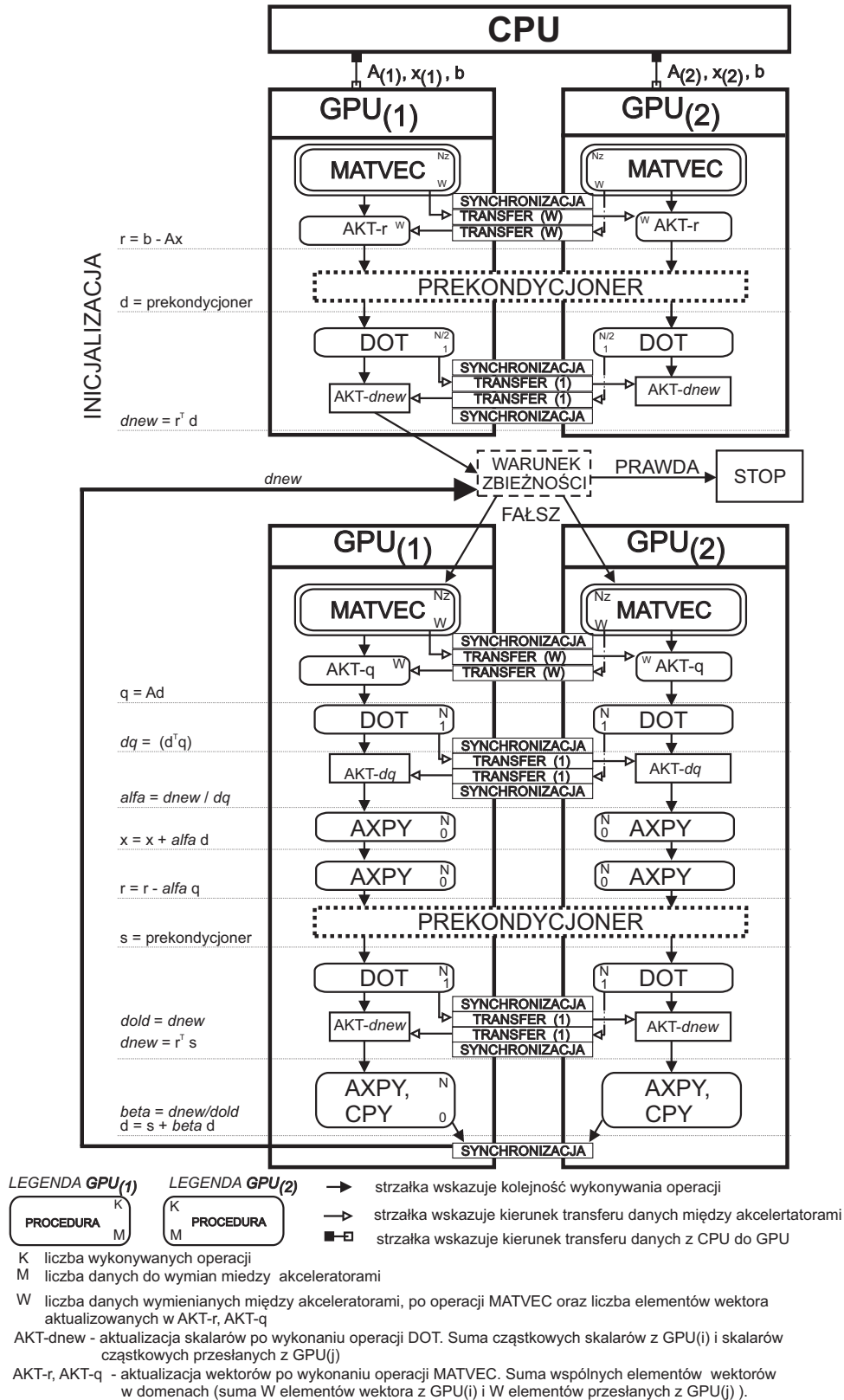
Dzięki temu każda poddziedzina  $\Omega_i$  zawiera średnio tyle samo oczek siatki oraz ograniczono liczbę wspólnych krawędzi w grafie. Podział dziedziny przed konstrukcją macierzy nie jest bardzo kosztowny czasowo (liczba czworościanów jest znacznie mniejsza niż rozmiar macierzy). Dodatkowy czas przeznaczony na ten etap jest bardzo mały i został „wchłonięty” przez czas generacji macierzy rzadkich. W przypadku podziału czworościanów na dwie ( $K_{GPU} = 2$ ) dziedziny obliczeniowe dwukrotnie uruchomiono proces generacji macierzy globalnych, najpierw dla grupy czworościanów należących do dziedziny  $\Omega_1$ , a potem należących do dziedziny  $\Omega_2$ . W efekcie otrzymano macierze sztywności  $\mathbf{S}_1$ ,  $\mathbf{T}_1$  oraz  $\mathbf{S}_2$ ,  $\mathbf{T}_2$ , przy użyciu których w następnym kroku obliczono macierze  $\mathbf{A}_1 = \mathbf{S}_1 - k_0^2 \mathbf{T}_1$  i  $\mathbf{A}_2 = \mathbf{S}_2 - k_0^2 \mathbf{T}_2$  odpowiednio dla  $\Omega_1$  i  $\Omega_2$ . Macierze  $\mathbf{A}_1$  i  $\mathbf{A}_2$  mają następujące właściwości. Po pierwsze, obydwie macierze mają ok.  $\frac{N}{2}$  niepustych wierszy i kolumn, które przynależą wyłącznie do danej macierzy (dziedziny) i w drugiej macierzy w tych wierszach i kolumnach występują zera<sup>2</sup>. Po drugie, macierze mają mniej niż 1% wspólnych wierszy, które są pokłosem wspólnych krawędzi dziedzin i w celu zapewnienia poprawności wykonywania operacji *matvec* są wymieniane między akceleratorami. Opisane powyżej właściwości macierzy pozwoliły na redukcję liczby elementów wektora wynikowego operacji *matvec*, które muszą być wymienione pomiędzy akceleratorami oraz na równomierne rozdzielenie obliczeń pomiędzy akceleratory graficzne w procesie iteracyjnym.

W (7.2) zdefiniowano macierz  $\mathbf{F}_{LW}$ , która informuje o liczbie wierszy ( $L_i$ ), które należą wyłącznie do macierzy  $\mathbf{A}_i$  oraz ( $W$ ) które są wspólne dla obydwu macierzy. Na przykładzie macierzy  $\mathbf{A}$  o rozmiarze  $N = 806811$  w (7.3) przedstawiono wartości  $L_i$  i  $W$ . Zauważyć można, iż liczba wspólnych wierszy  $W$  stanowi niewielki odsetek (0,22%) wszystkich wierszy macierzy  $\mathbf{A}$  (podobne proporcje występują dla wszystkich macierzy testowych z Tab. 4.1).

$$\mathbf{F}_{LW} = \begin{bmatrix} L_1 & W \\ W & L_2 \end{bmatrix} = \begin{bmatrix} \text{l. niepustych wierszy macierzy } \mathbf{A}_1 & \text{l. wspólnych wierszy } \mathbf{A}_1 \text{ i } \mathbf{A}_2 \\ \text{l. wspólnych wierszy } \mathbf{A}_1 \text{ i } \mathbf{A}_2 & \text{l. niepustych wierszy macierzy } \mathbf{A}_2 \end{bmatrix} \quad (7.2)$$

$$\mathbf{F}_{LW} = \begin{bmatrix} 407649 & 1752 \\ 1752 & 400914 \end{bmatrix} = \begin{bmatrix} 50,5\% & 0,22\% \\ 0,22\% & 49,7\% \end{bmatrix} \quad (7.3)$$

<sup>2</sup>Właściwość ta jest bardzo istotna z punktu widzenia implementacji operacji *matvec* z wykorzystaniem formatu Sliced ELLR-T i żeby uniknąć przechowywania w pamięci dodatkowych zerowych wierszy w etapie formowania reprezentacji Sliced ELLR-T należy pominąć te wiersze.



RYSUNEK 7.3: Schemat wykonania obliczeń w metodzie gradientów sprzężonych na dwóch akceleratorach. Wariant z dekompozycją dziedzin obliczeniowych. Operacja *matvec* oraz operacje na wektorach wykonywane równolegle na dwóch akceleratorach  $GPU_1$  i  $GPU_2$ .

Poniżej opisano implementację metody iteracyjnej, w której wykorzystano dekompozycję dziedzin obliczeniowych i obliczenia wykonano na ( $K_{GPU} = 2$ ) akceleratorach.

rach. W pierwszej kolejności opisano implementację mnożenia macierzy przez wektor na dwóch akceleratorach  $\mathbf{q}_i = f(\mathbf{A}_i, \mathbf{A}_j, \mathbf{d}_i, \mathbf{d}_j)$  (Wydruk C.8). Na każdym z  $GPU_i$  wykonywano obliczenia w wierszach, które pochodzą od czworoscianów wewnątrz  $i$ -tej dziedziny ( $id$ ) i w wierszach, które pochodzą od czworoscianów z krawędzi wspólnych dla sąsiednich dziedzin ( $iw$ ). Stosując nomenklaturę przyjętą w (7.2) określić można liczbę indeksów ( $id$ ) i ( $iw$ ) jako, odpowiednio,  $L_i - W$  i  $W$ . Po wykonaniu operacji *matvec* między akceleratorami przesyła się elementy wektorów wynikowych, których indeksy należą do zbioru ( $iw$ ). W wariancie z dekompozycją dziedzin przesłanie fragmentów wektorów odbywa się w **obydwu** kierunkach (Wydruk C.8, linie: 15-16). W rezultacie po wykonaniu operacji *matvec* i wymianie fragmentów wektorów wynikowych w pamięci  $GPU_i$  przechowywane są trzy wektory:

$\mathbf{q}_{(id)}$  - wektor wynikowy obliczony na  $GPU_i$ ,  $L_i - W$  niezerowych wartości zapisanych w komórkach z tablicy indeksów  $id$ ,

$\mathbf{q}_{(iw)}$  - wektor wynikowy obliczony na  $GPU_i$ ,  $W$  indeksów niezerowych wartości zapisanych w komórkach z tablicy indeksów  $iw$ ,

$\mathbf{q}_{(jw)}$  - wektor wynikowy obliczony na  $GPU_j$ ,  $W$  indeksów niezerowych wartości zapisanych w komórkach z tablicy indeksów  $iw$ ,

Kolejnym etapem jest aktualizacja wektora wynikowego, która polega na sumie  $W$  elementów wektorów wynikowych o indeksach ( $iw$ ) i ( $jw$ ). Warto podkreślić, że po tak wykonanych procedurach wektory wynikowe  $\mathbf{q}_i$  przechowywane na  $GPU_i$  w komórkach o indeksach  $\{id, iw\}$  mają te same wartości co wektory wynikowe obliczone w wariancie *matvec* na pojedynczym GPU<sup>3</sup>.

Aby rozdzielić procedury na wektorach w metodzie iteracyjnej na  $K_{GPU}$  akceleratorów graficznych, należało zmodyfikować implementację iloczynu skalarnego, tak by  $GPU_i$  wykonywało obliczenia wyłącznie na indeksach  $id, iw$  oraz aby na każdym z akceleratorów wynikowe skalary ( $\alpha, \beta$ ) były takie same jak ich odpowiedniki w wariancie na pojedynczym akceleratorze (Wydruk C.4, rys. 6.17). Spełnienie pierwszego warunku przyniosło korzyść w postaci równomiernego rozłożenia obliczeń pomiędzy akceleratory. Spełnienie drugiego warunku było konieczne, do odpowiedniego wyznaczenie wektora  $\mathbf{d}$  nowego kierunku poszukiwań oraz aktualizację wektora rozwiązania  $\mathbf{x}$  i wektora residuum  $\mathbf{r}$ . Obliczenia i wymiana danych pomiędzy akceleratorami w nowej implementacji iloczynu skalarnego  $dq = \mathbf{d}^T \mathbf{q}$  przeprowadzono podobnie do opisanej wcześniej operacji *matvec* z Wydruku C.8. Na każdym z  $GPU_i$  wykonywano obliczenia na wierszach o indeksach ( $id$ ) i ( $iw$ ) i w rezultacie otrzymano cząstkowe wyniki iloczynu skalarnego ( $dq_{(id)}, dq_{(iw)}$ ). Skalary będące rezultatem obliczeń na indeksach  $id$  wymieniono pomiędzy akceleratorami. W rezultacie, po wykonaniu powyższych procedur w pamięci  $GPU_i$  przechowywane były trzy skalary:

$dq_{(id)}$  - cząstkowy skalar obliczony dla  $(L_i - W)$  indeksów  $id$  wektorów  $GPU_i$ ,

$dq_{(iw)}$  - cząstkowy skalar obliczony dla  $(W)$  indeksów  $iw$  wektorów  $GPU_i$ ,

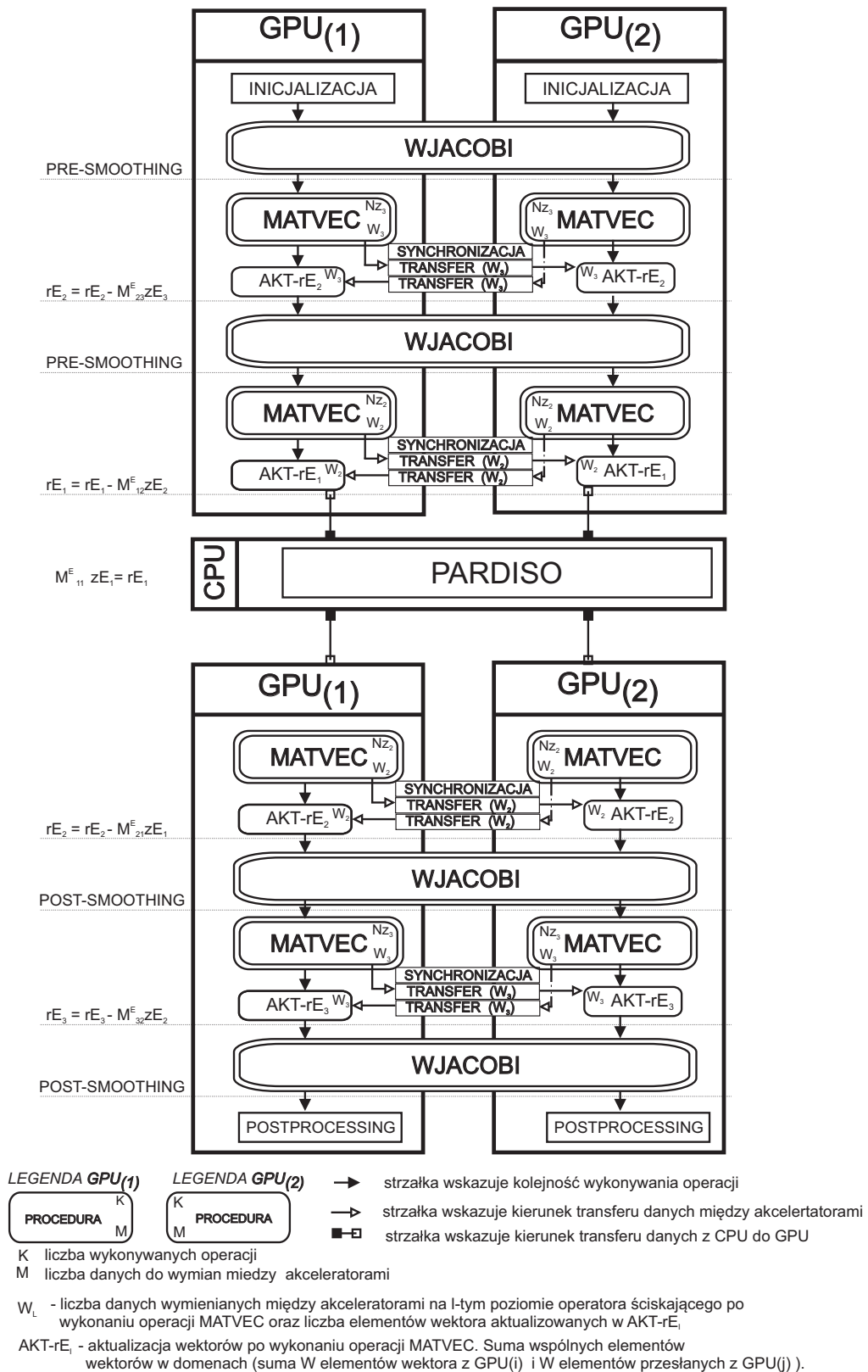
$dq_{(jd)}$  - cząstkowy skalar obliczony dla  $(W)$  indeksów  $jw$  wektorów  $GPU_j$ .

Ostatnim etapem wyznaczenia końcowego wyniku jest suma powyższych skalarów przeprowadzona niezależnie na akceleratorach. Tak wykonane procedury zagwarantują, że na każdym z akceleratorów  $dq_i = dq_j$  oraz równy jest iloczynowi skalarnemu otrzymanemu w implementacji na pojedynczym GPU (Wydruk C.4, mianownik z linii: 29).

Podsumowując, w odróżnieniu do algorytmu z (rys. 7.2), docelowy algorytm zaproponowany w niniejszej rozprawie ma następujące zalety (rys. 7.3):

- zredukowana liczba elementów ( $W$  z macierzy (7.2)), które są wymieniane między

<sup>3</sup>Ewentualne różnice są wynikiem innej kolejności wykonanych obliczeń.



RYSUNEK 7.4: Schemat wykonania obliczeń na dwóch akceleratorach graficznych w wielopoziomowym operatorze ściskającym o cyklu V. Wariant z dekompozycją dziedzin obliczeniowych: operacja *matvec* oraz operacje na wektorach wykonywane równolegle na dwóch akceleratorach  $GPU_1$  i  $GPU_2$ .



akceleratorami po wykonaniu operacji mnożenia macierzy rzadkiej przez wektor,

- równomierne rozłożenie operacji na wektorach na dwa akceleratory, tak by na  $GPU_i$  wykonany były operacje na  $L_i$  wartościach wektorów (7.2).

W porównaniu z implementacją z Wydruk C.7, w algorytmie bazującym na dekompozycji dziedziny występują dodatkowe synchronizacje pomiędzy akceleratorami i potrzeba wymiany danych (pojedyncze skalary) pomiędzy akceleratorami w trakcie operacji liczenia iloczynu skalarnego. Testy numeryczne wykazały, iż narzut wynikający z dodatkowej synchronizacji i wymiana pojedynczych danych był do zaakceptowania wobec zysków wynikających z redukcji ilości danych wymienianych pomiędzy akceleratorami w trakcie wykonywania operacji *matvec*. W przypadku zastosowania wielopoziomowego operatora ściskającego o schemacie V zadbane również o pełen rozdział obliczeń na dwa akceleratory (rys. 7.4). Operacje *matvec* wykonywano analogicznie jak w głównej pętli metody gradientów sprzężonych (Wydruk C.8). Także w tym przypadku potrzebna jest wymiana fragmentów wektorów wynikowych pomiędzy akceleratorami. Co więcej, porcja wymienianych danych ( $W_1, W_2, W_3$ ) jest mniejsza niż w pętli głównej metody gradientów sprzężonych ( $W$ ), gdyż macierze  $M_{ij(1)}^E$  i  $M_{ij(2)}^E$  są mniejszych rozmiarów niż macierze  $A_1$  i  $A_2$ . Nieznacznej modyfikacji wymaga także etap bezpośredniego rozwiązania układu równań na CPU na najniższym poziomie operatora ściskającego. Najpierw przesłane są prawe strony ( $r_{E1(1)}$  i  $r_{E1(2)}$ ) z  $GPU_1$  i  $GPU_2$  do pamięci CPU. Następnie wektory są zsumowane na CPU i dla tak powstałej prawej strony rozwiązuje się układ przy wykorzystaniu pakietu Intel MKL Pardiso. Po wykonaniu obliczeń wektor wynikowy rozwiązania bezpośredniego przesyłany jest na obydwa akceleratory i kolejne obliczenia operatora ściskającego ponownie są rozdzielone na dwa akceleratory (rys. 7.4).

Z racji tego, że obliczenia operacji *matvec* zostały wykonywane z wykorzystaniem formatu Sliced ELLR-T (p. 6.2.2), należało odpowiednio przygotować macierz. Wykonanie podstawowego przetwarzania wstępnego macierzy do formatu Sliced ELLR-T (podział macierzy na „plastry” o  $S = 32, 64, 128, 192, 256$  wierszy powodowałby, iż w pamięci przechowywane byłyby zerowe wiersze wynikające z dekompozycji dziedziny. Aby tego uniknąć, zmodyfikowano przetwarzanie wstępne macierzy tak, aby nie przechowywać zerowych wierszy. Drugą poważną modyfikacją było dodanie macierzy  $M_{11}^E = M_{11(1)}^E + M_{11(2)}^E$  na CPU, tak aby przy użyciu biblioteki Intel MKL Pardiso wykonać faktoryzację i rozwiązanie na pełnej macierzy z najniższego poziomu operatora ściskającego.

### 7.3 Podsumowanie strategii iteracyjnego rozwiązywania układów równań dla kilku GPU

W p 6.3 wskazano, iż operacja mnożenia macierzy rzadkiej przez wektor (*matvec*) ma decydujący wpływ na szybkość rozwiązania układu metodą iteracyjną. Z tego powodu najpierw pokazano i omówiono rezultaty otrzymane dla operacji *matvec* w zależności od implementacji na GPU:

- Sliced ELLR-T- format opisany w p. 6.2.2,
- Sliced ELLR-T\* - format opisany w p. 6.2.2, w którym zastosowano eliminację pustych wierszy,



TABELA 7.1: Czas (w milisekundach) operacji *matvec* w zależności od strategii podziału macierzy i liczby akceleratorów użytych w obliczeniach. DtoD (ang. Device to Device) - etap wymiany danych między akceleratorami graficznymi.

matvec implementacja	Sliced ELLR-T	Sliced ELLR-T	Sliced ELLR-T	Sliced ELLR-T*
$K_{GPU}$	1	2	2	2
strategia	-	$N/K_{GPU}$	$\Omega_i$	$\Omega_i$
1. matvec(DtoD-przed) [ms]	0,00	10,18	0,00	0,00
2. matvec( $\mathbf{q} = \mathbf{Ad}$ ) [ms]	50,0	33,46	28,30	24,32
3. matvec(DtoD-po) [ms]	0,00	4,02	0,04	0,04
4. matvec(aktualizacja) [ms]	0,00	0,74	0,41	0,50
(1.-4.) matvec [ms]	50,00	48,40	28,75	24,86
Przyspieszenie (2.)	1,00	1,49	1,77	2,06
Przyspieszenie (1.-4.)	1,00	1,03	1,74	2,00

TABELA 7.2: Czas iteracyjnego rozwiązania układu równań PCG-V<sub>(GPU+CPU)</sub> z wykorzystaniem  $K_{GPU} = \{1,2\}$  akceleratorów graficznych Tesla K20c.

<i>matvec</i>	Sliced ELLR-T	Sliced ELLR-T*	Przyspieszenie
$K_{GPU}$	1	2	2 vs. 1
$N = 1\,998\,492$	8,31	4,86	<b>1,71</b>
$N = 3\,370\,908$	14,70	8,73	<b>1,68</b>
$N = 5\,079\,849$	-	14,19	-

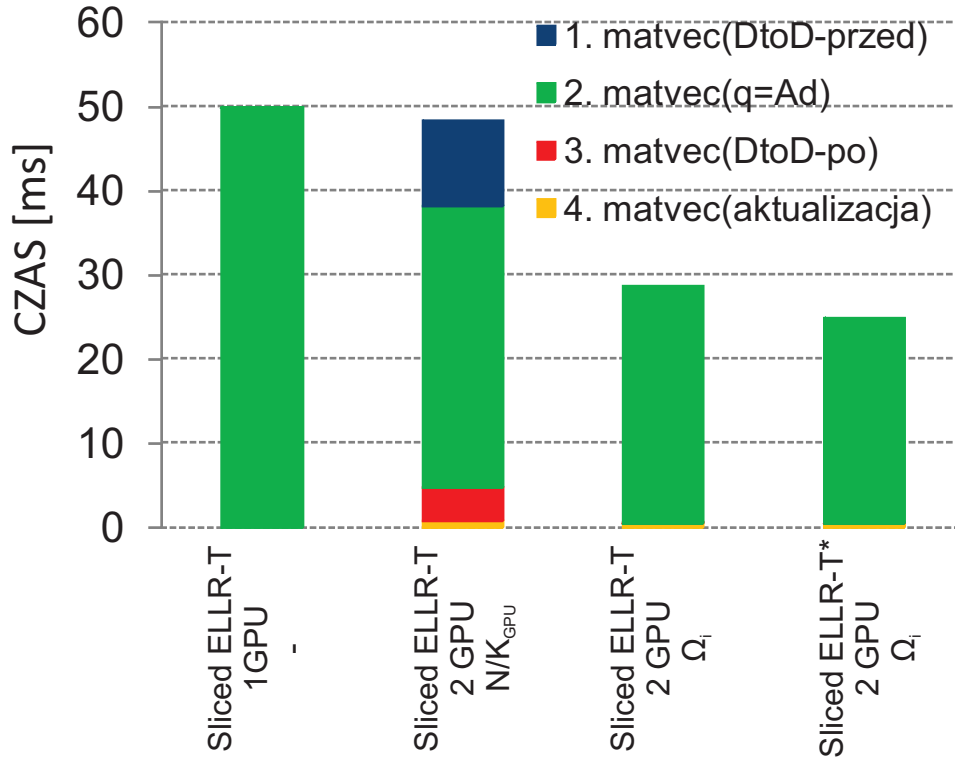
i zastosowanej strategii podziału macierzy:

- $N/K_{GPU}$  - podział macierzy ze względu na wiersze (rys. 7.1),
- $\Omega_i$  - podział macierzy ze względu na dziedziny obliczeniowe (rys. 7.2),

oraz liczby akceleratorów użytych w obliczeniach  $K_{GPU}$  (Tab. 7.1, rys.7.5). Testy zaprezentowane w tym punkcie zostały przeprowadzone na stacji roboczej (2x Tesla K20c, 1x Intel Xeon E5-2620 i 64 GB RAM)<sup>4</sup>.

W drugiej kolumnie Tab. 7.1 przedstawiono czasy referencyjnej implementacji na pojedynczym akceleratorze GPU (Tesla K20c), w której macierz przechowywana jest w formacie Sliced ELLR-T. W trzeciej kolumnie przedstawiono czasy wykonania operacji dla implementacji strategii opisanej w p. 7.1. Przed docelową operacją mnożenia  $\mathbf{q} = \mathbf{Ad}$  należy wykonać kosztowne przesłanie wektora  $\mathbf{d}$  ( $N$  elementów) z pamięci  $GPU_1$  do  $GPU_2$ . Po wykonaniu obliczeń z  $GPU_2$  do  $GPU_1$  przesyłana jest połowa ( $\frac{N}{2}$  elementów) wektora wynikowego  $\mathbf{q}$ , a następnie aktualizowane są wartości wektora  $\mathbf{q}$  na  $GPU_1$ . Co prawda w implementacji tej dla operacji  $\mathbf{q} = \mathbf{Ad}$  uzyskano ok. 1,5-krotne przyspieszenie względem wariantu referencyjnego, ale przez duży narzut wynikający z komunikacji między akceleratorami i aktualizację wektora wynikowego (Tab. 7.1, kolumna druga, operacje 1.,3.,4.), całkowite przyspieszenie wynosi zaledwie 1,03.

<sup>4</sup>Stacja robocza udostępniona autorowi w trakcie stażu pod kierunkiem profesora M. Okoniewskiego w ramach projektu NCN Etiuda (Uniwersytet w Calgary, Kanada), Maj-Sierpień 2014.



RYSUNEK 7.5: Czas (w milisekundach) operacji *matvec* w zależności od strategii podziału macierzy i liczby akceleratorów użytych w obliczeniach. DtoD - etap wymiany danych między akceleratorami graficznymi.

W czwartej kolumnie przedstawiono czasy otrzymane dla strategii podziału macierzy opisanej w p. 7.2 i gdy macierz przechowywana jest w formacie Sliced ELLR-T (bez eliminacji zerowych wierszy wynikających z podziału na dziedziny obliczeniowe). W podejściu tym zakłada się rozdział obliczeń w operacjach na wektorach na  $K_{GPU}$  akceleratorów i w procesie iteracyjnym nie ma potrzeby komunikacji między akceleratorami bezpośrednio przed  $\mathbf{q} = \mathbf{A}\mathbf{d}$ . Po wykonaniu obliczeń na macierzy rzadkiej akceleratorów wymieniają między sobą  $W_i$  wspólnych elementów (r. (7.2)). Dzięki temu, że stacja robocza umożliwia komunikację P2P oraz, że macierze mają bardzo mało wspólnych wierszy to koszt komunikacji między akceleratorami jest ok. dziesięciokrotnie mniejszy niż w przypadku poprzedniego wariantu, w którym przesyła się połowę wektora wynikowego. Aktualizacja wektorów również trwa krócej, gdyż odbywa się niezależnie na dwóch akceleratorach i aktualizowanych jest  $W_i$  elementów. Dzięki temu, że narzut spowodowany komunikacją między akceleratorami i aktualizacją wektorów wynikowych jest bardzo mały (operacje 3.,4. Tab. 7.1), to przyspieszenie uzyskane dla tego wariantu wynosi ok. 1,74 względem implementacji referencyjnej na pojedynczym GPU. W ostatniej kolumnie przedstawiono czasy uzyskane dla wariantu, w którym macierz przechowywana jest w formacie Sliced ELLR-T, ale w etapie przetwarzania wstępnego wyeliminowane zostały zerowe wiersze (w pamięci przechowywana tablica  $\mathbf{I}_{org}$  informująca o oryginalnym indeksie niezerowych wierszy) i w tabeli implementację oznaczono jako Sliced ELLR-T\*. Dla zoptymalizowanej implementacji zmniejszył się czas operacji  $\mathbf{q} = \mathbf{A}\mathbf{d}$ . Jeśli występują puste wiersze, wątki przy zapisie do wektora wynikowego  $\mathbf{q}$  „skaczą” po adresach. Po wyeliminowaniu pustych wierszy (w etapie wstępnego przetworzenia macierzy) wątki zapisują dane do kolejnych adresów  $\mathbf{q}$  co

TABELA 7.3: Efektywność przyspieszenia implementacji metody iteracyjnej PCG-V na dwóch akceleratorach graficznych wyrażona przez prawo Amdahla r. (7.4).

L. wierszy	F	$K=K_{GPU}$	$P_{Amdahl}$	$Eff = \frac{P}{P_{Amdahl}}$
1 998 492	0,98	2	1,96	$\frac{1,71}{1,96} = 87\%$
3 370 908	0,97	2	1,94	$\frac{1,68}{1,94} = 87\%$

powoduje, że częściej niż w poprzedniej implementacji występują zgrupowane zapisy do pamięci globalnej. Nieznacznie zwiększył się czas aktualizacji wektorów, ponieważ zapis do wektora wynikowego wymaga dodatkowego odczytu indeksu z tablicy *Iorg*. Zastosowane modyfikacje pozwoliły na dwukrotną redukcję czasu wykonania operacji *matvec* względem implementacji referencyjnej na pojedynczym GPU.

Zastosowanie schematu z rys. 7.3-7.4, formatu Sliced ELLR-T\* dedykowanego wykonaniu operacji *matvec* na GPU i równomierny rozdział obliczeń w operacjach na wektorach na dwóch akceleratorach, skróciło ok. 1,7 krotnie czas rozwiązania układu równań (Tab. 7.2). Analiza czasu wykonania poszczególnych etapów pozwoliła określić, dlaczego otrzymane przyspieszenie metody iteracyjnej (1,7x) jest mniejsze niż *matvec* (2,0x) z Tab. 7.1. Po pierwsze obydwie implementacje zawierają rozwiązanie układu równań na dolnym poziomie operatora ściskającego. W przypadku obliczeń wykonanych na dwóch akceleratorach procentowy udział rozwiązania na dolnym poziomie to ok. 12%-20% (podczas gdy dla implementacji na pojedynczym akceleratorze udział ten wynosi ok. 8%-16%). Hipotetyczne przyspieszenie uzyskane przy założeniu, że czasu rozwiązania układu równań na najniższym poziomie operatora ściskającego nie jest uwzględniany wynosi 1,82<sup>5</sup>. Ponadto w Tab. 7.3 przedstawiono wyniki, które weryfikują wydajność iteracyjnego rozwiązania przy użyciu implementacji PCG-V(GPU,CPU) na dwóch akceleratorach graficznych. Do weryfikacji skorzystano z prawa Amdahla, które pozwala określić maksymalne przyspieszenie w przypadku, gdy w implementacji można wydzielić dwa składniki: F - obliczenia które zostały zrównoleglone na GPU (operacje *matvec* i operacje na wektorach) i 1-F - obliczenia nie zrównoleglone na GPU (rozwiązanie układu równań na dolnym poziomie wielopoziomowego operatora ściskającego):

$$P_{Amdahl} = \frac{1}{(1 - F) + \frac{F}{K}} \quad (7.4)$$

gdzie  $K$  określa przyspieszenie czynnika F wynikające z użycia  $K_{GPU}$  akceleratorów graficznych. Dane przedstawione w Tab. 7.3 wskazują, że opracowana implementacja pozwala uzyskać 87% efektywnego przyspieszenia wynikającego z prawa Amdahla. Przyczynami nie pozwalającymi uzyskać większej efektywności są dodatkowe synchronizacje jakie należy wykonać w procesie iteracyjnym: w operacji iloczynu skalarnego, po wykonaniu ważonej metody Jacobiego i po wykonaniu obliczeń w każdej iteracji PCG-V (rys. 7.3-7.4).

Rozmiar pamięci (5GB) akceleratora graficznego K20c nie pozwolił na rozwiązanie układu równań o 5 mln niewiadomych. Zastosowanie dwóch akceleratorów pozwoliło przezwyciężyć ograniczenie pamięciowe. W Tab. 7.4 przedstawiono zmniejszenie zapotrzebowania na pamięć uzyskane dzięki zwiększeniu zasobów pamięciowych. Celowo

<sup>5</sup>Wykorzystując czas rozwiązania układu równań z Tab. 7.2 dla problemu  $N = 1998492$  i to, że sumaryczny czas rozwiązań układów równań na dolnym poziomie w procesie iteracyjnym wynosi 0,650 sekundy, hipotetyczne przyspieszenie bez uwzględniania obliczeń na CPU wynosi:  $\frac{8,310-0,650}{4,857-0,650} = 1,82$ .

TABELA 7.4: Zapotrzebowanie na pamięć akceleratora graficznego. Kolumny 2-4: wektory i macierz  $\mathbf{A}$  przechowywana w formacie Sliced ELLR-T (PCG- $V_{GPU+CPU}$  uruchomiony na pojedynczym GPU). Kolumny 5-7: wektory i macierz  $\mathbf{A}_i$  z dziedziny  $\Omega_i$  przechowywane na każdym z akceleratorów (macierz rzadka przechowywana w formacie Sliced ELLR-T\*, PCG- $V_{GPU+CPU}$  uruchomiony na dwóch GPU).

matvec	Sliced ELLR-T	Sliced ELLR-T	Sliced ELLR-T	Sliced ELLR-T*	Sliced ELLR-T*	Sliced ELLR-T*
$K_{GPU}$	1	1	1	2	2	2
N	macierz $\mathbf{A}$ [GB]	wektory [GB]	$\mathbf{A}$ , wektory [GB]	$\mathbf{A}_i$ [GB]	wektory [GB]	$\mathbf{A}_i$ , wektory [GB]
1 998 492	1,96	0,24	2,19	1,03	0,29	1,32
3 370 908	3,31	0,40	3,71	1,67	0,49	2,16
5 079 849	5,13	0,61	5,74	2,60	0,74	3,33

rozdzielono zapotrzebowanie na pamięć macierzy i wektorów. W przypadku implementacji na dwóch akceleratorach i przy zastosowaniu strategii podziału dziedziny obliczeniowej praktycznie dwukrotnie zredukowano zapotrzebowanie na pamięć macierzy rzadkich na każdym z akceleratorów (piąta kolumna). Inaczej jest w przypadku wektorów. Zapotrzebowanie jest większe ponieważ potrzebne są dodatkowe wektory pomocnicze wykorzystywane do przechowywania danych tymczasowych w trakcie komunikacji między akceleratorami graficznymi. **Podsumowując zastosowana strategia podziału macierzy ze względu na dziedziny obliczeniowe pozwoliła na ok. 1,7-krotne skrócenie czasu rozwiązania układu równań i ok. 1,7 zwiększyło rozmiar układu, który może być rozwiązany.**

## Rozdział 8

# Pełny cykl analizy z użyciem MES na CPU i GPU

W rozdziale tym zweryfikowano wpływ skrócenia czasu obliczeń konstrukcji macierzy rzadkich i rozwiązywania układu równań uzyskanego dzięki zastosowaniu akceleratora graficznego na skrócenie czasu analizy filtru grzebieniowego 9 rzędu pracującego w paśmie GSM (920-980 MHz) przy użyciu MES (rys. 4.1). W poniższych rozważaniach skupiono się na zagadnieniach o dużych rozmiarach, w których generowane macierze mają co najmniej 1998492 wierszy (Tab. 4.1).

Analiza filtru grzebieniowego odbywa się w następujący sposób. Najpierw wczytywana jest siatka oczek wygenerowana w programie NETGEN [58]. Następnie wykonywane są obliczenia związane z określeniem warunków brzegowych. W dalszej kolejności generowane są macierze sztywności i bezwładności. Później wyznaczana jest macierz reprezentująca pobudzenia we wrotach układu i dla  $f_m$  częstotliwości rozwiązywany jest układ równań liniowych. Testy referencyjne na CPU wykonano dla częściowo zoptymalizowanego kodu wykorzystywanego w symulatorze zagadnień elektromagnetycznych [90, 91]: w etapie generacji macierzy wykorzystano bibliotekę Intel MKL, w etapie rozwiązywania układu równań pakiet Intel MKL Pardiso (PARDISO\_64 w wariantcie wykorzystującym symetrię macierzy  $\mathbf{A}$ ), a w pozostałych etapach, tam gdzie było to możliwe, obliczenia zrównoleglono przy użyciu dyrektyw OpenMP (ang. Open Multi-Processing) [119].

TABELA 8.1: Czasy wykonania etapów i ich procentowy udział w analizie filtru grzebieniowego przy wykorzystaniu MES. Liczba częstotliwości w paśmie  $f_m = 1$ . Obliczenia wykonane wyłącznie na CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków).

N	1998492		3370908		5079849	
Etap	[s]	(%)	[s]	(%)	[s]	(%)
Odczyt siatki elementów	1,1	1%	1,2	1%	2,1	1%
Konstrukcja warunków brzegowych	2,0	2%	3,0	2%	4,1	1%
Generacja macierzy	48,1	55%	84,4	46%	114,9	31%
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	2,4	3%	4,6	3%	13,7	4%
Rozwiązanie układów równań	33,8	39%	89,2	49%	234,8	64%
MES	87,3	100%	182,3	100%	369,5	100%

TABELA 8.2: Czasy wykonania etapów i ich procentowy udział w analizie filtru grzebieniowego przy wykorzystaniu MES. Liczba częstotliwości w paśmie  $f_m = 1$ . W etapach generacji macierzy i iteracyjnego rozwiązania układu równań obliczenia wykonane na GPU (Tesla K40c), pozostałe obliczenia wykonane na CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków).

N	1998492		3370908		5079849	
Etap	[s]	(%)	[s]	(%)	[s]	(%)
Odczyt siatki elementów	1,1	3%	1,2	2%	2,1	2%
Konstrukcja warunków brzegowych	2,0	6%	3,0	5%	4,1	5%
Generacja macierzy	5,9	17%	9,7	17%	11,7	13%
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	2,4	7%	4,6	8%	13,7	16%
Rozwiązanie układów równań	22,3	66%	38,7	68%	55,4	64%
MES	33,7	100%	57,2	100%	87,0	100%

TABELA 8.3: Czasy wykonania etapów i ich procentowy udział w analizie filtru grzebieniowego przy wykorzystaniu MES. Liczba częstotliwości w paśmie  $f_m = 100$ . Obliczenia wykonane wyłącznie na CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków).

N	1998492		3370908		5079849	
Etap	[s]	(%)	[s]	(%)	[s]	(%)
Odczyt siatki elementów	1,1	0%	1,2	0%	2,1	0%
Konstrukcja warunków brzegowych	2,0	0%	3,0	0%	4,1	0%
Generacja macierzy	48,1	3%	84,4	1%	114,9	1%
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	2,4	0%	4,6	0%	13,7	0%
Rozwiązanie układów równań	1484,4	97%	5536,0	98%	17453,1	99%
MES	1537,9	100%	5629,1	100%	17587,8	100%

W Tab. 8.1 zebrano czasy wykonania poszczególnych etapów gdy analiza przeprowadzona jest dla jednej częstotliwości  $f_m = 1$  na CPU. Czasy poszczególnych etapów potwierdzają, że generacja macierzy i rozwiązanie układu równań to najbardziej kosztowne obliczeniowo etapy analizy filtru mikrofalowego przy użyciu MES. Obydwie operacje zajmują łącznie ponad 96% całego czasu analizy. Czasy otrzymane dla analizy filtru mikrofalowego wspartej przez zastosowanie akceleratora graficznego gdy  $f_m = 1$  zebrano w Tab. 8.2. Dziesięciokrotne skrócenie czasu generacji macierzy MES spowodowało, że dla problemu o największym rozmiarze etap ten stał się mniej kosztowny niż wyznaczenie macierzy pobudzeń we wrotach układu. Logicznym następnym krokiem jest więc przyspieszenie wyznaczenia macierzy pobudzeń poprzez wykorzystanie GPU. Wraz ze wzrostem rozmiaru problemu, gdy rozwiązanie metodą iteracyjną jest szybsze niż faktoryzacja numeryczna na CPU, zastosowanie akceleratora graficznego pozwala na większe skrócenie czasu obliczeń w fazie rozwiązania. Dla największego problemu ( $N=5079849$ ) zastosowanie akceleratora graficznego pozwoliło na ok. 4,2 krotne skrócenie czasu analizy MES (z 370 sekund do 87 sekund).

Aby określić charakterystyki transmisji i odbicia filtrów w paśmie, analizę należy przeprowadzić dla wielu częstotliwości. W analizowanym przypadku, gdy parametry materiałowe ośrodka nie zależą od częstotliwości, wczytanie siatki, określenie warun-



TABELA 8.4: Czasy wykonania etapów i ich procentowy udział w analizie filtru grzebiennowego przy wykorzystaniu MES. Liczba częstotliwości w paśmie  $f_m = 100$ . W etapach generacji macierzy i iteracyjnego rozwiązania układu równań obliczenia wykonane na GPU (Tesla K40c), pozostałe obliczenia wykonane na CPU (Intel Xeon Sandy Bridge E5-2687W, 8 wątków).

N	1998492		3370908		5079849	
Etap	[s]	(%)	[s]	(%)	[s]	(%)
Odczyt siatki elementów	1,1	0%	1,2	0%	2,1	0%
Konstrukcja warunków brzegowych	2,0	0%	3,0	0%	4,1	0%
Generacja macierzy	5,9	0%	9,7	0%	11,7	0%
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	2,4	0%	4,6	0%	13,7	0%
Rozwiązanie układów równań	1539,1	99%	2784,8	99%	3732,2	99%
MES	1550,5	100%	2803,4	100%	3763,7	100%

ków brzegowych, generacja macierzy sztywności i wyznaczenie macierzy pobudzeń w wrotach wykonywane są jednokrotnie. Rozwiązanie układu równań wykonywane jest dla  $f_m$  częstotliwości i dla każdego z wrót (r. (6.2)). W Tab. 8.3 zebrano czasy wykonania poszczególnych etapów MES na CPU dla  $f_m = 100$  częstotliwości w paśmie 920-980 MHz. Charakterystyki odbicia i transmisji uzyskane dla  $f_m = 100$  przedstawiono na rys. 6.21. W analizie w szerokim paśmie czas rozwiązania układu równań jest czynnikiem dominującym. Procentowy udział tego etapu wynosi ponad 97% czasu analizy przeprowadzonej na CPU i rośnie wraz ze wzrostem rozmiaru problemu. Analiza z użyciem MES, w której obliczenia wykonywane są na CPU trwała w zależności od liczby oczek, na które zostały podzielony analizowany filtr od ok. 25 minut do niespełna pięciu godzin. Czasy otrzymane dla analizy filtru mikrofalowego wspartej przez zastosowanie akceleratora graficznego w paśmie częstotliwości zebrano w Tab. 8.4. Przy wzroście liczby punktów częstotliwości maleje znaczenie użycia GPU do generacji macierzy. Powtórzyć można wniosek, że dla problemów o dużym rozmiarze rozwiązanie układu równań metodą iteracyjną jest szybsze niż faktoryzacja numeryczna na CPU i zastosowanie GPU pozwala na znaczące skrócenie czasu obliczeń w fazie rozwiązania. W efekcie dzięki zastosowaniu akceleratora graficznego dla problemu o rozmiarze 3370908 uzyskano 2-krotne skrócenie czasu analizy z wykorzystaniem MES (użycie akceleratora graficznego pozwoliło skrócić czas obliczeń z 94 do 47 minut). Dla większego problemu (5079849 niewiadomych) analiza w której część obliczeń wykonywane jest na akceleratorze graficznym jest jeszcze bardziej efektywna, gdyż uzyskano ok. 4,7 krotne skrócenie czasu obliczeń (z niespełna pięciu godzin do ok. godziny).

W ramach podsumowania określono etapy, dla których wysiłek poniesiony na opracowanie nowych zrównoleglonych algorytmów na GPU przyniósł szczególną korzyść. Dla przyjętych w tej rozprawie sformułowań, dla których elementy macierzy nie zależą od częstotliwości i generacja macierzy odbywa się dla pierwszej częstotliwości a rozwiązanie układu równań wykonywane jest dla wielu częstotliwości, skrócenie czasu całego cyklu analizy MES zostało osiągnięte dzięki zrównoleglonym obliczeniom w iteracyjnej metodzie rozwiązywania układu równań (PCG- $V_{(\text{GPU}+\text{CPU})}$ ). Trud poniesiony na zrównoleglenie generacji macierzy sztywności i bezwładności ma duże znaczenie jeżeli dla każdej częstotliwości są generowane nowe macierze sztywności i bezwładno-

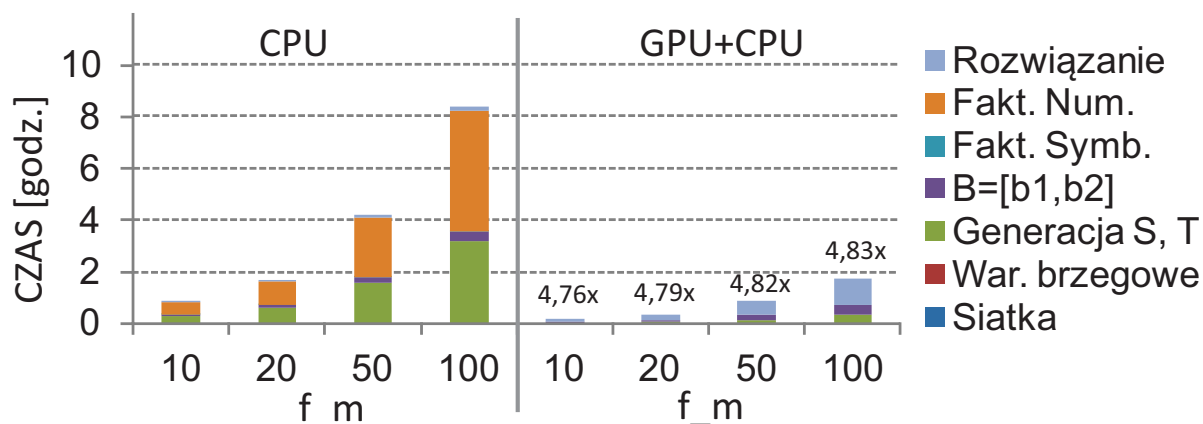
TABELA 8.5: Hipotetyczne scenariusze pełnego cyklu MES w przypadku, gdy elementy macierzy sztywności i bezwładności są zależne od częstotliwości.

Liczba wykonań	I	II
Odczyt siatki elementów	1	1
Konstrukcja warunków brzegowych	1	1
Generacja macierzy	$f_m$	$f_m$
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	$f_m$	1
Rozwiązanie układów równań	$2 \cdot f_m$	$2 \cdot f_m$

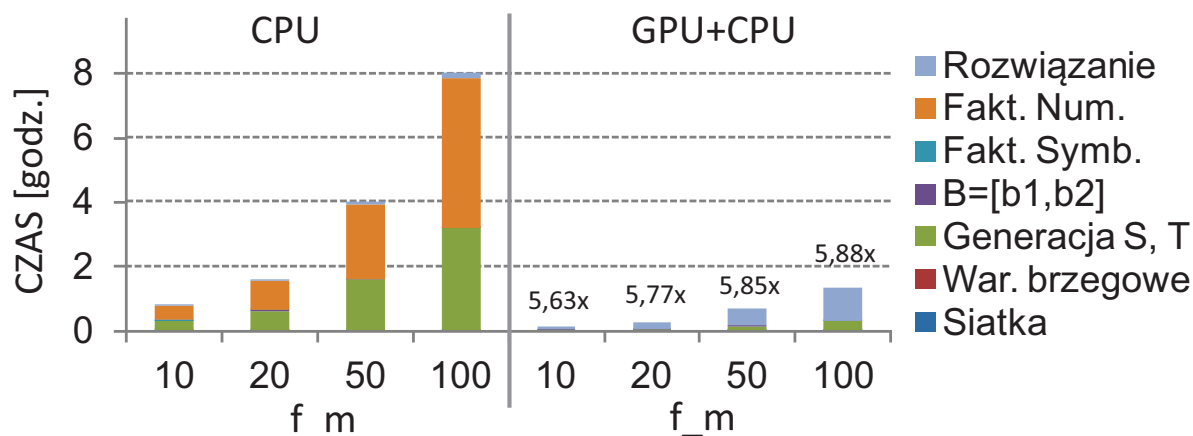
ści. Taka sytuacja ma miejsce gdy wartości macierzy zależą od częstotliwości i poniżej przedstawiono rozważania dla dwóch scenariuszy z Tab. 8.5. Na rys. 8.1 i 8.2 przedstawiono estymowane czasy wykonania pełnego cyklu MES dla odpowiednio scenariusza I i II gdy liczba częstotliwości w paśmie  $f_m = \{10, 20, 50, 100\}$  dla problemu o rozmiarze  $N=5079849$ . W obliczeniach wykorzystano czasy uzyskane dla poszczególnych etapów z Tab. 6.10, Tab. 8.1, Tab. 8.2. Zauważyć, można że wysiłek włożony w zrównoleglenie generacji macierzy znacząco redukuje czas analizy MES dla obydwu scenariuszy i wraz z hybrydową implementacją iteracyjnej metody rozwiązywania układu równań umożliwia znaczące (ok. 4,8 i 5,9 krotne) skrócenie czasu całej analizy MES (odpowiednio dla scenariusza I i II). Analizując Tab. 8.6 (scenariusz I) gdzie przedstawiono estymowane czasy etapów analizy MES i ich procentowy udział widać, że czynnikiem, który w obecnej implementacji znacząco wpływa na czas symulacji jest czas wyznaczenia macierzy pobudzeń i w przyszłości podjęta zostanie próba zrównoleglenia tego etapu.

TABELA 8.6: Estymowany czas wykonania poszczególnych etapów analizy MES wg. scenariuszy I i II. CPU - obliczenia wykonane wyłącznie na CPU, GPU+CPU - obliczenia generacji macierzy i rozwiązywania układu równań wykonane na GPU, pozostałe etapy wykonane na CPU.

Scenariusz	I				II			
	CPU		GPU+CPU		CPU		GPU+CPU	
Implementacja								
Etapy	[godz.]	[%]	[godz.]	[%]	[godz.]	[%]	[godz.]	[%]
Odczyt siatki elementów	0,00	0%	0,00	0%	0,0	0%	0,00	0%
Konstrukcja warunków brzegowych	0,00	0%	0,00	0%	0,0	0%	0,00	0%
Generacja macierzy	3,19	38%	0,33	19%	3,2	40%	0,33	24%
Wyznaczenie pobudzeń $\mathbf{B}=[\mathbf{b}_1, \mathbf{b}_2]$	0,38	5%	0,38	22%	0,0	0%	0,00	0%
Rozwiązanie układów równań	4,85	58%	1,04	59%	4,8	60%	1,04	76%
MES	8,42	100%	1,74	100%	8,0	100%	1,37	100%



RYSUNEK 8.1: Estymowany czas wykonania pełnego cyklu analizy MES wykonanego zgodnie ze scenariuszem I. Liczba częstotliwości w paśmie wynosi  $f_m = \{10, 20, 50, 100\}$ . Rozmiar generowanej macierzy  $N=5079849$ .



RYSUNEK 8.2: Estymowany czas wykonania pełnego cyklu analizy MES wykonanego zgodnie ze scenariuszem II. Liczba częstotliwości w paśmie wynosi  $f_m = \{10, 20, 50, 100\}$ . Rozmiar generowanej macierzy  $N=5079849$ .



# Rozdział 9

## Podsumowanie

W niniejszej rozprawie wykorzystano sformułowania zaproponowane przez [63] do rozwiązania wektorowego równania falowego Helmholtza z uwzględnieniem warunków brzegowych. Do opisu trójwymiarowych krzywoliniowych elementów skończonych użyto hierarchicznych wektorowych funkcji bazowych.

W pracy wykazano postawione w rozdziale pierwszym tezy opierając się na wynikach otrzymanych dla opracowanych strategii, algorytmów i ich implementacji na pojedynczym i dwóch akceleratorach graficznych. Zaproponowane implementacje pozwoliły na skrócenie czasu obliczeń dwóch najbardziej kosztownych obliczeniowo etapów metody elementów skończonych, czyli generacji globalnych macierzy sztywności i bezwładności oraz rozwiązania układu równań liniowych.

W kontekście generacji macierzy w MES:

1. Opracowano strategie i ich implementacje pozwalające na masywne zrównoleglenie obliczeń w trakcie generacji dużych macierzy sztywności i bezwładności na pojedynczym i wielu akceleratorach graficznych, w szczególności:
  - czas całkowania numerycznego został skrócony dzięki równoległemu przetwarzaniu porcji czworoscianów, zrównoleglonych obliczeniach kwadratury Gaussa, równoległym obliczeniom na macierzach gęstych oraz zastosowaniu współbieżnych strumieni w zależności od ośrodka,
  - czas konstrukcji macierzy został skrócony dzięki zrównolegleniu obliczeń na etapie składania macierzy w formacie COO jak oraz zaproponowano nowy zrównoleglony algorytm konwersji reprezentacji macierzy między formatami COO i CRS z eliminacją duplikatów.

Zaproponowane strategie i ich implementacje pozwoliły uzyskać ok. dziesięciokrotne skrócenie czasu generacji macierzy w MES przy użyciu akceleratora graficznego Tesla K40c względem zoptymalizowanej implementacji na CPU uruchomionej na wydajnym ośmiordzeniowym procesorze Intel Xeon Sandy Bridge. Wyniki prac dotyczących zagadnienia budowy macierzy były przedmiotem czterech publikacji w czasopiśmie z listy JCR (ang. Journal Citation Reports) [86–89].

W kontekście rozwiązywania układów równań w MES:

1. Opracowano i opublikowano w trzech artykułach w czasopiśmie z listy JCR [81, 106, 114] hybrydową implementację iteracyjnej metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym pozwalającą rozwiązać układ równań liniowych z wykorzystaniem pojedynczego akceleratora, w szczególności:

- Opracowano nowy format zapisu macierzy rzadkiej (Sliced ELLR-T) dedykowany wykonaniu operacji mnożenia macierzy rzadkiej przez wektor na GPU, który jest wydajniejszy niż implementacje dostępne w bibliotece CUSPARSE oparte na formacie CRS [106].
  - Zaproponowano zastąpienie metody Gaussa-Seidela przez ważoną metodę Jacobiego w operacjach „wygładzających” wielopoziomowego operatora ścisającego [77,81,114]. W ważonej metodzie Jacobiego wykorzystano operację mnożenia macierzy rzadkiej przez wektor (*matvec*) z [77].
  - Przeprowadzono analizę i dokonano optymalnego wyboru metody i implementacji rozwiązania układu równań na najniższym poziomie operatora ścisającego [77,81,114].
2. Opracowano hybrydową implementację iteracyjnej metody gradientów sprzężonych z wielopoziomowym operatorem ścisującym pozwalającą rozwiązać układ równań liniowych z wykorzystaniem kilku akceleratorów graficznych<sup>1</sup>. Podejście oparte na podziale dziedziny obliczeniowej, zastosowaniu formatu Sliced ELLR-T oraz dzięki zwielokrotnieniu zasobów obliczeniowych i pamięciowych pozwala na rozwiązanie większych układów równań w krótszym czasie (dla dwóch akceleratorów Tesla K20c można rozwiązać 1,7 krotnie większy problem w ok. 1,7 krótszym czasie).

Zaproponowane strategie i ich implementacje pozwoliły uzyskać kilkukrotne skrócenie czasu rozwiązywania dużych kilkumilionowych układów równań w MES na akceleratorze Tesla K40c względem implementacji bezpośredniej metody rozwiązywania układów równań na CPU uruchomionych na wydajnym ośmiordzeniowym procesorze Intel Xeon Sandy Bridge E5-2687W.

Zastosowanie akceleratora graficznego do wykonania obliczeń najbardziej kosztownych obliczeniowo etapów MES pozwoliło na ok. 4,7 krotne skrócenie czasu analizy MES dla największego problemu (5 milionów niewiadomych). Czas analizy MES został skrócony z niespełna pięciu godzin (gdy obliczenia wykonywane są wyłącznie na CPU) do ok. godziny gdy obliczenia w etapach generacji macierzy i iteracyjnego rozwiązania układu równań wykonywane są na GPU.

Z przedstawionych w pracy wyników jasno wynika, że współczesne karty graficzne zgodne z architekturą CUDA umożliwiają efektywną realizację obliczeń numerycznych kosztownych obliczeniowo etapów rozwiązania problemów elektromagnetycznych metodą elementów skończonych. Zastosowane strategie i ich implementacje masywnego zrównoleglenia obliczeń mają istotne znaczenie dla elektrodynamiki obliczeniowej w której wykorzystano MES z funkcjami bazowymi wyższego rzędu. Ponadto niektóre z opracowanych zagadnień (format zapisu macierzy rzadkich, rozwiązanie układu równań metodą iteracyjną na pojedynczym jak i kilku akceleratorach graficznych, konwersja między formatami reprezentacji macierzy rzadkiej) są ogólniejsze i mają zastosowanie dla szerszej klasy zagadnień numerycznych.

---

<sup>1</sup>Opisany w rozdziale 7 algorytm i wyniki będą przedmiotem przygotowywanej publikacji w czasopiśmie z listy JCR.

## Przewidywane kierunki rozwoju

Uzyskane wyniki zachęcają do prac nad optymalizacją metody elementów skończonych przy użyciu akceleratorów graficznych kompatybilnych z architekturą CUDA. Jednym z możliwych kierunków jest implementacja innych metod przestrzeni Krylowa, w tym iteracyjnych schematów dla problemów własnych (np. LOBPCG [120, 121], ISIA [122], ISIL [123], Jacobi-Davidson [124]).

Z uwagi na ograniczone zasoby pamięciowe, istotnym zagadnieniem wydaje się też opracowanie, dostosowanego do GPU, formatu zapisu macierzy uwzględniającego jej symetrię. Prace w tym kierunku zostały podjęte i opisane w publikacji będącej obecnie w drugiej recenzji w czasopiśmie *SIAM Journal on Scientific Computing* [125] oraz zaprezentowane na konferencji *GPU Technology Conference 2014* [126].

Inny kierunek rozwoju jest oparty na wykorzystaniu klastra obliczeniowego wyposażonego w  $K$  węzłów (w każdym węźle znajduje się co najmniej jeden akcelerator graficzny). Jednym z możliwych scenariuszy jest podział dziedziny obliczeniowej na  $K$  poddziedzin i generacja  $K$  macierzy sztywności  $\mathbf{S}_k$  i bezwładności  $\mathbf{T}_k$  w każdym z węzłów  $k = \{1, \dots, K\}$  i następnie rozwiązanie układu równań przez  $K$  akceleratorów graficznych. W etapie generacji macierzy  $\mathbf{S}_k$  i  $\mathbf{T}_k$  można wykorzystać algorytm iteracyjny zaproponowany dla jednego węzła (rozdział 5), z tą różnicą, że grupa elementów skończonych będzie przesyłana do  $K$  węzłów, a nie bezpośrednio do  $K$  akceleratorów w jednym węźle tak jak to miało miejsce w niniejszej rozprawie. W etapie rozwiązywania układów równań można zastosować podobny scenariusz wymiany danych wspólnych między węzłami jak to zostało zaproponowane w rozdziale 7 pod warunkiem, iż karty sieciowe klastra będą wykonane w technologii *Infiniband*, która pozwala na efektywną komunikację między akceleratorami graficznymi w różnych węzłach.





# Dodatek A

## Metoda elementów skończonych

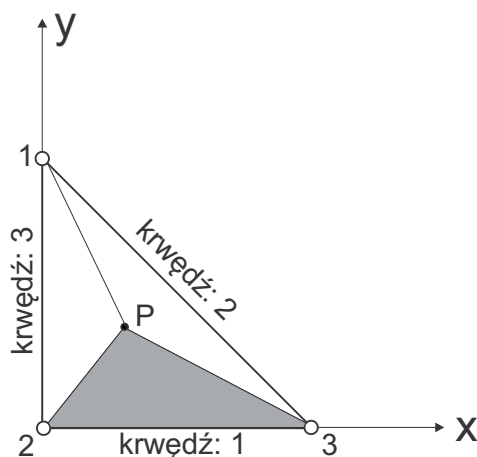
### A.1 Simpleksowy układ współrzędnych

W celu uproszczenia opisu w metodzie elementów skończonych wprowadzono simpleksowy układ współrzędnych z elementem referencyjnym, który na podstawie prostego skalowania można zaadaptować dla każdego elementu skończonego z siatki elementów. W ogólności współrzędne simpleksowe są definiowane jako współczynniki długości, obszaru oraz objętości na jakie dzieli punkt wewnątrz linii, trójkąta i czworościanu odpowiednio dla struktur jednowymiarowych, dwuwymiarowych i trójwymiarowych [1].

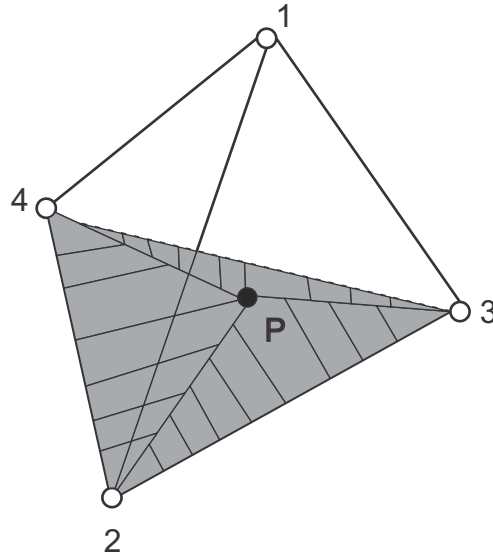
Dla struktur dwuwymiarowych wyróżnić można trzy współrzędne simpleksowe  $\lambda_1, \lambda_2, \lambda_3$ . Współrzędna  $\lambda_i$  r. (A.1) odpowiada stosunkowi pola trójkąta wyznaczonego przez punkt (P) oraz pozostałe węzły (j,k) trójkąta do pola trójkąta wyznaczonego przez węzły (i,j,k) trójkąta (rys. A.1).

$$\lambda_i = \frac{\Delta_{Pjk}}{\Delta_{ijk}} \quad (\text{A.1})$$

Analogicznie, dla struktur trójwymiarowych wyróżnić można cztery współrzędne simpleksowe  $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ . Współrzędna  $\lambda_i$  r. (A.2) odpowiada stosunkowi objętości czworościanu wyznaczonego przez punkt(P) oraz pozostałe węzły (j,k,l) czworościanu, i ob-



RYSUNEK A.1: Referencyjny element skończony w przestrzeni dwuwymiarowej (trójkąt).



RYSUNEK A.2: Referencyjny element skończony w przestrzeni trójwymiarowej (czworościan).

jętości czworościanu wyznaczonego przez węzły (i,j,k,l) czworościanu (rys. A.2).

$$\lambda_i = \frac{V_{Pjkl}}{V_{ijkl}} \quad (\text{A.2})$$

Współrzędne simpleksowe mają kilka ważnych właściwości. Po pierwsze współrzędne są unormowane (r. (A.3)). Po drugie, zarówno w przestrzeni dwuwymiarowej jak i trójwymiarowej gradient współrzędnej simpleksowej  $\lambda_i$  jest stały i prostopadły odpowiednio do  $i$ -tej krawędzi trójkąta i  $i$ -tej ścianki czworościanu.

$$\sum \lambda_i = 1 \quad (\text{A.3})$$

Zastosowanie współrzędnych simpleksowych upraszcza wyprowadzenie funkcji bazowych. Funkcje bazowe  $N_1, N_2, N_3$  mają takie właściwości, iż  $N_i$  w węźle  $i$  są równe 1, a w pozostałych węzłach są równe 0. W ogólności funkcje  $N_1$  można zapisać w postaci wielomianu:

$$N_1 = a + bx + cy \quad (\text{A.4})$$

W trzech węzłach można wyznaczyć wartości współczynników:

$$\text{węzeł 1 : } x = 0, y = 1, N_1(0, 1) = a + 0 + c = 1 \rightarrow a = -c$$

$$\text{węzeł 2 : } x = 0, y = 0, N_1(0, 0) = a + 0 + 0 = 0 \rightarrow a = 0$$

$$\text{węzeł 3 : } x = 1, y = 0, N_1(1, 0) = a + b + 0 = 0 \rightarrow b = 0$$

$a = 0, b = 0, c = 1$  funkcji  $N_1$  (A.5). W analogiczny sposób można określić funkcje  $N_2$  i  $N_3$  (A.6)-(A.7).

$$N_1 = y \quad (\text{A.5})$$

$$N_2 = 1 - x - y \quad (\text{A.6})$$

$$N_3 = x \quad (\text{A.7})$$

Tę samą postać funkcji  $N_i$  można uzyskać wykorzystując formuły z (A.8)-(A.10), co prowadzi do wniosku, iż w układzie simpleksowym  $N_i = \lambda_i$ .

$$\lambda_1 = \frac{\Delta_{P23}}{\Delta_{123}} = \frac{0.5 \cdot \text{podstawa} \cdot \text{wysokość}}{0.5 \cdot 1 \cdot 1} = y \quad (\text{A.8})$$

$$\lambda_2 = \frac{\Delta_{P13}}{\Delta_{123}} = \frac{0.5 \cdot (1 - x - y)}{0.5} = 1 - x - y \quad (\text{A.9})$$

$$\lambda_3 = \frac{\Delta_{P12}}{\Delta_{123}} = \frac{0.5 \cdot x \cdot 1}{0.5} = x \quad (\text{A.10})$$

Na koniec warto jeszcze wskazać, iż funkcje bazowe nie są liniowo niezależne, a dokładniej, tylko  $N_2$  i  $N_3$  są niezależne, a  $N_1$  jest kombinacją  $N_2$  i  $N_3$  r. (A.11).

$$N_1 = 1 - N_2 - N_3 \quad (\text{A.11})$$

## A.2 Metoda residuów ważonych

Punktem wyjścia jest rozwiązanie równania różniczkowego (2.1), czyli znalezienie funkcji  $\Phi$ , która zeruje błąd residualny  $r$ . (A.12). Poszukiwana funkcja  $\Phi$  aproksymowana jest przez szereg z r. (A.13), w którym  $a_k$  to współczynniki, a  $N_k$  to funkcje bazowe [1].

$$\mathcal{R} = \mathcal{L}\Phi - h = 0 \quad (\text{A.12})$$

$$\Phi \approx \sum_{k=1}^K a_k N_k \quad (\text{A.13})$$

Łącząc (A.12) i (A.13) błąd residualny można zapisać w następujący sposób:

$$\mathcal{R} = \mathcal{L} \sum_{i=k}^K a_k N_k - h = 0 \quad (\text{A.14})$$

Do rozwiązywania równania (A.14) można użyć metody residuów ważonych. W metodzie tej wprowadzane są funkcje wagi (A.15) przez które mnożone jest r. (A.14). Tak otrzymane równanie jest całkowane i w efekcie otrzymujemy postać całkową, którą z wykorzystaniem właściwości (A.16), można zapisać w postaci (A.17).

$$w = \sum_{m=1}^M w_m \quad (\text{A.15})$$

$$\langle b, e \rangle = \int_D b e \, dV \quad (\text{A.16})$$

$$\langle w_m, \mathcal{L}a_k N_k \rangle = \langle w_m, h \rangle \quad (\text{A.17})$$

W metodzie elementów skończonych częstym sposobem doboru funkcji wagi  $w_m$  jest zastosowanie metody Galerkina, w której funkcje wagi  $w_m$  wybiera się spośród zestawu funkcji bazowych  $N_k$ .

### A.3 Element Whitneya

W sekcji A.1 przedstawiono opis współrzędnych simpleksowych dla skalarnych funkcji bazowych. Zastosowanie skalarnych funkcji bazowych sprawdza się jeżeli  $\Phi$  w równaniu różniczkowym (2.1) jest polem skalarnym. W przypadku gdy  $\Phi$  jest polem wektorowym to dokładniejszą aproksymację pól wektorowych uzyskuje się stosując wektorowe funkcje bazowe [56].

Element Whitney-a jest opisany w układzie współrzędnych simpleksowych  $\lambda_i$  (patrz. Dodatek A.1, r. (A.8)-(A.10)) [1]. W przestrzeni dwuwymiarowej (trójkąt) gradienty współrzędnych simpleksowych  $\nabla \lambda_i$  są prostopadłe do „i”-tej krawędzi:

$$\nabla \lambda_1 = \hat{y} \quad (\text{A.18})$$

$$\nabla \lambda_2 = \hat{x} + \hat{y} \quad (\text{A.19})$$

$$\nabla \lambda_3 = \hat{x} \quad (\text{A.20})$$

Przepisane funkcje bazowe we współrzędnych kartezjańskich można zapisać w postaci:

$$\vec{N}_1 = \lambda_2 \nabla \lambda_3 - \lambda_3 \nabla \lambda_2 = (1 - x - y)\hat{x} - x(-\hat{x} - \hat{y}) = (1 - y)\hat{x} + x\hat{y} \quad (\text{A.21})$$

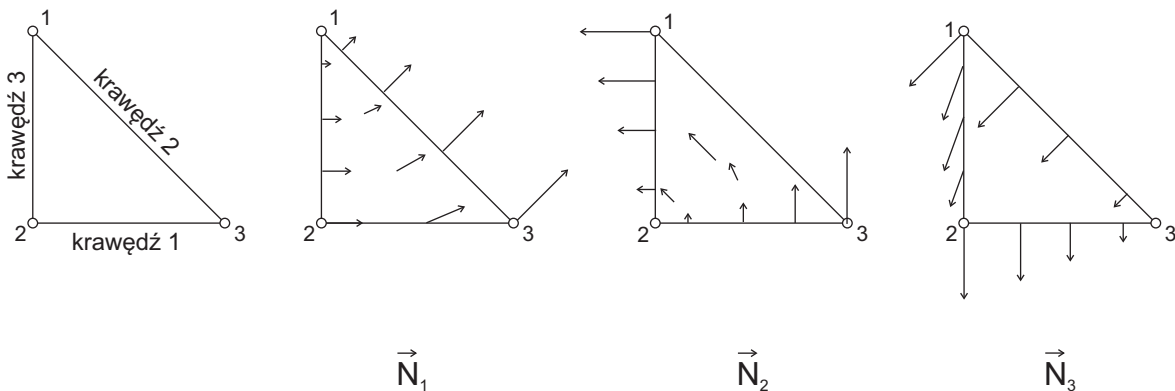
$$\vec{N}_2 = \lambda_1 \nabla \lambda_3 - \lambda_3 \nabla \lambda_1 = (-y)\hat{x} + x\hat{y} \quad (\text{A.22})$$

$$\vec{N}_3 = \lambda_1 \nabla \lambda_2 - \lambda_2 \nabla \lambda_1 = (-y)\hat{x} + (x - 1)\hat{y} \quad (\text{A.23})$$

Analizując funkcję  $\vec{N}_1$  i rys. A.3 można zaobserwować, iż:

- jest prostopadła wzdłuż krawędzi 2 i 3,
- funkcja rośnie wzdłuż krawędzi 2 (od węzła 1 do węzła 2),
- funkcja rośnie wzdłuż krawędzi 3 (od węzła 1 do węzła 3),
- wzdłuż krawędzi 1 funkcja ma dwie składowe (styczną i normalną). Składowa styczna, związana z  $\hat{x}$ , jest stała  $(1-y) = 1-0 = 1$ . Składowa normalna, związana z  $\hat{y}$ , jest liniowo zmienna ( $x$ ).

Analogiczne właściwości można wykazać dla funkcji związanych z krawędziami 2 i 3. Takie funkcje bazowe określa się jako CT\LN (ang. constant tangential, linear normal).



RYSUNEK A.3: Wektorowe funkcje bazowe dla elementu skończonego (trójkąt w przestrzeni dwuwymiarowej).

## A.4 Hierarchiczne funkcje bazowe - element Webba

W elemencie Webba występuje 30 funkcji bazowych opisanych w tabeli A.1:

- 6 funkcji o właściwości CT\LN (składowa styczna jest stała, składowa normalna jest liniowo zmienna) i każda z funkcji związana jest z jedną krawędzią czworościanu,
- 6 funkcji o właściwości LT\LN (składowe styczna i normalna są liniowo zmienne) i każda z funkcji związana jest z jedną krawędzią czworościanu,
- 2x4 funkcji o właściwości LT\QN (składowa styczna jest liniowo zmienna, składowa normalna jest funkcją kwadratową) i po dwie funkcje przypadają na jedną ze ścianek czworościanu,
- 10 funkcji o właściwości QT\QN (składowe styczna i normalna są kwadratowymi funkcjami zmiennych) i sześć związanych jest z krawędziami, cztery związane są ze ściankami czworościanu.

TABELA A.1: Zestaw funkcji bazowych opisujących element Webba [67].

Grupa	Opis	Liczba funkcji
I.	$\lambda_i \nabla \lambda_j - \lambda_j \nabla \lambda_i$	6 (krawędzie)
II.	$\lambda \lambda_i \lambda_j$	6 (krawędzie)
III.	$\lambda_j \lambda_k \nabla \lambda_i + \lambda_i \lambda_k \nabla \lambda_j - 2 \lambda_i \lambda_j \nabla \lambda_k$	$2 \cdot 4$ (ścianki)
IV.	$\nabla(\lambda_i \lambda_j [\lambda_i - \lambda_j])$ $\nabla \lambda_i \lambda_j \lambda_k$	6 (krawędzie) 4 (ścianki)





# Dodatek B

## Programowanie GPU

### B.1 Skuteczne programowanie GPU

Tak jak to zostało napisane w rozdziale 3 architektura CUDA przezwyciężyła wcześniejsze bariery wykorzystania GPU w aplikacjach ogólnego zastosowania [8] i umożliwia programowanie GPU przy użyciu języka wysokiego poziomu (rozszerzony język C). Ważną cechą CUDA jest zwiększenie elastyczności programowania poprzez możliwość swobodnego oraz szybkiego gromadzenia (ang. *gather*) i rozrzucania (ang. *scatter*) danych w DRAM w porównaniu do innych architektur GPU. Ponadto właściwości architektur Fermi i Kepler umożliwiają efektywniejsze tworzenie algorytmów z maszynowo zrównoleglonymi obliczeniami. Aby zwiększyć wydajność obliczeń wykonywanych na GPU należy maksymalnie zrównoleglić implementowany algorytm i optymalizować użycie pamięci [83].

Poniżej przedstawiono kilka najważniejszych reguł zrównoleglenia algorytmów:

- komunikacja między wątkami tego samego bloku powinna odbywać się poprzez pamięć wspólną (ang. *shared memory*), a nie pamięć globalną (ang. *global memory*),
- komunikacja między wątkami z różnych bloków powinna odbywać się poprzez pamięć globalną lub program należy rozdzielić na kilka kerneli,
- jeżeli problem zostanie rozbity na kilka kerneli to mogą być one wykonane równolegle (właściwość *concurrent kernels*), pod warunkiem, że wykorzystuje się oddzielne zasoby pamięci (Uwaga! Właściwość dostępna tylko dla architektury Fermi i Kepler), dodatkowo warto rozważyć możliwość wykonania kerneli wewnątrz kernela (*dynamic parallelism*), ale jest to właściwość dostępna tylko dla architektury Kepler,

oraz optymalizacji użycia pamięci:

- należy ukrywać opóźnienia poprzez wykonanie w tym samym czasie dwóch rodzajów operacji: pobierania/wysyłanie danych z/do pamięci akceleratora i wykonywania obliczeń, np. nie czekając na pobranie wszystkich danych można wykonywać obliczenia na już dostępnych danych,
- przesyłanie danych pomiędzy CPU i GPU powinno być grupowane (jednorazowe przesłanie dużej porcji danych jest wydajniejsze niż kilka przesyłów mniejszych porcji danych),

- należy maksymalizować użycie pamięci wspólnej, która jest wielokrotnie szybsza niż pamięć globalna,
- pamięć wspólna jest podzielona na 16 banków i przy dostępie do nich należy unikać konfliktów, które występują jeśli wiele wątków jednocześnie żąda dostępu do tego samego banku,
- optymalne użycie pamięci globalnej uzyskuje się poprzez zapewnienie dostępu łącznego<sup>1</sup> do tej pamięci. Dostęp łączny ma miejsce, np. gdy 32 wątki (1 warp) odczytują przyległe 4-bajtowe dane (np. zmienna typu float). W takim przypadku 128 (=32·4) bajtów jest jednorazowo (łącznie) pobranych z pamięci,
- gdy nie można zagwarantować łącznego dostępu do pamięci globalnej należy wykorzystać pamięć tekstur (*texture memory*),
- w zależności od złożoności problemu rozwiązywanego na GPU, programista powinien wybrać opcję konfigurowalnej pamięci podręcznej, która zapewni efektywne wykonanie kernela (Uwaga! Właściwość dostępna tylko dla architektury Fermi i Kepler).

## B.2 Przykład uruchomienia obliczeń na GPU.

W ramach podsumowania w Wydruku B.1 przedstawiono opisany w rozdziale 3 model programowania na GPU na przykładzie dodawania dwóch wektorów. Na początku dane są alokowane i inicjalizowane na CPU (linie: 19-31). Następnie wykonywana jest alokacja danych w pamięci GPU (linie: 33-37) i kopiowanie danych z CPU do GPU (linie: 40-41). Przed uruchomieniem obliczeń na GPU (linia: 48), ustawiane są parametry uruchomienia kernela, czyli określana jest liczba wątków w bloku i liczba bloków wątków (linie: 45-47). W kernelu (`kernel_dodanieWektorow`) każdy wątek (indeks wątku w bloku: `threadIdx.x`, indeks globalny wątku: `i`) ma za zadanie odczyt danych z wektorów (`X[i]`, `Y[i]`), wykonanie sumy oraz zapis obliczeń do wektora wynikowego (`Z[i]`). Po zakończeniu obliczeń na GPU dane wynikowe są kopiowane z pamięci GPU do pamięci CPU (linia: 51). Na zakończenie programu zwalniana jest pamięć GPU (linia: 54) i CPU (linia: 55).

---

<sup>1</sup>Przez niektórych autorów dostęp łączny jest też tłumaczony jako złączony.

```

1  /* Model programowania na GPU na przykładzie operacji dodawania wektorów  $Z = X + Y$ 
2  (Faza-1) Alokacja na CPU (host)
3  (Faza-2) Alokacja na GPU (device)
4  (Faza-3) Przesłanie danych z CPU (host) na GPU (device)
5  (Faza-4) Wykonanie obliczeń na GPU ( $Z = X + Y$ )
6  (Faza-5) Przesłanie danych z GPU (device) do CPU (host)
7  (Faza-6) Zwolnienie pamięci GPU (device) i CPU (host)
8  */
9  // kernel_dodanieWektorow - funkcja wywołana z CPU (host) i wykonana na GPU (device)
10 __global__ void kernel_dodanieWektorow(const float *X, const float *Y, float *Z, int N){
11     int i = blockDim.x * blockIdx.x + threadIdx.x;
12     if (i < N)
13         Z[i] = X[i] + Y[i];
14     __syncthreads(); // synchronizacja w tym przypadku jest nadmiarowa
15 }
16
17 int main(void){
18     // rozmiar wektora:
19     int N = 2000000;
20     size_t size = N * sizeof(float);
21
22     // (Faza-1) Alokacja na CPU (host):
23     float *h_X = (float *)malloc(size);
24     float *h_Y = (float *)malloc(size);
25     float *h_Z = (float *)malloc(size);
26
27     // Inicjalizacja danych na CPU (host):
28     for (int i = 0; i < N; i++){
29         h_X[i] = rand();
30         h_Y[i] = rand();
31     }
32
33     // (Faza-2) Alokacja na GPU (device):
34     float *d_X, *d_Y, *d_Z;
35     cudaMalloc((void **)&d_X, size);
36     cudaMalloc((void **)&d_Y, size);
37     cudaMalloc((void **)&d_Z, size);
38
39     // (Faza-3) Przesłanie danych z CPU (host) na GPU (device):
40     cudaMemcpy(d_X, h_X, size, cudaMemcpyHostToDevice);
41     cudaMemcpy(d_Y, h_Y, size, cudaMemcpyHostToDevice);
42
43     // (Faza-4) Wykonanie obliczeń na GPU:
44     // liczba wątków w bloku
45     int watkowPerBlok = 256;
46     // liczba bloków w siatce
47     int blokowPerSiatka = (N + watkowPerBlok - 1) / watkowPerBlok;
48     kernel_dodanieWektorow <<< blokowPerSiatka, watkowPerBlok>>>(d_X, d_Y, d_Z, N);
49
50     // (Faza-5) Przesłanie danych z GPU (device) do CPU (host):
51     cudaMemcpy(h_Z, d_Z, size, cudaMemcpyDeviceToHost);
52
53     // (Faza-6) Zwolnienie pamięci GPU (device) i CPU (host):
54     cudaFree(d_X); cudaFree(d_Y); cudaFree(d_Z);
55     free(h_X); free(h_Y); free(h_Z);
56 }

```

WYDRUK B.1: Przykład wykonania operacji dodawania wektorów  $Z = X + Y$  z wykorzystaniem GPU.



## Dodatek C

# Schematy, kody i pseudokody implementacji na GPU

Lista schematów, kodów i pseudokodów:

1. Schemat całkowania numerycznego na GPU z mieszaną kwadraturą Gaussa (Wydruk C.1)
2. Kod wykonania operacji mnożenia macierzy gęstych w fazie całkowania numerycznego na GPU (Wydruk C.2).
3. Kod wykonania operacji mnożenia macierzy rzadkich przez wektor przy użyciu formatu Sliced ELLR-T (Wydruk C.3).
4. Pseudo-kod metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V (Wydruk C.4).
5. Pseudo-kod wielopoziomowego operatora ściskającego o cyklu V (Wydruk C.5).
6. Kod ważonej metody Jacobiego (wJacobi) zastosowanej do operacji wygładzających wielopoziomowego operatora ściskającego (Wydruk C.6).
7. Schemat obliczeń metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V z wariantem wykonania operacji *matvec* dla macierzy z rys. 7.1 (Wydruk C.7).
8. Schemat obliczeń operacji *matvec* na dwóch akceleratorach w wariancie z dekompozycją domen (Wydruk C.8).

```

1
2
3 // WEJŚCIE: podzbiór M czworościanów, macierze funkcji bazowych (N),
4 //         rotacje funkcji bazowych (NR), ośrodek (e,u)
5 // WYJŚCIE: M lokalnych macierzy sztywności i bezwładności SE, TE o
6 //         rozmiarze (M*50*50)
7
8
9 void numIntegWithMixedGaussQuadrature (... , int start, int stop,
10                                     double * TE, double * SE){
11
12     // selekcja silnie i słabo krzywoliniowych czworościanów
13     selectTetrahedra<<<...>>>(..., tetsIndexQ81, tetsIndexQ24);
14     // wyjście:
15     // tetsIndexQ81, tetsIndexQ24 - tablice z indeksami silnie
16     // i słabo krzywoliniowych czworościanów w
17     // podzbiorze M czworościanów
18     // tetsQ81, tetsQ24 - rozmiar tablic tetsIndexQ81 i tetsIndexQ24
19
20     // -----
21     // Całkowanie numeryczne z kwadraturą Gaussa wyższego 10-tego rzędu:
22     // przygotowanie kwadratury 10-tego rzędu na podstawie zasad z [72]
23     prepareW81<<<...>>>(...);
24     prepareL81<<<...>>>(...);
25     // obliczenie funkcji bazowych N, NR (według [64])
26     buildBasisFunQ81<<<...>>>(...);
27
28     // obliczenie lokalnych macierzy elementów
29     // (Se r.(4.6) i Te r.(4.8)):
30     for(int t = 0; t < tetsQ81; t=t+parallelTets){
31         // t - indeks silnie krzywoliniowego czworościanu
32         // parallelTets - liczba czworościanów przetwarzanych
33         // równolegle w porcji
34         // obliczenie macierzy Jacobiego (J) i
35         // odwróconych macierzy Jacobiego inv(J)
36         matmat3x10_10x3_Q81<<<...>>>(..., tetsIndexQ81);
37         calcDetInvJ_Q81<<<...>>>(..., tetsIndexQ81);
38
39         // zrównoleglona Q=81-punktowa kwadratura Gaussa
40         // (współbieżne strumienie stream0 i stream1)
41         // T12 = N * (inv(J)^T * e * inv(J))
42         T12<<< ..., stream0 >>>(..., tetsIndexQ81);
43         // Te = suma (T12 * N^T)
44         Te <<< ..., stream0 >>>(..., tetsIndexQ81, double * TE);
45         // S12 = NR * (J * u * J^T)
46         S12<<< ..., stream1 >>>(..., tetsIndexQ81);
47         // Se = suma (S12 * NR^T)
48         Se <<< ..., stream1 >>>(..., tetsIndexQ81, double * SE);
49     } // zakończenie pętli NI dla silnie krzywoliniowych czworościanów
50
51

```

```

52 // -----
53 // Całkowanie numeryczne z kwadraturą Gaussa niższego 6-tego rzędu:
54
55 // przygotowanie kwadratury 6-tego rzędu na podstawie zasad z [72]
56 prepareW24<<<...>>>(...);
57 prepareL24<<<...>>>(...);
58 // obliczenie funkcji bazowych N, NR (według [64])
59 buildBasisFunQ24<<<...>>>(...);
60
61
62 // obliczenie macierzy lokalnych macierzy elementów
63 // (Se r.(4.5) i Te r.(4.7)):
64 for(int t = 0; t < tetsQ24; t=t+parallelTets){
65
66     // t - indeks słabo krzywoliniowego czworościanu
67     // parallelTets - liczba czworościanów przetwarzanych
68     // równoległe w porcji
69     matmat3x10_10x3_Q24<<<...>>>(..., tetsIndexQ24);
70     calcDetInvJ_Q24<<<...>>>(..., tetsIndexQ24);
71     // zrównoleglona Q=81-punktowa kwadratura Gaussa:
72     // (współbieżne strumienie stream0 i stream1)
73     // T12 = N * (inv(J)^T * e * inv(J))
74     T12<<< ..., stream0 >>>(..., tetsIndexQ24);
75     // Te = T12 * N^T
76     Te <<< ..., stream0 >>>(..., , tetsIndexQ24, double * TE);
77     // S12 = NR * (J * u * J^T)
78     S12<<< ..., stream1 >>>(..., tetsIndexQ24);
79     // Se = S12 * NR^T
80     Se <<< ..., stream1 >>>(..., tetsIndexQ81, double * SE);
81 } // zakończenie pętli NI dla słabo krzywoliniowych czworościanów
82 }

```

WYDRUK C.1: Schemat implementacji całkowania numerycznego (NI) z kwadraturą mieszaną na GPU.



```

1
2 /*
3 WEJŚCIE:
4     N (50x3) x 81    // 81 macierz funkcji bazowych N
5     JJ (3x3) x 81    // 81 macierzy JJ
6 WYJŚCIE:
7     T12 (50,3) = N (50,3) * JJ (3,3) dla 81 punktów kwadratury
8     T12 (50x3) x 81 //
9
10 Zrównoleglenie:
11 blockIdx.x - jeden wątek wykonuje obliczenia w jednym wierszu
12 blockIdx.y (0,...,parallelTets-1) elementy przetwarzane równolegle
13             w porcji zawierającej parallelTets czworościanów
14 blockIdx.z (0,...,80)    81-punktowa kwadratura
15
16
17 N      - 81 macierzy funkcji bazowych o rozmiarze macierzy (NLen=50, k=3)
18 JJ     - iloczyn macierzy Jacobianów o rozmiarze (JJLen=3 x JJLen=3)
19 T12    - macierz wynikowa o rozmiarze (T12Len=50, k=3)
20 T12Len - liczba wierszy macierzy T12
21 quad   - liczba punktów kwadratury
22 */
23 __global__ void mgNi_T12_Q( const double *N, int NLen, const double *JJ,
24                             int JJLen, double * T12, int T12Len,
25                             int k, int quad ){
26     const int inx = threadIdx.x;
27     const int iny = threadIdx.y;
28
29     const int iby = blockIdx.y;
30     int row, bz;
31     // rezerwacja pamięci wspólnej na macierz T12
32     // dla pojedynczego punktu kwadratury
33     __shared__ double t12[50][3];
34
35
36     if(inx + iny*16 < 50){
37         row = inx + iny*16;
38         bz = blockIdx.z;    // punkt kwadratury
39
40         // row      - numer wiersza w N oraz T12
41         // bz*50*3 - przeskok do jednej z 81 macierzy N
42         // bz*50*3 - przeskok do jednej z 81 macierzy T12
43
44         // iby      - numer czworościanu
45         // iby*3*3*81 - przeskok do macierzy JJ analizowanego czw.
46         // iby*50*3*81 - przeskok do macierzy T12 analizowanego czw.
47
48         N += row + bz*50*3;
49         JJ += inx + iny*JJLen + bz*3*3 + iby*quad* 3*3;
50         T12 += row + bz*50*3 + iby*quad*50*3;
51

```

```

52         double c[3] = {0,0,0};
53         __shared__ double sh_JJ[4][4];
54         // zapis macierzy JJ do pamięci wspólnej
55         if(iny<3 && inx < 3){
56             sh_JJ[iny][inx] = JJ[0];
57         }
58         __syncthreads();
59
60         // N*JJ
61         // c[0], c[1], c[2] - jeden wiersz z mnożenia
62         c[0] = N[0]*sh_JJ[0][0] +
63             N[NLen]*sh_JJ[1][0] +
64             N[2*NLen]*sh_JJ[2][0];
65         c[1] = N[0]*sh_JJ[0][1] +
66             N[NLen]*sh_JJ[1][1] +
67             N[2*NLen]*sh_JJ[2][1];
68         c[2] = N[0]*sh_JJ[0][2] +
69             N[NLen]*sh_JJ[1][2] +
70             N[2*NLen]*sh_JJ[2][2];
71
72         t12[row][0] = c[0]*0.16666666666666666;
73         t12[row][1] = c[1]*0.16666666666666666;
74         t12[row][2] = c[2]*0.16666666666666666;
75         //0.16666666666666666 = 1/6
76
77         T12[0] = t12[row][0]; T12 += T12Len;
78         T12[0] = t12[row][1]; T12 += T12Len;
79         T12[0] = t12[row][2];
80     }
81 }

```

WYDRUK C.2: Kod mnożenia macierzy gęstych małych rozmiarów użytego w etapie całkowania numerycznego ( $T12 = N \cdot JJ$ ).

```
1
2 // WEJŚCIE:
3 // macierz rzadka (A) reprezentowana poprzez wektory:
4 // d_ella - wartości niezerowe
5 // d_ja    - indeksy kolumn wartości niezerowych
6 // d_rls   - informacja o liczbie elementów przetwarzanych
7 //         przez wątek w wierszy
8 // d_slice - wskaźnik do pierwszej wartości niezerowej poszczególnego
9 //         plastra (slice) macierzy
10 // nrows   - liczba wierszy macierzy
11 // ali      - przesunięcie gwarantująca łączny dostęp do el. macierzy
12 // d_x      - wektor prawej strony,
13
14 // WYJŚCIE:
15 // d_y      - wektor wynikowy
16
17 extern __shared__ double shared_data[]; // pamięć shared
18
19 // W przypadku gdy wektor d_x jest „zbindowany
20 // w pamięci tekstur to niezbędne jest zdefiniowanie (WARIANT-II):
21 texture<int2,1> text_xd;
22 static __inline__ __device__ double fetch_double(texture<int2, 1> t, int i){
23     int2 v = tex1Dfetch(t,i);
24     return __hiloint2double(v.y, v.x);
25 }
26
27 // kernel wykonujący operację SpMV:
28 __global__ void spmv_SELLR4T(double *d_ella, int *d_ja,
29                             int *d_rls, int *d_slice,
30                             int nrows, int ali,
31                             double alfa, double beta,
32                             // double *d_x WARIANT-II,
33                             double *d_y ){
34
35     // wskaźnik do pamięci shared
36     double * array = &shared_data[0];
37
38     // lokalny indeks wątku w bloku
39     int tx = threadIdx.x;
40
41     // indeks wątku pracującego na porcji el. niezerowych w wierszu
42     int txm = tx % 4;
43
44     // globalny indeks wątku
45     int txg = (blockIdx.x*blockDim.x+threadIdx.x) ;
46
47     // indeks wiersza na którym pracują wątki
48     int n = (blockIdx.x*blockDim.x+threadIdx.x) >> 2;
49
50     if (n < nrows){
51         // sub przechowuje składnik wartości y[n]
```

```

52 // obliczony przez jeden z 4 wątków pracujących w wierszu
53 double sub = 0.0;
54 // określenie maksymalnej liczby elementów przetwarzanych przez watek
55 // w plastrze (slice)
56 int max = d_rls[txg];
57
58 // pętla obliczająca składniki (sub) wartości wynikowych
59
60 for(int i=0; i < max; i++){
61     // indeks wartości niezerowej macierzy rzadkiej A
62     int ind = i*ali+slice[blockIdx.x]+tx;
63     // wartość niezerowa macierzy rzadkiej A
64     double value = d_ella[ind];
65     // indeks kolumny wartości niezerowej macierzy rzadkiej A
66     int col = d_ja[ind];
67
68     // składnik wartości wektora wynikowego obliczona przez
69     // jeden z 4 wątków pracujących w wierszu
70     // WARIANT-I: odczyt z pamięci globalnej
71     //sub += value * d_x[col];
72
73     // WARIANT-II: odczyt z pamięci tekstur
74     sub += value * fetch_double(text_xd, col); // lub __ldg(&d_x[col])
75 }
76 // zapisanie składników wektora wynikowego do
77 // tablicy (array) w pamięci shared:
78 array[tx] = sub;
79
80 // redukcja na tablicy array celem zsumowania
81 // składników obliczonych przez 4 wątki pracujące w wierszu
82 if(txm < 2){
83     array[tx] += array[tx+2];
84     if(txm == 0)
85         d_y[n] = array[tx] + array[tx+1];
86 }
87 }
88 }

```

WYDRUK C.3: Kod  $y = Ax$  na GPU z wykorzystaniem formatu Sliced ELLR-T, gdy liczba wątków wykonujących obliczenia w pojedynczym wierszu  $T = 4$ .

```

1 // Podział macierzy na potrzeby wielopoziomowego operatora ściskającego:
2
3 /* macierz dla trzypoziomowego operatora ściskającego:
4
5     
$$A = A_{V3} = \begin{bmatrix} M_{11}^E & M_{12}^E & M_{13}^E \\ M_{21}^E & M_{22}^E & M_{23}^E \\ M_{31}^E & M_{32}^E & M_{33}^E \end{bmatrix}$$

6
7 */
8 // metoda PCG z wielopoziomowym operatorem ściskającym cyklu V
9 // WEJŚCIE:

```

```

9 // A - macierz współczynników ( $A = S + k^2 T$ , gdzie S i T
10 //      to macierze sztywności i bezwładności)
11 // b - wektor pobudzenia
12 // iter - liczba iteracji metody gradientów sprzężonych
13 // iterWJ - tablica przechowująca w komórce „i” liczbę iteracji w
14 // ważonej metodzie Jacobiego na „i”-tym poziomie operatora ściskającego
15 //  $\epsilon$  - oczekiwany poziom zbieżności metody iteracyjnej, np.  $\epsilon = 1e^{-4}$ 
16 // WYJŚCIE:
17 // x - poszukiwany wektor wynikowy
18
19 void pcg_V(double * A,
20           double * b, double * x,
21           int iter, int * iterWJ, double  $\epsilon$ ){
22     i = 0
23      $r = b - \begin{bmatrix} M_{11}^E & M_{12}^E & M_{13}^E \\ M_{21}^E & M_{22}^E & M_{23}^E \\ M_{31}^E & M_{32}^E & M_{33}^E \end{bmatrix} x$  //II-SpMV
24     prekondycjoner_V( $M_{11}^E, \dots, M_{33}^E, r, d$ ) //I,II
25      $d_{new} = r^T d$  //I-dot
26      $d_0 = d_{new}$ 
27     while (i < iter &&  $d_{new} > \epsilon^2 d_0$ ){
28          $q = \begin{bmatrix} M_{11}^E & M_{12}^E & M_{13}^E \\ M_{21}^E & M_{22}^E & M_{23}^E \\ M_{31}^E & M_{32}^E & M_{33}^E \end{bmatrix} d$  //II-SpMV
29          $\alpha = \frac{d_{new}}{d^T q}$  //I-dot
30          $x = x + \alpha d$  //I-axpy
31          $r = r - \alpha q$  //I-axpy
32         prekondycjoner_V( $M_{11}^E, \dots, M_{33}^E, r, s$ ) //I,II
33          $d_{old} = d_{new}$ 
34          $d_{new} = r^T s$  //I-dot
35          $\beta = \frac{d_{new}}{d_{old}}$ 
36          $d = s + \beta d$  //I-axpy
37         i = i + 1
38     }
39 }
```

WYDRUK C.4: Pseudo-kod metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V. W komentarzach operacjom metody iteracyjnej przypisano kategorie: I,II.

```

1
2 // Wielopoziomowy operator ściskający o cyklu V
3 // WEJŚCIE:
4 //  $r_{Ek}$  - wektor residualny określony w głównej iteracji PCG
5 // WYJŚCIE:
6 //  $z_{Ek}$  - wektor wynikowy operatora ściskającego
7
8 prekondycjoner_V(double *  $M_{11}^E, \dots, double * M_{33}^E,$ 
9                 double * r, double * z, int * iterWJ){
10     //  $N_1$  - liczba wierszy macierzy  $M_{11}^E$ 
```

```

11 // N2 - liczba wierszy macierzy  $M_{22}^E$ 
12 // N3 - liczba wierszy macierzy  $M_{33}^E$  // N12 = N1 + N2
13 // N123 = N1 + N2 + N3
14 // K - liczba poziomów wielopoziomowego operatora ściskającego
15 // iterWJ - tablica przechowująca w i-tej komórce liczbę iteracji
16
17 //          ważonej metody Jacobiego (wJacobi) jakie trzeba
18 //          wykonać na i-tym poziomie operatora ściskającego
19
20 rE = r; //I-cpy
21 zE = [0,...,0]; //I-scal
22
23 rE1 = r(1 : N1, 1); //I-cpy
24 rE2 = r(N1 + 1 : N12, 1); //I-cpy
25 if (K==3)
26     rE3 = r(N12 + 1 : N123, 1); //I-cpy
27
28 zE1 = zE(1 : N12, 1); //I-cpy
29 zE2 = zE(N1 + 1 : N12, 1); //I-cpy
30 if (K==3)
31     zE3 = zE(N12 + 1 : N123, 1); //I-cpy
32
33 // poziom 3:
34 if (K == 3){
35     zE3 = wJacobi( $M_{33}^E$ , rE3, zE3, w, iterWJ[3]);
36     rE2 = rE2 -  $M_{23}^E$  zE3;
37 }
38 // poziom 2:
39 zE2 = wJacobi( $M_{22}^E$ , rE2, zE2, w, iterWJ[2]);
40 rE2 = rE2 -  $M_{21}^E$  zE1; //II-SpMV
41
42 // poziom 1:
43  $M_{11}^E$  zE1 = rE1; //Pardiso
44
45 // poziom 2:
46 rE2 = rE2 -  $M_{21}^E$  zE1; //II-SpMV
47 zE2 = wJacobi( $M_{22}^E$ , rE3, zE3, w, iterWJ[2]);
48
49 // poziom 3:
50 if (K == 3){
51     rE3 = rE3 -  $M_{32}^E$  zE2; //II-SpMV
52     zE3 = wJacobi( $M_{33}^E$ , rE3, zE3, w, iterWJ[3]);
53 }
54 z(1 : N1, 1) = zE1; //I-cpy
55 z(N1 + 1 : N12, 1) = zE2; //I-cpy
56 z(N12 + 1 : N123, 1) = zE3; //I-cpy
57 }

```

WYDRUK C.5: Pseudo-kod wielopoziomowego operatora ściskającego o cyklu V. W komentarzach operacjom operatora ściskającego przypisano kategorie: I,II.

```

1
2 // ważona metoda Jacobiego, rozwiązanie układu  $\mathbf{M}_{kk}^E \mathbf{z}_{Ek} = \mathbf{r}_{Ek}$ 
3 // WEJŚCIE:
4 //  $\mathbf{M}_{kk}^E$  - macierz na poziomie  $k$ 
5 //  $\mathbf{D}_{kk}$  - macierz zawierająca diagonalę macierzy  $\mathbf{M}_{kk}^E$ 
6 //  $\mathbf{r}_{Ek}$  - prawa strona układu równań
7 //  $\mathbf{z}_{Ek}$  - początkowe rozwiązanie układu
8 //  $w$  - waga w ważonej metodzie Jacobiego
9
10 // iterWJ - liczba iteracji w ważonej metodzie Jacobiego
11 // WYJŚCIE:
12 //  $\mathbf{z}_{Ek}$  - wektor wynikowy
13
14 void wJacobi(double *  $\mathbf{M}_{kk}^E$ , double *  $\mathbf{r}_{Ek}$ ,
15             double *  $\mathbf{z}_{Ek}$ , double  $w$ , int iterWJ){
16
17     for (int p = 0; p < iterWJ; p++){
18          $\mathbf{r} = \mathbf{r}_{Ek} - \mathbf{M}_{kk}^E \mathbf{z}_{Ek}$  //II-SpMV
19          $\mathbf{z}_{Ek} = \mathbf{z}_{Ek} + w * \mathbf{D}_{kk}^{-1} \mathbf{r}$  //III-SpMV
20     }
21 }

```

WYDRUK C.6: Kod ważonej metody Jacobiego (wJacobi) zastosowanej do operacji wygładzających wielopoziomowego operatora ściskającego (Wydruk C.6). W komentarzach operacje metody rozpisane zostały na kategorie: I,II.

```

1
2 void spmv_2GPU_domeny(double *  $\mathbf{A}_{(1)}$ , double *  $\mathbf{A}_{(2)}$ , double *  $\mathbf{q}_{(1)}$ , double *  $\mathbf{q}_{(2)}$ ,
3                      double *  $\mathbf{d}_{(1)}$ , double *  $\mathbf{d}_{(2)}$ ){
4     // ---- GPU(1) ---- // ---- GPU(2) ----
5     // (1) - wektory, skalary // (2) - wektory, skalary
6     // przechowywane na GPU(1) // przechowywane na GPU(2)
7     // (1d) - indeksy wierszy należących // (2d) - indeksy wierszy
8     // do domeny 1 // należących do domeny 2
9     // (1w) - indeksy wspólnych wierszy // (2w) - indeksy wspólnych wierszy
10    // należących do domeny 1 i 2 // należących do domeny 1 i 2
11
12     $\mathbf{q}_{(1d)} = \mathbf{A}_{(1d)} \mathbf{d}_{(1d)}$   $\mathbf{q}_{(2d)} = \mathbf{A}_{(2d)} \mathbf{d}_{(2d)}$ 
13     $\mathbf{q}_{(1w)} = \mathbf{A}_{(1w)} \mathbf{d}_{(1w)}$   $\mathbf{q}_{(2w)} = \mathbf{A}_{(2w)} \mathbf{d}_{(2w)}$ 
14
15    SYNCHRONIZACJA
16     $\leftarrow \mathbf{q}_{(2w)}(\mathbf{DtD})$ 
17     $\rightarrow \mathbf{q}_{(1w)}(\mathbf{DtD})$ 
18
19     $\mathbf{q}_{(1)} = \mathbf{q}_{(1d)} + (\mathbf{q}_{(2w)} + \mathbf{q}_{(1w)})$   $\mathbf{q}_{(2)} = \mathbf{q}_{(2d)} + (\mathbf{q}_{(1w)} + \mathbf{q}_{(2w)})$ 
20 }

```

WYDRUK C.8: Schemat obliczeń operacji *matvec* na dwóch akceleratorach w wariacie z dekompozycją domen.



```

1 void pcg_2GPU(double * A(1), double * A(2), double * b,
2             double * x(1), double * x(2), double ε){
3 // ---- GPU(1) ----                      ---- GPU(2) ----
4 // (1) - wektory, skalary                (2) - wektory, skalary
5 //   przechowywane na GPU(1)              przechowywane na GPU(2)
6 /* macierz dla trzypoziomowego operatora ściskającego:
7
8     
$$\mathbf{A} = \mathbf{A}_{(1)} + \mathbf{A}_{(2)} = \begin{bmatrix} \mathbf{M}_{11(1)}^E & \mathbf{M}_{12(1)}^E & \mathbf{M}_{13(1)}^E \\ \mathbf{M}_{21(1)}^E & \mathbf{M}_{22(1)}^E & \mathbf{M}_{23(1)}^E \\ \mathbf{M}_{31(1)}^E & \mathbf{M}_{32(1)}^E & \mathbf{M}_{33(1)}^E \end{bmatrix} + \begin{bmatrix} \mathbf{M}_{11(2)}^E & \mathbf{M}_{12(2)}^E & \mathbf{M}_{13(2)}^E \\ \mathbf{M}_{21(2)}^E & \mathbf{M}_{22(2)}^E & \mathbf{M}_{23(2)}^E \\ \mathbf{M}_{31(2)}^E & \mathbf{M}_{32(2)}^E & \mathbf{M}_{33(2)}^E \end{bmatrix}$$

9
10 */
11 i = 0
12 r(1) = A(1)x(1)                                r(2) = A(2)x(2)
13
14                                     SYNCHRONIZACJA
15                                     ← r(2)(DtD)
16 r(1) = b - (r(1) + r(2)) // r(1) = b - ([r(1); r(2)])
17 d = prekondycjoner
18 d(1)new = r(1)Tr(1)
19 dx = d(1)new
20 while (i < iter && dnew > ε2dx){
21                                     SYNCHRONIZACJA
22                                     → d(1)(DtD)
23 q(1) = A(1)d(1)                                q(2) = A(2)d(2)
24                                     SYNCHRONIZACJA
25                                     ← q(2)(DtD)
26 q(1) = q(1) + q(2) // q(1) = [q(1); q(2)]
27 dq(1) = d(1)Tq(1)
28 α =  $\frac{d_{(1)new}}{dq_{(1)}}$ 
29 x(1) = x(1) + αd(1)
30 r(1) = r(1) - αq(1)
31 s(1) = prekondycjoner
32 d(1)old = d(1)new
33 d(1)new = r(1)Ts(1)
34 β =  $\frac{d_{(1)new}}{d_{(1)old}}$ 
35 d(1) = s(1) + βd(1)
36 i = i + 1
37 }
38 }

```

WYDRUK C.7: Schemat obliczeń metody gradientów sprzężonych z wielopoziomowym operatorem ściskającym o cyklu V z wariantem wykonania operacji *matvec* dla macierzy z rys. 7.1.



## Dodatek D

# FSMA - algorytm sumowania macierzy rzadkich

Po wygenerowaniu cząstkowych macierzy sztywności  $\mathbf{S}_i$  i bezwładności  $\mathbf{T}_i$ <sup>1</sup> w wariancie iteracyjnym zaprezentowanym na rysunku 5.1 macierze w formacie CRS ( $\mathbf{Sval}, \mathbf{Sval}, \mathbf{STcol}, \mathbf{STptr}$ ) są gotowe do przesłania z pamięci GPU do pamięci CPU RAM. Po zakończeniu przesyłania danych zostaje ustawiona flaga, która informuje, że zostało ono zakończone. Wartość tej flagi jest sprawdzana przez wątek zarządzający obliczeniami na CPU  $T_{master}$  (rys. 5.1, r. (5.1)).

Następnie na CPU wywoływany jest zrównoleglony algorytm sumowania macierzy rzadkich zapisanych w formacie CRS, który oznaczono w skrócie jako algorytm FSMA (ang. *Fast Sparse Matrix Addition*, [89]), którego autorem jest dr inż. Piotr Sypek. Algorytm stosowany jest do przeprowadzenia dodawania macierzy cząstkowych do macierzy wyjściowych:  $\mathbf{S} = \mathbf{S} + \mathbf{S}_i$ ,  $\mathbf{T} = \mathbf{T} + \mathbf{T}_i$ . Sumowanie macierzy sztywności  $\mathbf{S}$  i bezwładności  $\mathbf{T}$  przeprowadzenie jest równocześnie. Cechą szczególną algorytmu FSMA jest możliwość przeprowadzenia jednoczesnego sumowania kilku ( $N$ ) macierzy cząstkowych:  $\mathbf{S} = \mathbf{S} + \sum_{i=1}^n \mathbf{S}_i$ ,  $\mathbf{T} = \mathbf{T} + \sum_{i=1}^n \mathbf{T}_i$  dla  $n = 1, 2, \dots, N$ .

Ponieważ algorytm FSMA umożliwia przeprowadzenie sumowania  $N$  macierzy cząstkowych równocześnie w czasie znacznie krótszym od czasu przeprowadzenia  $N$  sumowań tych macierzy, to możliwość ukrycia obliczeń na CPU względem obliczeń na GPU znacznie wzrasta. Wskazana zaleta algorytmu FSMA jest szczególnie istotna w sytuacji, gdy wiele akceleratorów generuje macierze cząstkowe z szybkością na tyle dużą, że algorytm FSMA nie byłby w stanie równie szybko przeprowadzić ich sumowania realizowanego oddzielnie dla każdej pary macierzy cząstkowych. Rozwiązaniem sytuacji, w której CPU nie nadąża z obliczeniami względem szybko działającej generacji macierzy cząstkowych na GPU, jest zastosowanie wielu buforów danych na CPU RAM (w których przechowywane są dane generowane przez kernele działające na GPU) w połączeniu z jednoczesnym sumowaniem wielu macierzy cząstkowych realizowanym przez algorytm FSMA.

Sumowanie macierzy cząstkowych nie zawsze jest przeprowadzane automatycznie po otrzymaniu danych z GPU. W sytuacji, gdy obliczenia są przeprowadzane na wielu akceleratorach graficznych wątek zarządzający obliczeniami na CPU uruchamia obliczenia, gdy w buforze pojawi się liczba macierzy cząstkowych równa liczbie akceleratorów stosowanych w obliczeniach lub jej wielokrotności.

---

<sup>1</sup>Indeks „i” - określa numer iteracji w iteracyjnym wariancie generacji macierzy w metodzie elementów skończonych (patrz. 5.1)

Algorytm FSMA realizuje zrównoleglone sumowanie macierzy poprzez rozdzielenie obliczeń na wątki systemowe. Kluczowym parametrem jest liczba podmacierzy, na którą zostanie podzielona macierz  $\mathbf{S}$  i  $\mathbf{T}$ . Każda z tych podmacierzy obejmuje wszystkie kolumny i wybrany zakres wierszy macierzy podstawowej. Algorytm FSMA realizuje zadane obliczenia poprzez przydział wybranej podmacierzy do wątku FSMA nieobciążonego obliczeniami. Zalecane jest, aby w algorytmie FSMA jeden wątek przetwarzał dane z wielu podmacierzy. Ponieważ domyślnie liczba tych podmacierzy jest znacznie większa w porównaniu do liczby uruchomionych wątków ( $T_{Mt}$ , r. (5.1)), to algorytm FSMA realizuje obliczenia z równoważeniem obciążenia rdzeni CPU (ang. *load balancing*).

Każda z podmacierzy zarządzanych przez algorytm FSMA opisana jest przez osobne tablice danych, tak iż algorytm FSMA nie alokuje jednego obszaru pamięci dla zbioru indeksów wierszy i kolumn oraz wektora wartości macierzy  $\mathbf{S}$  i  $\mathbf{T}$  zapisanych w formacie CRS. Opisana zasada umożliwia dynamiczne zwiększenie rozmiaru pamięci przydzielonej do reprezentacji każdej z podmacierzy w sytuacji, gdy po dodaniu kolejnych macierzy cząstkowych liczba elementów niezerowych w zadanej podmacierzy wzrasta.

Po dodaniu macierzy cząstkowych  $\mathbf{S}_i$  i  $\mathbf{T}_i$  do macierzy wynikowych  $\mathbf{S}$  i  $\mathbf{T}$  przeprowadzana jest zmiana wartości wewnętrznych flag, tak iż buforzy zawierające te macierze cząstkowe oznaczone zostają jako puste co umożliwia kopiowanie do nich nowych danych.

Po zakończeniu generacji wszystkich macierzy cząstkowych przez kernele GPU i przeprowadzeniu ich sumowania wymagane jest zbudowanie macierzy wynikowych  $\mathbf{S}$  i  $\mathbf{T}$  na podstawie podmacierzy zdefiniowanych w algorytmie FSMA. Ten etap algorytmu FSMA określony został jako postprocessing algorytmu FSMA i jest również zrównoleglony. Dopiero po zakończeniu obliczeń realizowanych przez wątki w etapie postprocessingu algorytmu FSMA macierze  $\mathbf{S}$  i  $\mathbf{T}$  zapisane w formacie CRS posiadają osobne wektory z indeksami kolumn i skompresowanymi indeksami wierszy.

## Dodatek E

# Konwersja macierzy z formatu COO do formatów CCS lub CRS

### E.1 Konwersja sekwencyjna

Funkcja `UMFPACK_triplet_to_col` dostępna w bibliotece `UMFPACK` umożliwia konwersję macierzy rzadkiej z formatu COO do formatu CCS z eliminacją duplikatów. Przed wykonaniem konwersji macierz rzadka reprezentowana jest przez trzy wektory  $I$  - indeksy wierszy,  $J$  - indeksy kolumn,  $V$  - indeksy wartości niezerowych. Po wykonaniu konwersji, macierz reprezentowana jest w formacie CCS bez duplikatów. Ponieważ funkcja `UMFPACK_triplet_to_col` konwertuje macierze z formatu COO do formatu CCS to w tym dodatku sformułowano właśnie taką konwersję. Warto jednak zauważyć, iż zamiana na wejściu wektora indeksów wierszy ( $I$ ) z wektorem indeksów kolumn ( $J$ ) zmienia rodzaj formatu wyjściowego na CRS. Co więcej, w przypadku macierzy symetrycznych (i taka sytuacja występuje dla macierzy generowanych w niniejszej rozprawie) reprezentacja macierzy w formacie CRS i CCS jest identyczna.

Poszczególne etapy konwersji zaprezentowano w Algorytm 1 i zademonstrowano dla przykładowej macierzy rzadkiej o rozmiarze  $3 \times 3$  (10 elementów niezerowych w tym 4 duplikaty, rys. E.1). Etapy konwersji można podzielić na dwie części. W etapach (A1-1)-(A1-3) macierz konwertowana jest do formatu CRS i eliminowane są duplikaty (A1-4). W drugiej części, macierz rzadka konwertowana jest do formatu CCS. To podejście pozwala uzyskać reprezentację macierzy w postaci trzech wektorów:  $A_p$ ,  $A_i$  i  $A_x$ , bez duplikatów i z elementami posortowanymi wewnątrz kolumn w porządku rosnących indeksów wierszy.

<b>Macierz w formacie COO</b>		
$I = [2, 0, 1, 2, 1, 0, 2, 0, 1, 0]$ $J = [0, 2, 1, 0, 1, 0, 2, 0, 0, 0]$ $V = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]$		
6.0 8.0 10.0		2.0
9.0	3.0 5.0	
1.0 4.0		7.0

RYSUNEK E.1: Macierz rzadka w formacie COO wybrana do ilustracji procesu konwersji.

---


**ALGORYTM 1:** Etapy konwersji dostępnej w bibliotece UMFPACK (funkcja `UMFPACK_triplet_to_col`).

---

- (A1-1) Wyznaczenie liczby elementów niezerowych w każdym wierszu (również duplikatów)
  - (A1-2) Kompresja wektora indeksów wierszy macierzy z duplikatami
  - (A1-3) Budowa macierzy w formacie CRS (z duplikatami)
  - (A1-4) Eliminacja duplikatów
  - (A1-5) Wyznaczenie liczby elementów niezerowych w każdej kolumnie (bez duplikatów)
  - (A1-6) Kompresja wektora indeksów kolumn macierzy (bez duplikatów)  $A_p$
  - (A1-7) Budowa macierzy w formacie CCS (bez duplikatów)  $A_i, A_x$
- 

### (A1-1) Wyznaczenie liczby elementów niezerowych w każdym wierszu


W pierwszym etapie konwersji obliczana jest liczba elementów niezerowych w każdym wierszu (rys. E.2). Przed tym etapem wektor pomocniczy  $W$  został zainicjalizowany wartościami zerowymi. Po wykonaniu tego etapu liczba elementów niezerowych w  $i$ -tym wierszu przechowywana jest w  $W[i]$ .

<pre>// (A1-1) for (k = 0 ; k &lt; nz ; k++){     i = I [k] ;     W [i]++ ; }</pre>		<pre>W[0] = 4 W[1] = 3 W[2] = 3</pre>
---	---	---------------------------------------

RYSUNEK E.2: Wyznaczenie liczby elementów niezerowych w każdym wierszu.

### (A1-2) Kompresja wektora indeksów wierszy macierzy z duplikatami

W następnym kroku przy użyciu algorytmu *prefix sum* kompresowany jest wektor indeksów wierszy  $R_p$  (rys. E.3).

<pre>// (A1-2) Rp [0] = 0 ; for (i = 0 ; i &lt; n_row ; i++){     Rp [i+1] = Rp [i] + W [i];     W [i] = Rp [i]; }</pre>		<pre>//Rp - prefix sum //W[i] = Rp[i]  Rp[0] = 0,  W[0] = 0 Rp[1] = 4,  W[1] = 4 Rp[2] = 7,  W[2] = 7 Rp[3] = 10</pre>
--	---	--

RYSUNEK E.3: Kompresja wektora indeksów wierszy macierzy z duplikatami.

### (A1-3) Budowa macierzy w formacie CRS (z duplikatami)

W kolejnym kroku konstruowane są wektory indeksów kolumn i wartości niezerowych (rys. E.4). Po wykonaniu tego etapu, macierz rzadka reprezentowana jest w formacie CRS z duplikatami:  $R_p$  - skompresowany wiersz ( $R_p[i]$  wskazuje na początek  $i$ -ego wiersza),  $R_j$  - wektor indeksów kolumn,  $R_x$  - wektor wartości niezerowych.

```
// (A1-3)
for (k = 0 ; k < nz ; k++){
    p = W [ I [k] ]++ ;
    Rj [p] = J [k] ;
    Rx [p] = V [k] ;
}

// Rx - wartości niezerowe
// Rj - indeksy column
// Rp - skompresowany wiersz
```

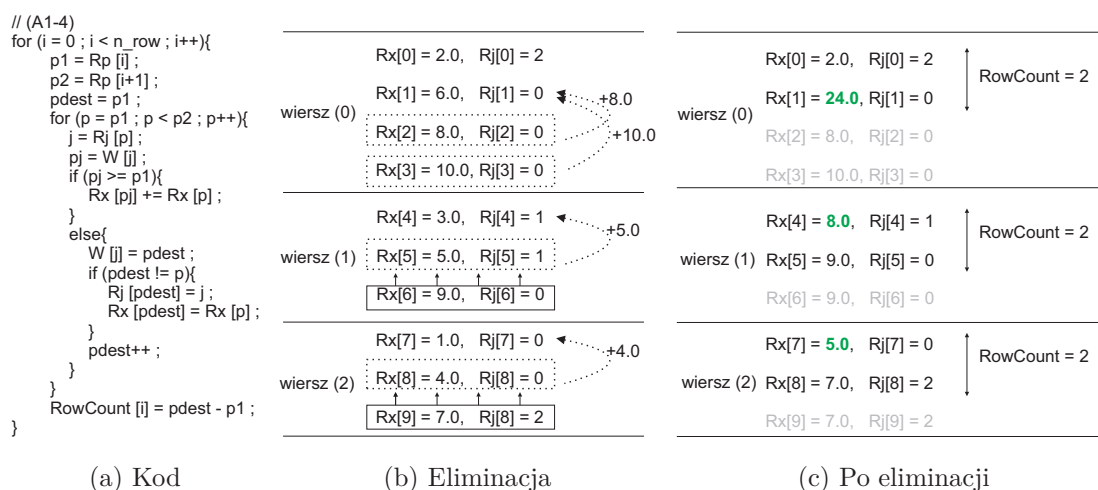
➔

wiersz (0)	Rx[0] = 2.0, Rj[0] = 2 Rx[1] = 6.0, Rj[1] = 0 Rx[2] = 8.0, Rj[2] = 0 Rx[3] = 10.0, Rj[3] = 0
wiersz (1)	Rx[4] = 3.0, Rj[4] = 1 Rx[5] = 5.0, Rj[5] = 1 Rx[6] = 9.0, Rj[6] = 0
wiersz (2)	Rx[0] = 1.0, Rj[0] = 0 Rx[1] = 4.0, Rj[1] = 0 Rx[2] = 6.0, Rj[2] = 2

RYSUNEK E.4: Budowa macierzy w formacie CRS (z duplikatami).

### (A1-4) Eliminacja duplikatów

W każdym wierszu macierzy, w pętli, sprawdza się czy występują elementy o takich samych indeksach kolumn (rys. E.5). Jeśli tak, elementy są sumowane, jeśli nie, to ustalana jest nowa pozycja wartości niezerowych w wektorach Rj i Rx. W wektorze RowCount zapisuje się liczbę elementów w wierszu po eliminacji duplikatów.



RYSUNEK E.5: Eliminacja duplikatów

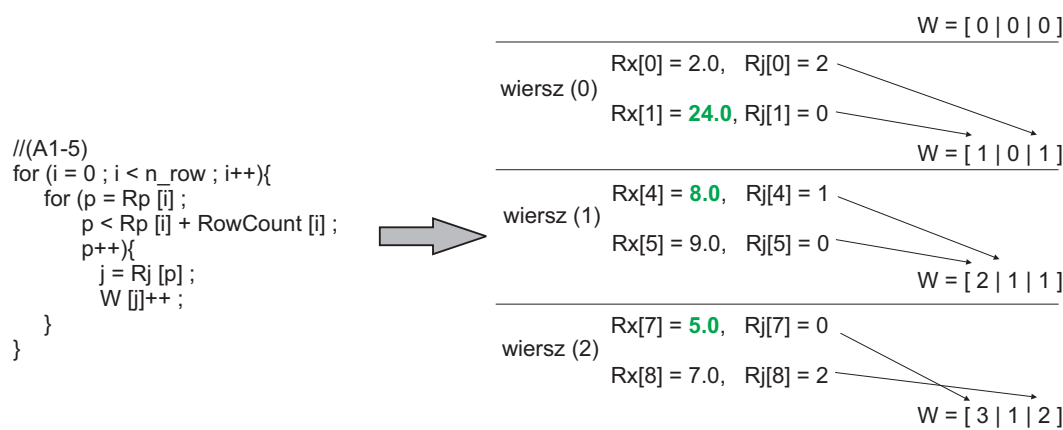
### (A1-5) Wyznaczenie liczby elementów niezerowych w każdej kolumnie (bez duplikatów)

W piątym etapie konwersji określana jest liczba elementów niezerowych w każdej kolumnie (rys. E.6). Wartość W[j] zwiększana jest dla każdej wartości niezerowej w j=Rj[p]-ej kolumnie.

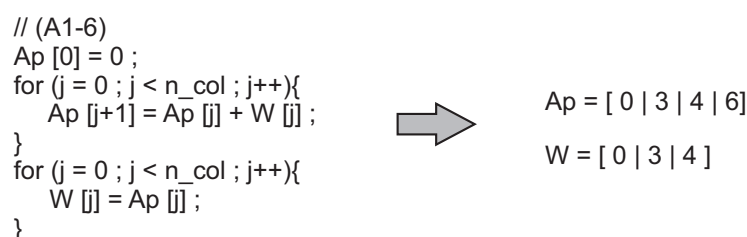
### (A1-6) Kompresja wektora indeksów kolumn macierzy (bez duplikatów)

W następnym kroku przy użyciu algorytmu *prefix sum* kompresowany jest wektor indeksów kolumn Ap (rys. E.7).





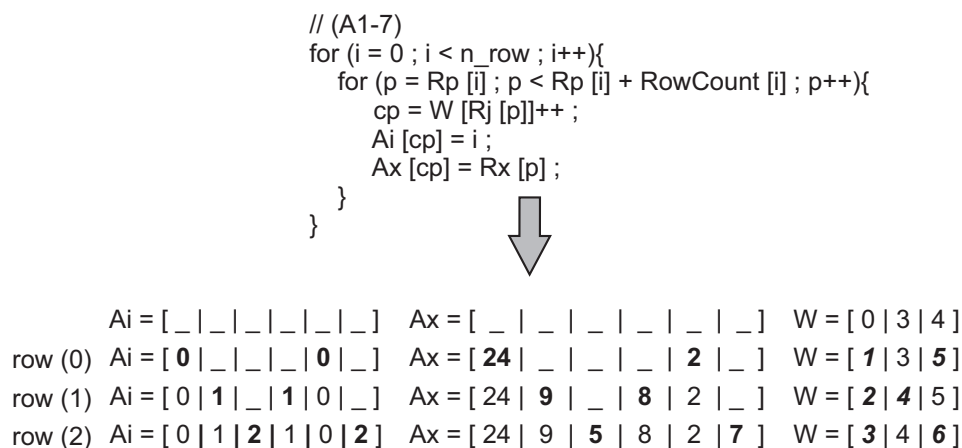
RYSUNEK E.6: Wyznaczenie liczby elementów niezerowych w każdej kolumnie



RYSUNEK E.7: Kompresja wektora indeksów kolumn macierzy (Ap)

### (A1-7) Budowa macierzy w formacie CCS (bez duplikatów)

W ostatnim etapie tworzone są wektory indeksów wierszy i wartości niezerowych (rys. E.8) i macierz rzadka reprezentowana jest w formacie CCS:  $A_i$  - wektor indeksów wierszy (posortowany w porządku rosnącym w każdej kolumnie),  $A_x$  - wektor wartości niezerowych,  $A_p$  - skompresowana kolumna ( $A_p[i]$  wskazuje na początek i-tej kolumny macierzy rzadkiej (rys. E.9)).

RYSUNEK E.8: Budowa macierzy w formacie CCS ( $A_i, A_x$ )

**CCS :**  
 $A_p = [0, 3, 4, 6]$   
 $A_i = [0, 1, 2, 1, 0, 2]$   
 $A_x = [24.0, 9.0, 5.0, 8.0, 2.0, 7.0]$

24		2.0
9.0	8.0	
5.0		7.0

RYSUNEK E.9: Macierz rzadka reprezentowana w formacie CCS. Kolor zielony wskazuje wyeliminowane duplikaty.

## E.2 Zrównoleglenie konwersji z biblioteki UMFPACK

Analiza implementacji konwersji z p. E.1 wskazała, że efektywnie można zrównoleglić etapy, w których wykonuje się algorytm *prefix sum*. W pozostałych etapach zrównoleglenie konwersji (Algorytm 2) napotyka następujące problemy:

1. w trakcie równoległych obliczeń w etapach (A2-1), (A2-3), (A2-5), (A2-7) mogą wystąpić konflikty podczas zapisu do pamięci i aby zapewnić poprawność obliczeń należy zastosować operacje atomowe.  
Przykład (etap (A2-1), obliczenie liczby elementów w wierszu): jeżeli wątek  $k$  jest przypisany do wartości niezerowej o indeksie  $k$ , to mogą wystąpić konflikty podczas zapisu danych w  $W[i]++$ , gdzie  $i=I[k]$  jest indeksem wiersza (rys. E.2),
2. w etapie (A2-4), w pętli `for(p=p1; p<p2; p++){...}` (rys. E.5), należy wykonać kosztowne sekwencyjne wyszukiwanie duplikatów,
3. równoległa praca wątków nie gwarantuje tego, że elementy w wektorach wyjścio-

---

**ALGORYTM 2:** Bezpośrednie zrównoleglenie konwersji UMFPACK\_triplet\_to\_col. (+) oznacza możliwość efektywnego zrównoleglenia obliczeń, (-) oznacza problemy ze zrównolegleniem obliczeń.  $N$  - liczba wierszy macierzy,  $Nz$  - liczba elementów niezerowych wraz z duplikatami,  $Nz'$  - liczba elementów niezerowych bez duplikatów.

---

(A2-1) Wyznaczenie liczby elementów niezerowych w każdym wierszu [(-)  $Nz$  operacji atomowych]

(A2-2) Kompresja wektora indeksów wierszy macierzy [(+) zrównoleglone skanowanie *prefix sum*]

(A2-3) Budowa macierzy w formacie CRS [(-)  $Nz$  operacji atomowych]

(A2-4) Eliminacja duplikatów [(+/-) jeden wątek przypisany do obliczeń w jednym wierszu, sekwencyjne wyszukanie duplikatów]

(A2-5) Wyznaczenie liczby elementów niezerowych w każdej kolumnie [(-)  $Nz'$  operacji atomowych]

(A2-6) Kompresja wektora indeksów kolumn macierzy ( $A_p$ ) [(+) zrównoleglone skanowanie *prefix sum*]

(A2-7) Budowa macierzy w formacie CCS ( $A_i$ ,  $A_x$ ) [(-)  $Nz$  operacji atomowych]

(A2-8) Sortowanie ( $A_i$ ,  $A_x$ ) [(-/+)] dodatkowa operacja, jeden wątek przypisany do obliczeń w jednej kolumnie]

---

wych są posortowane ( $A_i$ ,  $A_x$ ) w porządku rosnącym i dlatego należy wykonać dodatkowe sortowanie (A2-8).

# Podziękowania

Pragnę serdecznie podziękować promotorowi Profesorowi Michałowi Mrozowskiemu, który w okresie prac nad niniejszą rozprawą nie szczędził mi swojego czasu, zawsze służył wieloma cennymi radami, wyrozumiałością i cierpliwością.

Dziękuję Profesorowi Michałowi Okoniewskiemu za możliwość odbycia trzymiesięcznego stażu na Uniwersytecie w Calgary w ramach Stypendium Doktorskiego NCN Etiuda 1, możliwość wykonania badań oraz testów na stacji roboczej z dwoma akceleratorami Tesla K20c, konsultacje i owocną pracę w zespole naukowo-badawczym Profesora.

Szczególne podziękowania kieruję do promotora pomocniczego dra inż. Adama Lamęckiego. Opracowane przeze mnie strategie, algorytmy i ich implementacje ze zrównoleglonymi obliczeniami na akceleratorach graficznych zostały opracowane dla sformułowań MES użytych w symulatorze *InventSim*. Dziękuję również za wszelką pomoc w okresie prac nad niniejszą rozprawą.

Dziękuję bardzo dr inż. Piotrowi Sypkowi za liczne dyskusje i pomoc w opracowywaniu nowych rozwiązań algorytmicznych podczas tworzenia niniejszej rozprawy.

Mojej kochanej i wspierającej żonie dziękuję za nieustanną wiarę we mnie, wsparcie i natchnienie. Jestem świadom, że poświęciła wiele bym mógł w spokoju pracować nad rozprawą.



# Bibliografia

- [1] D. B. Davidson. *Computational Electromagnetics for RF and Microwave Engineering*. Cambridge University Press, 2005.
- [2] M. O. Sadiku. *Numerical Techniques in Electromagnetics 2nd ed.* CRC Press, 2001.
- [3] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Boston, 2003.
- [4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [5] ANSYS HFSS, <http://www.ansys.com/Products/Simulation+Technology/Electronics/Signal+Integrity/ANSYS+HFSS>.
- [6] Intel® Many Integrated Core Architecture (Intel® MIC Architecture), <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>.
- [7] J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. NVIDIA Corporation, 2011.
- [8] General-Purpose computing on Graphics Processor Units, <http://gpgpu.org>.
- [9] Stanford University Graphics Lab. BrookGPU - Brook stream program language for modern graphics hardware, <http://graphics.stanford.edu/projects/brookgpu/index.html>.
- [10] ATI CTM, [http://ati.amd.com/companyinfo/researcher/documents/ATI\\_CTM\\_Guide.pdf](http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf).
- [11] NVIDIA Co. Computed Unified Device Architecture, [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html).
- [12] Exascale Supercomputing, <http://www.nvidia.com/object/exascale-supercomputing.html>.
- [13] S. E. Krakiwsky, L. E. Turner, M. Okoniewski. Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU). *Microwave Symposium Digest, 2004 IEEE MTT-S International*, wolumen 2, strony 1033–1036, 2004.

- [14] M. J. Inman, A. Z. Elsherbeni. Programming Video Cards for Computational Electromagnetics Applications. *Antennas and Propagation Magazine, IEEE*, 47(6):71–78, 2005.
- [15] D. De Donno, A. Esposito, L. Tarricone, L. Catarinucci. Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]. *Antennas and Propagation Magazine, IEEE*, 52(3):116–122, 2010.
- [16] S. Adams, J. Payne, R. Boppana. Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors. *High Performance Computing Modernization Program Users Group Conference*, strony 334–338, June 2007.
- [17] P. Sypek, A. Dziekonski, M. Mrozowski. How to Render FDTD Computations More Effective Using a Graphics Accelerator. *Magnetics, IEEE Transactions on*, 45(3):1324–1327, 2009.
- [18] C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, M. Okoniewski. Speed It Up. *Microwave Magazine, IEEE*, 11(2):70–78, April 2010.
- [19] M. Livesey, J. F. Stack, F. Costen, T. Nanri, N. Nakashima, S. Fujino. Development of a CUDA Implementation of the 3D FDTD Method. *IEEE Antennas and Propagation Magazine*, 54(5):186–195, 2012.
- [20] C. Y. Ong, M. Weldon, D. Cyca, M. Okoniewski. Acceleration of Large-Scale FDTD Simulations on High Performance GPU Clusters. *Antennas and Propagation Society International Symposium, 2009. APSURSI '09. IEEE*, strony 1–4, June 2009.
- [21] T. Topa, A. Noga, A. Karwowski. Adapting MoM With RWG Basis Functions to GPU Technology Using CUDA. *Antennas and Wireless Propagation Letters, IEEE*, 10:480–483, 2011.
- [22] S. Peng, Z. Nie. Acceleration of the Method of Moments Calculations by Using Graphics Processing Units. *Antennas and Propagation, IEEE Transactions on*, 56(7):2130–2133, 2008.
- [23] T. P. Stefanski, T. D. Drysdale. Acceleration of the 3D ADI-FDTD Method Using Graphics Processor Units. *Microwave Symposium Digest, 2009. MTT '09. IEEE MTT-S International*, strony 241–244, 2009.
- [24] F. Rossi, P. P. M. So. Hardware Accelerated Symmetric Condensed Node TLM Procedure for NVIDIA Graphics Processing Units. *IEEE Antennas Propag. Soc. Int. Symp. (APSURSI'09)*, strony 1–4, 2009.
- [25] J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley and Sons Inc., 2002.
- [26] J. L. Volakis, A. Chatterjee, L. C. Kempel. *Finite Element Method for Electromagnetics. Antennas, Microwave Circuits and Scattering Applications*. IEEE Series on Electromagnetic Wave Theory, 1998.



- [27] T. Cabel, J. Charles, S. Lanteri. Multi-GPU Acceleration of a DGTD Method for Modeling Human Exposure to Electromagnetic Waves. Research Report RR-7592, 2011.
- [28] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon, J.-M. Jin. GPU Accelerated Finite-Element Computation for Electromagnetic Analysis. *Antennas and Propagation Magazine, IEEE*, 56(2):39–62, 2014.
- [29] K. Kourtis, G. Goumas, N. Koziris. Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression. *2013 42nd International Conference on Parallel Processing*, strony 511–519, 2008.
- [30] F. Vazquez, E. M. Garzon, J. A. Martinez, J. J. Fernandez. The sparse matrix vector product on GPUs. *Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering*, wolumen 2, strony 1081–1092, 2009.
- [31] N. Bell, M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Raport instytutowy, NVIDIA Co., 2008.
- [32] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishop. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
- [33] H. Anzt, S. Tomov, J. Dongarra. Implementing a Sparse Matrix Vector Product for the SELL-C\SELL-C- $\sigma$  formats on NVIDIA GPUs. Raport instytutowy, University of Tennessee, Department of Electrical Engineering & Computer Science, 2014.
- [34] D. Langr, P. Tvrdik. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems*, (w druku).
- [35] R. Li, Y. Saad. GPU-Accelerated Preconditioned Iterative Linear Solvers. *Journal of Supercomputing*, 63(2):443–466, 2013.
- [36] S. Georgescu, H. Okuda. Conjugate Gradients on Multiple GPUs. *International Journal for Numerical Methods in Fluids*, 64:1254–1273, 2010.
- [37] M. Naumov. Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, <http://docs.nvidia.com/cuda/incomplete-lu-cholesky/>.
- [38] D. Goeddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, S. Turek. Using GPUs to Improve Multigrid Solver Performance on a Cluster. *International Journal on Computer Science and Engineering*, 4(1):36–55, 2008.
- [39] Matrix Algebra on GPU and Multicore Architectures, <http://icl.cs.utk.edu/magma/index.html>.
- [40] AmgX, <https://developer.nvidia.com/amgx>.

- [41] Sparse Parallel Robust Algorithms Library, <http://www.numerical.rl.ac.uk/spral/>.
- [42] NVIDIA cuSOLVER (CUDA 7.0), <https://developer.nvidia.com/cusolver>.
- [43] Intel Math Kernel Library (Intel MKL), <http://software.intel.com/en-us/articles/intel-mkl>.
- [44] P. Plaszewski, K. Banas, P. Maciol. Higher order FEM numerical integration on GPUs with OpenCL. *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT)*, 2010.
- [45] P. Maciol, P. Plaszewski, K. Banas. 3D finite element numerical integration on GPUs. *Procedia Computer Science*, 1(1):1093–1100, 2010.
- [46] K. Banas, P. Plaszewski, P. Maciol. Numerical Integration on GPUs for Higher Order Finite Elements. *Computers and Mathematics with Applications*, 67:1319–1344, 2014.
- [47] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, S. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1):80–97, 2013.
- [48] C. Cecka, A. J. Lew, E. Darve. Assembly of Finite Element Methods on Graphics Processors. *International Journal for Numerical Methods in Engineering*, 85(3):640–669, 2011.
- [49] S. Georgescu, P. Chow, H. Okuda. GPU Acceleration for FEM-Based Structural Analysis. *Archives of Computational Methods in Engineering*, 20(2):111–121, 2013.
- [50] I. Z. Reguly, M. B. Giles. Finite Element Algorithms and Data Structures on Graphical Processing Units. *International Journal of Parallel Programming*, 43(2):203–239, 2015.
- [51] Z. Fu, T. J. Lewis, R. M. Kirby, R. T. Whitaker. Architecting the Finite Element Method Pipeline for the GPU. *Journal of Computational and Applied Mathematics*, 257(0):195 – 211, 2014.
- [52] O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu. *The Finite Element Method for Fluid Dynamics, In The Finite Element Method for Fluid Dynamics (Seventh Edition)*. Butterworth-Heinemann, Oxford, 2014.
- [53] O. C. Zienkiewicz, R. L. Taylor, D. Fox. *The Finite Element Method for Solid and Structural Mechanics, In The Finite Element Method for Solid and Structural Mechanics (Seventh Edition)*. Butterworth-Heinemann, Oxford, 2014.
- [54] S. M. Musa. *Computational Finite Element Methods in Nanotechnology*. CRC Press, 2012.
- [55] J. P. Swartz, D. B. Davidson. Curvilinear Vector Finite Elements Using a Set of Hierarchical Basis Functions. *IEEE Transactions on Antennas and Propagation*, 55(2):400–406, 2007.

- [56] G. Pelosi, R. Coccioli, S. Selleri. *Quick Finite Elements for Electromagnetic Waves*. Artech House, Inc., 2009.
- [57] A. Polycarpou. *Introduction to the Finite Element Method in Electromagnetics*. Morgan & Claypool Publishers, 2006.
- [58] J. Schöber. NETGEN An Advancing Front 2D 3D Mesh Generator Based on Abstract Rules. *Computing and Visualization in Science*, 1:41–52, 1997.
- [59] TETGEN, <http://wias-berlin.de/software/tetgen/>.
- [60] GMSH, <http://geuz.org/gmsh/>.
- [61] Y. Zhu, A. Cangellaris. *Multigrid Finite Element Methods For Electromagnetic Field Modeling*. Wiley-Interscience, 2006.
- [62] T. S. Chu, T. Itoh. Generalized Scattering Matrix Method for Analysis of Cascaded and Offset Microstrip Step Discontinuities. *IEEE Transactions on Microwave Theory and Techniques*, 34(2):280–284, 1986.
- [63] J. Rubio, J. Arroyo, J. Zapata. Analysis of Passive Microwave Circuits by Using a Hybrid 2-D and 3-D Finite-Element Mode-Matching Method. *IEEE Transactions on Microwave Theory and Techniques*, 47(9), 1999.
- [64] P. Ingelström. A New Set of H(curl)-Conforming Hierarchical Basis Functions for Tetrahedral Meshes. *Microwave Theory and Techniques, IEEE Transactions on*, 54:106–114, 2006.
- [65] S. M. Schnepp, T. Weiland. Efficient Large Scale Electromagnetics Simulations Using Dynamically Adapted Meshes with the Discontinuous Galerkin Method. *Journal of Computational and Applied Mathematics*, 236(18):4909–4924, 2012.
- [66] P. Ingelström, V. Hill, R. Dyczij-Edlinger. Goal-Oriented Error Estimates for hp-Adaptive Solutions of the Time-Harmonic Maxwell’s Equations Electromagnetic Field Computation. *12th Biennial IEEE Conference on*, strony 396–396, 2006.
- [67] J. P. Webb. Hierarchal Vector Basis Functions of Arbitrary Order for Triangular and Tetrahedral Finite Elements. *IEEE Transactions on Antennas and Propagation*, 47(8):1244–1253, 1999.
- [68] J. S. Savage. Comparing High Order Vector Basis Functions. *In Proceedings of the 14th Annual Review of Progress in Applied Computational Electromagnetics*, strony 742–749, 1988.
- [69] J. P. Webb, B. Forghani. Hierarchal Scalar and Vector Tetrahedra. *IEEE Transactions on Magnetism*, 29:1495–1498, 1993.
- [70] L. S. Andersen, J. L. Volakis. Hierarchical Tangential Vector Finite Elements for Tetrahedra. *IEEE Microwave Guided Wave Letter*, 8:127–129, 1998.
- [71] J. C. Nédélec. Mixed Finite Elements in  $\mathbb{R}^3$ . *Numerische Mathematik, Springer-Verlag*, 35(3):315–341, 1980.

- [72] L. Zhang, T. Cui, H. Liu. A Set of Symmetric Quadrature Rules on Triangles and Tetrahedra. *Journal of Computational Mathematics*, 26(3):1–16, 2008.
- [73] CUSPARSE Library v.7.0, <http://docs.nvidia.com/cuda/cusparse/>.
- [74] J. R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Raport instytutowy, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.
- [75] Y. Zhu, A. C. Cangellaris. Hierarchical Multilevel Potential Preconditioner for Fast Finite-Element Analysis of Microwave Devices. *IEEE Transactions on Microwave Theory and Techniques*, 50(8):1984–1989, 2002.
- [76] T. Sogabe, S.-L. Zhang. A COCR Method for Solving Complex Symmetric Linear Systems. *Journal of Computational and Applied Mathematics*, 199(2):297 – 303, 2007. Special Issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004) Special Issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004).
- [77] A. Dziekonski, A. Lamecki, M. Mrozowski. Jacobi and Gauss-Seidel Preconditioned Complex Conjugate Gradient Method with GPU Acceleration for Finite Element Method. *Microwave Conference (EuMC), 2010 European*, strony 1305–1309, 28-30 Sept. 2010.
- [78] M. M. Ilıc, B. M. Notaros. Higher Order Large-Domain Hierarchical FEM Technique for Electromagnetic Modeling Using Legendre Basis Functions on Generalized Hexahedra. *Electromagnetics*, 26(7):517–529, 2006.
- [79] E. Jorgensen, J. L. Volakis, P. Meincke, O. Breinbjerg. Higher Order Hierarchical Legendre Basis Functions for Electromagnetic Modeling. *Antennas and Propagation, IEEE Transactions on*, 52(11):2985–2995, Nov 2004.
- [80] W. L. Briggs, V. E. Henson, S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2000.
- [81] A. Dziekonski, A. Lamecki, M. Mrozowski. GPU Acceleration of Multilevel Solvers for Analysis of Microwave Components With Finite Element Method. *Microwave and Wireless Components Letters, IEEE*, 1:1–3, 2011.
- [82] Nvidia. *CUDA Programming Guide Version 4.0*. Nvidia, czerwiec 2011. [http://www.nvidia.com/object/cuda\\_develop.html](http://www.nvidia.com/object/cuda_develop.html).
- [83] A. Dziekonski. Zastosowanie procesorów graficznych do rozwiązywania zagadnień elektrodynamiki obliczeniowej. Praca magisterska, Politechnika Gdańska, 2009.
- [84] NVIDIA Co. Whitepaper-NVIDIA’s Next Generation CUDA Compute Architecture Kepler GK110, <http://www.nvidia.com/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
- [85] CUBLAS Library v.7.0, <http://docs.nvidia.com/cuda/cublas/>.

- [86] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Finite Element Matrix Generation on a GPU. *Progress In Electromagnetics Research*, 128:249–265, 2012.
- [87] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Accuracy, Memory, and Speed Strategies in GPU-Based Finite-Element Matrix-Generation. *Antennas and Wireless Propagation Letters, IEEE*, 11:1346–1349, 2012.
- [88] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. GPU-Accelerated Finite-Element Matrix Generation for Lossless, Lossy, and Tensor Media. *IEEE Antennas and Propagation Magazine [EM Programmer's Notebook]*, 56(5):186–197, 2014.
- [89] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Generation of Large Finite-Element Matrices on Multiple Graphics Processors. *International Journal for Numerical Methods in Engineering*, 94(2):204–220, 2012.
- [90] A. Lamecki, L. Balewski, M. Mrozowski. An Efficient Framework for Fast Computer Aided Design of Microwave Circuits Based on the Higher-Order 3D Finite-Element Method. *Radioengineering*, 23(4):970–978, 2014.
- [91] InventSim - 3D FEM Electromagnetic Simulation and Optimization Framework, <http://eminvent.com>.
- [92] CUSPARSE Library v.6.0, <http://docs.nvidia.com/cuda/cusparse/>.
- [93] SuiteSparse is a suite of sparse matrix algorithms, <http://faculty.cse.tamu.edu/davis/suitesparse.html>.
- [94] UMFPACK (unsymmetric multifrontal sparse LU factorization package), <http://www.cise.ufl.edu/research/sparse/umfpack>.
- [95] SPARSKIT: A basic toolkit for sparse matrix computations, <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/>.
- [96] Supernodal LU, <http://crd-legacy.lbl.gov/~xiaoye/SuperLU/>.
- [97] TAUCS, A Library of Sparse Linear Solvers, <http://www.tau.ac.il/~stoledo/taucs/>.
- [98] MUMPS: a MULTifrontal Massively Parallel sparse direct Solver, <http://mumps.enseeiht.fr/>.
- [99] D. Mukunoki, D. Takahashi. Optimization of Sparse Matrix-Vector Multiplication for CRS Format on NVIDIA Kepler Architecture GPUs. *Computational Science and Its Applications - ICCSA 2013*, 7975:211–223, 2013.
- [100] K. Kourtis, G. Goumas, N. Koziris. Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems. *ACM Trans. Archit. Code Optim.*, 7(3):1544–3566, 2010.
- [101] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, Z. Shao. Optimization of Sparse Matrix-Vector Multiplication with Variant CSR on GPUs. *Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, strony 165–172, Dec 2011.



- [102] T. Oberhuber, A. Suzuki, J. Vacata. New Row-Grouped CSR Format for Storing the Sparse Matrices on GPU with Implementation in CUDA. *Acta Technica*, 56:447–466, 2011.
- [103] A. Dziekonski, P. Sypek, L. Kulas, M. Mrozowski. Implementation of Matrix-Type FDTD Algorithm on a Graphics Accelerator. *Microwaves, Radar and Wireless Communications, 2008. MIKON 2008. 17th International Conference on*, strony 1–4, May 2008.
- [104] A. Monakov, A. Lokhmotov, A. Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. *High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science*, 5952:111–125, 2010.
- [105] F. Vazquez, G. Ortega, J. J. Fernandez, E. M. Garzon. Improving the Performance of the Sparse Matrix Vector Product with GPUs. *IEEE 10th International Conference on Computer and Information Technology (CIT)*, strony 1146–1151, 2010.
- [106] A. Dziekonski, A. Lamecki, M. Mrozowski. A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.
- [107] X. Liu, M. Smelyanskiy, E. Chow, P. Dubey. Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors. *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013.
- [108] S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.
- [109] J. Sim, A. Dasgupta K. Hyesoon, R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. *SIGPLAN Not.*, 47(8):11–22, Luty 2012.
- [110] S. Williams. Kolekcja macierzy rzadkich,  
<http://www.cise.ufl.edu/research/sparse/matrices/Williams/index.html>.
- [111] A. Dziekoński - strona domowa, <http://mwave.eti.pg.gda.pl/~adziekonski/downloads.html>.
- [112] E. Cuthill, J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. *Proceedings of the 1969 24th National Conference*, ACM '69, strony 157–172, 1969.
- [113] W. M. Chan, A. George. A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm. *BIT Numerical Mathematics*, 20:8–14, 1980.
- [114] A. Dziekonski, A. Lamecki, M. Mrozowski. Tuning A Hybrid GPU-CPU V-cycle Multilevel Preconditioner for Solving Large Real and Complex Systems of FEM Equations. *Antennas and Wireless Propagation Letters, IEEE*, 10:619–622, 2011.

- [115] O. Schenk, M. Bollhöfer, R. A. Römer. On Large Scale Diagonalization Techniques For The Anderson Model of Localization. *SIAM Review*, 50:91–112, 2008.
- [116] O. Schenk, A. Wächter, M. Hagemann. Matching-Based Preprocessing Algorithms to the Solution of Saddle-Point Problems in Large-Scale Nonconvex Interior-Point Optimization. *Computational Optimization and Applications*, 36:2–3, 2007.
- [117] X. Yang, S. Parthasarathy, P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proceedings of the VLDB Endowment (PVLDB)*, 4:231–242, 2011.
- [118] METIS - Graph Partitioning and Fill-reducing Matrix Ordering, [glaros.dtc.umn.edu/gkhome/views/metis](http://glaros.dtc.umn.edu/gkhome/views/metis).
- [119] Open Multi-Processing (OpenMP), <http://openmp.org/wp/>.
- [120] A. V. Knyazev. Preconditioned Eigensolvers: Practical Algorithms. Raport instytutowy, Denver, CO, USA, 1999.
- [121] H. Anzt, S. Tomov, J. Dongarra. Accelerating the LOBPCG Method on GPUs Using a Blocked Sparse Matrix Vector Product. Raport instytutowy, Department of Electrical Engineering and Computer Science, University of Tennessee, 2014.
- [122] M. A. Freitag. *Inner-outer Iterative Methods for Eigenvalue Problems - Convergence and Preconditioning*. Praca doktorska, University of Bath, 2007.
- [123] M. Rewienski, A. Lamecki, M. Mrozowski. An Extended Basis Inexact Shift-Invert Lanczos for the Efficient Solution of Large-Scale Generalized Eigenproblems. *Computer Physics Communications*, 184(9):2127 – 2135, 2013.
- [124] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, PA, 2000.
- [125] P. Mironowicz, A. Dziekonski, M. Mrozowski. A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU. *SIAM Journal on Scientific Computing* (złożony do drugiej recenzji 27.01.2014r.).
- [126] P. Mironowicz, A. Dziekonski, M. Mrozowski. Efficient Symmetric Sparse Matrix-Vector Product on a GPU, *Graphics Technology Conference 2014* [http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4222\\_sparse\\_matrix\\_vector\\_symmetric.pdf](http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4222_sparse_matrix_vector_symmetric.pdf).





# Prawo rozpowszechniania

Niniejszym wyrażam zgodę na wykorzystanie wyników mojej pracy w pracach badawczych i publikacjach przygotowywanych przez pracowników Politechniki Gdańskiej lub pod ich kierownictwem. Wykorzystanie wyników wymaga wskazania niniejszej rozprawy doktorskiej jako źródła.



# Sylwetka autora

Adam Dziekoński jest absolwentem Wydziału Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej. Pracę magisterską pod tytułem *Zastosowanie procesorów graficznych do rozwiązywania zagadnień elektrodynamiki obliczeniowej* obronił w Katedrze Inżynierii Mikrofalowej i Antenowej w grudniu 2009. Stacjonarne studia doktoranckie rozpoczął na Wydziale ETI Politechniki Gdańskiej w 2009 roku.

Mgr inż. Adam Dziekoński jest współautorem ośmiu publikacji, które wg. ISI Web of Science były cytowane 76 razy przez autorów obcych (H=5).

## Lista publikacji naukowych indeksowanych w pismach ISI:

1. P. Mironowicz, A. Dziekonski, M. Mrozowski. A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU. *SIAM Journal on Scientific Computing* (złożony do drugiej recenzji 27.01.2014r.).
2. A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. GPU-Accelerated Finite-Element Matrix Generation for Lossless, Lossy, and Tensor Media. *IEEE Antennas and Propagation Magazine [EM Programmer's Notebook]*, 56(5):186–197, 2014.
3. A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Generation of Large Finite-Element Matrices on Multiple Graphics Processors. *International Journal for Numerical Methods in Engineering*, 94(2):204–220, 2012.
4. A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Accuracy, Memory, and Speed Strategies in GPU-Based Finite-Element Matrix-Generation. *Antennas and Wireless Propagation Letters, IEEE*, 11:1346–1349, 2012.
5. A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Finite Element Matrix Generation on a GPU. *Progress In Electromagnetics Research*, 128:249–265, 2012.
6. A. Dziekonski, A. Lamecki, M. Mrozowski. Tuning A Hybrid GPU-CPU V-cycle Multilevel Preconditioner for Solving Large Real and Complex Systems of FEM Equations. *Antennas and Wireless Propagation Letters, IEEE*, 10:619–622, 2011.
7. A. Dziekonski, A. Lamecki, M. Mrozowski. A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.
8. A. Dziekonski, A. Lamecki, M. Mrozowski. GPU Acceleration of Multilevel Solvers for Analysis of Microwave Components With Finite Element Method. *Microwave and Wireless Components Letters, IEEE*, 1:1–3, 2011.

9. P. Sypek, A. Dziekonski, M. Mrozowski. How to Render FDTD Computations More Effective Using a Graphics Accelerator. *Magnetics, IEEE Transactions on*, 45(3):1324–1327, 2009.

**Lista referatów konferencyjnych (z uwzględnieniem prac popularnonaukowych):**

1. J. Mamza, P. Makyla, A. Dziekonski, A. Lamecki, M. Mrozowski. Multi-core and multiprocessor implementation of numerical integration in Finite Element Method. *Microwave Radar and Wireless Communications (MIKON), 2012 19th International Conference on*, strony 457–461, May 2012.
2. A. Dziekonski, A. Lamecki, M. Mrozowski. Jacobi and Gauss-Seidel Preconditioned Complex Conjugate Gradient Method with GPU Acceleration for Finite Element Method. *Microwave Conference (EuMC), 2010 European*, strony 1305–1309, 28-30 Sept. 2010.
3. A. Dziekonski, M. Mrozowski. Krylov space iterative solvers on graphics processing units. *Microwave Radar and Wireless Communications (MIKON), 2010 18th International Conference on*, strony 1–4, June 2010.
4. A. Dziekonski, M. Mrozowski. Tuning matrix-vector multiplication on GPU. *Information Technology (ICIT), 2010 2nd International Conference on*, strony 457–461, May 2010.
5. A. Dziekonski, P. Sypek, L. Kulas, M. Mrozowski. Implementation of Matrix-Type FDTD Algorithm on a Graphics Accelerator. *Microwave Radar and Wireless Communications (MIKON), 2008 17th International Conference on*, strony 1–4, May 2008.

**Lista stypendiów przyznanych w trybie konkursowym:**

1. InnoDoktorant – stypendia dla doktorantów, VI edycja, przyznawane przez Marszałka Województwa Pomorskiego, 2014r.
2. Stypendium doktorskie Etiuda 1, finansowane przez NCN, 2013r.
3. Stypendium Fundacji na rzecz Nauki Polskiej - Program START (2012,2013).
4. Stypendium na badania naukowe dla doktorantów i grantów dla młodych pracowników naukowych Wydziału Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej, 2011-2013r.
5. Stypendium z dotacji projakościowej, 2011–2013r.
6. Stypendium dla doktorantów z projektu Rozwój interdyscyplinarnych studiów doktoranckich na Politechnice Gdańskiej w zakresie nowoczesnych technologii, 2010–2013r.
7. MTT-S Undergraduate Fall 2008 Scholarship przyznane przez IEEE Microwave Theory and Techniques Society w 2012r.

**Nagrody i wyróżnienia:**

1. Nagroda Rektora Politechniki Gdańskiej dla Młodych pracowników Nauki (2012r.)
2. Dyplom Dziekana Wydziału ETI za wyróżniający dorobek publikacyjny (2012r.)
3. Przyznanie statusu *CUDA Research Center* zespołowi naukowemu Profesora Michała Mrozowskiego (2012,2013,2014).

**Udział w projektach krajowych i międzynarodowych:**

1. „Metody podprzestrzeni Kryłowa wykorzystujące wiele akceleratorów graficznych dla rozwiązywania problemów elektromagnetycznych średniej skali otrzymanych metodą elementów skończonych” projekt realizowany w ramach programu NCN OPUS 7 (2015-2017) - główny wykonawca.
2. „Szybkie projektowanie złożonych układów filtrów i multiplekserów o zwartej konstrukcji dla nowych systemów komunikacji bezprzewodowej z wykorzystaniem trójwymiarowych symulatorów elektromagnetycznych” projekt realizowany w ramach programu LIDER i finansowanego przez Narodowe Centrum Badań i Rozwoju (NCBiR), (2011-2013) - wykonawca.
3. „Systemy Antenowe i Sensory dla społeczeństwa Informacyjnego - nowe metody analizy teoretycznej i szybkiej symulacji numerycznej” w ramach projektu COST ASSIST ic0603 - Antenna Systems & Sensors for Information Society Techniques, (2009-2011) - wykonawca.
4. „Modelowanie kryształów i światłowodów fotonicznych z wykorzystaniem makro-modeli i specjalizowanego szybkiego mini-klastra ze sprzętowymi akceleratorami obliczeniowymi”, projekt badawczy nr 3 T11F 037 30, (2007-2008) - praktyki studenckie.