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

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. Niecodzienność struktury tej rodziny procesorów czyni tworzenie programów rozwiązujących danych 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 mikro-architekturę 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 mikro-architekturę 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 wyjątkiem 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 wektorze 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 przyjmiemy, że tylko n_z elementów w macierzy jest niezerowych teoretycznie możliwe jest następujące zmniejszenie zużycia pamięci i przyśpiesznie obliczeń:

$$R = \frac{MN}{n_z} \tag{2.1}$$

R – krotność zmniejszenia zużycia pamięci oraz krotność przyśpieszenia 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:

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

– 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:

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

– 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:

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

– 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:

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

– 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 $\frac{M}{C}$ zaokrąglajac 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 role, 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 $C = 2$:

$$\begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 6 & 0 & 8 & 0 \end{bmatrix} \rightarrow \text{floatdata} = \left[\begin{array}{ccc} 1 & 4 & \\ 2 & 3 & \\ \hline 5 & 7 & 8 \\ 6 & 8 & X \end{array}\right] \quad \text{columnIndices} = \left[\begin{array}{ccc} 0 & 1 & \\ 1 & 2 & \\ \hline 0 & 3 & 4 \\ 1 & 3 & X \end{array}\right]$$

$$\text{rowOffsets} = \begin{bmatrix} 0 & 4 & 10 \end{bmatrix}$$

– 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 $B_s = 2$:

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

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

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

3. Mikro-architektura, interfejs Vulkan i shadery obliczeniowe

3.1. Mikro-architektura nowoczesnych procesorów graficznych

Podejście 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 następnego logicznego kroku. Kolejno 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, natomiast chce wykonać skomplikowane operacje na wygenerowanych fragmentach, nie istniał sposób, aby krzem przeznaczony do przetwarzania geometrii mógł przyczynić się 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. Sami twórcy architektury również mieli problemy z przewidzeniem w jaki sposób zbalansować ilość krzemu przeznaczoną dla każdego z kroków.

Rozwiązaniem było stworzenie ogólnej jednostki, będącej w stanie wykonywać zadane obliczenia bez przywiązania do jakiejkolwiek abstrakcyjnej wizji. Wraz z premierą architektury *Fermi* w mikroprocesorach *NVIDIA*, wprowadzono pojęcie multi-procesora strumieniowego (ang. Stream Multiprocesor), w skrócie *SM*. Posiadał on 8 rdzeni 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* 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 użytkowniku 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 konceptami 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 różnego 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 a 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 wie, na jakich danych ma operować. Dane do i z shaderów są przenoszone poprzez sampler tekstur lub Shader Storage Blocks (*SSBO*). *SSBO* są buferami 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*, jest to język pośredni przechowywany w formacie binarnym. 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 przechowany 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 zostaje 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 macierzy 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, stworzenie puli zasobów, z których korzystać 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 wzorzec 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 aplikacja 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 działać 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 różnić w innym środowisku pracy.

Wybór fizycznego urządzenia odbywa się poprzez zapytanie 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 opisania 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 zmiennoprzecinkowego podwójnej precyzji, który domyślnie jest wyłączona. Dodatkowo, istnieje możliwość zdefiniowania rozszerzeń, które mają zostać włączone. Rozszerzeniami są nowe niezdefiniowane przez specyfikacje 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 ostatnie 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.

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 będąca odpowiedzialna za przechowywanie danych potrzebnych do opisania macierzy rzadkiej 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* [2]. Każda

```
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

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 przekątnej dla zaoszczędzenia miejsca. Podczas wczytywania danych program musi odbić wartości z nad przekątnej 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 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

```
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

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.

```
#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

4.2.2. Format CSR

Kod 4 przedstawia strukturę będąca odpowiedzialna za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie CSR.

```
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

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 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*.

```
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

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 różnica 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 przyśpiesza wykonanie shadera.

4.2.3. Format CSC

Kod 7 przedstawia strukturę będąca odpowiedzialna za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie CSC.

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 wewnętrzna pętle 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 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 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

```
#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

gwarantuje wyłączności do danej komórki wektora wyjściowego. Uniknięcie wyścigu danych wymaga wykorzystania atomicznego operacji dodawania.

```
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

```
#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

6. Podsumowanie

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] R. Pozo R. F. Boisvert i K. A. Remington. *The matrix Market Exchange Formats: Initial Design*. U. S. Department of Commerce. 1996.