AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Inżynierii Metali i Informatyki Przemysłowej


# PROJEKT INŻYNIERSKI

pt.

# „Realizacja frontalnego solwera MES z wykorzystaniem technologii OpenCL”

| | |
|---|---|
| Imię i nazwisko dyplomanta: | **Paweł Wal** |
| Kierunek studiów: | **Informatyka Stosowana** |
| Nr albumu: | **240202** |
| Opiekun: | dr inż. Łukasz Rauch |

Podpis dyplomanta: Podpis opiekuna:

Kraków 2014

***Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.***

Kraków, dnia ……..…  Podpis dyplomanta…….

## Spis treści

# 1 WSTĘP

Ponad czterdzieści lat temu, w roku 1970, Bruce Irons opublikował swoją koncepcję programu rozwiązującego układy równań w postaci macierzy rzadkich metodą frontalną. Jego główną motywacją do opracowania tej techniki była nie tylko wydajność, ale również zużycie pamięci [5]. Irons pracował z komputerem ICT 1905, który w najlepszym razie mógł mieć 96 kilobajtów pamięci operacyjnej w postaci 32 768 słów po 24 bity. Nic więc w tym dziwnego, że poszukiwał rozwiązania jak najbardziej optymalnego pamięciowo. Dziś co prawda pamięć jest znacznie mniej ograniczonym zasobem – przeciętny komputer biurowy dysponuje na ogół kilkoma gigabajtami, a większość kart, które wykorzystuje się do obliczeń nie posiada mniej niż gigabajt. Jak jednak mówi przysłowie – apetyt rośnie w miarę jedzenia. Wraz ze wzrostem pamięci i mocy obliczeniowej wzrastają również rozmiary problemów – w tym tych z zakresu metody elementów skończonych, które interesują badaczy.

Wraz z biegiem lat technologia obliczeniowa ewoluowała. Kulminacją tej ewolucji są nowoczesne wielordzeniowe procesory i koprocesory obliczeniowe oraz karty graficzne. Szczególnie interesujące z punktu widzenia obliczeń wysokiej wydajności są te ostatnie. Składają się z jednego lub więcej procesorów strumieniowych, z których każdy dysponuje na ogół dużą ilością rdzeni obliczeniowych. Każdy z tych rdzeni może wykonywać obliczenia już nie tylko równolegle, ale w sposób masowo równoległy, niedostępny dla urządzeń opartych na klasycznych procesorach. Dla kodu obliczeniowego uruchamianego na takich urządzeniach istnieje jednak szereg ograniczeń [4]: transfery z pamięci dostępu ogólnego (pamięci hosta) do pamięci urządzenia są kosztowne czasowo, gdyż wymuszają okresy, w których ani CPU, ani GPU nie wykonują obliczeń, operacje atomowe na pamięci globalnej są powolne, problem stanowi również nadmierna dywergencja wątków.

Niniejsza praca jest poświęcona implementacji solwera podążającego za ideą solwera frontalnego zaprezentowanego w 1970 roku przez Ironsa, rozumianej jako rozłożenie złożonego problemu na serię częściowo tylko od siebie zależnych, lżejszych pamięciowo i znacznie prostszych obliczeniowo podproblemów. Celem przyświecającym tej pracy jest również stworzenie oprogramowania wykorzystującego w najlepszy sposób możliwości masowej równoległości oferowane przez nowoczesne, wysokowydajne urządzenia obliczeniowe.

Mimo, iż oprogramowanie podąża za myślą Ironsa, postulowana przez niego metoda rozwiązywania układów równań liniowych nie została dosłownie zastosowana. Zasada matematyczna, na której opiera się stworzone oprogramowanie jest wyprowadzona ze

zmodyfikowanej metody eliminacji Gaussa. Modyfikacje mają na celu wykorzystanie mocnych stron zastosowanych urządzeń obliczeniowych, jednocześnie omijając ich ograniczenia i potencjalne słabości.

Do realizacji samego oprogramowania została wykorzystana architektura i zestaw bibliotek OpenCL. Każdy z wiodących producentów sprzętu obliczeniowego ma zazwyczaj swoją własną architekturę, którą obsługują jego urządzenia – jest to na przykład oprogramowanie CUDA dla urządzeń firmy NVIDIA czy Stream od firmy ATI. W odróżnieniu od nich OpenCL jest otwartym standardem, którego implementacje dla konkretnych urządzeń należą wprawdzie do ich producentów, lecz jego zastosowanie jest możliwe na urządzeniach NVIDIA, ATI, Intel, a nawet na niektórych procesorach w architekturze ARM pod kontrolą systemu Android [12]. Standard OpenCL dostarcza warstwę abstrakcji, dzięki której urządzenie, na którym wykonywany jest kod nie ma znaczenia dla twórcy oprogramowania tak długo, jak implementacja OpenCL dla tej platformy jest z nim zgodna.

Stworzony kod został przetestowany pod kątem wydajności dla szeregu macierzy o różnych rozmiarach. Do testów została wykorzystana karta graficzno-obliczeniowa NVIDIA Tesla M2090 oraz procesor Intel Xeon X5650.

# 2 WPROWADZENIE TEORETYCZNE

## 2.1 METODY ROZWIĄZYWANIA UKŁADÓW RÓWNAŃ LINIOWYCH

Koniunkcję pewnej liczby równań liniowych, w skład których wchodzi ten sam zestaw zmiennych nazywa się układem równań liniowych. Układ taki można zapisać w postaci równania macierzowego (1).

$$Ax = b \tag{1}$$

Ogólną reprezentację macierzową układu przedstawia równanie (2).

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2}$$

Ze względu na wygodę, w przykładach umieszczanych w literaturze, często stosowana jest również reprezentacja w postaci macierzy rozszerzonej (3).

$$A_{(aug)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{bmatrix} \tag{3}$$

Ze względu na czytelność, powyższa forma będzie stosowana w pozostałej części tego dokumentu. W literaturze zaproponowanych zostało wiele metod rozwiązywania układów równań. Oprogramowanie komputerowe służące do realizacji tej funkcjonalności nazywane jest solwerem. Metody służące do rozwiązywania układów równań liniowych można podzielić na dwie główne kategorie [7]: metody iteracyjne oraz metody bezpośrednie. Do metod bezpośrednich zaliczana jest między innymi metoda eliminacji Gaussa oraz metoda faktoryzacji (dekompozycji) LU.

## 2.2 METODA ELIMINACJI GAUSSA

Metoda eliminacji Gaussa składa się z dwóch zasadniczych faz tj. fazy eliminacji w przód oraz fazy podstawiania wstecz. Uważna analiza obu tych faz pozwala wysnuć wnioski kluczowe w modyfikacji algorytmu dla oprogramowania równoległego.

### 2.2.1 ELIMINACJA W PRZÓD

Pierwsza z faz opiera się na liniowej operacji algebraicznej [9], czyli na wiedzy, iż każde równanie w układzie równań liniowych może zostać zastąpione równaniem powstałym z połączenia tego równania z dowolnym innym występującym w tym układzie. Przedstawiając układ równań w postaci macierzowej, daje to podstawy do stosowania jednej z macierzowych operacji elementarnych, czyli dodawania lub odejmowania od siebie wierszy bądź ich wielokrotności. Celem fazy eliminacji w przód jest doprowadzenie układu równań w postaci macierzowej do górnej macierzy trójkątnej (znanej również jako macierz schodkowa). Dokonuje się tego eliminując, przy pomocy operacji elementarnych, pewną liczbę niewiadomych z poszczególnych równań reprezentowanych przez wiersze w macierzy [1,9].

Na podstawie [1] przeanalizowany zostanie przykład fazy eliminacji w przód. Na rysunku 1 dany jest układ równań w postaci macierzy rozszerzonej:

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ -4 & 6 & -4 & 1 & 1 \\ 1 & -4 & 6 & -4 & 0 \\ 0 & 1 & -4 & 5 & 0 \end{bmatrix}$$

*Rysunek 1. Układ równań liniowych w postaci macierzowej*

Jest to układ równań wygenerowany przez prosty problem metody elementów skończonych. Na pierwszy rzut oka widać że jest symetryczna oraz pasmowa mimo, że pasmo to jest bardzo szerokie. Symetria oraz pasmowość to istotne cechy macierzy opisujących problemy metody elementów skończonych.

Pierwszym krokiem eliminacji w przód jest wyeliminowanie zmiennych $a_{21}$ oraz $a_{31}$ przy pomocy pierwszego wiersza macierzy pomnożonego przez odpowiednią liczbę. Po przeprowadzeniu tej operacji, macierz wygląda jak na rysunku 2.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & -\frac{16}{5} & \frac{29}{5} & -4 & 0 \\ 0 & 1 & -4 & 5 & 0 \end{bmatrix}$$

*Rysunek 2. Układ równań liniowych w postaci macierzowej po pierwszym kroku eliminacji w przód*

Następnym krokiem jest postępowanie analogicznie w stosunku do zmiennych $a_{32}$ oraz $a_{42}$, co zostało przedstawione na rysunku 3.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{8}{7} \\ 0 & 0 & -\frac{20}{7} & \frac{65}{14} & -\frac{5}{14} \end{bmatrix}$$

*Rysunek 3. Układ równań liniowych w postaci macierzowej po drugim kroku eliminacji w przód*

Ostatnim krokiem w przypadku tej przykładowej macierzy jest zredukowanie wyrazu $a_{43}$, jak zostało to uwidocznione na rysunku 4.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{8}{7} \\ 0 & 0 & 0 & \frac{5}{6} & \frac{7}{6} \end{bmatrix}$$

*Rysunek 4. Układ równań liniowych w postaci macierzowej po zakończeniu eliminacji w przód*

Analizując przebieg tego przykładu można wysnuć dwa pomocne wnioski: dla każdego równania, czyli wiersza macierzy, można wyznaczyć ilość operacji elementarnych, które są konieczne do doprowadzenia go do pożądanej postaci, przy założeniu wykonywania operacji w sposób sekwencyjny. W powyższym przykładzie dla wiersza 1 było to zero operacji elementarnych, wiersz numer 2 wymagał jednej operacji elementarnej, zaś wiersze 3 i 4 po dwie. Wychodząc z powyższego wniosku postawić można kolejny, iż dla danej

macierzy można szybko wyznaczyć maksymalną ilość operacji elementarnych konieczną do sprowadzenia jej do macierzy schodkowej przypadającej na wiersz. Jak łatwo zauważyć, ta ilość operacji $o$ jest związana bezpośrednio z szerokością pasma $d$ macierzy: $o = d - 1$.

Drugi wniosek wyprowadzić można z faktu, iż eliminacja w przód sprowadza macierz do postaci macierzy schodkowej. Cechą charakterystyczną tej macierzy jest to, iż pierwszy wyraz niezerowy w danym wierszu musi znajdować się na diagonali macierzy; bezpośrednio oznacza to, że żadne dwa wiersze nie mogą mieć identycznego pierwszego wyrazu niezerowego.

### 2.2.2 PODSTAWIANIE WSTECZ

Drugą z faz rozwiązania układu równań liniowych przy pomocy metody Gaussa jest faza podstawiania wstecz (ang. back substitution). Etap ten przeprowadzany jest na macierzy sprowadzonej do górnej macierzy trójkątnej, czyli macierzy w postaci schodkowej utworzonej w fazie eliminacji w przód. Rysunek 5 przedstawia macierz w formie schodkowej gotową do przeprowadzenia fazy podstawiania wstecz.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{8}{7} \\ 0 & 0 & 0 & \frac{5}{6} & \frac{7}{6} \end{bmatrix}$$

*Rysunek 5. Układ równań liniowych w postaci macierzy schodkowej*

Kontynuując analizę tego przykładu na podstawie [9], zauważyć można iż wartość wyrazu $x_4$ może zostać wyliczona bez przeprowadzania żadnych dodatkowych operacji, podczas gdy pozostałe wyrazy wymagają operacji odpowiednio więcej. Wyraz $x_4$ zostaje więc wyliczony poprzez obustronne dzielenie jak uwidoczniono na rysunku 6.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{8}{7} \\ 0 & 0 & 0 & 1 & \frac{7}{5} \end{bmatrix}$$

*Rysunek 6. Macierz schodkowa po przeprowadzeniu pierwszego kroku podstawienia wstecz*

Wiedząc iż $x_4 = \frac{7}{5}$, możliwe jest teraz wyliczenie $x_3$ za pomocą równania danego trzecim wierszem macierzy: $\frac{15}{7}x_3 = \frac{8}{7} + \frac{20}{7}x_4$. Po wykonaniu odpowiednich operacji, uzyskujemy wynikową macierz przedstawioną na rysunku 7.

$$A_{(aug)} = \begin{bmatrix} 5 & -4 & 1 & 0 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & 1 \\ 0 & 0 & 1 & 0 & \frac{12}{5} \\ 0 & 0 & 0 & 1 & \frac{7}{5} \end{bmatrix}$$

*Rysunek 7. Macierz schodkowa po przeprowadzeniu drugiego kroku podstawienia wstecz*

W drodze analogicznych działań dla dwóch pozostałych poszukiwanych wyrazów, $x_2$ oraz $x_1$ danych równaniami opartymi odpowiednio na drugim i pierwszym wierszu macierzy w formie zaprezentowanej na rysunku 7, uzyskiwana jest ostatecznie macierz przedstawiona na rysunku 8.

$$A_{(aug)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 10\frac{66}{175} \\ 0 & 1 & 0 & 0 & 13\frac{4}{7} \\ 0 & 0 & 1 & 0 & 2\frac{2}{5} \\ 0 & 0 & 0 & 1 & \frac{7}{5} \end{bmatrix}$$

*Rysunek 8. Finalna postać macierzy – rozwiązany układ równań liniowych*

Macierz w tej postaci zawiera wyłącznie rozwiązanie układu równań liniowych, czyli cel zastosowania metody eliminacji Gaussa.

Na podstawie powyższego przykładu można wysnuć kolejny istotny wniosek. W przypadku fazy podstawiania wstecz, każdy kolejny krok (wyliczenie kolejnego wyrazu) wymaga wyliczenia wszystkich wyrazów, od których jest zależny. Inaczej mówiąc, wyliczenie wyrazu $x_i$ wymaga, co można przyjąć jako ogólną regułę, wyliczenia wyrazów $x_{i+1}, \dots, x_n$.

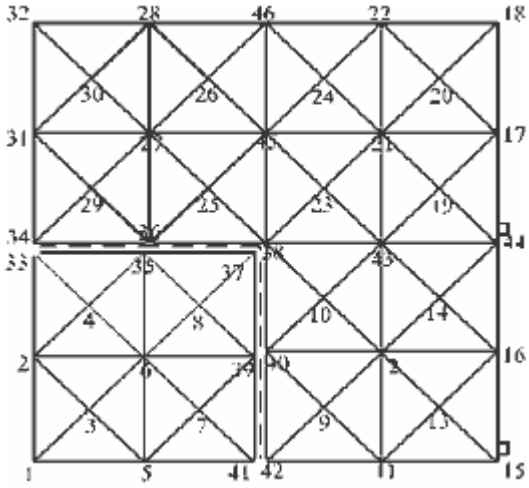
Drugim istotnym spostrzeżeniem jest fakt, iż kolejność rozwiązywania równań z dołu do góry, typowa dla górnej macierzy trójkątnej, jest pewną abstrakcją. W istocie nieistotne jest,

którym wierszem macierzy dany jest wyraz rozwiązywanego układu równań, tak długo jak istnieje metoda identyfikacji właściwego dla danego wyrazu wiersza.

## 2.3 MACIERZE

Proponowane rozwiązanie ma służyć przede wszystkim jako solwer dla układów równań liniowych przedstawionych w postaci macierzowej wygenerowanych przez oprogramowanie rozwiązujące problemy z zakresu metody elementów skończonych. Wychodząc z tego założenia można wyciągnąć pewne wnioski, co do charakteru rozpatrywanych macierzy.

### 2.3.1 CHARAKTERYSTYKA MACIERZY W METODZIE ELEMENTÓW SKOŃCZONYCH



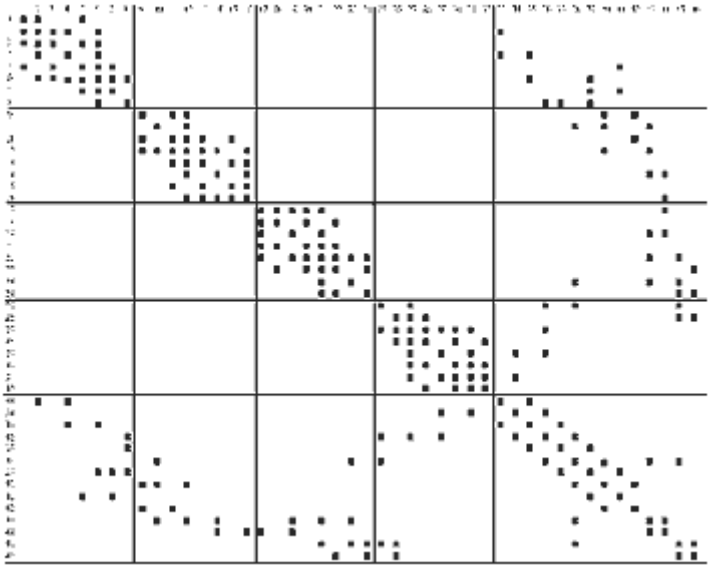
*Rysunek 8. Siatka elementów skończonych*[1]

Analizując siatkę elementów skończonych przedstawioną na rysunku 8, łatwo zauważyć, iż każdy element, a nawet cała siatka jako taka, może być postrzegana jako rodzaj grafu nieskierowanego. Dla każdego elementu wyliczana jest lokalna macierz sztywności. Następnie zawarte w niej wartości są umieszczane w globalnej macierzy sztywności. Mapowanie pozycji wyrazów w macierzy lokalnej do pozycji w macierzy globalnej określa numeracja węzłów oraz elementów [8].

Wychodząc z powyższych założeń można powiedzieć, że macierz sztywności tworzona w metodzie elementów skończonych jest w swojej strukturze podobna do macierzy sąsiedztwa dla grafu nieskierowanego, które są inherentnie symetryczne. Intuicyjnie „jeśli wierzchołek 1 jest sąsiadem wierzchołka 2 to wierzchołek 2 jest sąsiadem wierzchołka 1”.

[1] Rysunek został zaczerpnięty z http://icis.pcz.czest.pl/~roman/mat_dyd/prz_rown/mac_rzadkie/4_2.html [dostęp 23-12-2013].

Identyczną logikę można zastosować w przypadku siatki elementów skończonych. Dzięki temu pewnym jest, iż wygenerowana macierz będzie symetryczna.



*Rysunek 9. Macierz sztywności dla metody elementów skończonych*[2]

Na rysunku 9 została uwidoczniona macierz sztywności dla przedstawionej powyżej siatki elementów skończonych. Na tym przykładzie dobrze uwidoczniona jest druga z interesujących cech macierzy, którymi posługuje się metoda elementów skończonych: macierze te są rzadkie. W praktyce oznacza to, iż wiele z ich elementów to zera.

W pracy [10] Reginald Tewarson postulował, że macierz rzadka to taka, która przy wymiarach $n\,x\,n$ ma około $n$ niezerowych elementów, w praktyce – dwa do dziesięciu elementów niezerowych przypadających na każdy wiersz dla dużych $n$. Są to wprawdzie definicje skonstruowane w zupełnie innych czasach, praca Tewarsona została opublikowana w 1973 roku, i niezbyt precyzyjne (nie jest na przykład sprecyzowane, co autor miał na myśli przez duże $n$), lecz pozwalają na wytworzenie definicji intuicyjnej: macierz jest rzadka, jeśli więcej niż połowa jej elementów to zera. Praktycznie każda macierz wygenerowana przez metodę elementów skończonych spełnia to założenie.

Specyficzną formą macierzy rzadkich są macierze pasmowe, czyli takie macierze rzadkie w których elementy niezerowe są skupione w paśmie ułożonym na przekątnej, którego środek wyznacza główna przekątna macierzy i zero lub więcej przekątnych po obu jej stronach. Macierze pasmowe bardzo dobrze nadają się do zastosowania metody Gaussa i jej

[2] Rysunek został zaczerpnięty z http://icis.pcz.czest.pl/~roman/mat_dyd/prz_rown/mac_rzadkie/4_2.html [dostęp 23-12-2013].

pochodnych, gdyż z samej ich definicji wynika, iż niewiele operacji jest koniecznych, by sprowadzić je do postaci macierzy schodkowej – wymagana jest jedynie eliminacja elementów znajdujących się pod główną przekątną macierzy.

W metodzie elementu skończonego uzyskanie macierzy pasmowych wymaga odpowiedniego ponumerowania elementów i węzłów. Niektóre solwery [3] integrują się z rozwiązaniem samego problemu metody elementów skończonych do tego stopnia, że same zmieniają numerację węzłów i elementów, by uzyskać korzystniejszą dla ich metody działania strukturę macierzy. Ujmuje to jednak takim rozwiązaniom uniwersalności i utrudnia ich implementację w rozwiązaniach zewnętrznych.

### 2.3.2 METODY PRZECHOWYWANIA MACIERZY RZADKICH

Duże macierze rzadkie na ogół przechowuje się w pamięci komputera w postaci skompresowanej [10]. Dzięki temu możliwe jest przechowanie danej macierzy kosztem mniejszej ilości pamięci. Kompresja pozwala też przechować macierz która, składowana w klasycznej strukturze danych, byłaby większa niż może pomieścić pamięć urządzenia.

W kontekście solwera działającego przeważnie na kartach graficznych zużycie pamięci jest istotne, co najmniej z dwóch powodów. Po pierwsze, dzięki kompresji można umieścić w pamięci urządzenia obliczeniowego większą macierz na raz. Po drugie, szybkość całkowitego działania solwera będzie związana między innymi z czasem transferu danych na urządzenie za pośrednictwem szyny danych PCI-Express [4]. Z tych dwóch względów konieczne jest rozważenie najpopularniejszych schematów przechowywania macierzy rzadkich.

Jednym z najprostszych schematów jest metoda koordynatowa (Coordinate Format), w której elementy niezerowe macierzy rzadkiej przechowywane są w formie tripletów $(i, j, n)$, gdzie $i$ oraz $j$ stanowią odpowiednio wiersz i kolumnę w której znajduje się wartość $n$. Ta metoda jest najprostsza, lecz zużywa $3t$ miejsc w pamięci (dla uproszczenia kalkulacji pominięto kwestię stricte implementacyjna, tj. użytych typów danych), gdzie $t$ to ilość niezerowych elementów w macierzy.

Wariacją na temat metody koordynatowej i jednym z popularniejszych schematów przechowywania macierzy rzadkich, jest metoda Compressed Sparse Row Format[3] (CSR). W tej metodzie przechowywane w pamięci są trzy wektory. Pierwszy z nich, o długości $t$, zawiera niezerowe wartości z macierzy ułożone w kolejności od lewej do prawej oraz od góry

[3] Opis za https://www.icm.edu.pl/kdm/Numeryka:_Macierzy#Compressed_Sparse_Row_Format

do dołu macierzy. Drugi wektor zawiera indeksy kolumn, w których konkretne wartości w pierwszym wektorze znajdowały się w macierzy (we właściwych sobie wierszach). Trzeci wektor zaś zawiera informację o tym, od których indeksów zaczynają się poszczególne wiersze w dwóch poprzednich wektorach.

Przykład metody CSR przechowywania macierzy został przedstawiony na rysunku 10.

```
AA: 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
JA:   1   4   1   2   4   1   3   4   5    3    4    5
IA:   1   3   6  10  12  13
```

*Rysunek 10. Przykład macierzy zapisanej w formacie CSR*

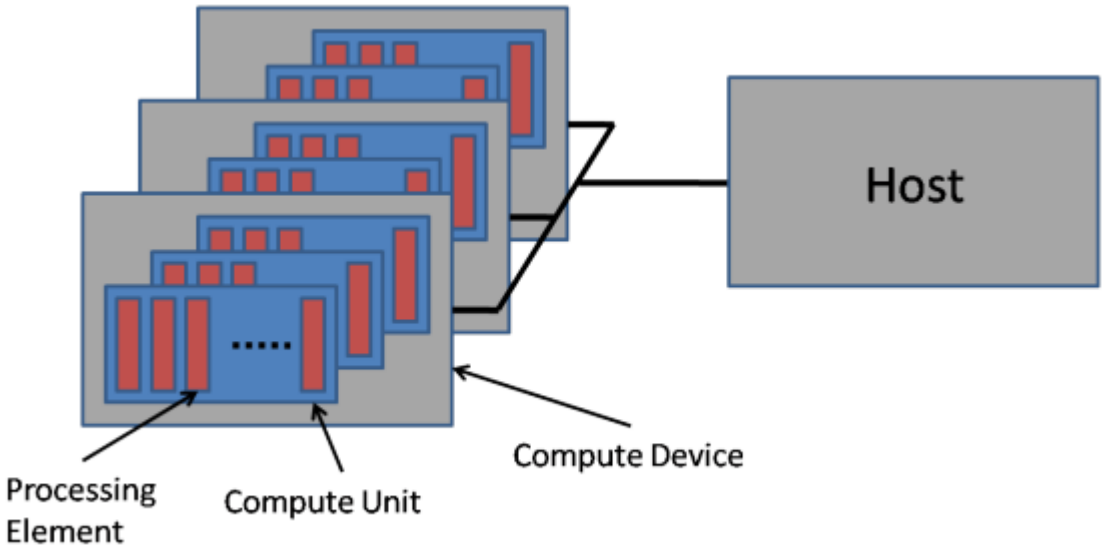
W tym przypadku ilość zużytej pamięci wynosi $t + t + n$. Dwa pierwsze wektory muszą mieć długość równą ilości niezerowych wartości w macierzy, trzeci zaś w typowym dla metody elementów skończonych przypadku ma długość równą wymiarowi macierzy. Jest tak oczywiście przy założeniu, że w macierzy nie ma wierszy, w których nie ma żadnych danych, które dla metody elementów skończonych jest zasadne. Ponieważ z reguły niezerowych elementów w macierzy wygenerowanej w metodzie elementów skończonych będzie więcej niż wynosi wymiar macierzy $n$, metoda ta wygrywa kompresją z metodą koordynatową ($2t + n$ jest mniejsze niż $3n$). Istnieją również wariacje na temat tej metody. Najoczywistszą z nich jest format Compressed Sparse Column (CSC), różniący się tym, iż drugi wektor przechowuje indeksy wierszy, a trzeci początki kolumn w pozostałych wektorach.

## 2.4 ARCHITEKTURA OPENCL

Jak wspomniano we wstępie do niniejszej pracy, OpenCL (lub Open Computing Language) jest stworzonym przez Khronos Group otwartym standardem tworzenia oprogramowania dla nowoczesnych urządzeń obliczeniowych. Architektura ta posiada kilka specyficznych cech, których zrozumienie jest konieczne dla jej skutecznego wykorzystania.

### 2.4.1 SKŁADNIKI ŚRODOWISKA OPENCL

Idea funkcjonowania platformy OpenCL została przedstawiona na rysunku 11. Środowisko składa się z hosta – „gospodarza”, który zleca wykonanie konkretnych obliczeń urządzeniom obliczeniowym. Każde z nich posiada wiele jednostek obliczeniowych, z których każda posiada więcej niż jeden element przetwarzający.




*Rysunek 11. Składniki środowiska dla architektury OpenCL[6]*

### 2.4.2 SKŁADNIKI ARCHITEKTURY OPENCL

Sama architektura składa się z trzech zasadniczych części [6]: specyfikacji języka, API platformy i API czasu wykonania. Specyfikacja języka OpenCL definiuje składnię programów i kerneli, które zostaną uruchomione z użyciem pozostałych dwóch części architektury. Jest oparty na standardzie ISO C99, lecz ze zmodyfikowanymi, dodanymi lub usuniętymi słowami kluczowymi, a także bez niektórych funkcjonalności.

API platformy zapewnia programistom dostęp do funkcji przy pomocy których mogą sprawdzić dostępność urządzeń wspierających OpenCL. Realizuje ono również koncepcję kontekstu. Kontekst jest kontenerem grupującym urządzenie z przeznaczoną dla niego zawartością pamięci oraz kolejkami zadań. Przy pomocy API platformy realizowane są transfery danych między gospodarzem a urządzeniem obliczeniowym.

API czasu wykonania wykorzystuje dostarczone przez platformę konteksty do kontrolowania kompatybilnych urządzeń. Przy jego pomocy odbywa się zarządzanie kolejkami zadań, obiektami pamięci i kernelami. Za pośrednictwem API czasu wykonania odbywa się również kolejkowanie kerneli na konkretnych urządzeniach.

### 2.4.3 PARALELIZM DANYCH

OpenCL, podobnie jak karty graficzne, które są główną grupą urządzeń wykorzystujących tą technologię, działa na zasadzie paralelizmu danych [15]. W praktyce oznacza to, iż w przeciwieństwie do paralelizmu zadań, który zakłada wykonanie różnych

zadań w tym samym czasie, OpenCL zakłada wielokrotne wykonanie identycznego (lub podobnego) zadania na elementach pewnego zbioru danych.

W tabeli 1 zostało przedstawione porównanie standardowego kodu skalarnego w języku C oraz kodu paralelnego względem danych w języku OpenCL.

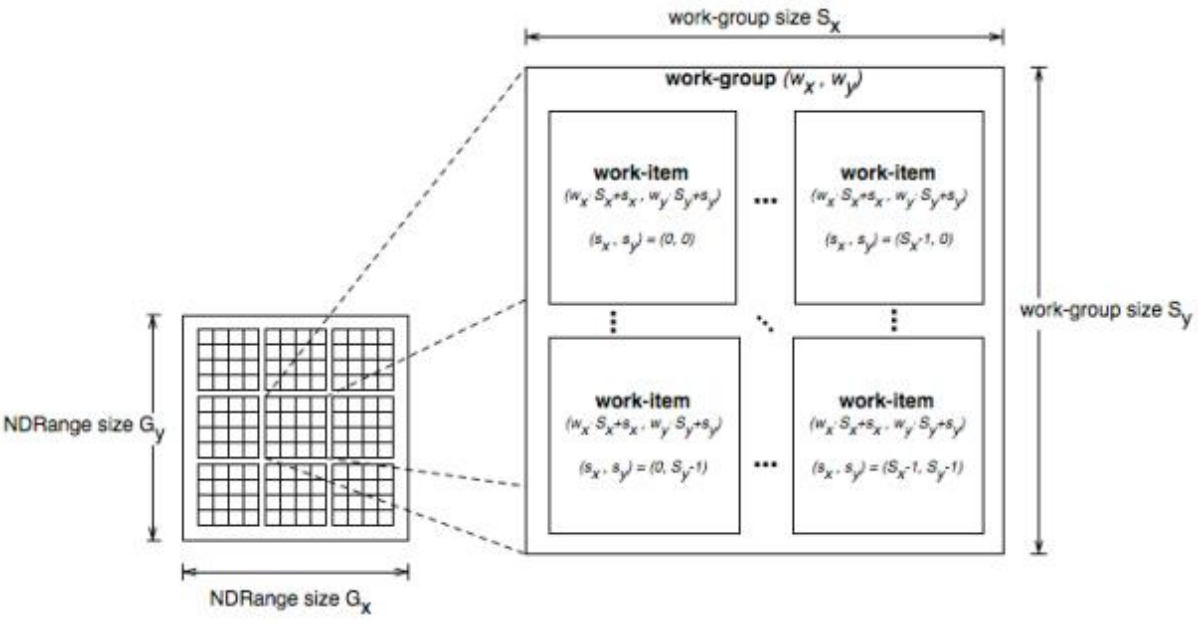
*Tabela 1. Porównanie kodu paralelnego względem danych z funkcją skalarną (za [6]).*

| Skalarna funkcja w języku C | Funkcja paralelna względem danych |
|---|---|
| `void sqrt(int n, const float *a, float *res)`<br><br>`{`<br>`  int i;`<br>`  for(i=0; i<n; i++)`<br>`  res[i] = a[i]*a[i];`<br>`}` | `kernel void dp_square`<br><br>`  (global const float *a, global float *res)`<br><br>`{`<br><br>`  int id = get_global_id(0);`<br><br>`  res[id] = a[id] * a[id];`<br><br>`}` |

Paralelizm danych jest w OpenCL realizowany za pośrednictwem programów, składających się z jednego lub więcej kerneli. Program może też zawierać dodatkowe funkcje wykorzystywane przez kernele oraz stałe dane. Podczas wykonania danego kernela wiele jego kopii jest równolegle wykonywanych na elementach przetwarzających urządzenia obliczeniowego. Wykonania te zwane są work-itemami[15]. W dalszej części niniejszej pracy będą wykorzystywane terminy „zadanie”, by określić work-item, i „grupa robocza”, by określić work-group.

### 2.4.4 ZADANIA I GRUPY ROBOCZE

Zadania są powiązane ze sobą w grupy robocze, jak uwidoczniono na rysunku 12.




*Rysunek 12. Zadania i grupy robocze w OpenCL [15].*

Przedstawiona po lewej stronie powierzchnia *NDRange* to metoda organizacji pamięci i rozplanowania zadań w architekturze OpenCL. *NDRange* to *N*-wymiarowa przestrzeń (OpenCL pozwala na wykorzystanie jedno-, dwu- lub trójwymiarowej przestrzeni). Przestrzeń ta jest dzielona na grupy robocze. Każda z grup roboczych dysponuje indeksami lokującymi ją w konkretnym punkcie *NDRange*. Każde zadanie, stanowiące odrębną instancję kernela, dysponuje unikalnym globalnym numerem identyfikacyjnym oraz unikalnym wewnątrz grupy roboczej identyfikatorem lokalnym.

Z punktu widzenia paralelizmu danych, *NDRange* jest swoistym mapowaniem konkretnych instancji kernela na konkretne elementy obszaru pamięci, który ma zostać przetworzony. Do dodania do siebie dwóch macierzy *N* x *N* można wykorzystać kernele uruchomione przy dwuwymiarowym *NDRange* z *Gx* = *Gy* = *N*. Do wyliczenia iloczynu skalarnego z kolei należałoby wykorzystać kernele uruchomione na jednowymiarowym *NDRange*.

Podział zadań na grupy robocze służy też do podziału barier na dwa rodzaje. W kernelach OpenCL można wykorzystywać bariery lokalne, które synchronizują tylko wątki wewnątrz grupy roboczej, bądź bariery globalne, które zsynchronizują wszystkie wątki działające na *NDRange*.

### 2.4.5 ZARZĄDZANIE PAMIĘCIĄ

Jednym z istotnych założeń architektury OpenCL, o którym należy pamiętać projektując dla niej oprogramowanie jest fakt, iż pojedyncza grupa robocza będzie wykonywana symultanicznie na jednej jednostce obliczeniowej (a wielu elementach przetwarzających). Ta wiedza przydatna jest podczas analizy modelu pamięci OpenCL.

W OpenCL wyróżniamy cztery osobne przestrzenie w pamięci [15]. Są to:

a) **Pamięć stała.** W tej kategorii pamięci przechowywane są wartości niezmienne podczas całego wykonania kernela. Może ona zostać zainicjalizowana przez „gospodarza".
b) **Pamięć globalna.** Do tej pamięci mają dostęp (zapis i odczyt) wszystkie wątki we wszystkich grupach roboczych. Jeśli urządzenie na to zezwala, operacje na tej pamięci mogą być przechowywane w pamięci podręcznej. Jest to zazwyczaj największy dostępny obszar pamięci na urządzeniu obliczeniowym, ale najczęściej oferuje również najwyższy czas dostępu.
c) **Pamięć prywatna.** Do tego rodzaju pamięci ma dostęp tylko pojedyncze zadanie, które ją zaalokowało.
d) **Pamięć lokalna.** Pamięć współdzielona przez wszystkie wątki w danej grupie roboczej, niedostępna poza grupą roboczą. Jeśli jest to wspierane przez konkretne urządzenie, może być mapowana na obszar pamięci podręcznej przy konkretnej jednostce obliczeniowej, dzięki czemu dostęp do takiej pamięci jest znacznie szybszy dla danej jednostki i uruchomionych na niej zadań. Jeśli nie ma takiej możliwości, jako pamięć lokalna dla danej grupy roboczej zostanie zmapowany pewien obszar pamięci globalnej, co oczywiście nie pozwoli na uzyskanie takiego przyspieszenia jak prawdziwa pamięć lokalna.

Tabela 2 pokazuje jakie urządzenia i sposób, w jaki mogą alokować poszczególne rodzaje pamięci.

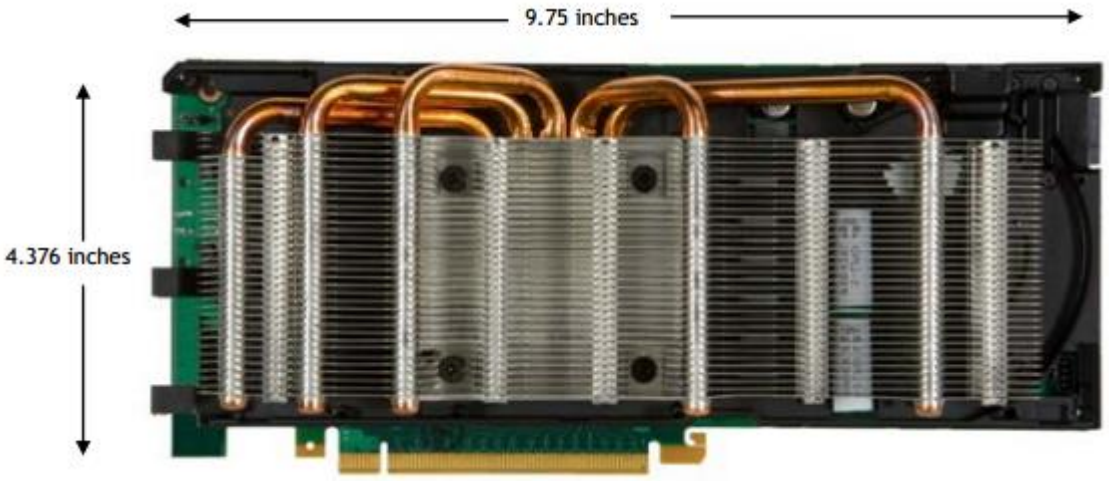
*Tabela 2. Dostęp do pamięci oraz możliwości jej alokacji w OpenCL (za [15])*

| | Globalna | Stała | Lokalna | Prywatna |
|---|---|---|---|---|
| **Gospodarz** | Dynamiczna alokacja<br>Odczyt i zapis | Dynamiczna alokacja<br>Odczyt i zapis | Dynamiczna alokacja<br>Brak dostępu | Brak alokacji<br>Brak dostępu |
| **Kernel** | Brak alokacji<br>Odczyt i zapis | Statyczna alokacja<br>Odczyt | Statyczna alokacja<br>Odczyt i zapis | Statyczna alokacja<br>Odczyt i zapis |

## 2.5 WYKORZYSTANE URZĄDZENIA

### 2.5.1 NVIDIA® TESLA™ M2090

Jako jedno z urządzeń do testów została wykorzystana karta obliczeniowa NVIDIA Tesla M2090, przedstawiona na rysunku 13.




*Rysunek 13. NVIDIA Tesla M2090*

Urządzenia obliczeniowe Tesla M-class są jednymi z najszybszych istniejących kart obliczeniowych do obliczeń wysokiej wydajności [14], zaś Tesla M2090 to najszybsza z kart w serii 20. Karty te oparto na architekturze CUDA noszącej nazwę kodową „Fermi".

Wewnątrz znajduje się aż 512 rdzeni CUDA (z punktu widzenia OpenCL są to elementy przetwarzające) zebranych w 16 jednostek obliczeniowych. Każdy z rdzeni jest taktowany zegarem o częstotliwości 1300 MHz. Maksymalna wydajność karty podczas obliczeń

podwójnej precyzji wynosi 665 GFLOPsów (tysięcy operacji zmiennoprzecinkowych na sekundę), zaś podczas obliczeń w pojedynczej precyzji 1331 GFLOPsów.

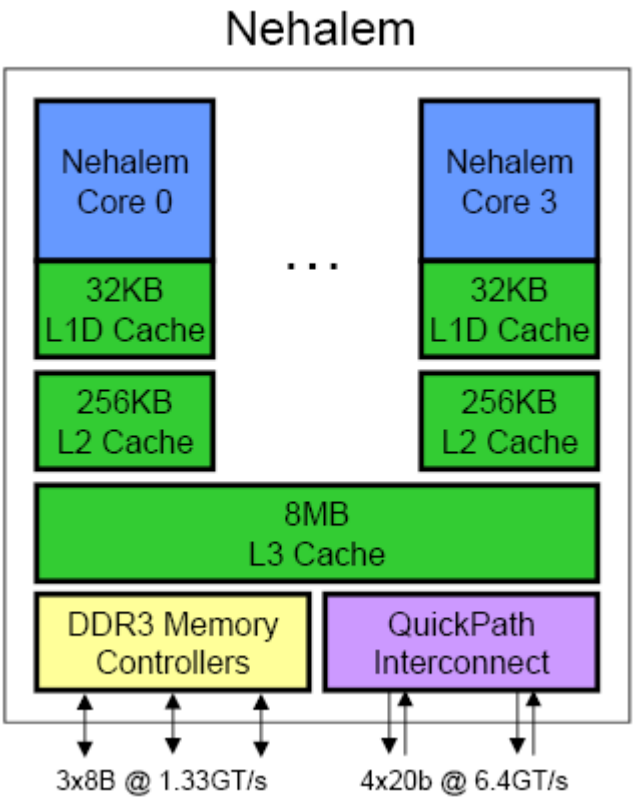
Akceleratorowi Tesla M2090 nie brakuje również pamięci. Został wyposażony w 6GB pamięci RAM typu GDDR5, taktowanej z efektywną prędkością 3700 MHz. Co więcej jest to jedyny akcelerator NVIDIA który zawiera dwa silniki DMA, pozwalając na równoczesną komunikację przez złącze PCI-Express w obu kierunkach [13]. Efektywnie pozwala to na równoczesne pobieranie i wysyłanie danych przez gospodarza.

Ze względu na zastosowanie w karcie Tesla M2090 512 rdzeni obliczeniowych, przewidywane jest uzyskanie najlepszych wyników przy wybraniu globalnej ilości wątków równej wielokrotności tej liczby. Należy tu jednak zauważyć iż urządzenie powinno pozwalać na zadania o rozmiarze globalnym wynoszącym najwyżej 8192. Zezwala na maksymalny rozmiar grupy roboczej równy 1024 wątkom. Wspiera ona standard OpenCL 1.1.

### 2.5.2 INTEL® XEON® X5650

Do testów został wykorzystany również procesor CPU Intel® Xeon® model X5650. Posiada sześć rdzeni, które mogą na raz przetwarzać dwanaście wątków. Pracuje w zakresie prędkości 2,67 GHz – 3,06 GHz.

Jest to urządzenie zrealizowane w architekturze Nehalem. Jak zostało uwidocznione na rysunku 14 zawiera niedużo pamięci poziomu L1 i L2, czyli tych o najszybszym dostępie.




*Rysunek 14. Architektura Nehalem*[4]

Jest to pewien mankament w porównaniu do na przykład procesorów z serii Core 2, które zawierają 6MB pamięci poziomu L2 na wszystkie rdzenie, zaś Nehalem zawiera 256KB na rdzeń czyli 1,5MB na wszystkie rdzenie. W architekturze Nehalem rdzenie nie muszą się dzielić ze sobą pamięcią, czyli odpada problem wzajemnego blokowania sobie dostępów do niej. Dostępy do pamięci L2 powinny być dość szybkie, co jest istotne z punktu widzenia proponowanego rozwiązania które silnie wykorzystuje pamięć lokalną.

Urządzenie to dostarcza 24 jednostki przetwarzania równoległego. Dzięki temu możliwe jest uruchamianie na nim zadań przy maksymalnym rozmiarze grupy roboczej równym 8192 wątki. Procesor wraz z odpowiednim SDK od Intela wspiera standard OpenCL w najnowszej wersji 1.2.

[4]Rysunek został zaczerpnięty z http://www.realworldtech.com/nehalem/2/ [dostęp: 2013-01-04]

# 3 IMPLEMENTACJA SOLWERA

## 3.1 ZAŁOŻENIA OGÓLNE

### 3.1.1 PARADYGMAT „CZARNEJ SKRZYNKI”

Jednym z głównych założeń przyświecających projektowaniu proponowanego rozwiązania jest paradygmat solwera jako „czarnej skrzynki”. Jak wspomniano w rozdziale 2.3.1, niektóre solwery są bardzo mocno zintegrowane z oprogramowaniem rozwiązującym konkretny problem. Dzięki temu można spowodować, iż oprogramowanie, na przykład realizujące metodę elementu skończonego, wygeneruje macierz o lepszych parametrach z punktu widzenia przyjętej metody rozwiązania niż gdyby pominąć tą integrację. Dzieje się tak jednak kosztem ogólności rozwiązania i łatwości jego implementacji.

Proponowany solwer nie wymaga integracji z oprogramowaniem obliczeniowym na tym poziomie. Dzięki temu jego włączenie w istniejące rozwiązanie wymaga minimalnego wysiłku ze strony użytkownika. Co więcej, solwer nie jest właściwie uzależniony od rodzaju problemu przedstawionego w formie układu równań liniowych w postaci macierzowej. Przyjęto założenie, że rozwiązywane przy jego pomocy będą głównie układy równań wygenerowane przez metodę elementów skończonych. Dzięki zastosowaniu paradygmatu czarnej skrzynki nie wykorzystuje on jednak żadnej wiedzy o naturze problemu. Jedynym wymaganiem jest podanie na wejściu oznaczony układ równań liniowych

### 3.1.2 BIBLIOTEKA NAGŁÓWKOWA

Drugim istotnym założeniem było stworzenie solwera w postaci biblioteki nagłówkowej. Dzięki temu jego dołączenie do istniejącego rozwiązania staje się jeszcze prostsze dla użytkownika końcowego, gdyż nie musi on najpierw kompilować kodu źródłowego solwera do postaci biblioteki, a następnie dołączać nagłówków do swojego oprogramowania i linkować go ze skompilowaną biblioteką. Wystarczającym jest dołączenie nagłówka i linkowanie programu z odpowiednią dla wykorzystywanego sprzętu implementacją biblioteki OpenCL.

### 3.1.3 POWIĄZANIE Z KARTAMI GRAFICZNYMI

Jako że OpenCL pozwala na uruchamianie kodu masowo równoległego na całej gamie platform, podczas tworzenia oprogramowania konieczne było skupienie się przynajmniej na jednej kategorii urządzeń obliczeniowych. Jako docelowa platforma dla optymalizacji rozwiązania zostały wybrane karty graficzne, przy pomocy których można realizować

akcelerację obliczeń. Podczas tworzenia opisanego w niniejszej pracy rozwiązania najłatwiej dostępnym urządzeniem do testów było kilka różnych kart graficznych firmy NVIDIA, które szerzej zostały opisane w rozdziale 2.5.

## 3.2 RÓWNOLEGŁA METODA GAUSSA

W obliczu wysnutych w rozdziale 2.2 wniosków można pokusić się o analizę potencjalnych możliwości zrównoleglenia metody eliminacji Gaussa.

### 3.2.1 ELIMINACJA W PRZÓD

Kluczowe wnioski jeśli chodzi o zrównoleglenie fazy eliminacji w przód zostały wyciągnięte w rozdziale 2.2.1. Cała macierz musi zostać sprowadzona do postaci macierzy schodkowej, co oznacza iż dla każdego wiersza konieczne jest wykonanie pewnej ilości eliminacji wyrazów. Aby eliminacja pierwszego niezerowego wyrazu w danym wierszu była efektywna i nie spowodowała niepożądanych efektów, eliminację należy przeprowadzić przy pomocy wiersza o identycznej charakterystyce, tj. tym samym pierwszym niezerowym wyrazie. Co więcej, eliminacje takie muszą trwać, dopóki dla żadnego wiersza w macierzy nie będzie takiego innego wiersza, który miałby ten sam pierwszy niezerowy wyraz (zgodnie z faktem iż w macierzy schodkowej pierwszy niezerowy wyraz w wierszu jest unikalną charakterystyką każdego wiersza).

Można zatem wyznaczyć pewną operację, która musi zostać wykonana na wszystkich elementach pewnego zbioru danych, czyli operacja eliminacji w przód może zostać zrównoleglona w zgodzie z paradygmatem paralelizmu względem danych.

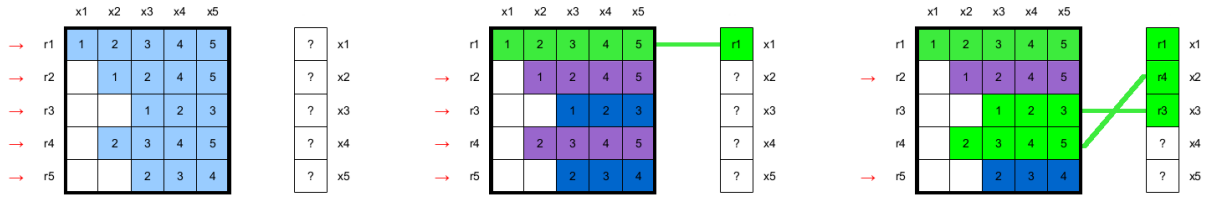
### 3.2.2 ZAPEWNIENIE FORMY SCHODKOWEJ MACIERZY PO WYKONANIU ELIMINACJI

Powstaje wszakże inny problem – ponieważ OpenCL nie gwarantuje ani kolejności wykonania poszczególnych grup roboczych, ani zadań w ich obrębie, może się zdarzyć iż eliminacje prowadzące macierz do formy, którą można sprowadzić do macierzy schodkowej przy pomocy operacji zamiany wierszy, nie dadzą macierzy schodkowej.

Optymalne dla rozwiązania było również uniknięcie całkowicie operacji zamiany wierszy, gdyż ma znacznie większy narzut pamięciowy i obliczeniowy niż operacje dodawania i mnożenia wiersza przez wyraz (jedyne dwie operacje konieczne do przeprowadzenia eliminacji w przód). Stąd do rozwiązania został wprowadzono dodatkowy wektor, obok pamięci przechowującej macierz oraz wektor prawej strony. Wektor ten stanowi swoiste mapowanie rzeczywistej pozycji wiersza w macierzy do pozycji, którą zajmowałby w

macierzy schodkowej (zgodnie z założeniem że wiersz o pierwszym niezerowym wyrazie $x_i$ będzie zajmował $i$-tą pozycję w macierzy). Mapowanie takie rozwiązuje jeszcze problem szybkiego zweryfikowania czy dany wiersz posiada unikalny pierwszy niezerowy wyraz, a jeżeli nie, to względem jakiego wiersza należy go wyeliminować. Informacja ta jest zapisana w dodatkowym wektorze i oszczędza kosztownych przeszukań dużych obszarów pamięci. Jedynym kosztem jest tutaj konieczność blokowania dostępu do wektora mapy na czas odczytu i zapisu informacji przez zadania.

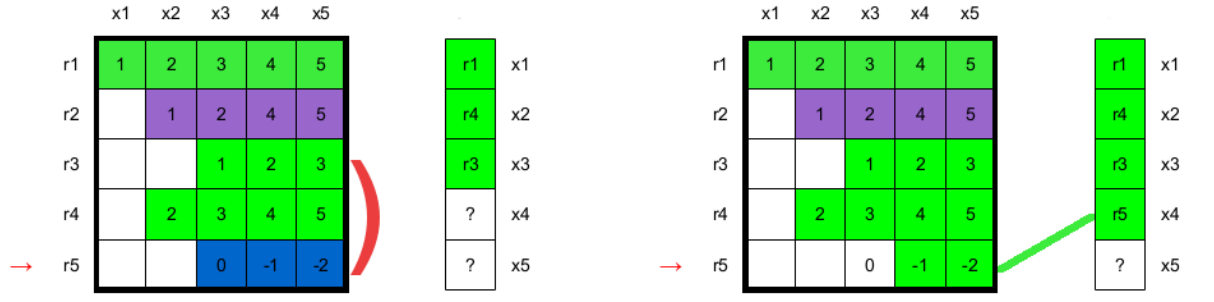
Na rysunku 15 można prześledzić pierwszy etap takiej operacji; przykład ten pokazuje działanie fazy eliminacji w przód proponowanego równoległego wariantu metody Gaussa w obrębie jednej grupy roboczej. Na rysunku przedstawiona została macierz oraz mapa, natomiast dla czytelności pominięty został wektor prawej strony.



*Rysunek 15. Pierwszy krok równoległego wariantu metody Gaussa*

Zadanie operujące na wierszu r1 znajduje pierwszy niezerowy wyraz w kolumnie numer 1. Ponieważ przed nim dostępu do mapy nie uzyskał żaden inny wątek, wpisuje on swoje ID do mapy na odpowiedniej pozycji i zakańcza funkcjonowanie.

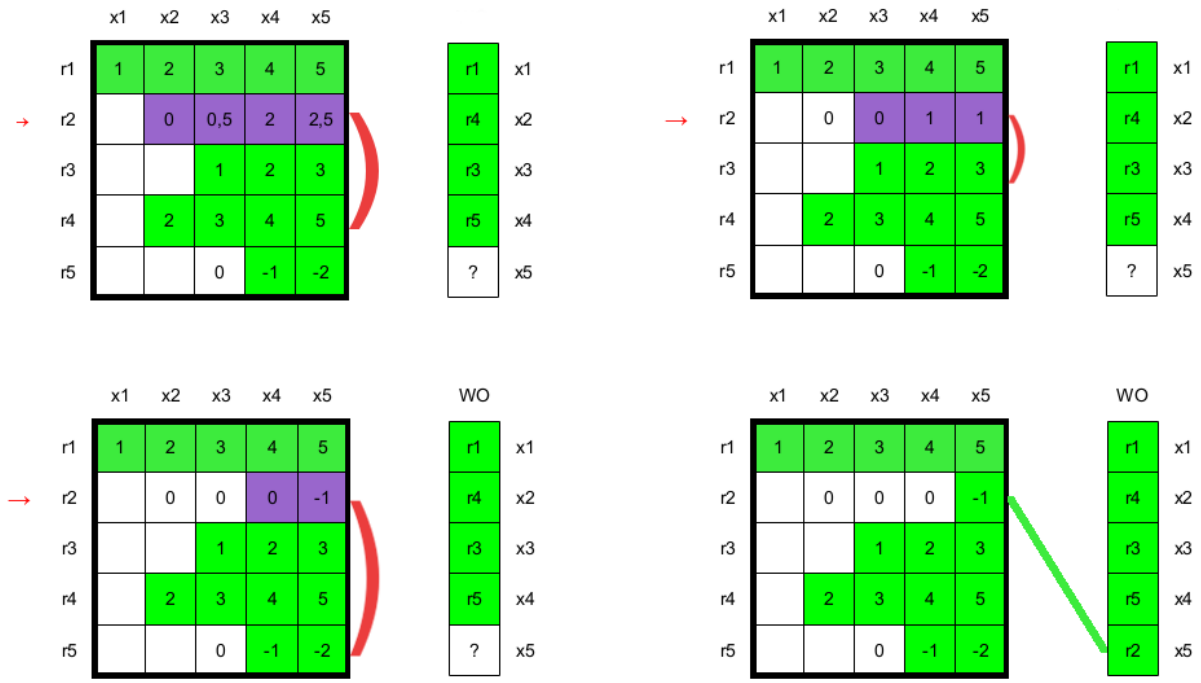
Podobnie dzieje się w przypadku wierszy r3 oraz r4; warto nadmienić tutaj, iż wiersze te zostały wybrane dla tego przykładu jako uzyskujące dostęp do mapy przed r2 oraz r5, by odwzorować typową dla OpenCL sytuację niesekwencyjnego przetwarzania zadań.



*Rysunek 16. Drugi element przykładu równoległego wariantu metody Gaussa*

Przykład kontynuowany jest na rysunku 16. Przyjęto tym razem, iż w następnej kolejności dostęp do mapy uzyskał wiersz r5. Zadanie przetwarzające wiersz r5 znajduje pierwszy niezerowy wyraz w kolumnie 3, lecz to miejsce w mapie zajmuje już inny wiersz.

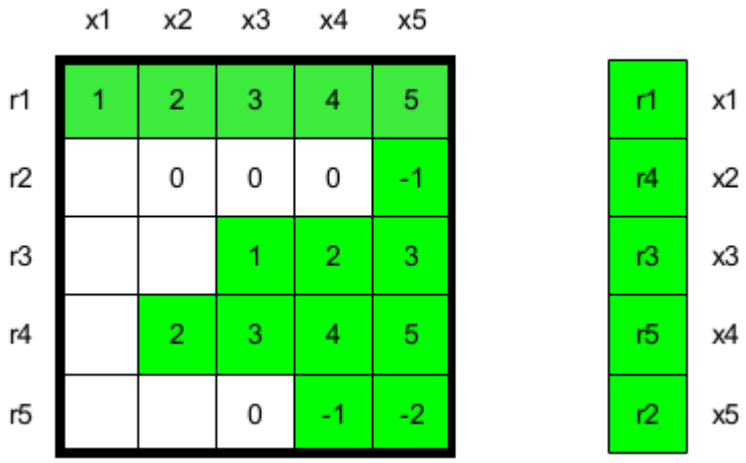
Na wierszu r5 jest zatem przeprowadzana eliminacja przy użyciu znalezionego w mapie wiersza r3. Po zakończeniu eliminacji wiersz r5 ma teraz pierwszy niezerowy wyraz w kolumnie 4. Zadanie znajduje niezajęte miejsce w wektorze mapy, wpisuje weń swoje ID i zakańcza wykonanie.




*Rysunek 17. Trzeci element przykładu równoległego wariantu metody Gaussa*

Rysunek 17 pokazuje ostatni element przykładu, w którym ostatnie zadanie, obsługujące wiersz r2, uzyskuje dostęp do wektora zawierającego mapę. Tym razem potrzebne są aż trzy eliminacje, by wiersz spełnił warunek unikalnego pierwszego wyrazu niezerowego. Po wykonaniu eliminacji i odnalezieniu wolnego miejsca w wektorze mapy zadanie zakańcza wykonanie. Ze względu na to że wszystkie wątki skończyły pracę, wychodzi również cała grupa robocza.

W pamięci pozostaje macierz przygotowana do fazy podstawiania wstecz, uwidoczniona na rysunku 18.




*Rysunek 18. Macierz przygotowana do fazy podstawiania wstecz przez równoległy wariant metody Gaussa.*

### 3.2.3 PODSTAWIENIE WSTECZ

Sposób przeprowadzenia podstawienia wstecz na macierzy wynikowej przedstawionej powyżej również wykorzystuje wektor mapy. Jedyną różnicą między tym wariantem a klasyczną metodą Gaussa jest to, iż podczas podstawiania wstecz algorytm porusza się od dołu do góry po wektorze mapy, wykorzystując wiersze o indeksach, na które kolejno natrafia zamiast w sposób naiwny poruszać się w górę po macierzy. W powyższym przykładzie kolejno przetworzone zostałyby wiersze r2, r5, r3, r4, r1.

Jeżeli chodzi o zrównoleglenie fazy podstawiania wstecz, wiedza zebrana w 2.2.2 wydaje się intuicyjnie sugerować, iż nie będzie to wykonalne. W istocie, z punktu widzenia programowania równoległego, wykonanie podstawiania wstecz dla wiersza $x_i$ jest operacją zależną od wykonania podstawiania wstecz dla $x_{i+1}, \ldots, x_n$. Operacje uzależnione od siebie w ten sposób nie mogą być łatwo zrównoleglone [2], czyli bez przeformułowania problemu tak, by usunąć z niego zależność następnego kroku od poprzednich.

## 3.3 PROBLEMY RÓWNOLEGŁOŚCI MASOWEJ

Zaprezentowane zostało ogólne rozwiązanie problemu zrównoleglenia fazy eliminacji w przód w metodzie Gaussa. By dostosować to rozwiązanie do uruchomienia w architekturze OpenCL i uzyskać adekwatną wydajność, konieczne jest wykorzystanie dodatkowych wniosków wynikających z analizy funkcjonowania tej platformy przeprowadzonej w rozdziale 2.4.

### 3.3.1 PODZIAŁ PROBLEMU NA ZADANIA

Zgodnie z proponowanym powyżej algorytmem, do każdego wiersza macierzy zostało przypisane jedno zadanie. Formułując jednakże rozwiązanie w ten sposób, niemożliwe byłoby rozwiązywanie układów równań o więcej niż kilku tysiącach niewiadomych. Docelowym rozmiarem problemów dla solwera były zaś macierze o setkach tysięcy lub milionach niewiadomych.

W związku z tym zostało stworzone rozwiązanie w duchu zaproponowanego przez Bruce'a Ironsa w jego oryginalnej pracy które pozwala na niezależne od siebie obrabianie części macierzy, a następnie łączenie wyników. Każdą z części można traktować jako osobny front rozwiązania. Dzięki zastosowaniu tego podziału można rozwiązywać układy o wielokrotnie większej ilości niewiadomych niż dostępna liczba wątków. Istotne jest również wynikające z tego podziału mniejsze zużycie pamięci.

Jedną ze zmiennych wejściowych dla solwera jest globalna ilość zadań które mają zostać uruchomione. Proponowane rozwiązanie dzieli macierz na części o rozmiarach równych tej wartości. Przykładowo, jeżeli dla macierzy o 10 000 niewiadomych zostanie wybrana globalna ilość zadań 1024, zostanie stworzonych 10 części, każda z nich obejmująca zakres 1024 wierszy.

Naturalnie w tym przypadku ostatnia, dziesiąta część obejmuje wiersze od 9216 do 10240. W kodzie kernela OpenCL zostały wprowadzone rozwiązania zapobiegające przekroczeniu granic pamięci w przypadku wystąpienia N większego niż rozmiary macierzy.

Zadania w obrębie jednej części są podzielone na grupy robocze. Podział ten jest wykonywany automatycznie przez OpenCL na podstawie lokalnej ilości zadań, podanej jako kolejna zmienna wejściowa do solwera. Przykładowo, jeżeli dla globalnej ilości zadań 1024 zostanie wybrana lokalna ilość zadań 128, utworzonych zostanie 8 grup roboczych. OpenCL wymaga, aby lokalna ilość zadań dokładnie dzieliła globalną. Każda z grup roboczych wykonywana jest na osobnej jednostce obliczeniowej w zakresie tego samego urządzenia obliczeniowego.

### 3.3.2 CYKL ROZWIĄZANIA

Jeden kompletny cykl rozwiązania składa się z dwóch etapów: pętli pod-cykli oraz złożenia rozwiązania.

Dla każdej wyznaczonej części macierzy (frontu) wykonywana jest pętla pod-cykli rozwiązania. Pod-cykl ten składa się z przywrócenia warunku unikalności pierwszego wyrazu niezerowego wewnątrz grupy roboczej oraz wewnątrz całego frontu.

Pierwszy z kerneli OpenCL działa na zasadzie przedstawionej powyżej. Dla każdej grupy roboczej tworzona jest osobna mapa, zaś do każdego wiersza w części macierzy przypisany jest wątek. W praktyce mapa została zrealizowana jako wektor typu *integer*. Pozycje nieobsadzone, tj. takie, dla których jeszcze żaden z wierszy w macierzy nie umieścił informacji w mapie, ustawione są domyślnie na wartość *-1*.

Wątek obsługujący dany wiersz sprawdza, czy pod pozycją w mapie odpowiadającą jego pierwszemu wyrazowi niezerowemu jest wpisana wartość inna niż *-1*. Jeżeli tak, używając wielokrotności tego wiersza dokonuje na sobie eliminacji. Operacja jest powtarzana aż wątek znajdzie nieobsadzone pole.

Ze względu na to, iż każda mapa odpowiada jednej grupie roboczej, a w obrębie jednej części macierzy funkcjonuje więcej niż jedna grupa robocza, zakończenie pracy grupy zapewnia przywrócenie warunku unikalności tylko wewnątrz niej. Konieczne jest więc przywrócenie warunku unikalności wewnątrz całej części macierzy.

Do tego celu został stworzony drugi kernel. Jego działanie jest mniej skomplikowane niż kernela przywracającego warunek unikalności wewnątrz mapy lokalnej. Ten kernel również uruchamiany jest na jednowymiarowym *NDRange*. W najprostszym przypadku, gdy liczba wierszy w mapach jest równa lub mniejsza od wybranej globalnej ilości zadań, numer identyfikacyjny zadania ponownie jest traktowany jako numer wiersza. W przeciwnym przypadku każde zadanie przetworzy odpowiednio więcej wątków.

Mapy lokalne dla grup roboczych przechowywane są w pamięci lokalnej, co zapewnia szybki dostęp i niski koszt operacji atomowych (ich wykorzystanie opisano w rozdziale 3.4.2). Przed zakończeniem wykonania grupy roboczej zadanie o lokalnym identyfikatorze 0 kopiuje uzupełnioną lokalną mapę z pamięci lokalnej do globalnej. Do przechowywania map w pamięci globalnej wykorzystywana jest macierz o rozmiarze *N x M*. *N* jest równe ilości grup roboczych przypadających na część macierzy, *M* zaś długości map lokalnych dla tej części.

Mapy lokalne wpisane są do mapy globalnej jako kolumny wspomnianej wyżej macierzy. Każde zadanie drugiego kernela przegląda przypisany mu wiersz w mapie globalnej. Pierwszy znaleziony wpis różny od *-1*, oznaczający, iż istnieje w macierzy wiersz o takim

pierwszym wyrazie niezerowym, traktowane jest jako kanoniczne, tj. ten wiersz jest wykorzystywany do redukcji ewentualnych innych wierszy które posiadają ten sam pierwszy wyraz niezerowy. Jeżeli podczas dalszego przeglądania wiersza w mapie zostaną znalezione numery innych wierszy które mają ten sam pierwszy wyraz niezerowy, na tych dalszych wierszach wykonywana jest eliminacja względem pierwszego znalezionego wiersza.

Ten kernel przywraca jedynie warunek unikalności globalnej nie zwracając z kolei uwagi na unikalność lokalną (wewnątrz bloków, tj. grup roboczych). Uruchomienie pierwszego, a potem drugiego kernela (w sposób blokujący, czyli niedopuszczający uruchomienia ich symultanicznie) jest traktowany jako jeden pod-cykl rozwiązania. Każda wykonana przez drugi kernel eliminacja jest zapisywana do odpowiedniej zmiennej jako „operacja"; tylko ta dana jest pobierana do hosta po każdym pod-cyklu rozwiązania, by zaoszczędzić na ilości danych koniecznych do przesłania przez złącze PCI-Express. Jeżeli ilość operacji globalnego przywracania jest niezerowa, jest to informacja, iż konieczne jest przeprowadzenie kolejnego pod-cyklu rozwiązania dla tego samego frontu, czyli ponowne uruchomienie obu kerneli dla tych samych danych.

Warunkiem zakończenia pętli pod-cykli dla frontu jest zakończenie wykonanie obu kerneli i pobrana ilość operacji jest *0*. Wówczas przetworzona część macierzy oraz wektora prawej strony jest pobierana z urządzenia. Następnie pobrane części są wpisywane z powrotem we właściwe miejsca w macierzy oraz wektorze po stronie gospodarza.

Pobierana jest również finalna postać mapy globalnej w formie *N x M*. Następnie, ponieważ po wykonaniu drugiego kernela z wynikiem zera operacji w każdym wierszu jest najwyżej jedno pole o wartości nie wynoszącej *-1,* mapa jest sprowadzana do pojedynczego wektora. Dla wierszy które zawierały wyraz nie wynoszący *-1* w wektorze zapisywany jest wyraz, dla pozostałych *-1*.

Powstały wektor można rozpatrywać jako globalną mapę dla danej części. Ponieważ części macierzy obrabiane są osobno, konieczne jest rozwiązanie analogicznego problemu jak w przypadku przywracania unikalności w obrębie pojedynczej części.

Po zakończeniu wykonania pod-cykli rozwiązania dla wszystkich części macierzy wygenerowane mapy globalne dla części są zbierane w strukturę podobną do mapy globalnej podczas przetwarzania części, również o wymiarach *N x M*. W tym przypadku *N* jest równe ilości części, *M* zaś ilości niewiadomych w macierzy.

Dla tej struktury wykonywana jest analogiczna do działania drugiego kernela w pod-cyklu rozwiązania dla części: przywracany jest globalny warunek unikalności pierwszego wyrazu niezerowego. Ta część rozwiązania jest wykonywana na CPU po stronie gospodarza, by uniknąć transferów dużych ilości danych po złączu PCI-Express. By przyspieszyć jej wykonanie, pętla po wierszach globalnych map została zrównoleglona przy pomocy API OpenMP.

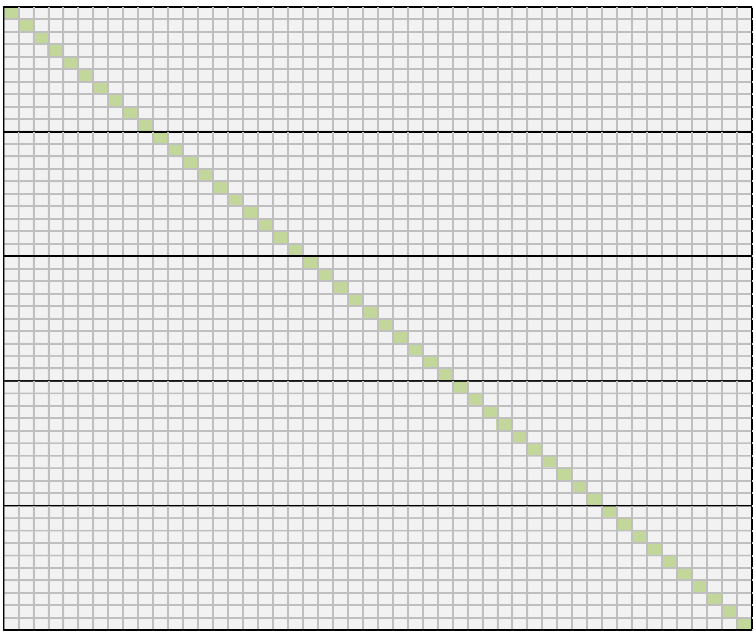
Analogicznie do pod-cyklu rozwiązania, zliczana jest ilość wykonanych operacji redukcji wierszy. Ilość zerowa powoduje zakończenie fazy eliminacji w przód, niezerowa zaś wykonanie kolejnego cyklu rozwiązania, wraz ze wszystkimi pod-cyklami dla poszczególnych części.

## 3.4 WYKORZYSTANIE PAMIĘCI

Jak zostało wcześniej nadmienione, jednym z wąskich gardeł w przypadku urządzeń obliczeniowych jest przesyłanie danych przez złącze PCI-Express. W trakcie wykonywania samych obliczeń należy również unikać blokujących dostępów do pamięci globalnej, takich jak na przykład operacje atomowe. Poniżej opisano metody ograniczenia ilości przesyłanych danych i wykorzystanie pamięci lokalnej.

### 3.4.1 WYBÓR WYCINKA MACIERZY DLA CZĘŚCI

Na rysunku 19 została przedstawiona przykładowa macierz podzielona na pięć części. Dla uproszczenia przykładu wybrano macierz pasmową zawierającą wyrazy niezerowe (oznaczone kolorem zielonym) wyłącznie na przekątnej. Pozostałe pola (szare) zawierają zera.

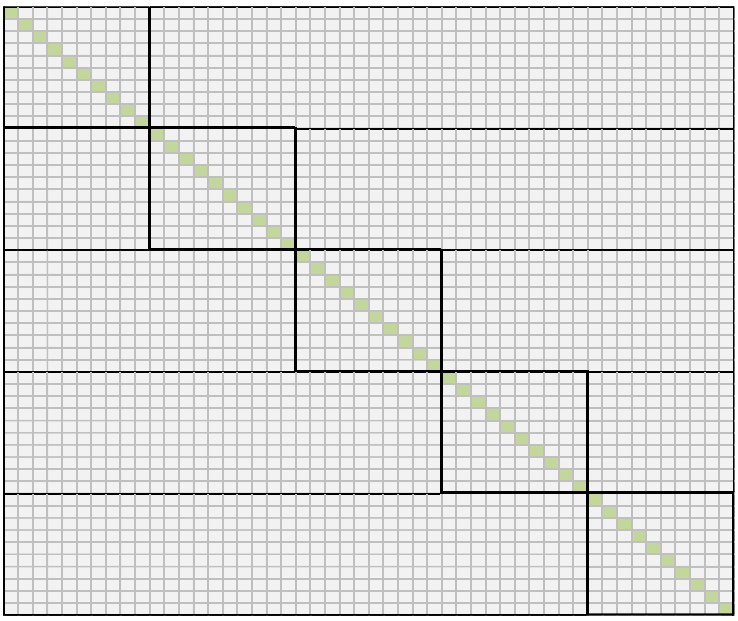


*Rysunek 19. Przykładowa macierz pasmowa*

Jeśli zostałoby wybrane najbardziej naiwne rozwiązanie, tj. przesyłanie za każdym razem cały fragment macierzy odpowiadający części, wielokrotnie odbywałyby się transfery zbędnych danych.

Posługując się przykładem eliminacji Gaussa przeprowadzonym wcześniej można wysunąć dwa kolejne stwierdzenia. Po pierwsze, dla danej macierzy podczas eliminacji w przód nie zajdzie żadna zmiana w kolumnach o indeksach mniejszych niż najniższy indeks kolumny zawierającej wyraz niezerowy. Po drugie, dla danej macierzy nie zajdzie żadna zmiana w kolumnach o indeksach większych niż najwyższy indeks kolumny zawierającej wyraz niezerowy.

W związku z tym, pozycje które nie mogą zostać zmienione nie są alokowane po stronie urządzenia obliczeniowego. Obszary które zostałyby przesłane dla przykładowej macierzy zostały oznaczone na rysunku 20 pogrubionym obramowaniem.



*Rysunek 20. Przykładowa macierz pasmowa*

Na rysunkach przedstawiona jest oczywiście sytuacja idealna, lecz przyjęty schemat zapewnia dostateczną kompresję dla macierzy generowanych przez metodę elementów skończonych.

Jak zostało pokazane w rozdziale 3.2, długość wymaganej dla danego fragmentu macierzy mapy wynika bezpośrednio z tego, ile znajduje się w nim niewiadomych. Dzieje się tak dlatego, iż numer wiersza w którym pierwszy wyraz niezerowy jest wpisywany do mapy pod indeksem odpowiadającym temu wyrazowi.

Jak widać na powyższym przykładzie, dzięki ograniczeniu szerokości fragmentów macierzy, każdy z nich zawiera mniej niewiadomych. Mapy dla danej części macierzy alokowane są więc nie o długości równej ilości wszystkich niewiadomych, a jedynie szerokości fragmentu. Dzięki temu możliwe jest umieszczenie ich w pamięci lokalnej jednostki obliczeniowej nawet dla dużych macierzy.

Warto zauważyć, że podczas podstawiania wstecz konieczne jest dla danego wiersza sprawdzenie wartości wszystkich pól od brzegu macierzy do konkretnego wyrazu. W przypadku pesymistycznym, czyli dla wiersza zawierającego pierwszy wyraz niezerowy pod indeksem *1,* jest to N sprawdzeń. Jednakże zastosowany schemat daje wiedzę o tym, jaki jest najwyższy indeks kolumny zawierającej wyraz niezerowy w danej części macierzy. Wykorzystano tą wiedzę i zamieniono przejście od brzegu macierzy do wyrazu na przejście od kolumny zawierającej najbardziej wysunięty na prawo wyraz niezerowy do aktualnie obrabianego wyrazu.

### 3.4.2 PAMIĘĆ LOKALNA

Podczas analizy problemu okazało się, iż nie jest konieczne blokowanie dostępu do całego wektora mapy podczas wykonywania zadań. Z racji tego, iż dla danego wiersza w danym momencie musi być zapewniony synchroniczny dostęp tylko do jednej pozycji w tym wektorze, konieczne jest zablokowanie dostępu tylko do tej jednej pozycji. Zarzucony został pomysł wykorzystania wykluczenia wzajemnego (mutexów); zamiast tego w kernelu zastosowano operację atomową wbudowaną w OpenCL funkcją `atomic_cmpxchg`[5].

Funkcja ta przyjmuje trzy parametry: wskaźnik na miejsce docelowe w pamięci, wartość do porównania z nim i wartość do ewentualnego podstawienia. W momencie, kiedy zadanie rozpatrujące dany wiersz ustali, jaki jest jego pierwszy wyraz niezerowy, wywołuje funkcję `atomic_cmpxchg` w następujący (zapisany poglądowo) sposób:

```
atomic_cmpxchg(&mapa[pierwszy_wyraz_niezerowy], -1, numer_wiersza)
```

Dla takich danych funkcja `atomic_cmpxchg` porówna pozycję w wektorze mapy podaną pozycję z -1. Jeżeli są takie same, na miejsce to zostanie wstawiona wartość zmiennej *numer_wiersza*, a funkcja zwróci -1. Jeżeli nie są, tj. w mapie na podanej pozycji jest już wpis różny od -1, na mapie nie zostanie wykonana żadna operacja, funkcja natomiast zwróci wartość, która została znaleziona w mapie. W ten sposób wywołaniem jednej funkcji w mapie zostaje umieszczona właściwa informacja, bądź do zadania trafia informacja o tym, względem którego wiersza należy wyeliminować pierwszy wyraz niezerowy w aktualnie przetwarzanym wierszu.

Najprostszym i najbardziej intuicyjnym rozwiązaniem byłoby umieszczenie pojedynczej mapy w pamięci globalnej, gdzie mogą ją osiągnąć wszystkie zadania. Jednakże jak zostało nadmienione wcześniej, operacje atomowe na pamięci globalnej są inherentnie powolne. Znacznie lepszym w kontekście prędkości wykonania jest przechowanie mapy w pamięci lokalnej.

W praktyce w kernelach OpenCL zastosowanych w rozwiązaniu dla każdej grupy roboczej tworzona jest osobna mapa, w identycznej postaci jak ta zaproponowana powyżej w opisie równoległego algorytmu.

Zadanie ustawienia początkowych wartości w lokalnej mapie zostało scedowane w każdej grupie roboczej na pierwsze z jej zadań, tj. to o lokalnym numerze identyfikacyjnym

[5] http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomic_cmpxchg.html

zero. Po kodzie wykonującym tą operację została umieszczona lokalna bariera, dzięki czemu żadne z zadań w grupie roboczej nie rozpocznie wykonania dalszego kodu zanim mapa nie zostanie wypełniona wpisami *-1*.

### 3.4.3 PRZECHOWANIE MACIERZY PO STRONIE GOSPODARZA

O ile na urządzenie obliczeniowe wysyłane są tylko fragmenty macierzy, po stronie gospodarza konieczne jest przechowanie całości macierzy w sposób pozwalający na zapis i odczyt dowolnych pól w macierzy. Jak jednakże łatwo obliczyć, próba wpisania do pamięci wszystkich pól macierzy o stu tysiącach niewiadomych jako typu *double* zużyłaby około 75GB.

Różne schematy kompresji macierzy rzadkich zostały zaprezentowane w rozdziale 2.3.2. Na potrzeby oprogramowania zrezygnowano z implementacji jednej z metod, zamiast tego wykorzystując część biblioteki Boost o nazwie uBLAS. Dostarcza ona klas do efektywnego przechowywania macierzy rzadkich, między innymi zastosowanej w proponowanym rozwiązaniu klasy *compressed_matrix*.[6] Klasa ta domyślnie przechowuje macierz przy pomocy opisanej wcześniej metody CSR.

### 3.4.4 PRZECHOWANIE KOLEKCJI MAP PO STRONIE GOSPODARZA

Ze względu na zastosowaną koncepcję w skali globalnej istnieje tyle map o długości równej ilości niewiadomych w macierzy, ile jest części macierzy. Jak wspomniano wcześniej, dla macierzy o rozmiarze 10 000 x 10 000, przy globalnej maksymalnej ilości wątków 1024, byłoby to 10 map o długości 10 000.

Nie jest to jeszcze duża ilość danych. Dlatego początkowo mapa przechowywana była w strukturze typu *std::vector* z biblioteki STL C++. Jednakże przy macierzach opisujących układy o więcej niż milionie niewiadomych zaczęło to powodować błędy. Działo się tak dlatego, iż w standardzie C++ od wektora wymagane jest podczas konstrukcji zaalokowanie ciągłego, odpowiednio długiego bloku pamięci [16]. Przy próbie konstrukcji mapy dla macierzy o milionie niewiadomych przy globalnej maksymalnej ilości wątków, czyli 977 częściach, następowała próba alokacji 977 * 1 000 000 zmiennych typu *integer*, co daje nieco ponad 1.8GB ciągłej pamięci.

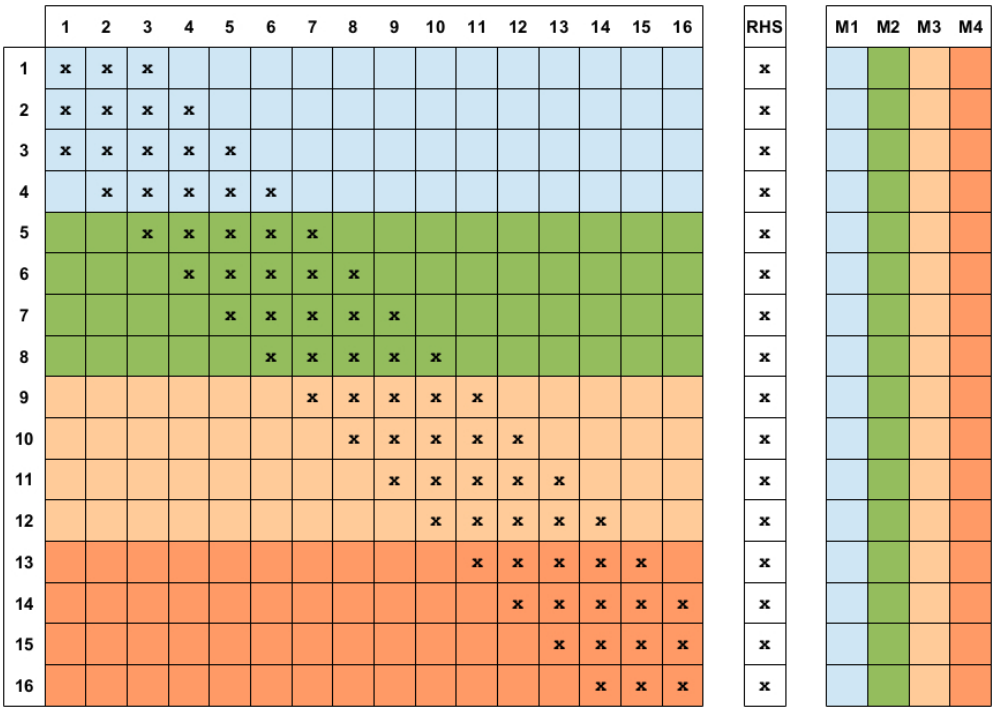
[6] http://www.boost.org/doc/libs/1_52_0/libs/numeric/ublas/doc/matrix_sparse.htm#2CompressedMatrix

Ponieważ kolekcja map jest zasadniczo macierzą, uwaga została skierowana w kierunku wykorzystywanej już i opisanej powyżej struktury *compressed_matrix* z biblioteki uBLAS. Jednakże domyślną wartością dla niezaalokowanej pozycji jest tam *0*, a nie *-1*. Dlatego wokół klasy *compressed_matrix* stworzony został wrapper modyfikujący nieznalezioną wartość.

## 3.5 PRZYKŁAD DZIAŁANIA PROPONOWANEGO ALGORYTMU

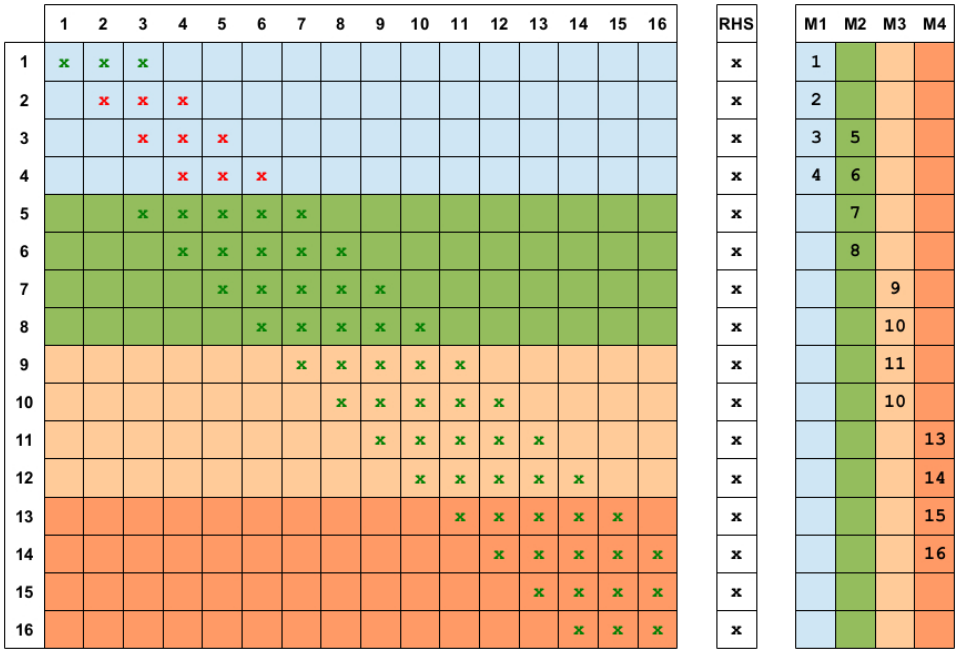
Na rysunku 21 została przedstawiona przykładowa macierz o wymiarze N równym 16. Na potrzeby tego przykładu założono, iż zadania rozwiązujące tą macierz zostały podzielone na cztery bloki oraz że w macierzy utworzona została tylko jedna część. Działanie części synchronizującej mapy pochodzące z części na CPU jest analogiczne z drugim z przedstawionych kerneli, zostało więc pominięte by nie przedłużać przykładu.

Obok macierzy i wektora prawej strony przedstawione zostały cztery mapy lokalne oznaczone M1,…,M4, ułożone tak, jak zostaną wpisane do mapy globalnej; kolor symbolizuje przypisanie map lokalnych do grup roboczych.



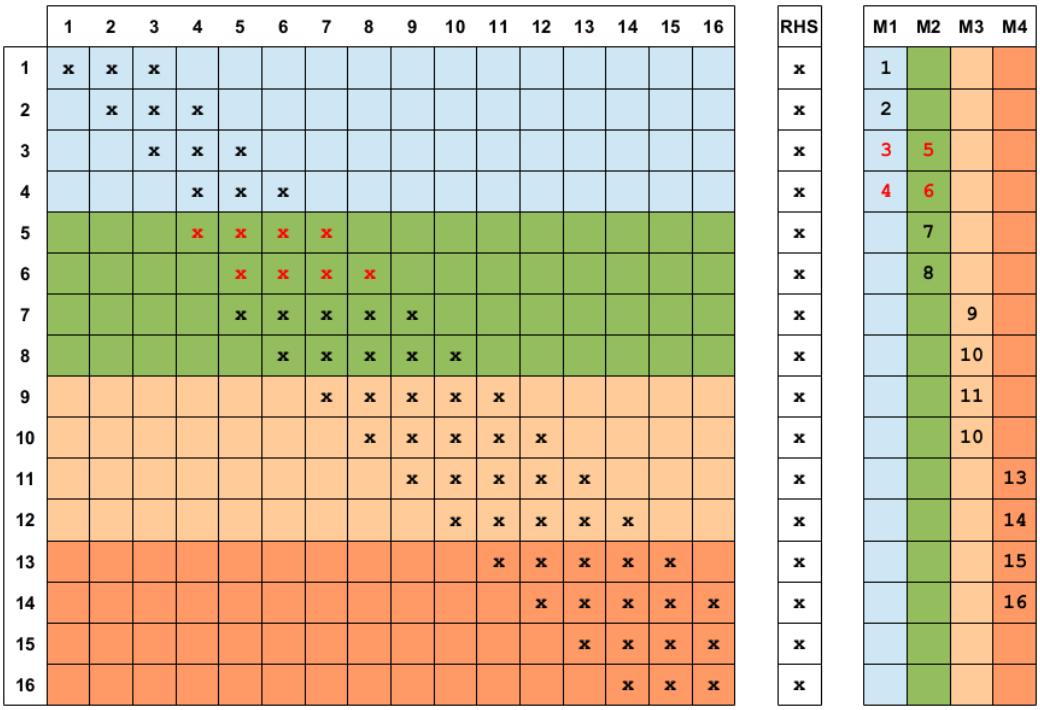

*Rysunek 21. Przykładowa macierz do rozwiązania proponowanym algorytmem*

Na rysunku 22 został przedstawiony wynik działania pierwszego z opisywanych kerneli, przywracającego warunek unikalności lokalnej. Kolorem zielonym oznaczono wiersze których ID zostały umieszczone w mapach lokalnych bez konieczności przeprowadzenia żadnych dodatkowych eliminacji. Kolorem czerwonym oznaczono wiersze, które uległy eliminacji.



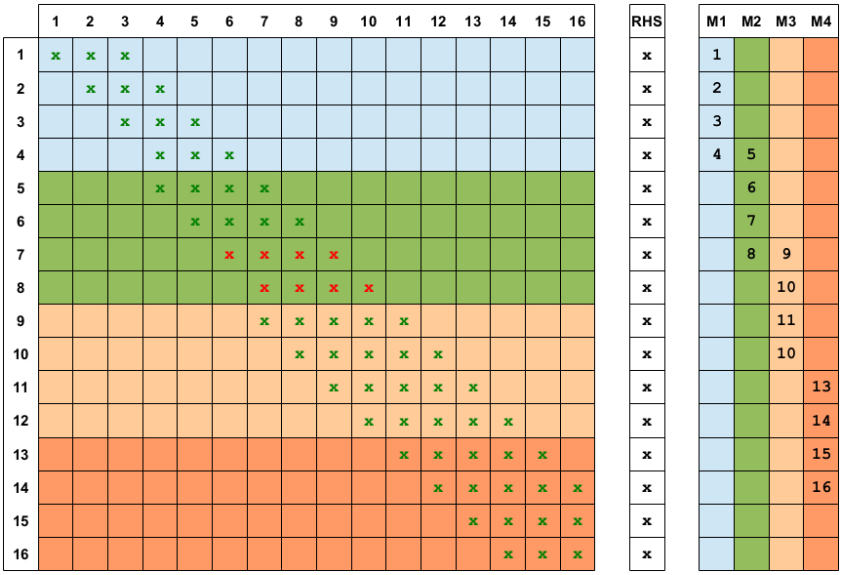

*Rysunek 22. Efekt wykonania pierwszego kernela w pierwszym cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

Jak można łatwo zauważyć, warunek unikalności pierwszego wyrazu niezerowego został przywrócony wewnątrz grup roboczych, lecz nie jest spełniony w skali globalnej. Na rysunku 23 zostało przedstawione działanie drugiego kernela, który przywraca jego spełnienie globalnie. Kolorem czerwonym w macierzy zostały zaznaczone wiersze, które uległy eliminacji. Kolorem czerwonym w mapach zostały zaznaczone pozycje, w których zostały znalezione rozwiązane właśnie konflikty.



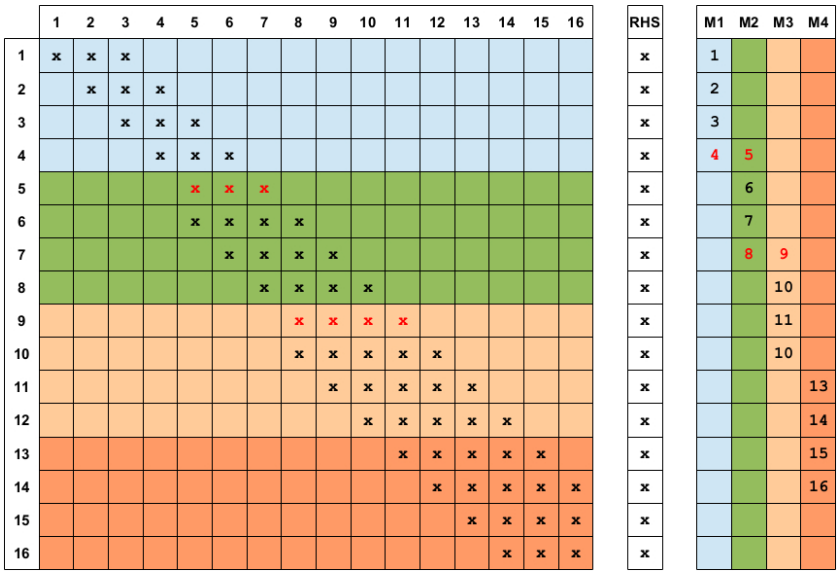

*Rysunek 23. Efekt wykonania drugiego kernela w pierwszym cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

W wyniku działania drugiego kernela zostanie zarejestrowana informacja, iż wykonano dwie operacje. Dana ta zostanie pobrana z powrotem z urządzenia OpenCL na hosta, w związku z czym zostanie podjęta decyzja o wykonaniu kolejnego cyklu rozwiązania. Efekt wykonania pierwszego kernela w drugim cyklu rozwiązania został przedstawiony na rysunku 24. Ponownie kolorem zielony zostały zaznaczone wiersze, dla których nie zostały wykonane żadne operacje, zaś kolorem czerwonym wiersze, dla których zostały przeprowadzone eliminacje, by przywrócić warunek unikalności pierwszych wyrazów niezerowych w obrębie grupy roboczej.




*Rysunek 24. Efekt wykonania pierwszego kernela w drugim cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

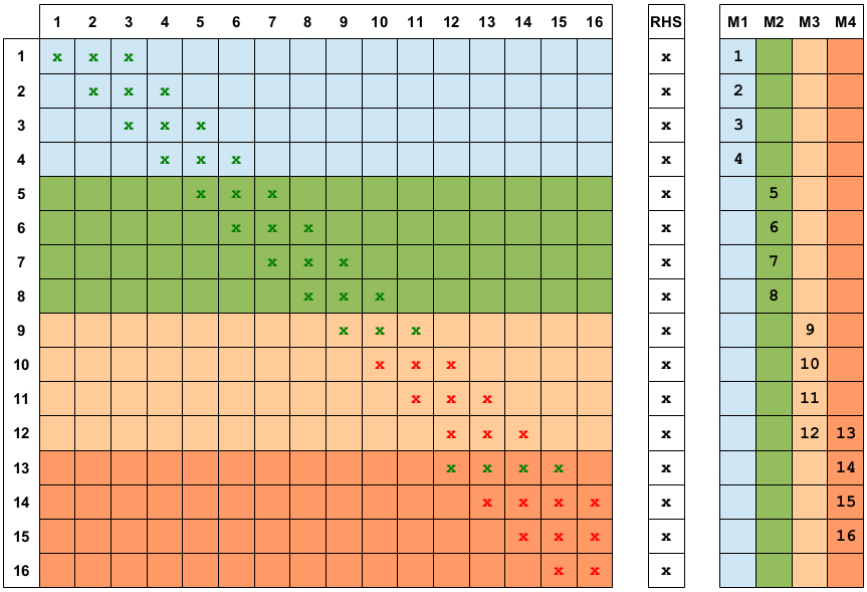
W wyniku wykonania pierwszego kernela powstały nowe globalne konflikty unikalności. Odpowiadające im wiersze w mapie globalnej zostały zaznaczone kolorem czerwonym na rysunku 25.




*Rysunek 25. Efekt wykonania drugiego kernela w drugim cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

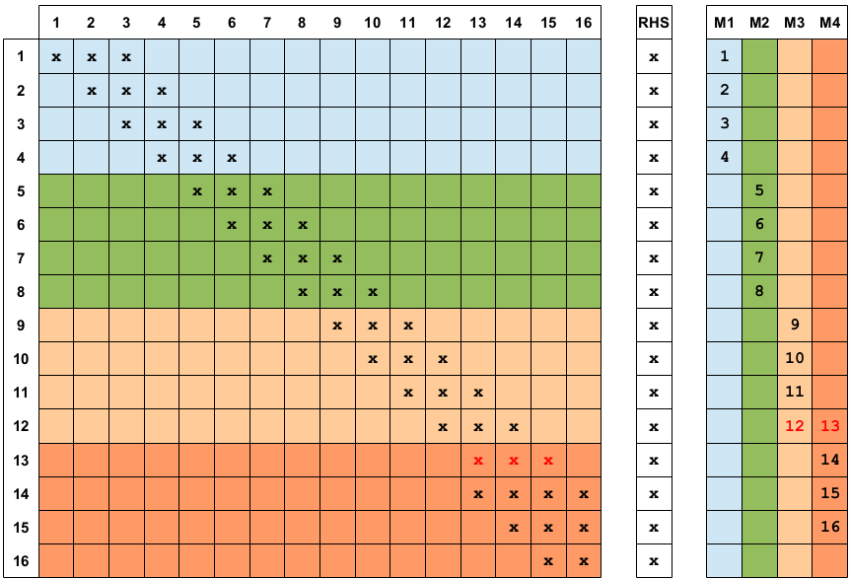
Drugi kernel w drugim cyklu rozwiązania ponownie wykonał dwie operacje eliminacji konfliktów globalnych. Ta informacja zostanie ściągnięta z urządzenia obliczeniowego do programu gospodarza i w związku z nią zostanie znowu podjęta decyzja o przeprowadzeniu kolejnego cyklu rozwiązania.

Dla zwiększenia czytelności przykładu grafiki reprezentujące dwa następne cykle rozwiązania zostaną pominięte. Zamiast tego przedstawiony zostanie cykl prowadzący do uzyskania finalnego rozwiązania układu równań. Na rysunku 26 przedstawione zostało wykonanie pierwszego kernela w przedostatnim cyklu prowadzącym do rozwiązania układu równań.



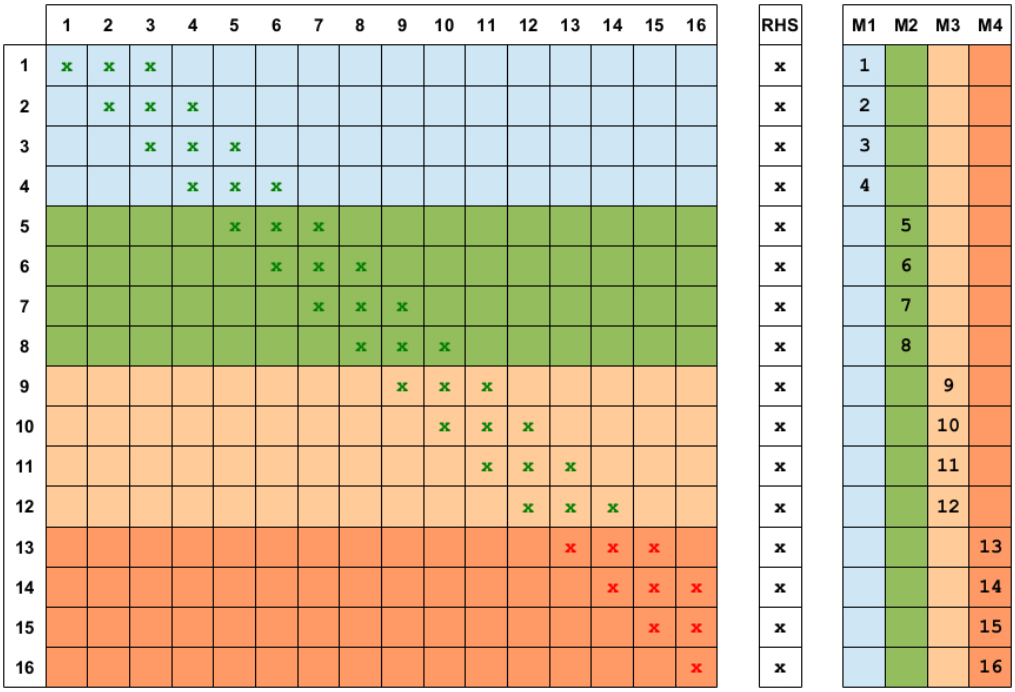

*Rysunek 26. Efekt wykonania pierwszego kernela w przedostatnim cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

Wygenerowany został już tylko jeden konflikt globalny. Na rysunku 27 zostało uwidocznione jego rozwiązanie przez drugi kernel.




*Rysunek 27. Efekt wykonania pierwszego kernela w przedostatnim cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

Z racji tego, iż została wykonana operacja eliminacji globalnej, zostanie przeprowadzony jeszcze jeden cykl rozwiązania. Jak jednakże widać na rysunku 28, tym razem kernel pierwszy, przywracający warunek unikalności, nie wytworzy już żadnych konfliktów globalnych, w związku z czym drugi kernel zwróci do gospodarza informację o niewykonaniu żadnych operacji.




*Rysunek 28. Efekt wykonania pierwszego kernela w finalnym cyklu rozwiązania układu równań liniowych proponowanym algorytmem*

Po otrzymaniu tej informacji oprogramowanie gospodarza zaprzestanie wykonywania kerneli. W następnym kroku rozwiązania macierz schodkowa, wektor prawej strony i ostateczna zawartość mapy globalnej dla tej części macierzy (na potrzeby tego przykładu – jedynej) zostanie pobrana z urządzenia obliczeniowego.

Macierz zawierająca mapę globalną zostanie sprowadzona do wektora zgodnego z ideą zaprezentowaną w rozdziale 3.2. Ponieważ jest tylko jedna część, kod synchronizujący mapy pochodzące z części nie wykona żadnych operacji i pętla rozwiązania zostanie zakończona. Następnie na uzyskanych strukturach danych zostanie przeprowadzona sekwencyjna operacja podstawienia wstecz z wykorzystaniem CPU, co zaowocuje finalnym wynikiem.

Warto zauważyć tutaj, iż do minimum zostały ograniczone transfery danych po złączu PCI-Express. Duże struktury danych, czyli fragmenty układu równań i wektora prawej strony oraz pamięć mapy globalnej zostaje przesłana w danym cyklu raz do urządzenia i raz z powrotem. Dzieje się to tylko na początku pętli pod-cykli rozwiązania dla tej części oraz na

jej końcu. W każdym pod-cyklu rozwiązania z urządzenia transferowana jest zbywalnie mała ilość danych, gdyż informacja o ilości przeprowadzonych operacji eliminacji globalnej w danej części jest zapisywana w pojedynczej zmiennej typu *unsigned int*.

# 4 BADANIA WYDAJNOŚCI

## 4.1 METODYKA BADAŃ

Badania wydajności zostały podzielone na dwie fazy. W fazie pierwszej oprogramowanie zostało uruchomione dla tej samej macierzy przy zmiennych lokalnej ilości wątków oraz globalnej ilości wątków. Na potrzeby tej fazy przyjęto również, iż lokalna ilość wątków będzie równa globalnej ilości wątków, dzięki czemu będzie można wyznaczyć najlepszą wartość pierwszego z tych parametrów.

Wyznaczona w fazie pierwszej optymalna wartość lokalnej ilości wątków zostaje zachowana jako stała dla fazy drugiej. W drugiej fazie globalna ilość wątków jest podnoszona aż do osiągnięcia maksymalnej ilości dostępnych wątków na danym urządzeniu. Na tej podstawie powinno zostać wyznaczone przyspieszenie. Pomiary te są wykonywane przy stałym rozmiarze macierzy.

Podczas wykonania fazy drugiej, dla każdego pomiaru przy stałym rozmiarze macierzy jest wykonywany pomiar przy rozmiarze macierzy proporcjonalnie większym. Jako proporcję wykorzystano tutaj stosunek lokalnej do globalnej ilości wątków. Przykładowo jeżeli użyto lokalnej ilości wątków równej *512*, oraz globalnej równej *1024,* a pierwszy pomiar przeprowadzono z macierzą o rozmiarze 10 000 niewiadomych, drugi zostanie przeprowadzony z macierzą dwukrotnie większą. Na tej podstawie wyznaczona zostanie skalowalność rozwiązania.

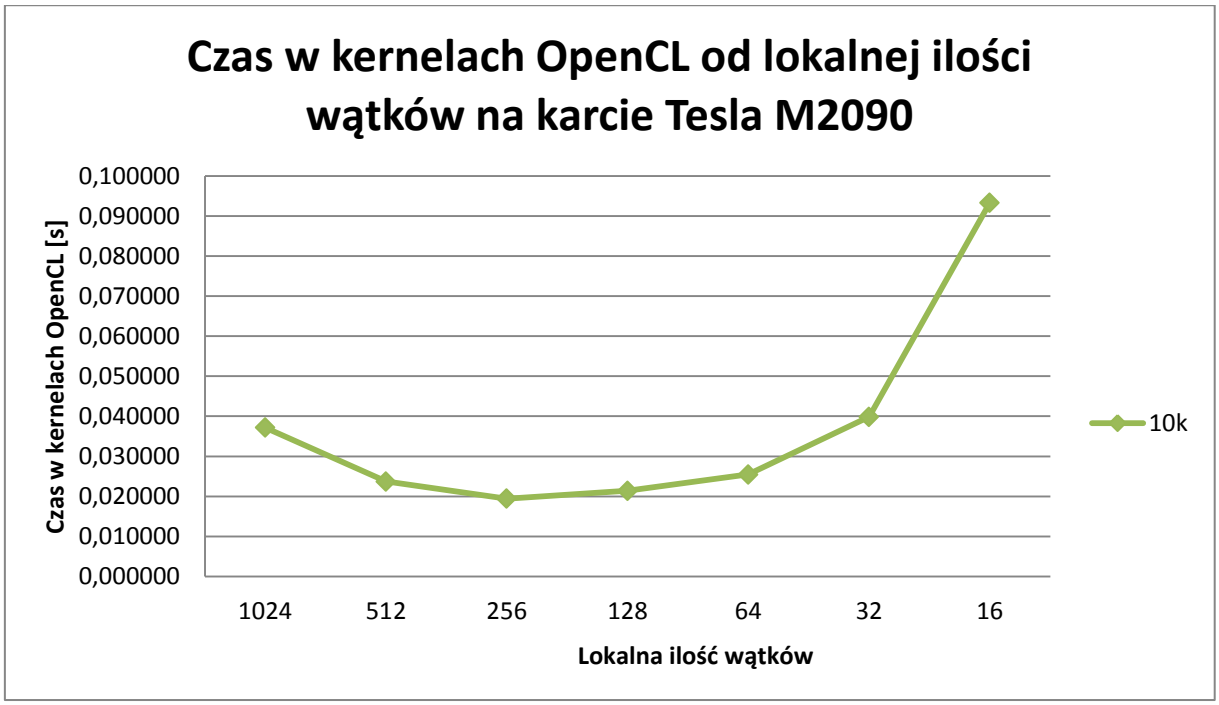
Jak zostało wspomniane, finalne składanie rozwiązania odbywa się na CPU. Zazwyczaj zajmuje to więcej czasu niż operacje wykonane na GPU. W przypadku niektórych testów jednakże część rozwiązania wykonywana na CPU zajęła o rząd wielkości więcej dla porównywalnych danych wejściowych. Badaniu całkowitych czasów wykonania i wyjaśnieniu tego zjawiska poświęcono więc osobny rozdział. W poniższych rozdziałach zostanie najpierw rozpatrzona wydajność części rozwiązania wykonywanej na karcie graficznej, by nie zaciemniać wyników badań pozornie przypadkowymi danymi dotyczącymi czasów wykonania na CPU.

## 4.2 WYZNACZENIE OPTYMALNEJ LOKALNEJ ILOŚCI WĄTKÓW

### 4.2.1 NVIDIA TESLA M2090

Celem wyznaczenia optymalnej lokalnej ilości wątków na karcie NVIDIA Tesla M2090 zostały przeprowadzone dwie serie testów, na macierzach o rozmiarach odpowiednio 10 000 niewiadomych oraz 100 000 niewiadomych.

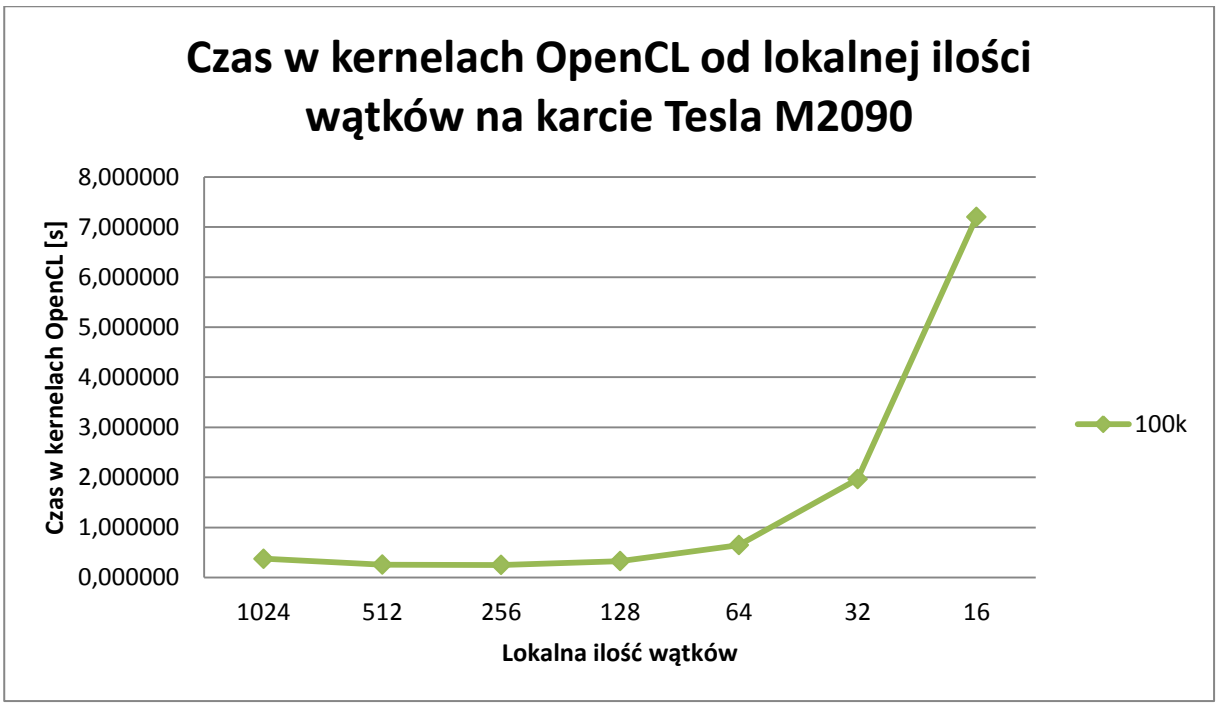
Na wykresie 1 zostały przedstawione wyniki testu przeprowadzonego na macierzy zawierającej układ równań o 10 000 niewiadomych.




*Wykres 1. Czas w kernelach OpenCL od lokalnej ilości wątków na karcie NVIDIA Tesla M2090 dla macierzy o 10 000 niewiadomych*

Na wykresie zauważyć można, iż najniższy czas wykonania rozwiązania uzyskano dla lokalnej ilości wątków (równej w tym teście globalnej) równej 256.

Wartość tą potwierdza drugi z przeprowadzonych pomiarów, którego wyniki przedstawiono na wykresie 2.



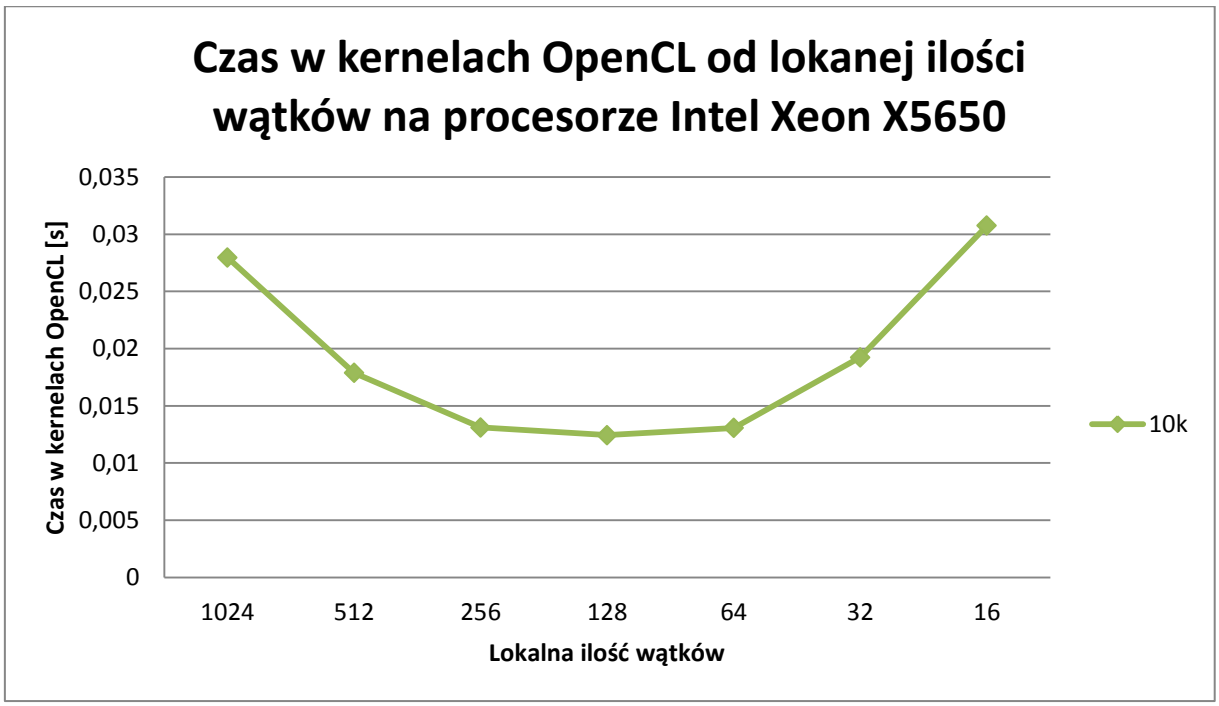

*Wykres 2. Czas w kernelach OpenCL od globalnej ilości wątków na karcie NVIDIA Tesla M2090 dla macierzy o 100 000 niewiadomych*

Dla tego wariantu wynik nie jest tak łatwy do odczytania z wykresu, jednakże dokładne dane nie pozostawiają wątpliwości. Dla lokalnej ilości wątków równej 512 totalny czas spędzony w kernelach OpenCL wyniósł 0,257143s, zaś dla 256 - 0,251640s. Wyniki te pozwalają więc skonkludować iż dla tego rozwiązania najlepszą lokalną ilością wątków jest 256.

Można również zauważyć, iż kernele OpenCL zdecydowanie tracą na wydajności wraz ze zmniejszaniem się rozmiaru grup roboczych, czego należało oczekiwać.

### 4.2.2 INTEL XEON X5650

Optymalna lokalna ilość wątków na procesorze Intel Xeon X5650 została wyznaczona w jednej serii testów na macierzy o rozmiarach 10 000 niewiadomych. Na wykresie 3 zostały przedstawione wyniki tego testu.




*Wykres 3. Czas w kernelach OpenCL od globalnej ilości wątków na procesorze Intel Xeon X5650 dla macierzy o 10 000 niewiadomych*

Dla procesora kształt wykresu jest inny; przede wszystkim nie traci on aż tak drastycznie wydajności dla małych rozmiarów grup roboczych. Z drugiej strony nie ma również dużego zysku przy dużych rozmiarach grup roboczych. Na wykresie widać, iż najniższy czas zarejestrowano dla lokalnej ilości wątków równej 128.

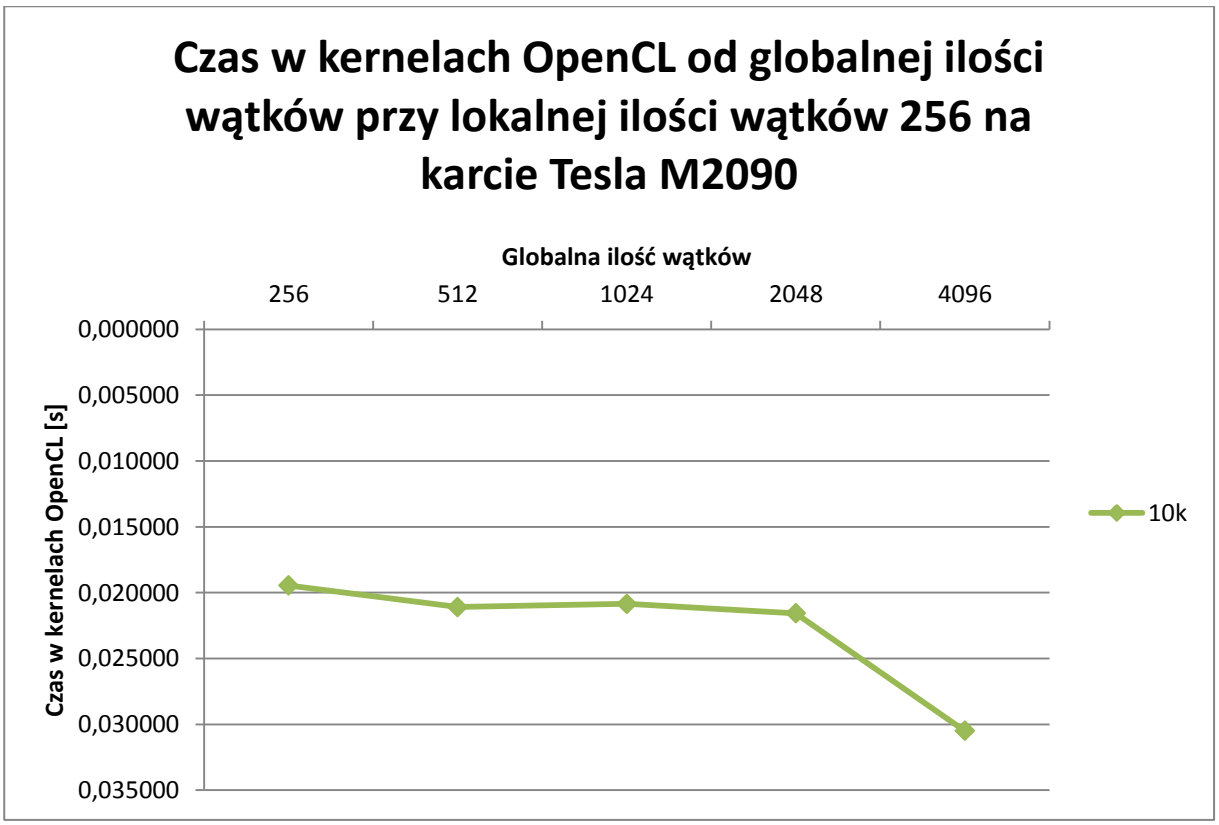
## 4.3 POMIARY PRZYSPIESZENIA

W tym punkcie przeanalizowane zostaną różne wartości dla globalnej ilości wątków przy zachowaniu stałej lokalnej ilości wątków, równej tej wyznaczonej jako optymalna dla danego urządzenia.

### 4.3.1 NVIDIA TESLA M2090

Dla urządzenia NVIDIA Tesla M2090 pomiary zostały przeprowadzone w dwóch seriach – o grubym ziarnie dla macierzy zawierającej 10 000 niewiadomych i o mniejszym ziarnie dla macierzy zawierającej 100 000 niewiadomych. Lokalna ilość wątków pozostała stała, równa 256.

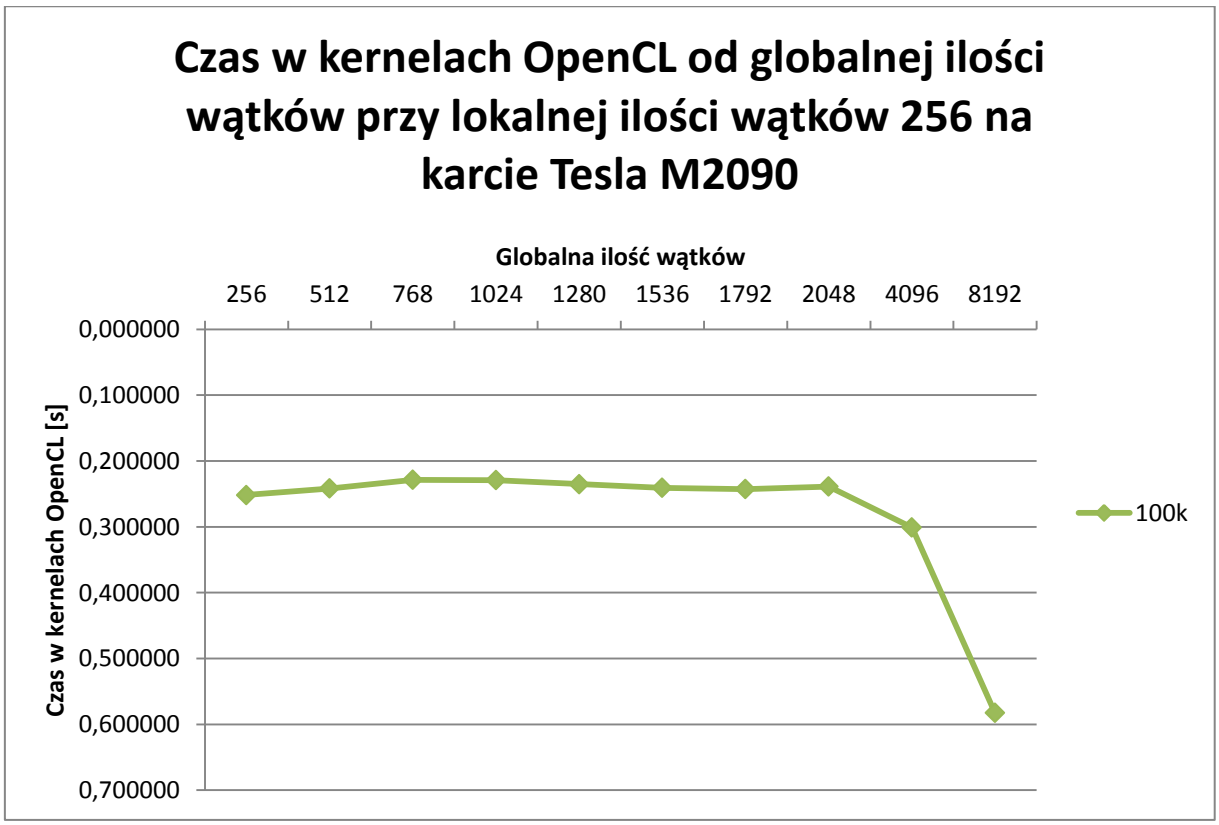
Wyniki pierwszego testu przedstawiono na wykresie 4.




*Wykres 4. Czas w kernelach OpenCL od globalnej ilości wątków na karcie NVIDIA Tesla M2090 przy stałej lokalnej ilości wątków.*

Zauważyć należy iż wykres przedstawiono z odwrotnie ułożonymi wartościami (wyżej to lepiej). Dla tej próbki nie widać jednak istotnego przyspieszenia – najkrótszy czas osiągnięto w oryginalnym teście przy lokalnej ilości wątków równej globalnej ilości wątków.

Na wykresie 5 przedstawiono rezultat drugiego testu na większej macierzy.

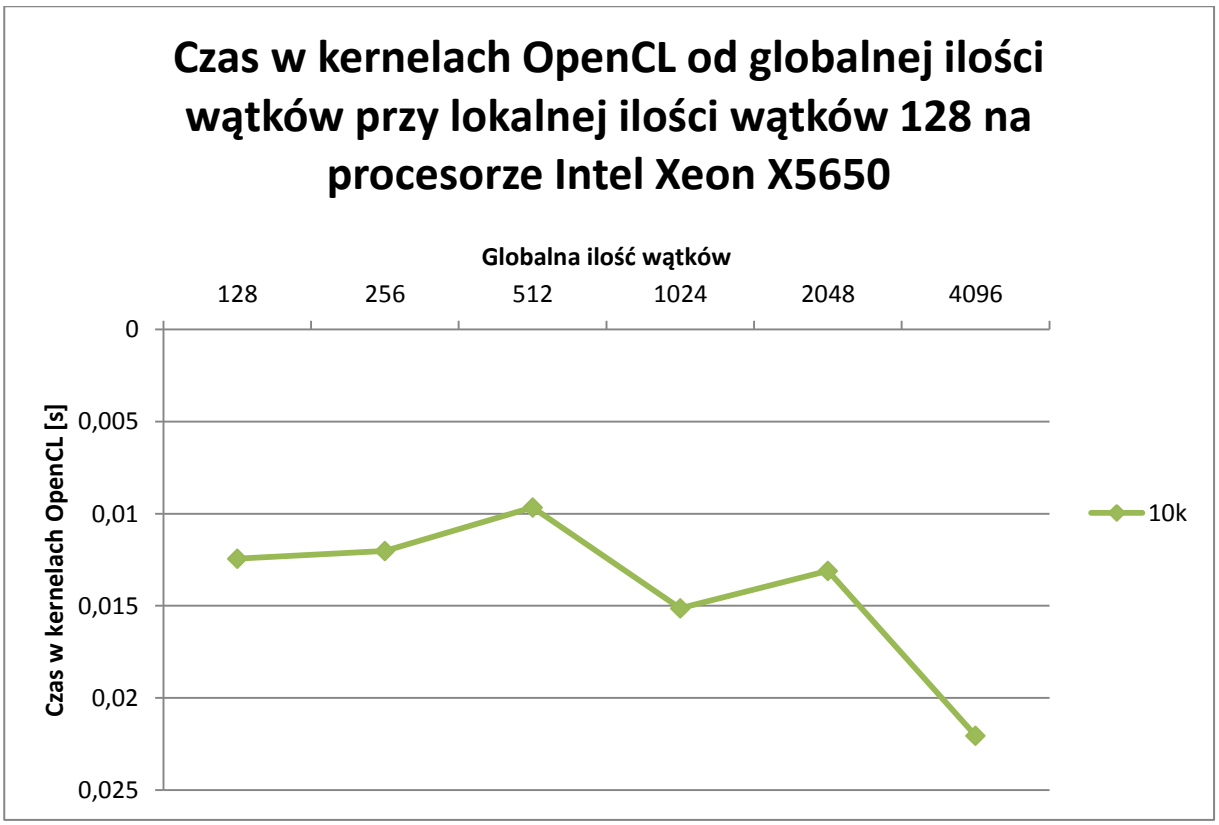



*Wykres 5. Czas w kernelach OpenCL od globalnej ilości wątków na karcie NVIDIA Tesla M2090 przy stałej lokalnej ilości wątków dla macierzy o 100 000 niewiadomych*

Dla większej macierzy przy drobniejszym ziarnie uzyskano bardziej interesujące rezultaty. Wykres ponownie zrealizowano z odwrotnie ułożonymi wartościami. Wynika z niego, iż dla większej macierzy zaobserwowano nieznaczne przyspieszenie dla globalnej ilości wątków trzykrotnie i czterokrotnie większej od lokalnej ilości wątków (wartości 768 i 1024).

### 4.3.2 INTEL XEON X5650

Na procesorze Intel Xeon X5650 ustalono optymalną lokalną ilość wątków na 128. Dla tej wartości przeprowadzono jedną serię testów przy zmiennej globalnej ilości wątków oraz stałym rozmiarze macierzy (10 000 niewiadomych). Wyniki tego testu przedstawiono na wykresie 6.




*Wykres 6. Czas w kernelach OpenCL od globalnej ilości wątków na procesorze Intel Xeon X5650 przy stałej lokalnej ilości wątków dla macierzy o 10 000 niewiadomych*

Najlepsze przyspieszenie osiągnięto przy globalnej ilości wątków równej 512. Warto zauważyć, iż podobnie jak w analogicznych testach dla karty graficznej Tesla M2090 jest to czterokrotność lokalnej ilości wątków.

## 4.4 POMIARY SKALOWALNOŚCI

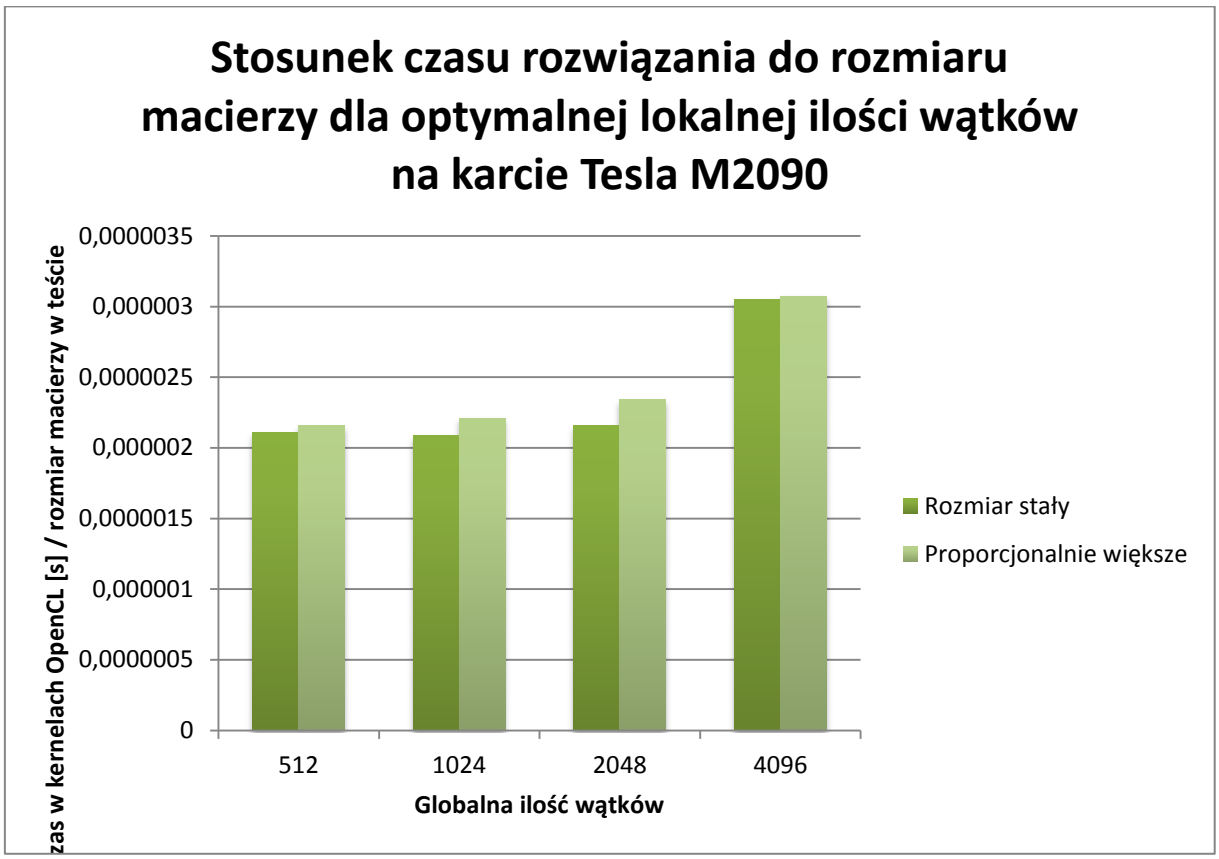
Podczas każdego z poprzednich pomiarów, dla każdej z wybranych globalnych ilości wątków przeprowadzono test na macierzy o stałym rozmiarze. W tej części przedstawiono wyniki testów, w których dla każdej globalnej ilości wątków przeprowadzono dodatkowy test na macierzy proporcjonalnie większej (zależnie od stosunku między lokalną a globalną ilością wątków).

Dla łatwiejszego porównania danych, na wykresach w niniejszym rozdziale przedstawiono czas spędzony w kernelach OpenCL podzielony przez rozmiar przetwarzanej macierzy.

### 4.4.1 NVIDIA TESLA M2090

Na tej karcie obliczeniowej optymalną ilością wątków lokalnych okazało się 256. Przeprowadzone więc zostały testy dla globalnych ilości wątków 512, 1024, 2048 i 4096. Do tych testów zastosowano odpowiednio dwukrotnie, czterokrotnie, ośmiokrotnie i

szesnastokrotnie większą macierz. Jako rozmiar podstawowy wykorzystano macierz o 10 000 niewiadomych. Wyniki tego testu przedstawiono na wykresie 7.

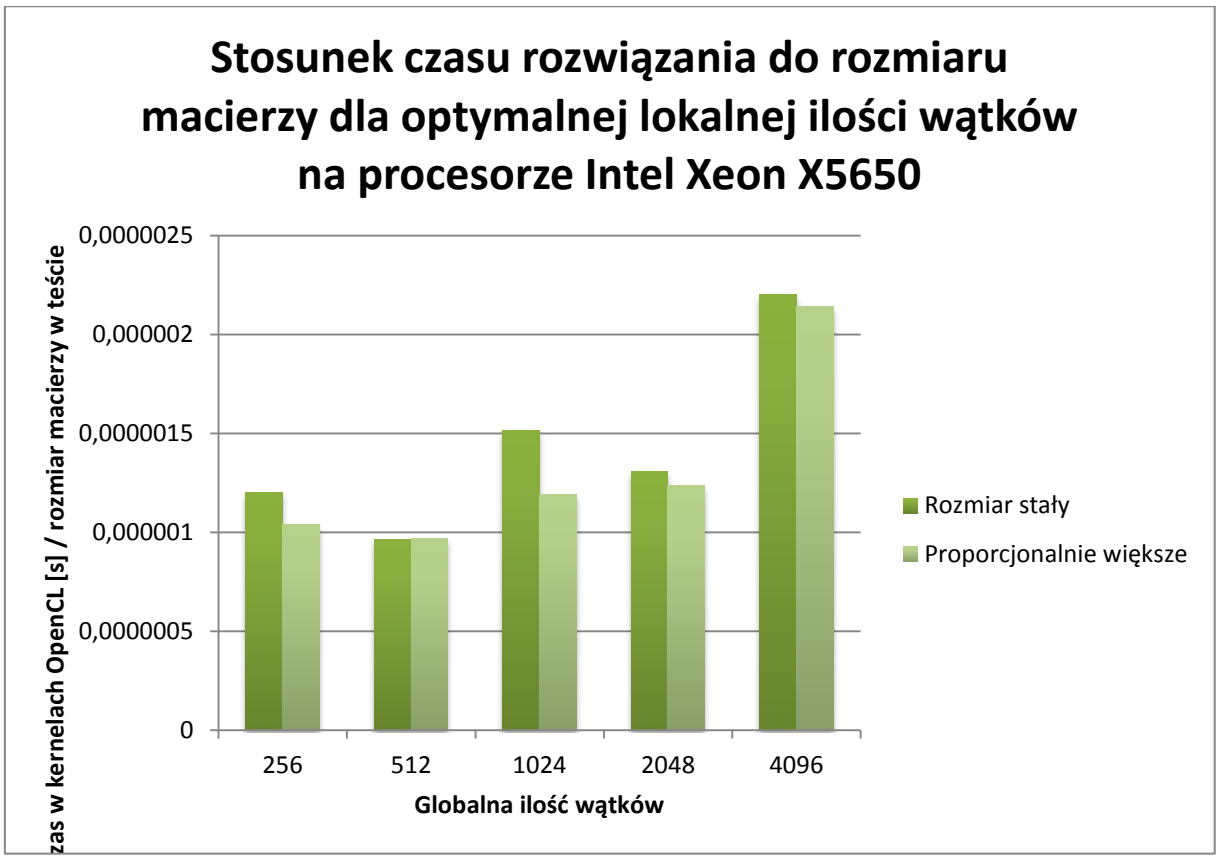



*Wykres 7. Stosunek czasu rozwiązania do rozmiaru macierzy dla optymalnej lokalnej ilości wątków na karcie NVIDIA Tesla M2090*

Jak można zauważyć na wykresie, stosunek czasu rozwiązania do rozmiaru macierzy dla konkretnych globalnych ilości wątków pozostaje stały, jest to zatem zależność liniowa.

### 4.4.2 INTEL XEON X5650

Analogiczny test został przeprowadzony dla procesora Intel Xeon X5650. Zastosowano identyczną metodologię jak w przypadku testów na karcie NVIDIA Tesla M2090, z tym, że zachowano optymalną dla badanej platformy lokalną liczbę wątków równą 128. Wyniki tego testu przedstawiono na wykresie 8.




*Wykres 8. Stosunek czasu rozwiązania do rozmiaru macierzy dla optymalnej lokalnej ilości wątków na procesorze Intel Xeon X5650*

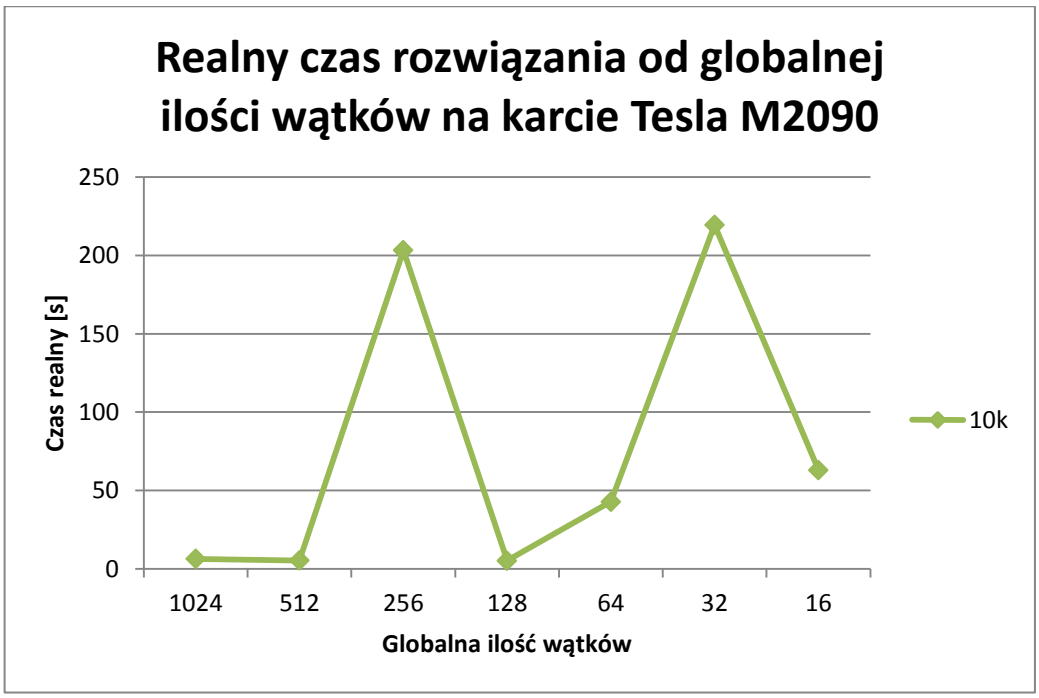
Jak można zauważyć na wykresie, czas rozwiązania na element macierzy w przypadku rozpatrywanego urządzenia jest mniejszy dla większych macierzy niż dla macierzy podstawowej. Oznacza to, iż osiągnięto nieco lepszą niż liniowa zależność czasu rozwiązania od rozmiaru macierzy dla konkretnych stosunków lokalnej do globalnej ilości wątków.

## 4.5 POMIARY CZASU RZECZYWISTEGO

W tej części ujęto pomiary czasu rzeczywistego dla obu urządzeń podczas testów z lokalną ilością wątków równą globalnej ilości wątków dla macierzy o rozmiarze 10 000 niewiadomych.

### 4.5.1 NVIDIA TESLA M2090

Wykres 9 przedstawia zmierzoną podczas testów zależność rzeczywistego czasu rozwiązania (w sekundach) od wybranej globalnej ilości wątków.



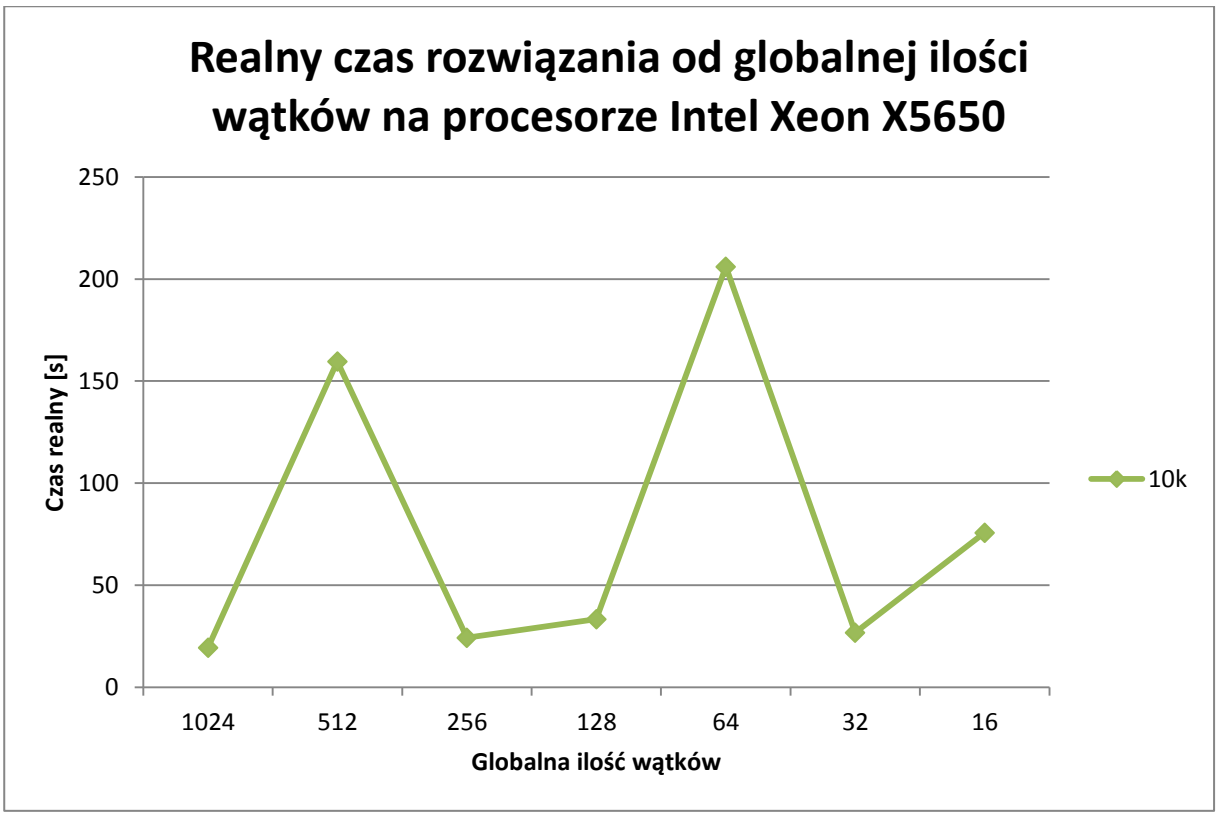

*Wykres 9. Rzeczywisty czas rozwiązania w stosunku do globalnej ilości wątków na karcie NVIDIA Tesla M2090*

Ilość map wymagających synchronizacji oraz złożenia w finalną mapę na CPU dla podstawienia wstecz jest odwrotnie proporcjonalna do globalnej ilości wątków. Można by więc spodziewać się, iż czas będzie rósł w pewnej względnie liniowej zależności w stosunku do malejącej globalnej ilości wątków.

Jak jednakże widać na wykresie, taka zależność nie zachodzi. Zamiast tego, paradoksalnie dla najlepszej z punktu widzenia wykonania kerneli wartości lokalnej ilości wątków rozwiązanie całościowe zajęło niemal najdłuższy czas.

### 4.5.2 INTEL XEON X5650

Wykres 10 został przygotowany analogicznie dla procesora Intel Xeon X5650.




*Wykres 10. Rzeczywisty czas rozwiązania w stosunku do globalnej ilości wątków na procesorze Intel Xeon X5650*

Jak widać wykres jest częściowo podobny do tego uzyskanego na karcie NVIDIA Tesla M2090, przynajmniej z punktu widzenia ilości pików, bo już nie ich rozłożenia. Ponownie piki wykresu nie są związane z żadną z mierzonych wartości w żaden zauważalny sposób.

Można pokusić się o stwierdzenie, iż być może w systemie pod kontrolą którego wykonywano testy, mimo iż jego czas obliczeniowy był zarezerwowany dla wykonania testów, mogły działać jakieś usługi systemowe bądź zaplanowane zadania. Z racji tego, że składanie rozwiązania na CPU to proces intensywny obliczeniowo i wykorzystujący wszystkie dostępne rdzenie wszystkich znalezionych procesorów CPU, nawet niewielkie nieplanowane obciążenie mogłoby diametralnie zmienić wyniki eksperymentu.

## 4.6 POZOSTAŁE WYNIKI

W tej części ujęto dodatkowe wyniki uzyskane podczas prowadzenia eksperymentów, niezwiązane konkretnie z żadną z prezentowanych wcześniej serią danych.

### 4.6.1 ZUŻYCIE PRĄDU PRZEZ KARTĘ GRAFICZNĄ

Podczas wykonywania testów na karcie NVIDIA Tesla M2090 śledzone było zużycie prądu przez kartę (za pośrednictwem narzędzia *nvidia-smi*). Według specyfikacji [13] maksymalna możliwa moc dla tej karty wynosi 225W. Podczas przeprowadzonych testów

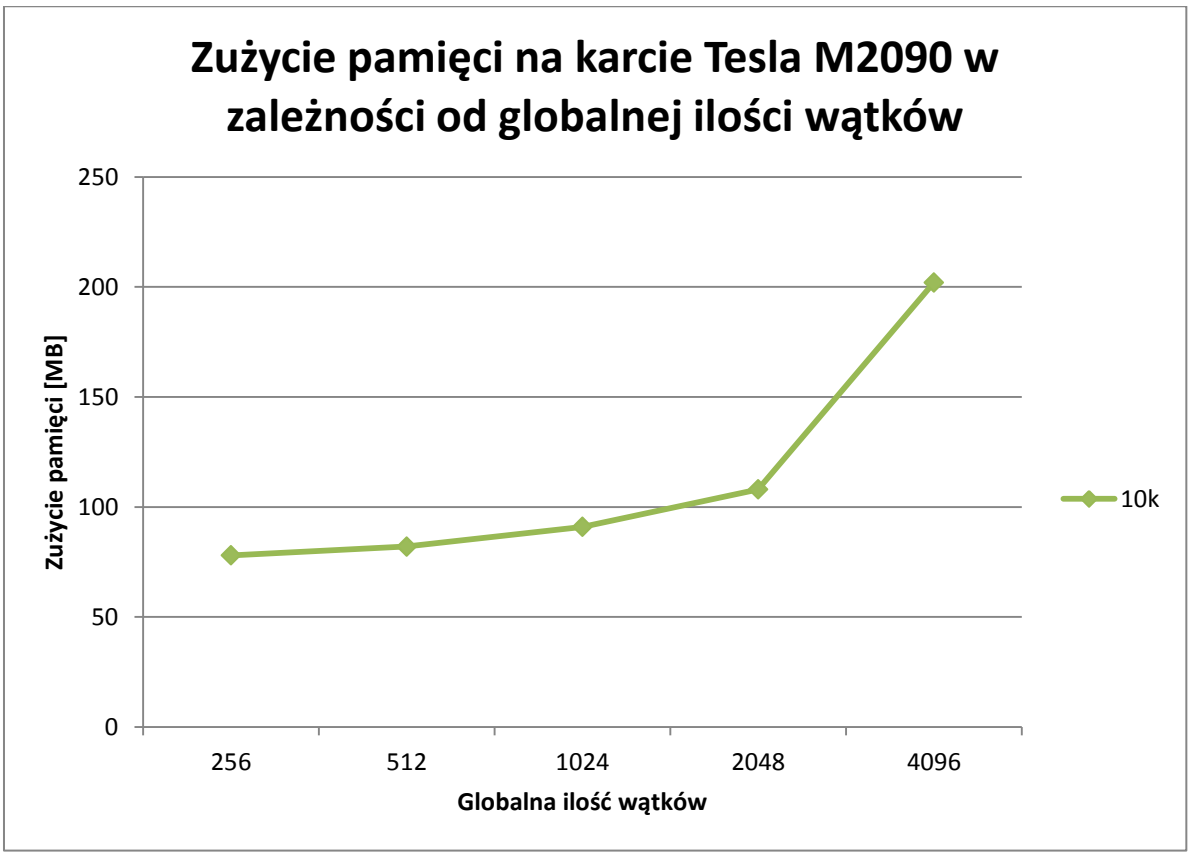
wykorzystywana przez kartę moc oscylowała między 81W a 87W w sposób niezależny od rozmiaru macierzy, założonych ilości wątków oraz czasu rozwiązania.

### 4.6.2 ZUŻYCIE PAMIĘCI KARTY GRAFICZNEJ

Wykorzystywana Tesla M2090 jest skonfigurowana tak, by wykorzystywać ECC (*error correcting codes* lub kody korekcji błędów). Dzięki temu istnieje niższe prawdopodobieństwo, że zajdą błędy odczytu lub zapisu pamięci, lecz ogólna ilość dostępnej pamięci spada z bazowych 6GB do około 5.2GB [13]. Zużycie pamięci mierzono przy użyciu narzędzia *nvidia-smi*.

Podczas bazowych testów (z globalną ilością wątków nie przekraczającą 1024) zużycie pamięci pozostawało na poziomie 78MB – 80MB. Co więcej, nie spadło poniżej tego poziomu nawet dla bardzo małych wartości globalnej ilości wątków. Ponieważ rozmiar części macierzy wysyłanej do urządzenia jest bezpośrednio związany z globalną ilością wątków, należałoby się spodziewać że przy minimalnych wartościach tej zmiennej zużycie pamięci będzie minimalne. Można więc stwierdzić, że około 80MB to minimalne zużycie pamięci dla proponowanego solwera.

Podczas testów z rosnącą globalną ilością wątków w stosunku do lokalnej ilości wątków zużycie pamięci rosło. Zależność przedstawiono na wykresie 11.




*Wykres 11. Zależność zużycia pamięci od globalnej ilości wątków przy stałej lokalnej ilości wątków na karcie NVIDIA Tesla M2090*

Ponieważ wraz ze wzrostem globalnej ilości wątków do urządzenia przesyłane są większe obszary macierzy, jest to zależność zgodna z teoretycznymi przewidywaniami.

### 4.6.3 OBCIĄŻENIE RDZENI PROCESORA PRZEZ OPENCL

Podczas wykonywania testów na procesorze Intel Xeon X5650 obciążenie jego rdzeni było obserwowane za pomocą narzędzia *htop*. Potwierdziło to między innymi zasadność wykorzystania OpenMP w części rozwiązania wykonywanej bezpośrednio na CPU. Wykazano tam niemal stuprocentowe obciążenie wszystkich dostępnych rdzeni procesora.

Podczas wykonania samych kerneli OpenCL stwierdzono zgodnie z przewidywaniami obciążenie tylko jednego z dwóch dostępnych procesorów (solwer współpracuje z jednym urządzeniem obliczeniowym na raz). W czasie obserwacji obciążenie rdzeni w obrębie jednego procesora niekiedy tylko osiągało 100%. Wartością typową podczas wykonywania kerneli OpenCL było około 60-70%.

# 5 WNIOSKI

W niniejszej pracy został zaproponowany frontalny solwer MES, zrównoleglony przy pomocy akceleratorów obliczeń GPU. Została również pokazana jego przenośność na inne rodzaje platform wspierające OpenCL, takie jak procesory firmy Intel. Przeanalizowany został potencjał masowo równoległej wersji metody eliminacji Gaussa oraz metody optymalizacji wykorzystania mocy i pamięci wielordzeniowych urządzeń obliczeniowych. W proponowanym rozwiązaniu zostały zidentyfikowane dwa wąskie gardła, czyli słabości powodujące spowolnienie uzyskania ostatecznego rozwiązania.

Pierwszym z nich jest konieczność przesyłania pewnej ilości danych z i do urządzenia. Narzut czasowy wynikający z tego problemu został zredukowany w sposób zadowalający dzięki podzieleniu rozwiązania na osobne fronty i ograniczenie transferów po złączu PCI-Express do minimum za pomocą kompresji danych.

Drugim jest konieczność wykonania dużej ilości operacji podczas ostatecznej synchronizacji map oraz fazy podstawiania wstecz samej metody Gaussa. Brak możliwości zrównoleglenia podstawiania wstecz został pokazany w rozdziale 3.2.3. Brak możliwości zrównoleglenia na GPU ostatecznej synchronizacji map wynika w tej fazie z konieczności dostępu do całości macierzy, rozmiar której zazwyczaj uniemożliwia umieszczenie jej w całości w pamięci urządzenia. Konieczność wykonania tych dwóch faz powoduje wydłużenie ogólnego czasu rozwiązania mimo zastosowania równoległości po stronie CPU z użyciem API OpenMP i możliwych optymalizacji podczas podstawienia wstecz.

Mimo tych mankamentów przedstawione rozwiązanie jest przenośne. Dość dobrze wykorzystuje również możliwości obliczeń masowo równoległych zapewniane przez zastosowane urządzenia obliczeniowe. Podczas testów ze zmiennym rozmiarem macierzy wykazano, że rozwiązanie jest dobrze skalowalne na GPU, a w badanym przypadku nawet lepiej skaluje się podczas obliczeń za pośrednictwem OpenCL z użyciem procesora CPU. Rozwiązanie było stabilne i konsekwentnie dawało wyniki dla oznaczonych układów równań.

Zaproponowany solwer jest obiecujący z jeszcze jednego powodu. O ile do testów jako GPU wykorzystano doskonałe, dedykowane urządzenie obliczeniowe NVIDIA Tesla M2090, dzięki OpenCL istnieje możliwość uruchomienia proponowanego oprogramowania również na konsumenckich kartach graficznych firm NVIDIA i AMD. Nawet konsumenckie układy mogą zapewnić proponowanemu algorytmowi dobrą wydajność w stosunku do koniecznej inwestycji w sprzęt. Dla przykładu, konsumencka karta NVIDIA GeForce GTX 550 Ti,

wykorzystywana we wczesnej fazie tworzenia proponowanego oprogramowania, zapewnia około 691 GFLOPs[7] w trybie pojedynczej precyzji, przy koszcie około 300 zł[8].

Niniejsza praca dowodzi, że masowo równoległy solwer frontalny, zbudowany w zgodzie z paradygmatem czarnej skrzynki, jest możliwy oraz może zapewnić dobrą wydajność. W przyszłości można skupić się na proponowanym algorytmie od strony ograniczenia narzutu obliczeniowego na CPU niezwiązanego z zastosowaniem OpenCL. Jeszcze lepszą wydajność może zapewnić wyeliminowanie lub modyfikacja kosztownego procesu ostatecznej synchronizacji map tak, by wykorzystywał możliwości nowoczesnych GPU. Możliwe jest również przeanalizowanie fazy podstawiania wstecz w kierunku dalszych optymalizacji.

[7] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-550ti/specifications
[8] Stan na styczeń 2014

# 6 BIBLIOGRAFIA

[1] Bathe, K. J.: Finite element procedures in engineering analysis. *Englewood Cliffs: Prentice-Hall*. 1982.

[2] Bernstein, A.J.: Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers.* 1966 Vol. EC-15, no. 5, pp. 757-763

[3] Butrylo, B. [et al.]: A Survey of Parallel Solvers for the Finite Element Method in Computational Electromagnetics. *International Journal for Computation and Mathematics in Electrical and Electronic Engineering.* 2004 Vol. 23, no. 2, pp. 531-546.

[4] Cook, S.: CUDA Programming. A Developer's Guide to Parallel Computing with GPUs. *Waltham: Elsevier.* 2013.

[5] Irons, B.: A Frontal Solution Program For Infinite Element Analysis. *International Journal for Numerical Methods in Engineering.* 1970 Vol. 2, pp. 5-32

[6] Introduction to OpenCL Programming – Training Guide. *AMD.* 2010

[7] Jamil, N.: A Comparison of Direct and Indirect Solvers for Linear Systems of Equations. *International Journal of Emerging Sciences.* 2012 Vol. 2, no. 2, pp. 310-321

[8] Milenin, A.: Podstawy MES. Zagadnienia termomechaniczne. *AGH*. 2010

[9] Rońda J., Oliver G.J., Introduction to numerical methods with Matlab procedures. *AGH*. 2010

[10] Tewarson, R.P.: Sparse matrices. *New York: Academic Press, Inc.* 1973

[11] ICT 1900 Series Central Processors 1904, 1905, *ICT Press release* (ICT), 1964 p. 4. [dostęp: 2013-11-02], Dostępny w Internecie: <http://bitsavers.trailing-edge.com/pdf/ict_icl/1900/brochures/1904_Central_Processor_Sep64.pdf>

[12] Conformant Products [online]. *Khronos Group*. [dostęp: 2013-12-11], Dostępny w Internecie: <http://www.khronos.org/conformance/adopters/conformant-products#opencl>

[13] Tesla M2090 Dual-slot Computing Processor Module. *NVIDIA.* [dostęp: 2013-01-02], Dostępny w Internecie: <http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf>

[14] Tesla M-Class GPU Computing Modules. Accelerating Science. *NVIDIA*. [dostęp: 2013-01-02], Dostępny w Internecie: <http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf>

[15] The OpenCL Specification [online]. *Khronos Group*. [dostęp: 2013-12-15], Dostępny w Internecie: <http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf>

[16] Working Draft, Standard for Programming Language C++. Revision N3225 [online]. [dostęp: 2013-01-04], Dostępny w Internecie: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>