



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ KATEDRA INFORMATYKI
STOSOWANEJ I MODELOWANIA

Projekt dyplomowy

*Wykorzystanie interfejsu Vulkan do implementacji mnożenia macierzy
rzadkiej z wektorem na procesorach graficznych*

*Using the Vulkan interface to implement sparse matrix-vector
multiplication on graphics processors*

Autor:

Mateusz Radomski

Kierunek studiów:

Informatyka Techniczna

Opiekun pracy:

Dr hab. inż. Krzysztof Banaś

Kraków, 2023

Spis treści

1. Wprowadzenie	6
1.1. Cele pracy	6
1.2. Zawartość pracy	6
2. Mnożenie macierz-wektor dla macierzy rzadkich	7
2.1. Problem mnożenia macierzy poprzez wektor	7
2.2. Macierze rzadkie	7
2.3. Problem mnożenia macierzy rzadkiej poprzez wektor	7
2.4. Formaty macierzy rzadkich	8
3. Mikro-architektura, interfejs Vulkan i shadery obliczeniowe	12
3.1. Mikro-architektura nowoczesnych procesorów graficznych	12
3.2. Interfejs Vulkan	13
3.3. Shadery obliczeniowe	14
4. Implementacja	16
4.1. Komunikacja z procesorem graficznym	16
4.2. Implementacja mnożenia macierz - wektor dla każdego z formatów	19
4.2.1. Format COO	20
4.2.2. Format CSR	22
4.2.3. Format CSC	23
5. Wyniki	28
6. Podsumowanie	33

1. Wprowadzenie

Procesory graficzne z wielu punktów widzenia są kompletnie różne od zwykłej jednostki centralnej w każdym komputerze. Mają nienaturalną konstrukcję, skupiającą się na posiadaniu jak największej ilości jednostek arytmetycznych w krzemie. Wynikiem tego jest nieosiągalnie duża dla zwykłych mikroprocesorów surowa moc obliczeniowa. Niecodziennosc struktury tej rodziny procesorów czyni tworzenie programów rozwiązujących dany problem bardziej skompilowane. Spektrum kodu, który zostanie zaakceptowane przez kartę graficzną jest szerokie w porównaniu do zwykłych mikroprocesorów. Normalnie procesor jest w stanie przyjąć jedynie instrukcje opisane i wspierane z danej standardowej architektury przemysłowej (ISA). Procesory graficzne mogą wspierać różne abstrakcje, które pomimo różnic udostępniają te same podstawowe operacje. Umożliwia to ciągle działający sterownik, który tłumaczy daną abstrakcję na odpowiedni dla tej mikro-architektury kod maszynowy. Taką abstrakcją jest język pośredni SPIR-V stworzony na potrzeby obliczeń wielowątkowych i graficznych. Załadowanie i uruchomienie tego kodu umożliwia specyfikacja Vulkan, opisująca zestaw interfejsów pozwalających na kontrolowanie zachowania procesora graficznego.

1.1. Cele pracy

Celem poniższej pracy jest implementacja algorytmów mnożenia macierzy rzadkich przez wektor oraz przedstawienie użytego interfejsu do procesora graficznego. Praca również omawia ogólną mikroarchitekturę procesorów graficznych i uruchamianie shaderów obliczeniowych na nich.

1.2. Zawartość pracy

W rozdziale 2 omówiono problem macierzy rzadkich i formatów do ich przetrzymywania. Następnie w rozdziale 3 przedstawiono ogólną mikroarchitekturę nowoczesnych procesorów graficznych. Opisano użyty interfejs Vulkan do zarządzania procesorem graficznym czym są shadery obliczeniowe. Sposób implementacji komunikacji z procesorem graficznych oraz poszczególnych shaderów obliczeniowych zawarto w rozdziale 4. Uzyskane wyniki używając tej implementacji przeanalizowano w rozdziale 5, a podsumowanie pracy w rozdziale 6.

2. Mnożenie macierz-wektor dla macierzy rzadkich

2.1. Problem mnożenia macierzy poprzez wektor

Operacja taka jest specjalnym przypadkiem mnożenia macierzy przez macierz, gdy liczba kolumn w drugiej macierzy równa jest jeden. Macierz o liczbie rzędów M i liczbie kolumn N zostaje pomnożona przez wektor o liczbie rzędów równej ilości kolumn macierzy. Dla macierzy A o wymiarach $M \times N$ i wektora X o wymiarach $N \times 1$ możemy następująco przedstawić operację mnożenia $Y = AX$. Wynikowy wektor Y będzie posiadał liczbę rzędów równą liczbie rzędów macierzy wejściowej, wymiary wektora to finalnie $M \times 1$.

2.2. Macierze rzadkie

Macierz rzadka jest specjalnym przypadkiem macierzy, w której większość jej elementów jest równa zero. Możemy zaobserwować, że dla elementów zerowych mnożenie zawsze będzie skutkować wynikiem równym zero. Poświęcanie czasu obliczeniowego do ewaluacji iloczynu tych elementów jest zbędne. Ilość pamięci jaka wymagana jest do przetrzymania macierzy rośnie kwadratowo wraz ze wzrostem jej wymiaru. Jeżeli przyjmujemy, że tylko n_z elementów w macierzy jest niezerowych teoretycznie możliwe jest następujące zmniejszenie zużycia pamięci i przyspieszenie obliczeń:

$$R = \frac{MN}{n_z} \quad (2.1)$$

R – krotność zmniejszenia zużycia pamięci oraz krotność przyspieszenia obliczeń

n_z – ilość elementów niezerowych w macierzy

M – liczba rzędów macierzy

N – liczba kolumn macierzy

2.3. Problem mnożenia macierzy rzadkiej poprzez wektor

Istnieje pole problemów, których rozwiązanie zawsze wymaga pracy z macierzami rzadkimi. Przykładem jest metoda elementów skończonych (MES), w której ostateczne określenie wartości w węzłach

jest rozwiązaniem układu równań, w którym macierz jest rzadka. Obliczenia na większych macierzach zazwyczaj dokonywane są poprzez metody iteracyjne. Podstawową operacją w tych metodach jest mnożenie macierzy rzadkiej przez wektor wiele razy, aby otrzymać jak najlepsze przybliżenie.

2.4. Formaty macierzy rzadkich

Problem główny w przechowywaniu macierzy rzadkich to sposób w jaki określony zostaje rząd oraz kolumna dla danego elementu. Samo przechowywanie tylko elementów niezerowych jest elementem wspólnym prawie wszystkich formatów. Występują formaty, które przetrzymują jednak część elementów zerowych, aby uprościć oszacowanie rzędu i kolumny. Zmniejsza to narzut pamięciowy przetrzymywania danych dodatkowych przy tablicy elementów niezerowych.

Formaty omówione to:

- COO (ang. Coordinate Format) - najprostszy format, który wykorzystuje trzy tablice *row*, *col*, *floatdata*. Odpowiednio przechowują one rząd, kolumnę i wartość elementu *i*'tego. Długość tablic równa jest ilości elementów niezerowych.

Reprezentacja wizualna transformacji przykładowej macierzy:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{array}{lcl} \text{floatdata} & = & \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 8 & 6 & 8 \end{bmatrix} \\ \text{row} & = & \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \text{col} & = & \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 1 & 3 \end{bmatrix} \end{array}$$

- CSR (ang. Compressed Sparse Row Format) - format rzędowo kompresuje macierz poprzez zapisywanie wszystkich wartości z danego rzędu bezpośrednio po sobie. Takie podejście umożliwia nieprzechowywanie rzędu elementu wprost, a jedynie indeks początkowego elementu wiersza. Dwie tablice *floatdata* i *columnIndices* są nadal długości równej ilości elementów niezerowych i przechowują one odpowiednio wartość i kolumnę elementu *i*'tego. Trzecia tablica *rowOffsets* ma długość o jeden większą niż ilość rzędów macierzy, co dla macierzy o większych zagęszczeniach pozwala zaoszczędzić na pamięci w porównaniu do formatu COO. Dla numeru rzędu *r* będący wartością z przedziału $[0, N - 1]$ przechowuje ona indeks pierwszego elementu w tablicach *floatdata* i *columnIndices* należącego do tego rzędu. Liczbę elementów niezerowych w danym wierszu można obliczyć jako różnicę pomiędzy indeksem w wierszu $r + 1$ a indeksem w wierszu *r*.

Reprezentacja wizualna transformacji przykładowej macierzy:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{array}{lcl} \text{floatdata} & = & \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 8 & 6 & 8 \end{bmatrix} \\ \text{columnIndices} & = & \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 4 & 1 & 3 \end{bmatrix} \\ \text{rowOffsets} & = & \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \end{array}$$

- CSC (ang. Compressed Sparse Column Format) - format kolumnowo kompresuje macierz w identyczny sposób jak format CSR ze zmianą kierunku kompresji. Niezmiennie macierz w tablicy *floatdata* przechowuje wartość elementu *i*'tego. Format ten zapisuje w pełni rząd elementu *i*'tego w tablicy *rowIndex* o długości równej ilości elementów niezerowych. Nieprzechowywane natomiast są kolumny zastąpione poprzez indeks początkowego elementu kolumny. Zadanie to pełni tablica *columnOffsets* o długości o jeden większa niż ilość kolumn w macierzy. Jeżeli liczba rzędów jest równa liczbie kolumn macierzy format ten będzie wymagał tę samą ilość pamięci, co format CSR. Najczęściej preferowanym formatem jest CSR ze względu na ogólną architekturę wszystkich mikroprocesorów.

Reprezentacja wizualna transformacji przykładowej macierzy:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{array}{lcl} \text{floatdata} & = & \begin{bmatrix} 1 & 5 & 4 & 2 & 6 & 3 & 7 & 8 & 8 \end{bmatrix} \\ \text{rowIndices} & = & \begin{bmatrix} 0 & 2 & 0 & 1 & 3 & 1 & 2 & 3 & 2 \end{bmatrix} \\ \text{columnOffsets} & = & \begin{bmatrix} 0 & 2 & 5 & 6 & 8 & 9 \end{bmatrix} \end{array}$$

- ELL (ang. ELLPACK Format) - format przechowuje podmacierz o wymiarach $M \times P$ gdzie P jest maksymalną ilością elementów niezerowych ze wszystkich wierszy. Jest to pierwszy format, który wybiera przechowywać większą ilość danych niż ilość elementów niezerowych. Jego wydajność wzrasta dla macierzy, w których średnia ilość elementów niezerowych w wierszu jest najbardziej zbliżona do P . Wykorzystuje dwie tablice, *floatdata* i *columnIndices* obie o wymiarach $M \times K$. Przechowują one kolejno wartości elementów i indeks kolumny, w której znajduje się ten element. Lokalizacja wartości należących do danego rzędu jest prosta, ze względu na stałą P . Zaczynając od indeksu rP następne P elementów należy do wiersza o indeksie r . Jeżeli wiersz posiada mniej elementów niezerowych niż P pola niewypełnione wartością lub kolumną są ustawione na specjalną wartość traktowaną jako flaga.

Reprezentacja wizualna transformacji przykładowej macierzy:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{array}{lcl} \text{floatdata} & = & \begin{bmatrix} 1 & 4 & X \\ 2 & 3 & X \\ 5 & 7 & 8 \\ 6 & 8 & X \end{bmatrix} \\ \text{columnIndices} & = & \begin{bmatrix} 0 & 1 & X \\ 1 & 2 & X \\ 0 & 3 & 4 \\ 1 & 3 & X \end{bmatrix} \end{array}$$

- SELL (ang. Sliced ELLPACK Format) - jest modyfikacją formatu ELL, w której wartość P jest obliczana w obrębie paska wierszy o wysokości C . Umożliwia to ograniczenie efektu, w którym wiersz z dużą ilością elementów sztucznie zwiększa ilość potrzebnej pamięci. Takie sporadyczne wiersze będą jedynie afektować rozmiar swojego paska. Najczęściej jednak wartość P pasków będzie zbliżona do wartości średniej ilości elementów niezerowych we wszystkich wierszach. Format wymaga dodatkowej tablicy *rowOffsets* o długości równej ilości pasków, ta może zostać obliczona jako $\frac{M}{C}$ zaokrąglając w górę. Tak samo jak w formacie CSR przetrzymuje ona indeks pierwszego

elementu w dwóch następnych tablicach odpowiadający i 'temu paskowi. Tablice te to *floatdata* i *columnIndices*, pełniące tę samą rolę, co tablice o tej samej nazwie w formacie ELL. Poprzez sterowanie wartością C zmieniamy zachowanie formatu. W przypadku, gdy $C = M$ mamy tylko jeden pasek, zatem format upraszcza się do ELL. Natomiast dla $C = 1$, format zachowuje się identycznie jak CSR.

Reprezentacja wizualna transformacji przykładowej macierzy dla wysokości paska $C = 2$:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{array}{l} \text{floatdata} = \begin{bmatrix} 1 & 4 \\ 2 & 3 \\ 5 & 7 & 8 \\ 6 & 8 & X \end{bmatrix} \\ \text{columnIndices} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 0 & 3 & 4 \\ 1 & 3 & X \end{bmatrix} \\ \text{rowOffsets} = [0 \quad 4 \quad 10] \end{array}$$

- BSR (ang. Block Compressed Sparse Row Format) - format dzielący i przechowujący macierz jako zbiór bloków o wymiarach $B_s \times B_s$. W specjalnym przypadku, gdy $B_s = 1$ format zachowuje się identycznie jak CSR. Dla wartości większych macierz zostaje skompresowana rzędowo przy założeniu, że najmniejszą komórką jest blok $B_s \times B_s$. W tablicy *floatdata* przechowywane są dane bloków następująco po sobie, długość tej tablicy równa jest liczbie niezerowych bloków pomnożona przez B_s^2 . Kolumnę bloku określa tablica *columnIndices* o długości równej liczbie niezerowych bloków. Określenie kolumny w przestrzeni macierzy dla elementu w bloku odbywa się poprzez pomnożenie kolumny bloku przez B_s oraz dodanie kolumny elementu wewnątrz bloku. Pierwszy blok należący do rzędu bloku o indeksie r_b znajduje się pod indeksem wskazanym przez wartość w tablicy *rowOffsets* pod indeksem r_b . Tablica ta ma długość równą $\frac{M}{B_s} + 1$. Obliczenie rzędu w przestrzeni macierzy dla elementu odbywa się podobnie jak dla kolumny, do iloczynu r_b i B_s zostaje dodany rząd elementu wewnątrz bloku. Dla macierzy o wymiarach nie będących wielokrotnością B_s wartości elementów wykraczające poza wymiar macierzy A zostają ustawione na wartość zerową.

Reprezentacja wizualna transformacji przykładowej macierzy dla $B_s = 2$:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 & 0 \\ 0 & 6 & 0 & 8 & 0 & 0 \end{bmatrix} \rightarrow \left[\begin{array}{cc|cc|cc} 1 & 4 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 & 0 \\ \hline 5 & 0 & 0 & 7 & 8 & 0 \\ 0 & 6 & 0 & 8 & 0 & 0 \end{array} \right]$$

$$A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix} \rightarrow \begin{array}{l} A_{00} = \begin{bmatrix} 1 & 4 \\ 0 & 2 \end{bmatrix}, A_{01} = \begin{bmatrix} 0 & 0 \\ 3 & 0 \end{bmatrix}, \\ A_{10} = \begin{bmatrix} 5 & 0 \\ 0 & 6 \end{bmatrix}, A_{11} = \begin{bmatrix} 0 & 7 \\ 0 & 8 \end{bmatrix}, A_{12} = \begin{bmatrix} 8 & 0 \\ 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{lll} \text{floatdata} & = & \begin{bmatrix} A_{00} & A_{01} & A_{10} & A_{11} & A_{12} \end{bmatrix} \\ \text{columnIndices} & = & \begin{bmatrix} 0 & 1 & 0 & 1 & 2 \end{bmatrix} \\ \text{rowOffsets} & = & \begin{bmatrix} 0 & 2 & 5 \end{bmatrix} \end{array}$$

3. Mikro-architektura, interfejs Vulkan i shadery obliczeniowe

3.1. Mikro-architektura nowoczesnych procesorów graficznych

Podjęcie do architektury, którą mikroprocesory graficzne mają implementować i ulepszać z generacji na generację wyewoluowało z wyspecjalizowanego na generyczne. Początkowo ogólną ideą było stworzenie akceleratora posiadającego wyspecjalizowane jednostki, będące odpowiedzialne tylko za ściśle określone operacje. Jednostki podzielono na trzy poziomy, każdy kolejny poziom miał na celu realizację innej operacji: przetwarzanie wierzchołków geometrii, generowanie fragmentów i ich finalne połączenie. Próba odwzorowania teoretycznej abstrakcji grafiki trójwymiarowej w krzemie stworzyła skomplikowany i mało elastyczny mikro-procesor. Przykładowo, jeżeli dana aplikacja chce narysować mało wierzchołków, a jednocześnie wykonać skomplikowane operacje na wygenerowanych fragmentach, nie było możliwości wykorzystania krzemu przeznaczonego do przetwarzania geometrii do wykonywania obliczeń na wierzchołkach. Można podać odwrotny przykład, w którym aplikacja chce wykorzystać wiele wierzchołków natomiast mieć proste obliczenia na wynikowych fragmentach. Wszystko, co wybiegało poza przewidzianą przez twórców architektury normę nie wykorzystywało krzemu w całości mimo tego, że surowa moc obliczeniowa w teorii była dostępna. Twórcy architektury również mieli problemy z optymalnym rozłożeniem ilości krzemu przeznaczonego dla poszczególnych kroków procesu.

Rozwiązaniem było stworzenie ogólnej jednostki, będącej w stanie wykonywać zadane obliczenia bez przywiązania do jakiegokolwiek abstrakcyjnej wizji. Wraz z premierą architektury *Fermi* w mikroprocesorach firmy *NVIDIA*, wprowadzono pojęcie multi-procesora strumieniowego (ang. Stream Multiprocessor), w skrócie *SM*. Posiadał on 8 jednostek wykonawczych (*ALU*), obsługujących podstawowe operacje na 32-bitowych liczbach zmiennoprzecinkowych, 2 jednostki specjalnego przeznaczenia (*SFU*), pozwalające na obliczanie skomplikowanych funkcji, na przykład *sin*, *exp* [1]. Aby dostarczyć dane do rdzeni, *SM* posiada pamięć podręczną instrukcji, pamięć współdzieloną pomiędzy jednostki wykonawcze, pamięć podręczną wartości stałych oraz kolejkę wątków do uruchomienia. Według taksonomii Flynna, architektura *Tesla* zostałaby nazwana *SIMT* (Single Instruction Multiple Threads), czyli pojedynczą instrukcję wykonuje wiele wątków jednocześnie. Zadania przypisywane są do kolejek wolnych *SM* w grupach 32 wątków nazwanych *warp*'ami. Opróżnienie kolejki zajmowało 4 cykle, jeżeli wszystkie instrukcje mogą zostać wykonane na rdzeniach *ALU*. Dla operacji wymagających wykorzystania rdzeni *SFU* wykonanie wszystkich przypisanych wątków trwa 16 cykli.

Najmniejszy logiczny blok adresowalny wewnątrz mikroprocesora, jakim jest *SM*, ma generyczną naturę, która sprawia, że w prosty sposób można zwiększyć moc obliczeniową mikro-procesora poprzez zwiększenie ich ilości znajdujących się w krzemie. Takie podejście pokrywało wszystkie możliwe przypadki, nieważne jak bardzo odbiegające od normy. Całość krzemu jest wykorzystywana, a to, jakie zadanie ma pełnić zostaje dynamicznie określone zależnie od typu pracy. W porównaniu do rdzenia procesora centralnego celem nie jest zwiększenie wydajności jednego wątku. Zmniejsza to potrzebę na implementowanie części spekulacyjnej procesora oraz dużych pamięci podręcznych. Ideą jest jak największa surowa moc obliczeniowa pożądana w takich dziedzinach jak algebra liniowa, metody numeryczne czy grafika komputerowa. Odblokowanie takich możliwości umożliwiło tworzenie o wiele bardziej skomplikowanych symulacji i tchnęło nowe życie w pole sztucznej inteligencji. Kolejne mikro-architektury budowały na koncepcie *SM*, zwiększając ich możliwości, moc oraz ilość dzięki postępom w litografii.

3.2. Interfejs Vulkan

Jednostka centralna komunikuje się z procesorem graficznym przy użyciu abstrakcyjnego interfejsu, który opisuje zestaw procedur oraz ich oczekiwany wynik działania. Sposób implementacji danego interfejsu zależy od sterownika graficznego. Idealny interfejs jest abstrakcyjny w takim stopniu, żeby pozwolić różnym producentom procesorów graficznych na elastyczne implementowanie procedur bez ścisłego powiązania z samą architekturą fizyczną. Dla użytkowników interfejs powinien dostarczać możliwie jak największej kontroli nad tym, co wykonuje procesor graficzny. Przykładowo, jeden z pierwszych interfejsów, *OpenGL* opierał się na wywoływaniu komend, które zmieniały globalną maszynę stanu, a ta następnie była interpretowana przez sterownik graficzny. Wysokopoziomowa abstrakcja miała na celu pozwolenie użytkownikowi na ignorowanie wielu operacji dziejących się bez jego wiedzy. Mimo licznych zalet, rozwiązanie takie miało również swoje wady. Osoby doświadczone nie miały możliwości niskopoziomowej kontroli, przez co pole optymalizacji było ograniczone. W samej specyfikacji istniały miejsca, w których wynik działania danej komendy był niedoprecyzowany. Finalne zachowanie było zależne od implementacji, przez co ten sam program wykonany na procesorach graficznych dwóch różnych producentów mógł wykazywać różne wyniki.

Sfinalizowany w roku 2016 interfejs *Vulkan* [2] ma na celu zastąpienie interfejsu *OpenGL*. Zbudowany został na podstawach interfejsu *Mantle*, który został stworzony oraz następnie przekazany grupie *Khronos* przez firmę *AMD*. W przeciwieństwie do swojego poprzednika, niskopoziomowa abstrakcja umożliwia wykorzystanie procesora graficznego w bardziej generyczny sposób. Aby to zrobić, użytkownikowi zostaje dostarczony zestaw procedur operujących na przesłanym przez niego stanie w poszczególnych obiektach. Odstąpienie od globalnego stanu umożliwiło wielowątkowe sterowanie procesorem graficznym, co przekłada się na zwiększoną wydajność w przypadkach, gdy procesor jest wąskim gardłem w programie. Wydajność zostaje również poprawiona poprzez ominięcie sprawdzania błędów przez sterownik graficzny podczas pracy programu, jest to zadaniem użytkownika, aby dostarczyć do sterownika poprawne dane. Na użytkownika ciąży wiele odpowiedzialności, zostaje mu powierzone zarządzanie

pamięcią oraz synchronizacją procesora graficznego. Wszystko to celem maksymalnego wykorzystania procesora graficznego, aby osiągnąć jak najwyższą wydajność. Doświadczony programista jest w stanie zarządzać dostępnymi mu zasobami na wcześniej niespotykaną skalę dokładności.

Posługiwanie się przez specyfikacje prostymi i kompatybilnymi z architekturą komputerów koncepcjami znacznie redukuje możliwość niesprecyzowania danego aspektu interfejsu. Twórcy sterowników graficznych mogą je znacznie uprościć tym samym redukując ilość błędów i poprawiając wydajność oraz zniesione zostaje rozgraniczenie pomiędzy interfejsem dla mobilnych i konwencjonalnych procesorów graficznych. Poprzednio dla systemów wbudowanych został stworzony interfejs *OpenGL ES*, będący podzbiorem interfejsu *OpenGL* dla komputerów stacjonarnych. Tworzy to sztuczny podział, w którym utrzymanie dwóch różnych systemów wymaga w najgorszym przypadku dwa razy więcej wysiłku. Podział ten nie istnieje dla *Vulkan*, ponieważ od początku celem było zunifikowanie wszystkich urządzeń i zmniejszenie liczby interfejsów do jednego. Dzisiaj ten sam interfejs wspierany jest w komputerach stacjonarnych, urządzeniach mobilnych, systemach wbudowanych i konsolach. Narzędzia stworzone do pracy z aplikacjami wykorzystującymi *Vulkan* mogą zostać wykorzystane we wszystkich typach urządzeń.

Mimo, że grafika komputerowa była głównym wspieranym celem, podczas tworzeniu interfejsu przewidziano wykorzystanie procesora graficznego do innych zadań. Rozwijające się pole sztucznej inteligencji zaczęło polegać na mocy obliczeniowej procesorów graficznych do budowania coraz to większych modeli uzyskujących coraz to lepsze wyniki. *Vulkan* przewiduje możliwość wykorzystania procesora graficznego do obliczeń naukowych, osiąga to poprzez tworzenie różnych typów *pipeline*ów. Obok *pipeline*’u graficznego istnieje *pipeline* obliczeniowy, który pozwala na uruchomienie wybranych shaderów obliczeniowych.

3.3. Shadery obliczeniowe

Shader to ogólnie przyjęta nazwa na program stworzony przez użytkownika, który ostatecznie zostanie uruchomiony na procesorze graficznym. Shaderem obliczeniowym jest program wielowątkowy, działający w modelu SIMT dokonujący arbitralnych obliczeń. Jednostka centralna żąda wywołania pewnej ilości grup shaderów, a ich ilość jest określona jako trójwymiarowa przestrzeń $X \times Y \times Z$. Ułatwia to wykonywanie obliczeń na problemach z natury wielowymiarowych. Przykładowo, ustawiając wymiar $Z = 1$ zostanie uruchomiona dwuwymiarowa grupa, którą można wykorzystać przy algorytmach działających na obrazach lub automatach komórkowych. Łączna liczba n_{SC} wszystkich uruchomionych shaderów obliczeniowych w danej inwokacji może zostać określona jako iloczyn wszystkich wymiarów $n_{SC} = XYZ$. Należy jednak rozgraniczyć grupę i pojedyncze wywołanie shadera. Ilość wywołań shaderów w pojedynczej grupie jest definiowane przez sam shader jako lokalny rozmiar shadera.

Wszystkie wartości wejściowe i wyjściowe shadera są zdefiniowane przez użytkownika. Shadery mają dostęp do swojego identyfikatora grupy oraz lokalnego identyfikatora wewnątrz tej grupy. Wywołania shader’a z tej samej grupy współdzielą identyfikator grupy, natomiast każdy z nich dostanie

unikatowy lokalny identyfikator. Na podstawie tych identyfikatorów shader może określić, na jakich danych ma operować. Dane do i z shaderów są przenoszone poprzez sampler tekstur lub Shader Storage Blocks (*SSBO*). *SSBO* są buforami przechowującymi dane w sekwencyjny sposób, których rozmiar może być dynamiczny, mają one niespójny model pamięci, to znaczy, że dane zapisane przez jeden wątek nie muszą być od razu widoczne przez drugi wątek. To samo zachodzi z zapisem, nie ma gwarancji, że dane zapisane przez jeden wątek nie zostaną nadpisane przez inny wątek przed trafieniem do pamięci głównej. Aby rozwiązać sytuację, w której więcej niż jeden wątek musi zapisać dane do tej samej komórki pamięci *SSBO* wspierają atomiczne operacje na pamięci.

Vulkan oczekuje, że wszystkie shadery będą przekazane do niego w formacie *SPIR-V*[3], jest to język pośredni przechowywany w formacie binarnym, stworzony na potrzeby obliczeń wielowątkowych i graficznych. Użycie języka pośredniego pozwala na tworzenie shaderów w różnych językach programowania, które należy jedynie sprowadzić do formy zgodnej ze specyfikacją *SPIR-V*. Językiem najczęściej wykorzystywanym jako wysokopoziomowa abstrakcja jest *GLSL* używany wcześniej w *OpenGL*. Wykorzystanie formatu binarnego w przeciwieństwie do *OpenGL*, gdzie cały kod *GLSL* był przechowywany jako ciąg czytelnych znaków, posiada wiele zalet. Największą z nich jest uproszczenie sterownika graficznego, który nie musi implementować całego kompilatora języka o podobnym stopniu skomplikowania, co język C. Konwersja odbywająca się w sterowniku jest prostsza i o wiele bardziej wydajna, ponieważ zbędna praca została wykonana wcześniej podczas konwersji na format *SPIR-V*. Zmniejsza to czas potrzebny na przygotowanie *pipeline*'u, przez co aplikacje mają większą swobodę w wykorzystywaniu większej ilości shaderów. Format binarny zajmuje również mniej miejsca na dysku oraz utrudnia inżynierię wsteczną własnościowych shaderów w publicznych aplikacjach.

4. Implementacja

W tym rozdziale opisana jest implementacja uruchomienia algorytmów mnożenia macierzy rzadkiej przez wektor na procesorze graficznym. Pierwszy podrozdział omawia opracowany sposób na komunikację z procesorem graficznym przy wykorzystaniu interfejsu *Vulkan*. Drugi przedstawia sposób przechowywania danego formatu macierzy rzadkiej, konwersji na ten format oraz shader obliczeniowy, którego zadaniem jest obliczenie wyniku mnożenia macierz - wektor dla tego formatu.

4.1. Komunikacja z procesorem graficznym

Aby rozpocząć komunikację z procesorem graficznym należy stworzyć obiekty skonfigurowane w taki sposób, w jaki zamierzamy ich używać. Inicjalizacja stanu początkowego w interfejsie Vulkan sprowadza się do stworzenia instancji, wybrania fizycznego urządzenia oraz na jego podstawie stworzenia urządzenia logicznego, stworzenia puli zasobów, z których korzystać będą obiekty komend i kwerend. Podczas tworzenia instancji użytkownik ma możliwość podania tablicy z warstwami, które mają zostać włączone oraz rozszerzeń, które mają zostać dodane do danej instancji. Warstwy to rozwiązanie problemu debugowania programów napisanych przy użyciu interfejsu *Vulkan*. Ponieważ sterownik graficzny nie sprawdza poprawności przesłanych do niego danych znajdowanie źródeł błędów jest o wiele trudniejsze. Rozwiązaniem tego problemu są warstwy, czyli dodatkowy kod wykonywany dookoła wywołania funkcji. Takie podejście można opisać jako wzorec projektowy dekorator. Istnieje wiele warstw, które pełnią różne funkcje, przykładowo dodając warstwę zapisującą kiedy dana funkcja została wywołana możemy stworzyć graf wywołań w danym programie. Najbardziej pomocną warstwą podczas tworzenia aplikacji od nowa jest warstwa walidująca. Będzie ona sprawdzać poprawność przesłanych danych względem tego, co mówi specyfikacja. Niezgodności z nią są wypisane na terminalu, towarzyszy temu wytłumaczenie co, dokładnie jest niezgodne ze specyfikacją. Warstwa walidująca upewnia się, że program będzie działał tak samo niezależnie od wersji sterownika lub typu procesora graficznego. Niezgodność nie oznacza, że dany kod będzie skutkował błędnym wynikiem, natomiast nie ma gwarancji, że wynik ten nie zmieni się jeżeli domyślny stan będzie się różnił w innym środowisku pracy.

Wybór fizycznego urządzenia odbywa się poprzez wysłanie zapytania do interfejsu o enumerację wszystkich obecnych urządzeń w tym systemie. Użytkownik następnie dokonuje wyboru, który procesor graficzny spełnia jego wymagania. Może dokonywać zapytań o to, jakie funkcjonalności wspiera dane urządzenie oraz jakie są jego specyfikacje. Nie można oczekiwać, że w systemie zawsze będzie

znajdować się tylko jeden procesor graficzny. Przykładami systemów posiadających więcej niż jeden procesor graficzny mogą być laptopy z procesorem graficznym dedykowanym oraz zintegrowanym z jednostką centralną lub serwery, na których często znajduje się wiele takich samych procesorów graficznych współpracujących ze sobą.

Na podstawie urządzenia fizycznego tworzone jest urządzenie logiczne. Wymaga ono opisanie z jakich kolejek będzie korzystać program. Przykładowo, program graficzny będzie korzystać z kolejki wspierającej operacje graficzne, natomiast program uruchamiający tylko i wyłącznie shadery obliczeniowe będzie korzystać z kolejki wspierającej arbitralne obliczenia. Następnie użytkownik ma możliwość włączenia danych specjalnych funkcji procesora graficznego, przykładowo obsługa typu zmienoprzecinkowego podwójnej precyzji, który domyślnie jest wyłączona. Dodatkowo, istnieje możliwość zdefiniowania rozszerzeń, które mają zostać włączone. Rozszerzeniami są nowe niezdefiniowane przez specyfikację komendy, struktury i wartości enumeracyjne. Przykładem jest rozszerzenie `VK_NV_COOPERATIVE_MATRIX`, pozwalające użytkownikowi na wykorzystanie rdzeni tensorowych, znajdujących się tylko na urządzeniach najnowszych generacji od NVIDIA. Dodanie takiej funkcjonalności nie może być dodane przez specyfikację, ponieważ nie wszystkie procesory graficzne będą wspierać to rozszerzenie. Mimo wszystko posiadanie rozszerzeń do urządzenia pozwala interfejsowi *Vulkan* być bardziej elastycznym w wspieraniu wielu pofragmentowanych rozwiązań. Po skonfigurowaniu tych opcji może zostać stworzone urządzenie logiczne, przy jego pomocy będzie wykonywana cała komunikacja z procesorem graficznym. Jako ostatni pobierany jest logiczny odnośnik do kolejki, na którą będzie wysyłana praca do wykonania przez procesor graficzny. Ostatni krok w ogólnej inicjalizacji to stworzenie dwóch pul pamięci dla komend i kwerend. Podejście takie pozwala na posiadanie tylko jednego określonego miejsca, w którym znajduje się pamięć danej rzeczy, tym samym zwiększając lokalność pamięci. Stworzenie puli dla komend wymaga podania indeksu kolejki na urządzeniu fizycznym, do której będą wysyłane komendy zaalokowane w tej puli. Pula dla kwerend natomiast wymaga logicznego urządzenia, na którym będą wykonywane kwerendy, ich typ oraz maksymalna ilość.

Na podstawie tego globalnego stanu zostają utworzone dodatkowe obiekty, będące odpowiedzialne za uruchomienie poszczególnych shaderów obliczeniowych dla danego formatu przechowywania macierzy. Dla każdego formatu są to te same obiekty tworzone w tej samej kolejności, natomiast różnią się na przykład rozmiarem lub ilością. Zbiór wszystkich obiektów potrzebnych do komunikacji z procesorem graficznym celem wywołania poszczególnego shadera dla wybranego typu macierzy nie będących stanem globalnym od tego miejsca nazwano scenariuszem. Każdy scenariusz zawiera następujące obiekty: pary buforów i pamięci, pulę deskryptorów, ich ułożenie oraz zbiór, definicję *pipeline*'u oraz bufor komend.

W zastosowanym rozwiązaniu każdy bufor przypisany jest do unikatowego obiektu pamięci, z tego względu zdecydowano zgrupować bufor i pamięć razem. W uproszczeniu, bufor jest wglądem do pamięci w zdefiniowanym zakresie, natomiast pamięć jest obiektem przy pomocy którego zarządzamy czasem życia danej alokacji pamięci. Takie rozwiązanie pozwala na stworzenie wielu różnych zmieniających się wglądów do zaalokowanej pamięci bez potrzeby jej ciągłej realokacji. Grupy bufora i pamięci zawsze

były tworzone w parze, jedna reprezentowała pamięć, do której jednostka centralna ma dostęp natomiast druga reprezentowała pamięć lokalną dla procesora graficznego. Jest to wymagane, ponieważ jednostka centralna nie może bezpośrednio dokonywać arbitralnych operacji na lokalnej pamięci procesora graficznego. Bufor wykorzystywany do przepisywania pamięci z jednostki centralnej do pamięci procesora graficznego nazywa się buforem przestojowym (ang. staging buffer). Zanim bufor zostanie stworzony należy określić jego rozmiar, sposób współdzielenia pomiędzy kolejkami i to, w jaki sposób będzie wykorzystywany, np. dla bufora przestojowego wykorzystywana jest możliwość przechowywania danych oraz bycie źródłem podczas transferu. Po stworzeniu bufora należy dokonać kwerendy celem określenia wyrównanego rozmiaru pamięci oraz jakie typy pamięci, które posiada urządzenie będą wspierać wszystkie zdefiniowane sposoby wykorzystywania. Zaalokowanie pamięci wymaga odpytania urządzenia jakie posiada typy pamięci oraz w jakich kopcach się znajdują. Wybór typu pamięci sprowadza się do tego, czy istnieje pamięć będąca w stanie obsłużyć wszystkie wymagania bufora i czy jest wystarczająco blisko pamięci samego rdzenia mikro-procesora graficznego. Po wskazaniu wyrównanej ilości pamięci do zaalokowania oraz który typ pamięci ma zostać wykorzystany zostaje stworzony obiekt pamięci. Ten, finalnie może zostać przypisany do bufora. Operacje na pamięci przez jednostkę centralną w buforach wspierających tę operację dokonywane są poprzez zmapowanie tej pamięci do przestrzeni adresowej danego programu. Po tym, procesor ma dostęp do wszystkich bajtów w sposób równoważny pamięci zaalokowanej na przykład przez funkcję *malloc*. Program ma obowiązek usunąć mapowanie pamięci przed wykorzystaniem jej do komunikacji z procesorem graficznym. Po przesłaniu wszystkich danych do buforów przestojowych następuje wykonanie komendy kopiowania danych z bufora do bufora.

Zaprzęgnięcie procesora graficznego do faktycznego wykonywania pracy wymaga wysłania bufora z nagranyimi komendami na daną kolejkę. Aby nagrać komendy potrzebne jest miejsce, które będzie zapisywać liniowy ciąg wywołanych komend, jest nim bufor komend, a do jego alokacji należy wskazać na pulę komend, która będzie zarządzać stworzonymi buforami komend. Taki bufor komend jest rozpoczynany i od tego momentu wszystkie wywołane procedury będące komendą są liniowo dodawane do tego bufora komend. Dla kopiowania danych bufora pamięci do drugiego bufora pamięci wywoływana jest komenda kopiowania buforów wskazująca, że należy przekopiować cały region, po tym zakończone jest nagrywanie bufora komend. Ten zostaje wysłany na kolejkę, która ma go wykonać, w przypadku implementowanego programu jest to kolejka obliczeniowa. Celem tworzonego programu jest zachowanie synchroniczności podczas komunikacji z procesorem graficznym, więc zaraz po zakolejkowaniu operacji program czeka, aż kolejka będzie bezczynna, co będzie wskazywać na wykonanie bufora komend. Dla kopiowania buforów pamięci bufor komend zostaje od razu zwolniony po jego wykonaniu.

Pula deskryptorów pełni taką samą rolę jak pula komend i kwerend, alokacje deskryptorów są lokalizowane do danej puli ułatwiając zarządzanie. Deskryptor reprezentuje zasób dostępny shaderowi, na przykład bufor lub sampler tekstur. Zdefiniowanie deskryptora wymaga stworzenia obiektu zestawu deskryptorów, a to z kolei wymaga opisanie układu deskryptorów w zestawie poprzez stworzenie obiektu układu zestawu deskryptorów. Podczas jego tworzenia określamy indeks w zestawie, pod którym ten

deskryptor będzie dostępny, jego typ oraz ilość, a gdy ta jest większa niż jeden umożliwia to stworzenie tablicy obiektów, na przykład takich jak samplery tekstur w shaderze. Na podstawie jednego lub większej ilości układów zestawu deskryptorów zostaje stworzony obiekt zestawu deskryptorów, który obecnie jest tylko sposobem interpretacji podłączonych zasobów, natomiast nie posiada żadnych zasobów powiązanych ze sobą. Przypisanie buforów odbywa się poprzez aktualizację zestawu deskryptorów, wymagane informacje to: który zestaw deskryptorów jest modyfikowany, który indeks w tym zestawie oraz ilość deskryptorów w tym indeksie, jakie należy zaktualizować. Sam zasób jest określany przez jeden z trzech różnych wskaźników na obrazy, bufor lub określony zakres bufora, dla buforów określany jest identyfikator bufora, przesunięcie względem początku oraz rozmiar, jaki ma być zmapowany. Ilość deskryptorów określa ilość elementów znajdujących się pod jednym z tych trzech wskaźników, to który z nich będzie wybrany określa typ deskryptora.

Tak stworzony i zaktualizowany z przypisanymi buforami zestaw deskryptorów może następnie być wykorzystany podczas tworzenia *pipeline*'u obliczeniowego. Będzie on uruchamiał dany shader, więc stworzony zostaje moduł shadera poprzez wykorzystanie danych binarnych w formacie SPIR-V. Następnie stworzony zostaje układ *pipeline*'u, który wymaga podania zestawu deskryptorów oraz opcjonalnie zestawu stałych, które mają zostać przesłane do shadera. Ostatnimi informacjami do określenia to struktura opisująca etap *pipeline*'u, posiada ona informacje o shaderze, takie jak jego moduł, nazwa funkcji wejściowej i typ etapu, w tym przypadku jest to etap obliczeniowy. Przy pomocy tych informacji zostaje stworzony *pipeline* obliczeniowy, który hermetyzuje sposób uruchomienia danego shadera i informacji z nim związanych.

Aby go uruchomić należy stworzyć nowy bufor komend, rozpocząć jego nagrywanie i wywołać następujące komendy: przypisanie *pipeline*'u, przypisanie zestawu deskryptorów, zrestartowanie puli kwerend, wpisanie obecnego znacznika czasowego do puli kwerend, wywołanie $X \times Y \times Z$ grup inwokujący shader z tego *pipeline*'u oraz wpisanego drugiego znacznika czasowego po zakończonych inwokacjach shadera, po czym nagrywanie bufora zostaje zakończone. Tak nagrany bufor komend zostaje przesłany na kolejkę obliczeniową urządzenia celem jego wykonania, gdy kolejka przejdzie w stan bezczynności wszystkie komendy zostały wykonane. Przy użyciu obiektu puli kwerend pobierane zostają dwa znaczniki czasowe, a różnica pomiędzy nimi opisuje czas potrzebny na wykonanie tego shadera dla użytych danych.

4.2. Implementacja mnożenia macierz - wektor dla każdego z formatów

Rozdział ten przedstawia sposób przechowywania w kodzie każdej z macierzy. Funkcję konwertującą na dany format oraz shader, który wykorzystuje ten format do obliczenia iloczynu macierz-wektor.

4.2.1. Format COO

Kod 1 przedstawia strukturę będącą odpowiedzialną za przechowywanie danych potrzebnych do opisanie macierzy rzadkiej w formacie COO.

```
typedef struct MatrixCOO
{
    uint32_t M, N, elementNum; // wymiar M x N i ilość elementów niezerowych
    float *floatdata;          // tablica wartości elementów
    uint32_t *row, *col;       // odpowiednio tablice rzędów i kolumn elementów
} MatrixCOO;
```

Listing 1. Struktura definiująca macierz w formacie COO

Konwersja na ten format macierzy odbywała się w sposób inny niż wszystkie pozostałe formaty. Dane potrzebne do wypełnienia tej macierzy pochodziły z pliku w formacie *MatrixMarket* [4]. Każda inna konwersja opierała się na danych z macierzy rzadkiej przechowywanej w innym formacie. Przetwarzanie danych z pliku opiera się na dużej ilości procesowania tekstu, więc dla ścisłości pominięto dokładny sposób implementacji części mniej krytycznych, w ich miejscu widnieje komentarz opisujący logiczny krok.

Pliki w formacie *MatrixMarket* na samym początku posiadają nagłówek. Widnieją w nim informacje o genezie macierzy, jej autorach oraz odwołania do pracy, z której pochodzi. Poza tym, nagłówek posiada flagi opisujące typ macierzy. Przykładowo flaga *symmetric* informuje użytkownika o tym, że macierz jest symetryczna względem przekątnej. W pliku zostają podane tylko wartości przekątnej oraz z nad przekątną dla zaoszczędzenia miejsca. Podczas wczytywania danych program musi odbić wartości z nad przekątną pod nią, aby poprawnie wczytać macierz. Pierwszy wiersz nie będący nagłówkiem posiada zapisane wartości wymiaru macierzy oraz liczbę elementów niezerowych. Każdy następny wiersz składa się z kolumny, rzędu oraz opcjonalnie wartości elementu. Niepodanie wartości elementu skutkuje zapisaniem wartości równej jeden w to miejsce. Ponieważ rzędy oraz kolumny podawane są zaczynając od jedynki, zmniejszono ich wartości o jeden, aby uzyskać indeksy zaczynające się od zera.

Funkcja 2 ustawia flagę *isSymmetric* zależnie od danych, które odczyta w nagłówku. Ustawia wymiar macierzy poprzez zamianę tekstu na liczbę funkcją *atoi*. Ustala ilość elementów niezerowych biorąc pod uwagę, że format podaje ich ilość w pliku zamiast w finalnej macierzy. Na podstawie tej wartości określona zostaje liczba bajtów potrzebna do przechowania jednej z tablicy. Następnie pamięć ta jest alokowana i przypisywana do wszystkich trzech tablic. W pętli przechodząc przez wszystkie pozostałe linie odczytywana i zapisana jest wartość elementu jego rząd oraz kolumna. Jeżeli element nie znajduje się na przekątnej, wartość zostaje odbita pod nią.

Shader ukazany w kodzie 3 oblicza wynik mnożenia macierzy w formacie COO z wektorem. Przyjmuje cztery bufory wejściowe i jeden wyjściowy. Pierwsze trzy opisują nagłówek macierzy, tablicę wartości, kolumn i rzędów elementów. Czwarta jest tablicą traktowaną jako wejściowy wektor, natomiast

```

MatrixCOO ReadMatrixFormatToCOO(const char *filename) {
    MatrixCOO result = { 0 };

    // Wczytaj plik do pamięci, przetwórz nagłówek celem wyciągnięcia potrzebnych flag
    // i pomiń komenatrze...

    result.M = atoi(getRowNumString());
    result.N = atoi(getColumnNumString());
    uint32_t k = isSymmetric ? 2 : 1; // dla macierzy symetrycznych względem przekątnej
    uint32_t elementNum = atoi(getElementNumString());
    result.elementNum = elementNum * k - (k - 1) * result.N;
    uint32_t toAllocate = result.elementNum * sizeof(result.floatdata[0]);
    result.floatdata = malloc(toAllocate);
    result.row = malloc(toAllocate);
    result.col = malloc(toAllocate);

    uint32_t elementIndex = 0;
    while((line = NextInSplit(&lineIter)).bytes != NULL) // pętla po wszystkich liniach
    {
        uint32_t row = atoi(getElementRowString());
        uint32_t col = atoi(getElementColumnString());
        char *valueString = getElementValueString();
        float value = strlen(valueString) == 0 ? 1.0f : atof(valueString);
        result.row[elementIndex] = row - 1;
        result.col[elementIndex] = col - 1;
        result.floatdata[elementIndex] = value;
        elementIndex += 1;

        if(isSymmetric && col != row) {
            result.row[elementIndex] = col;
            result.col[elementIndex] = row;
            result.floatdata[elementIndex] = value;
            elementIndex += 1;
        }
    }

    return result;
}

```

Listing 2. Funkcja tworząca macierz COO z pliku formatu MatrixMarket

piąta jest wektorem wyjściowym. Każda inwokacja shadera określa swój globalny identyfikator jako dostarczona przez shader zmienna *gl_GlobalInvocationID.x*. Zmienna ta określona na podstawie identyfikatora grupy, lokalnego identyfikatora wątku wewnątrz grupy oraz rozmiaru lokalnej grupy. Ta ustawiona jest na wartość równą 32, aby być kompatybilna z rozmiarem *warp*'u, czyli najmniejszej jednostki przydzielenia pracy do *SM*. Globalny identyfikator służy jako indeks do określenia jaką pracę ma wykonać każdy z wątków. Jeżeli indeks jest większy niż liczba wszystkich elementów wątek ten nie wykonuje

```

#version 450
#extension GL_EXT_shader_atomic_float: enable

#define WORKGROUP_SIZE 32
layout (local_size_x = WORKGROUP_SIZE, local_size_y = 1) in;

layout(set = 0, binding = 0) buffer bufA {
    uint elementNum, M, N;
    float data[];
};

layout(set = 0, binding = 1) buffer bufARow { uint rows[]; };
layout(set = 0, binding = 2) buffer bufACol { uint cols[]; };
layout(set = 0, binding = 3) buffer bufInVec { float inVec[]; };
layout(set = 0, binding = 4) buffer bufOutVec { float outVec[]; };

void main() {
    const uint i = gl_GlobalInvocationID.x;

    if(i < elementNum) {
        const uint row = rows[i];
        const uint col = cols[i];

        float prod = data[i] * inVec[col];
        atomicAdd(outVec[row], prod);
    }
}

```

Listing 3. Shader obliczeniowy wykorzystujący macierz COO

żadnej pracy. W przeciwnym przypadku, wątek pobiera rząd oraz kolumnę elementu znajdującego się pod tym indeksem. Określa iloczyn wartości elementu z wartością w wejściowym wektorze znajdującym się pod indeksem kolumny elementu. Następnie atomicznie dodaje go do wartości wektora wyjściowego znajdującej się pod indeksem rzędu elementu. Atomiczna operacja wymagana jest ze względu na fakt, iż wątek ten nie ma wyłącznego dostępu do zapisu w tej komórce. Inne wątki w tym samym czasie mogą chcieć zapisać do tego samego miejsca co skutkuje wyścigiem danych.

4.2.2. Format CSR

Kod 4 przedstawia strukturę będącą odpowiedzialną za przechowywanie danych potrzebnych do opisanie macierzy rzadkiej w formacie CSR.

Kod 5 przedstawia funkcję, która tworzy macierz CSR wykorzystując do tego wejściową macierz ELL. Przepisuje ona bezpośrednio wymiar oraz liczbę elementów niezerowych. Alokuje pamięć do każdej z trzech tablicy. Pierwsza i druga przechowuje tyle samo elementów; liczbę niezerowych elementów w macierzy. Tablica *rowOffsets* wymaga jest $M + 1$ elementów do przechowania przesunięcia do

```
typedef struct MatrixCSR
{
    uint32_t M, N, elementNum; // wymiar M x N i ilość elementów niezerowych
    float *floatdata;          // tablica wartości elementów
    uint32_t *columnIndices;    // tablica kolumn elementów
    uint32_t *rowOffsets;       // tablica pierwszego indeksu elementu w rzędzie
} MatrixCSR;
```

Listing 4. Struktura definiująca macierz w formacie CSR

dwóch poprzednich tablic. Zwiększanie długości o jeden odbywa się celem uproszczenia obliczania ilości elementów niezerowych znajdujących się w danym rzędzie. Iterując przez wszystkie rzędy macierzy wejściowej funkcja używa dwóch zmiennych: *head* do przechowania miejsca zapisu kolejnego elementu oraz *rowHead* celem przechowania miejsca zapisu kolejnego przesunięcia. Dla każdego rzędu iterowane zostają wszystkie kolumny, dopóki wartość w nich nie wskazuje na koniec danych w tym rzędzie. Przepisana zostaje wartość elementu i jej kolumna z macierzy ELL, zmienna *head* jest inkrementowana, aby następnym razem zapisać do nowej komórki. Po wyjściu z tej pętli zmienna *p* równa jest liczbie kolumn posiadających dane w tym rzędzie. Zostaje ona dodana do poprzedniej wartości przesunięcia, aby utworzyć nową wartość zapisaną w *rowOffsets*.

Shader ukazany w kodzie 6 oblicza wynik mnożenia macierzy w formacie CSR z wektorem. Bufery wejściowe są takie same jak w przypadku macierzy COO. Globalny identyfikator traktowany jest jako indeks rzędu, na którym mają zostać wykonane obliczenia. Jeżeli indeks jest większy niż liczba wszystkich rzędów wątek ten nie wykonuje żadnej pracy. W przeciwnym wypadku pobiera przesunięcie do tablic *floatdata* i *columnIndex*, pod którym zaczynają się elementy dla tego rzędu. Na podstawie tablicy *rowOffsets* ustala również ilość elementów niezerowych w tym rzędzie jako różnicę przesunięcia następnego rzędu a rzędu obecnego. Fakt, iż długość tablicy *rowOffsets* jest o jeden większa niż liczba rzędów, a ostatni element równy jest liczbie wszystkich elementów niezerowych pozwala na uniknięcie potrzeby wykorzystania specjalnego wyjątku, w którym dla ostatniego rzędu wykorzystywana jest wartość *elementNum*. Takie uproszczenie kodu pozwala mu na wyższą wydajność. Shader następnie, iterując po wszystkich niezerowych elementach, do lokalnej zmiennej *sum* sumuje wszystkie iloczyny wartości elementów występujących w tym rzędzie. Ponieważ każdy wątek odpowiedzialny jest za unikatowy rząd, miejsce w pamięci wyjściowej jest zapisywane tylko przez jeden wątek. Usuwa to potrzebę bazowania na operacjach atomicznych co przyspiesza wykonanie shadera.

4.2.3. Format CSC

Kod 7 przedstawia strukturę będącą odpowiedzialną za przechowywanie danych potrzebnych do opisanie macierzy rzadkiej w formacie CSC.

```

MatrixCSR ELLToMatrixCSR(MatrixELL matrix) {
    MatrixCSR result = { 0 };

    result.M          = matrix.M;
    result.N          = matrix.N;
    result.elementNum = matrix.elementNum;

    result.floatdata   = malloc(result.elementNum * sizeof(result.floatdata[0]));
    result.columnIndices = malloc(result.elementNum * sizeof(uint32_t));
    result.rowOffsets   = malloc((result.M+1) * sizeof(uint32_t));
    result.rowOffsets[0] = 0;

    uint32_t head = 0;
    uint32_t rowHead = 1;
    for(uint32_t row = 0; row < matrix.M; row++)
    {
        uint32_t rowOffset = row * matrix.P;
        uint32_t p = 0;
        for(; p < matrix.P && matrix.columnIndices[rowOffset + p] != INVALID_COLUMN; p++) {
            result.floatdata[head] = matrix.floatdata[rowOffset + p];
            result.columnIndices[head] = matrix.columnIndices[rowOffset + p];
            head += 1;
        }

        result.rowOffsets[rowHead] = result.rowOffsets[rowHead - 1] + p;
        rowHead += 1;
    }
    return result;
}

```

Listing 5. Funkcja tworząca macierz CSR na podstawie macierzy ELL

Kod 8 przedstawia funkcję, która tworzy macierz CSC wykorzystując do tego wejściową macierz ELL. Funkcja miejscami analogiczna jest z konwersją macierzy CSR z uwzględnieniem zamiany rzędów z kolumnami. Przy pomocy tablic *rowIndices* i *colFront* tablica *columnIndices* macierzy ELL zostaje przekształcona celem zmiany kierunku przechowywania danych z rzędowego na kolumnowy. W następnym kroku dla każdej kolumny wewnętrzna pętla przechodzi przez wszystkie wiersze w tej kolumnie tak długo jak indeks rzędu nie wskazuje na brak danych. Wiedząc, w którym rzędzie znajduje się następny element tej kolumny wyszukany zostaje indeks *pp*, pod którym wartość *columnIndices* równa się obecnej kolumnie iteracyjnej. Przy pomocy tego indeksu wyciągnięta zostaje wartość elementu, która przypisywana jest do wyjściowej macierzy razem z indeksem rzędu. Po przejściu przez wewnętrzną pętlę zostaje zapisane przesunięcie do tej kolumny w ten sam sposób co konwersji macierzy CSR.

Shader ukazany w kodzie 9 oblicza wynik mnożenia macierzy w formacie CSC z wektorem. Bufory wejściowe są takie same jak w przypadku macierzy COO. Globalny identyfikator traktowany jest jako indeks kolumny, na którym mają zostać wykonane obliczenia. Jeżeli indeks jest większy niż liczba

```

#version 450

#define WORKGROUP_SIZE 32
layout (local_size_x = WORKGROUP_SIZE, local_size_y = 1) in;

layout(set = 0, binding = 0) buffer bufA {
    uint elementNum, M, N;
    float floatdata[];
};

layout(set = 0, binding = 1) buffer bufAColumnIndex { uint columnIndex[]; };
layout(set = 0, binding = 2) buffer bufARowOffsets { uint rowOffsets[]; };
layout(set = 0, binding = 3) buffer inputVector { float inVec[]; };
layout(set = 0, binding = 4) buffer outputVector { float outVec[]; };

void main()
{
    const uint rowi = gl_GlobalInvocationID.x;
    if(rowi < M) {
        const uint rowOffset = rowOffsets[rowi];
        const uint nzCount = rowOffsets[rowi+1] - rowOffset;

        float sum = 0.0;
        for(uint coli = 0; coli < nzCount; coli++)
        {
            const uint cellOffset = rowOffset + coli;
            sum += inVec[columnIndex[cellOffset]] * floatdata[cellOffset];
        }
        outVec[rowi] = sum;
    }
}

```

Listing 6. Shader obliczeniowy wykorzystujący macierz CSR

```

typedef struct MatrixCSC
{
    u32 M, N, elementNum; // wymiar M x N i ilość elementów niezerowych
    float *floatdata; // tablica wartości elementów
    u32 *rowIndices; // tablica rzędów elementów
    u32 *columnOffsets; // tablica pierwszego indeksu elementu w kolumnie
} MatrixCSC;

```

Listing 7. Struktura definiująca macierz w formacie CSC

wszystkich rzędów kolumn ten nie wykonuje żadnej pracy. W przeciwnym wypadku pobiera przesunięcie do tablic *floatdata* i *rowIndex*, pod którym zaczynają się elementy dla tej kolumny. Na podstawie tablicy *columnOffsets* ustala również ilość elementów niezerowych w tej kolumnie jako różnica przesunięcia następnej kolumny a kolumny obecnej. Shader następnie, iterując po wszystkich niezerowych

```

MatrixCSC ELLToMatrixCSC(MatrixELL matrix) {
    MatrixCSC result = { 0 };

    result.M = matrix.M;
    result.N = matrix.N;
    result.elementNum = matrix.elementNum;
    result.floatdata = malloc(result.elementNum * sizeof(result.floatdata[0]));
    result.rowIndices = malloc(result.elementNum * sizeof(uint32_t));
    result.columnOffsets = malloc((result.N+1) * sizeof(uint32_t));
    result.columnOffsets[0] = 0;

    uint32_t *rowIndices = malloc(matrix.N * matrix.P * sizeof(uint32_t));
    uint32_t *colFront = calloc(1, matrix.N * sizeof(uint32_t));
    memset(rowIndices, INVALID_COLUMN, matrix.N * matrix.P * sizeof(uint32_t));
    for(uint32_t row = 0; row < matrix.M; row++) {
        for(uint32_t p = 0; p < matrix.P; p++) {
            uint32_t colIndex = matrix.columnIndices[row * matrix.P + p];
            if(colIndex != INVALID_COLUMN) {
                rowIndices[colFront[colIndex] + colIndex * matrix.P] = row;
                colFront[colIndex] += 1;
            }
        }
    }
    free(colFront);

    uint32_t head = 0;
    uint32_t colHead = 1;
    for(uint32_t col = 0; col < matrix.N; col++) {
        uint32_t p = 0;
        for(; p < matrix.P && rowIndices[col * matrix.P + p] != INVALID_COLUMN; p++) {
            uint32_t ri = rowIndices[col * matrix.P + p];
            uint32_t pp = 0;
            for(; (pp < matrix.P) && (matrix.columnIndices[ri * matrix.P + pp] != col); pp++) {}
            result.floatdata[head] = matrix.floatdata[ri * matrix.P + pp];
            result.rowIndices[head] = ri;
            head += 1;
        }

        result.columnOffsets[colHead] = result.columnOffsets[colHead - 1] + p;
        colHead += 1;
    }
    free(rowIndices);

    return result;
}

```

Listing 8. Funkcja tworząca macierz CSC na podstawie macierzy ELL

elementach, oblicza produkt pomiędzy wektorem wejściowym a danym elementem macierzy w tej kolumnie. Ze względu na przechodzenie po kolumnach zamiast po rzędach dana inwokacja shadera nie gwarantuje wyłączości do danej komórki wektora wyjściowego. Uniknięcie wyścigu danych wymaga wykorzystania atomicznej operacji dodawania.

```
#version 450
#extension GL_EXT_shader_atomic_float: enable

#define WORKGROUP_SIZE 32
layout (local_size_x = WORKGROUP_SIZE, local_size_y = 1) in;

layout(set = 0, binding = 0) buffer bufA {
    uint elementNum, M, N;
    float floatdata[];
};

layout(set = 0, binding = 1) buffer bufAColumnIndex { uint rowIndex[]; };
layout(set = 0, binding = 2) buffer bufARowOffsets { uint colOffsets[]; };
layout(set = 0, binding = 3) buffer inputVector { float inVec[]; };
layout(set = 0, binding = 4) buffer outputVector { float outVec[]; };

void main()
{
    const uint coli = gl_GlobalInvocationID.x;

    if(coli < N) {
        const float inVecTerm = inVec[coli];
        const uint colOffset = colOffsets[coli];
        const uint nzCount = colOffsets[coli+1] - colOffset;

        for(uint rowi = 0; rowi < nzCount; rowi++) {
            const uint cellOffset = colOffset + rowi;
            float prod = inVecTerm * floatdata[cellOffset];
            atomicAdd(outVec[rowIndex[cellOffset]], sum);
        }
    }
}
```

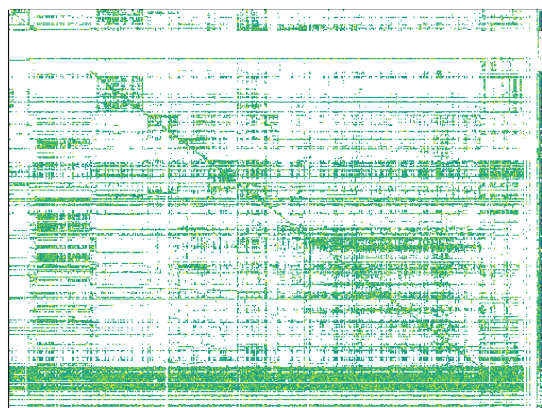
Listing 9. Shader obliczeniowy wykorzystujący macierz CSC

5. Wyniki

W tym rozdziale opisana jest metodologia testowania wydajności algorytmów mnożenia macierzy rzadkiej z wektorem dla wartości pojedynczej precyzji (*SpMV*) oraz dokonana jest analiza uzyskanych wyników.

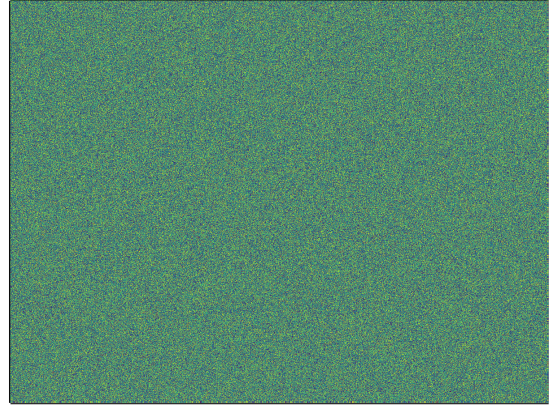
System wykorzystany do przeprowadzenia testów wydajności wyposażony był w procesor graficzny NVIDIA RTX 3060 (GA106) w konfiguracji 130W działający pod sterownikiem w wersji 528.24. Systemem operacyjnym był Windows 11 w wersji 22H2, kompilator *gslc* [5] w wersji v2021.2 oraz VulkanSDK [6] w wersji 1.2.182.0. Na potrzeby testów wykorzystano pięć macierzy: BEAFLW, DENSE2, BCSSTK32, SCIRCUIT i GA41AS41H72. Każda z nich reprezentuje różne cechy macierzy rzadkich: rozmiar, zagęszczenie, rozpiętość wartości w komórkach i sposób rozmieszczenia elementów. Każdy algorytm dla danego formatu uruchamiany był 1000 razy, celem zdobycia średniego czasu wykonania danej operacji.

BEAFLW ma wymiar 497×507 , z czego 21.193% jest elementami niezerowymi, rozkład przedstawiony na rysunku 5.1 wykazuje znikome skumulowanie względem przekątnej.



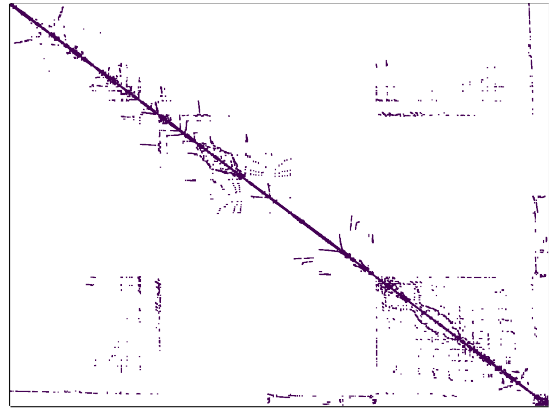
Rys. 5.1. Rozkład w macierzy BEAFLW

DENSE2 ma wymiar 2000×2000 , z czego 100% elementów ma wartości niezerowe, rozkład przedstawiony na rysunku 5.2.



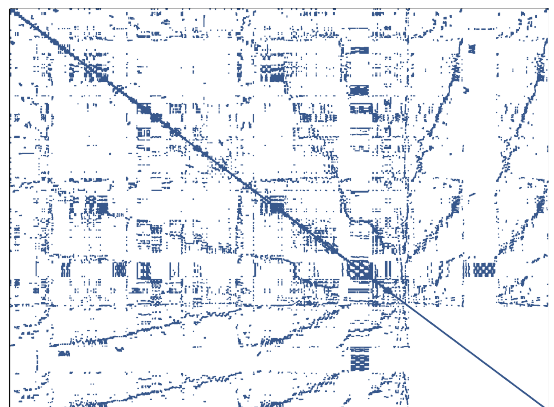
Rys. 5.2. Rozkład w macierzy DENSE2

BCSSTK32 ma wymiar 44609×44609 , z czego 0.1% jest elementami niezerowymi, rozkład przedstawiony na rysunku 5.3 wykazuje symetryczność względem przekątnej i niskie skumulowanie względem niej.



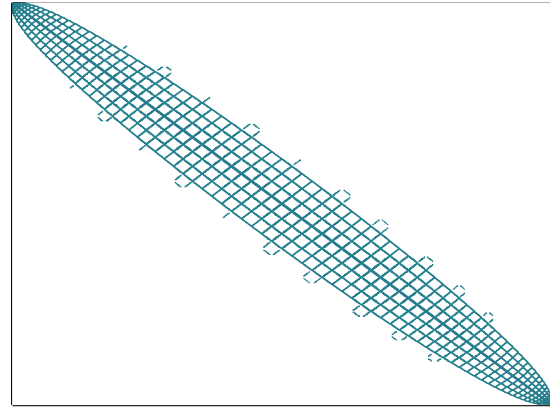
Rys. 5.3. Rozkład w macierzy BCSSTK32

SCIRCUIT ma wymiar 170998×170998 , z czego 0.003% jest elementami niezerowymi, rozkład przedstawiony na rysunku 5.4 wykazuje symetryczność względem przekątnej i średnie skumulowanie względem niej.



Rys. 5.4. Rozkład w macierzy SCIRCUIT

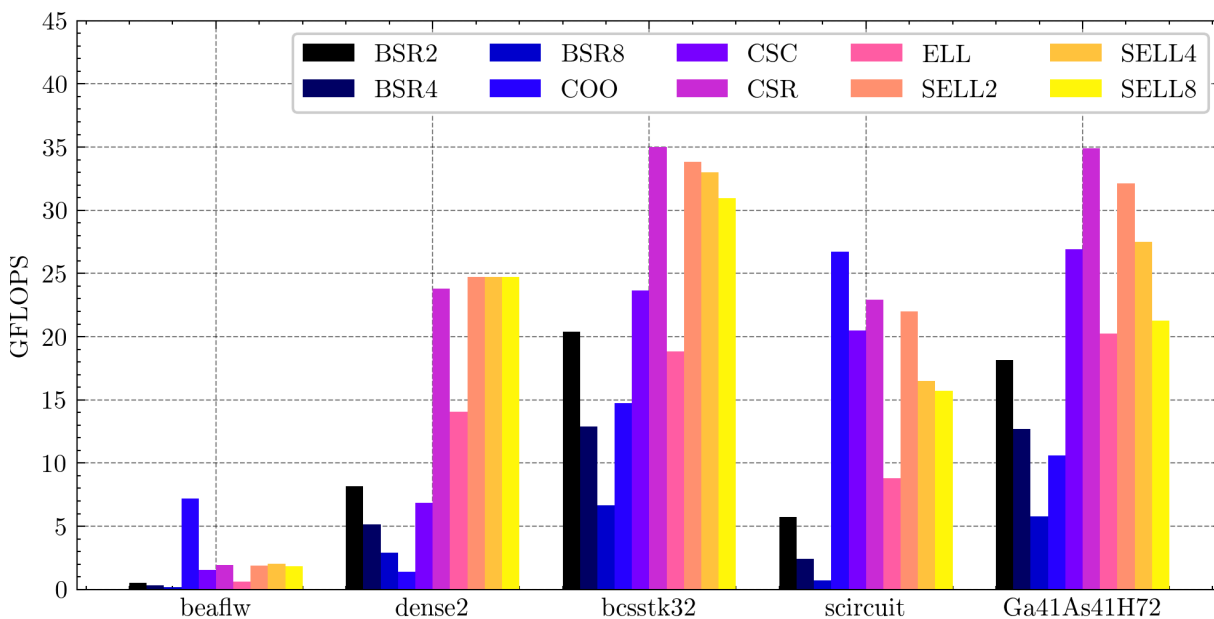
GA41AS41H72 ma wymiar 268096×268096 , z czego 0.026% jest elementami niezerowymi, rozkład przedstawiony na rysunku 5.5 wykazuje symetryczność względem przekątnej i wysokie skumulowanie względem niej.



Rys. 5.5. Rozkład w macierzy GA41AS41H72

Analizując wykres 5.6 zauważalna jest wysoka wydajność formatu CSR, w dwóch przypadkach jest najwyższa, natomiast w pozostałych jest blisko największej osiągniętej wydajności. Format SELL wraz ze zmniejszaniem C osiąga wyższe wyniki zbliżając się wydajnościowo do formatu CSR, co jest oczekiwane, ponieważ dla $C = 1$ format SELL będzie zachowywać się w silnie zbliżony sposób do formatu CSR. Wydajność formatu ELL nie wyróżniała się ponad inne formaty, pomimo znacznie większego zużycia pamięci. W ponad połowie przypadków format ten był szybszy niż COO i CSC, natomiast dla macierzy z małym zagęszczeniem lub małą ilością elementów format COO osiągał najwyższą wydajność ze wszystkich formatów. Pomimo najmniejszego zużycia pamięci format BSR osiągał jedne z najniższych wyników wydajnościowych, im większy rozmiar bloku tym bardziej wydajność spadała.

Wydajność różnych formatów macierzy w SpMV dla wybranych macierzy



Rys. 5.6. Wydajność różnych formatów macierzy

Wprowadzono pojęcie efektywności formatu, który jest sposobem porównania efektywności wykorzystania pamięci do uzyskania danego poziomu wydajności obliczeniowej, określonego przez stałą k :

$$k = \frac{1}{t \cdot s} \quad (5.1)$$

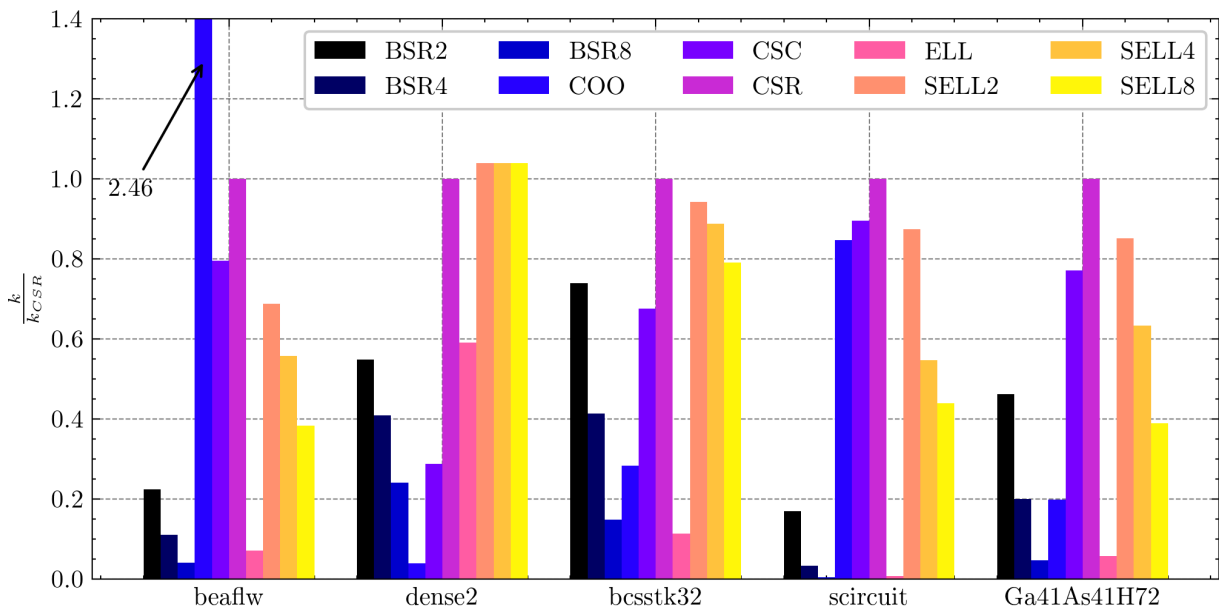
k – stała efektywności

t – czas obliczeń

s – rozmiar pamięci potrzebny do reprezentacji macierzy

Pozwala ona na porównanie w kontekście tej samej macierzy wydajności obliczeniowej w zależności od ilości wykorzystanej pamięci. Stała ta faworyzuje jak najmniejsze wykorzystanie pamięci i w tym samym momencie jak najszybsze wykonanie programu, format może wykorzystać więcej pamięci jeżeli czas wykonania na tym drastycznie zyskuje. Porównywanie różnych stałych może zostać dokonane tylko w obrębie tej samej macierzy. Celem ułatwienia porównań na wykresie 5.7 przedstawiono każdą wartość k przeskalowaną względem stałej k dla formatu CSR. W skutek tego w każdej macierzy format CSR zyskuje wartość $k = 1$, pozostałe wartości należy interpretować jako mnożnik efektywności tego formatu dla tej macierzy względem formatu CSR. Wybrano CSR ze względu na średnio najwyższe wyniki oraz stosunkowo niskie zużycie pamięci. W trzech przypadkach macierzy, format CSR jest najbardziej efektywnym formatem, dla macierzy *DENSE2* jest trochę mniej efektywny niż warianty SELL i dla macierzy *BEAFLW* jest drugi co do efektywności względem formatu COO, która jest 2.5 raza bardziej efektywna.

Efektywność k różnych formatów macierzy w SpMV dla wybranych macierzy



Rys. 5.7. Porównanie efektywności wykorzystania pamięci

Finalne wyniki posiadały pewne różnice pomiędzy sobą, największy błąd względny pojedynczego elementu wektora wyjściowego, dla którego bazą był wynik obliczony na jednostce centralnej formatem ELL wynosił 1.894% dla formatu CSC w macierzy GA41AS41H72. Różnice te są oczekiwanym rezultatem zmiany kolejności wykonywania obliczeń, ponieważ standard IEEE-754 [7] nie gwarantuje łączności obliczeń. Przez to należy rozumieć, iż zachodzi następująca nierówność:

$$(x + y) + z \neq x + (y + z) \tag{5.2}$$

Połączenie formatu, który rozdziela pracę w przeciwnym kierunku co format ELL oraz atomicznej redukcji do każdego elementu wektora wyjściowego, tworzy środowisko, w którym oczekiwane będzie, iż kolejność wykonywanych operacji będzie różna od tej która miała miejsce podczas obliczeń na jednostce centralnej. Dodatkowo, duża ilość elementów i niejednolite wartości macierzy GA41AS41H72 tylko potęgują ten efekt.

6. Podsumowanie

Praca obrała za cel stworzenie programu komunikującego się z procesorem graficznym przy pomocy interfejsu Vulkan. Przy jego pomocy zaprzęgnięto procesor graficzny do wykonywania arbitralnych obliczeń, które wykorzystano do obliczenia wyników mnożenia macierzy rzadkiej z wektorem. Operacje wykonano przy pomocy wielu różnych formatów przechowywania macierzy rzadkich na przykładowych macierzach rzadkich o szerokim spektrum cech charakterystycznych. Na podstawie uzyskanych wyników stwierdzono, że format CSR najczęściej będzie optymalnym lub najbliższym optymalnym formatem przechowywania macierzy rzadkiej, nie tylko w kontekście czystej wydajności obliczeniowej natomiast również z perspektywy efektywności zużycia pamięci. Nietrywialna wydajność obliczeń wskazuje, iż interfejs Vulkan posiada bardzo swobodny dostęp do wykorzystania wszystkich zasobów procesora graficznego celem dokonania arbitralnych obliczeń.

Tak jak w każdej dziedzinie inżynierskiej najlepsza decyzja to ta podjęta w odpowiednim kontekście, dla najlepszych wyników należałoby dokonać ewaluacji jak największej ilości formatów w przestrzeni rozwiązywanego problemu. Należy zwrócić uwagę, iż istnieją sytuacje, w których prędkość obliczeń nie gra roli, ponieważ liczy się możliwie największa kompresja macierzy rzadkiej w pamięci. Dla takich zastosowań nawet mało wydajny format BSR będzie w stanie dostarczyć najmniejszego zużycia pamięci dla niektórych macierzy rzadkich.

Temat nie został w pełni zgłębiony, perspektywą rozwoju może być przeniesienie części obliczeń w grach trójwymiarowych wykorzystujących silną symulację fizyczną na procesor graficzny, celem jej przyspieszenia lub wykorzystanie procesora graficznego w urządzeniu mobilnym, aby umożliwić mu lepsze reprezentowanie rozszerzonej rzeczywistości poprzez zwiększoną wydajność przetwarzania obrazów.

Bibliografia

- [1] S. Oberman E. Lindholm J. Nickolls i J. Montrym. „NVIDIA Tesla: A Unified Graphics and Computing Architecture”. W: *IEEE Micro* 28 (2 2008), s. 39–55.
- [2] *Vulkan 1.3.221 Specification*. <https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html>, dostęp 2022.07.14. Khronos Group.
- [3] *SPIR-V Specification Version 1.6, Revision 2: Unified*. <https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.pdf>, dostęp 2023.01.01. Khronos Group.
- [4] R. Pozo R. F. Boisvert i K. A. Remington. *The matrix Market Exchange Formats: Initial Design*. U. S. Department of Commerce. 1996.
- [5] *Shaderc compiler*. <https://github.com/google/shaderc>, dostęp 2023.01.01. Google.
- [6] *VulkanSDK*. <https://www.lunarg.com/vulkan-sdk/>, dostęp 2023.01.01. LunarG.
- [7] IEEE Computer Society. „IEEE Standard for Floating-Point Arithmetic”. W: *IEEE STD 754-2019* (2019), s. 1–84.