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

Serdecznie dziękuję mojej partnerce Weronice za pomoc w tworzeniu pracy

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 liczbie jednostek arytmetycznych w krzemie. Wynikiem tego jest nieosiągalnie duża dla zwykłych mikroprocesorów, surowa moc obliczeniowa. Niecodzienność struktury tej rodziny procesorów czyni tworzenie programów rozwiązujących dany problem bardziej skomplikowane. Liczba dostępnych języków programowania dla procesorów graficznych jest znikoma w porównaniu do języków pozwalających na programowanie przykładowo jednostki centralnej z architekturą x86. Procesory graficzne mogą wspierać różne ISA w tej samej chwili, umożliwia to ciągle działający sterownik, który tłumaczy daną abstrakcję na odpowiedni dla tej mikro-architektury kod maszynowy. Tak jak w nowoczesnych jednostkach centralnych, finalnie wykonywany kod maszynowy nazywany mikrokodem, często zmienia się pomiędzy architekturami; dla przykładu firma AMD udostępnia specyfikację mikrokodu publicznie[1], natomiast firma NVIDIA nie udostępnia takich danych. Przykładową abstrakcją wyższą 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 interfejsu użytego do zarządzania procesorem graficznym. 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 oraz czym są shadery obliczeniowe. Sposób implementacji komunikacji z procesorem graficznym oraz poszczególnych shaderów obliczeniowych zawarto w rozdziale 4. Uzyskane wyniki przy użyciu tej implementacji przeanalizowano w rozdziale 5, a podsumowanie pracy omówiono w rozdziale 6.

2. Mnożenie macierz-wektor dla macierzy rzadkich

2.1. Problem mnożenia macierzy przez 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 liczbie 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 przyjmiemy, że tylko n_z elementów w macierzy jest niezerowych, teoretycznie możliwe jest następujące zmniejszenie zużycia pamięci i przyspieszenie obliczeń:

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

R – krotność zmniejszenia zużycia pamięci oraz krotność przyśpieszenia obliczeń

n_z – liczba elementów niezerowych w macierzy

M – liczba rzędów macierzy

N – liczba kolumn macierzy

2.3. Problem mnożenia macierzy rzadkiej przez wektor

Istnieją problemy, 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 przez 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

Główny problem 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 jednak formaty, które przetrzymują 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 liczbie 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 przez zapisywanie wszystkich wartości z danego rzędu bezpośrednio po sobie. Takie podejście umożliwia nieprzechowywanie rzędu elementu wprost, a jedynie indeks początkowego elementu rzędu. Dwie tablice *floatdata* i *columnIndices* są nadal długości równej liczbie 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 będącego wartością z przedziału $[0, N-1]$ format przechowuje 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 w sposób kolumnowy 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 liczbie elementów niezerowych. Nieprzechowywane natomiast są kolumny zastąpione przez indeks początkowego elementu kolumny. Zadanie to pełni tablica *columnOffsets* o długości o jeden większej, niż liczba kolumn w macierzy. Jeżeli liczba rzędów jest równa liczbie kolumn macierzy, format ten będzie wymagał tej samej ilości pamięci, co format CSR. Dla problemu mnożenia macierz-wektor najczęściej preferowanym formatem jest CSR ze względu na swoją prostotę i wyższą wydajność osiąganą dzięki brakowi wymagania wykorzystania operacji atomowych.

  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 liczbą elementów niezerowych ze wszystkich wierszy. Jest to pierwszy format, który wybiera przechowywanie większej ilości danych, niż liczba elementów niezerowych. Jego wydajność wzrasta dla macierzy, w których średnia liczba elementów niezerowych w wierszu jest najbardziej zbliżona do P. Wykorzystuje dwie tablice, *floatdata* i *columnIndices* obie o wymiarach $M \times P$. 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 liczbą elementów sztucznie zwiększa ilość potrzebnej pamięci. Takie sporadyczne wiersze będą jedynie wpływać na rozmiar swojego paska. Najczęściej jednak wartość P w paskach będzie zbliżona do wartości średniej liczby elementów niezerowych we wszystkich wierszach. Format wymaga dodatkowej tablicy *rowOffsets* o długości równej liczbie 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 $C = 2$:

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

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

– 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 przez 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 niebędących 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 procesory graficzne mają implementować i ulepszać z generacji na generację wyewoluowało z wyspecjalizowanego na podejście 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 mikroprocesor. Przykładowo, jeżeli dana aplikacja chce narysować mało wierzchołków, a jednocześnie wykonać skomplikowane obliczenia 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 jakiejkolwiek abstrakcyjnej wizji. Wraz z premierą architektury *Tesla* w mikroprocesorach firmy *NVIDIA*, wprowadzono pojęcie multi-procesora strumieniowego (ang. Streaming 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* [2]. 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 zaklasyfikowana w kontekście SIMD (Single Instruction Multiple Data) jako procesor tablicowy (ang. Array Processor)[3], czyli jeden obiekt kontrolny sterujący daną liczbą połączonych elementów obliczeniowych, które same w sobie są niezależne, (na przykład mają własne rejestry), natomiast wszystkie operują na podstawie instrukcji wydanych przez obiekt kontrolny.

Obecnie najczęściej ta klasyfikacja opisywana jest jako SIMT (Single Instruction Multiple Threads), nazwa ta została rozpowszechniona przez firmę NVIDIA. 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 mikroprocesora poprzez zwiększenie ich liczby 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 implementowania części spekulacyjnej procesora oraz posiadania 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 mikroarchitektury budowały na koncepcie *SM*, zwiększając ich możliwości, moc oraz liczbę 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* [4] 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 jednostka centralna jest wąskim gardłem w programie. Wydajność została 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 synchronizacja 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 interfejsu prostymi i kompatybilnymi z architekturą komputerów konceptami znacznie redukuje możliwość niesprecyzowania danego aspektu interfejsu. Twórcy sterowników graficznych mogą je znacznie uprościć, tym samym, redukując liczbę błędów i poprawiając wydajność. Zniesione zostaje również 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 pracy. 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 tworzenia 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 potoków. Obok potoku graficznego istnieje potok 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 liczby grup shaderów, a ich liczba 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. Liczba wywołań shaderów w pojedynczej grupie jest definiowana przez sam shader jako lokalny rozmiar shadera, który również jest trójwymiarową wartością $X_l \times Y_l \times Z_l$.

Wszystkie wartości wejściowe i wyjściowe shadera są zdefiniowane przez użytkownika. Shadery mają dostęp do swojego identyfikatora grupy oraz lokalnego identyfikatora wewnątrz tej grupy. Wywołania shadera 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 Buffer Object (*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 atomowe (niepodzielne) operacje na pamięci.

Vulkan oczekuje, że wszystkie shadery będą przekazane do niego w formacie *SPIR-V*[5], 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 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 potoku, przez co aplikacje mają większą swobodę w wykorzystywaniu większej liczby 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 i 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 przesyłanych 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 aplikacji od nowa jest warstwa walidująca. Będzie ona sprawdzać poprawność przesyłanych 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. Wsparcie danej warstwy jest opcjonalne, przykładowo warstwy potrzebne tylko podczas tworzenia oprogramowania nie muszą być załączane razem z plikami programu, co zmniejsza jego rozmiar. Aby sprawdzić, czy dana warstwa jest w danym środowisku wspierana, należy wykorzystać funkcję *vkEnumerateInstanceLayerProperties*. Kod 1 przedstawia wykorzystanie tej funkcji do sprawdzenia czy przesłana nazwa

warstwy walidującej znajduje się we wspieranych warstwach. Warto zauważyć podejście do rozwiązania problemu dynamicznej ilości danych na granicy interfejsu, jakie wybrał Vulkan, funkcja przyjmuje dwa argumenty, z czego jeden to wskaźnik na liczbę całkowitą, a drugi to wskaźnik na tablicę elementów wyjściowych. Funkcja posiada dwustanowe zachowanie, jeżeli wskaźnik na tablicę elementów wyjściowych równy jest *NULL*, to liczba wszystkich dostępnych warstw jest wpisana przez wskaźnik zmiennej określającej liczbę elementów, funkcja nie wykonuje w tym stanie żadnych innych operacji. Po wyjściu z tej funkcji użytkownik wie, ile elementów musi dynamicznie zaalokować, żeby pomieścić wszystkie dostępne warstwy. W drugim stanie, jeżeli wskaźnik na tablicę elementów wyjściowych jest niezerowy, to odczytywana jest przez wskaźnik liczba elementów dostępnych w tablicy wyjściowej oraz pod koniec zostaje nadpisana faktyczną liczbą elementów wpisanych. W tablicy wyjściowej użytkownik ma dostęp do właściwości warstw, takich jak ich nazwa, wersja specyfikacji, wersja implementacji oraz krótki opis, jeżeli w tablicy znajduje się warstwa o takiej samej nazwie, jaka została przesłana do funkcji, zostaje zwrócona wartość boolowska wskazująca na prawdę lub fałsz w przypadku braku szukanej warstwy.

```
static bool
isValidationLayerSupported(const char *validationLayerName) {
    u32 layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount, 0x0);
    VkLayerProperties *layers = malloc(layerCount * sizeof(layers[0]));
    vkEnumerateInstanceLayerProperties(&layerCount, layers);

    bool result = false;
    for(u32 i = 0; i < layerCount; i++) {
        if(strcmp(layers[i].layerName, validationLayerName) == 0) {
            result = true;
            break;
        }
    }

    free(layers);
    return result;
}
```

Listing 1. Funkcja sprawdzająca wsparcie warstwy

Tworzenie instancji odbywa się przy użyciu funkcji *vkCreateInstance*, cechy charakterystyczne jej interfejsu są powielone we wszystkich pozostałych procedurach zdefiniowanych w *Vulkan*. Funkcja zawsze zwraca obiekt typu *VkResult*, który jest enumeracją wszystkich możliwych rezultatów wykonania. Jest to podstawowa forma raportowania błędów, pomyślnie wykonana funkcja zawsze zwróci wartość *VK_SUCCESS*. Dla porównania, OpenGL zwraca kod błędu poprzez wykorzystanie osobnej funkcji *glGetError*, a takie podejście wymaga globalnego stanu, który gorzej współpracuje z wieloma wątkami

chcącymi raportować błędy. Architektura *Vulkan* brała pod uwagę raportowanie błędów od samego początku tworzenia, kod błędu zwrócony jako wynik funkcji nie wymaga globalnego stanu, przez co wielowątkowe wykorzystanie *Vulkan* jest możliwe. Znany jest problem, w którym niemożliwe jest rozłożenie pracy podczas korzystania z OpenGL na wiele wątków, a przez to jeden wątek staje się mocniej obciążony niż reszta podczas wysyłania danych na kolejkę procesora graficznego. Stanowi to problem w aplikacjach wykorzystujących zaawansowaną grafikę trójwymiarową, głównie w grach, ponieważ w obecnych czasach prędkość pojedynczego rdzenia jednostki centralnej rośnie o wiele wolniej, niż ich liczba. Niewykorzystanie możliwości wielowątkowych dzisiejszych jednostek centralnych nakłada ograniczenia na rzeczy, które są możliwe do wykonania, odejście od globalnego stanu to jeden z kroków podjętych celem wspierania pracy wielowątkowej przez *Vulkan*.

Funkcja tworząca instancję przyjmuje trzy argumenty, które również ukazują, jakie rozwiązania zostały przyjęte podczas tworzenia samego interfejsu, argumenty to następująco: wskaźnik na obiekt opisujący jak stworzyć instancję, wskaźnik na obiekt opisujący jak zarządzać pamięcią po stronie jednostki centralnej oraz obiekt wyjściowy samej instancji. Konfigurację tworzonego obiektu przechowuje się w dostarczonych do tego strukturach, pozwala to na zmniejszenie liczby parametrów przesyłanych do każdej z funkcji, upraszczając jej wykorzystanie. Vulkan udostępnia możliwość sterowania alokacją pamięci po stronie jednostki centralnej programiście, wykorzystuje do tego obiekt struktury *VkAllocationCallbacks*, w którym znajdują się wskaźniki na funkcje alokujące i zwalniające pamięć. Częstym jest tworzenie niestandardowych obiektów zarządzających alokacją pamięci, celem zwiększenia wydajności programu oraz zmniejszenie szansy na wyciek pamięci. Jeżeli zamiast wskaźnika na obiekt alokatora zostanie przesłana wartość *NULL*, *Vulkan* wykorzysta domyślny interfejs *malloc* i *free* do zarządzania pamięcią. Ostatnim argumentem w funkcjach tworzących jest zawsze wskaźnik na obiekt tworzony, do którego będzie wpisany wynikowy obiekt, ponieważ wartość zwracana przez funkcję jest już zawsze wykorzystana do zwracania kodu błędu. Funkcja ukazana w kodzie 2 tworzy instancję *Vulkan* z załączoną warstwą walidującą, jeżeli taka jest dostępna. Konfiguracja instancji umożliwia podania informacji na temat aplikacji, takich jak jej nazwa, wersja, nazwa silnika oraz jego wersja i finalnie wersja API Vulkan. Dane te umożliwiają twórcom sterowników graficznych na optymalizację zachowania najpopularniejszych programów, na przykład silników do tworzenia gier, takich jak Unity czy Unreal Engine. Każda struktura w Vulkan posiada pole *sType* i odpowiadającą tej strukturze wartość enumeracyjną, pozwala to na dowolne rozszerzanie pewnych struktur poprzez tworzenie listy połączonej, w których to, jaka struktura znajduje się w danym elemencie listy określone jest przez pole *sType*. Po stworzeniu obiektu struktury *VkApplicationInfo*, jego adres pamięci zostaje zapisany w finalnym obiekcie konfigurującym tworzenie instancji, ten jest rozszerzony o warstwą walidującą, jeżeli taka jest wspierana. Wywołanie ostatecznej funkcji owinięte jest w makro sprawdzające, czy zwrócony kod błędu równy jest *VK_SUCCESS*.

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

```
static VkInstance
createInstance() {
    VkApplicationInfo appInfo = {
        .sType = VK_STRUCTURE_TYPE_APPLICATION_INFO,
        .pApplicationName = "Vulkan Compute",
        .applicationVersion = 0,
        .pEngineName = "notanengine",
        .engineVersion = 0,
        .apiVersion = VK_API_VERSION_1_2,
    };

    VkInstanceCreateInfo createInfo = {
        .sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,
        .pApplicationInfo = &appInfo,
    };

    const char *layerName = "VK_LAYER_KHRONOS_validation";
    if(isValidationLayerSupported(layerName)) {
        createInfo.enabledLayerCount = 1;
        createInfo.ppEnabledLayerNames = &layerName;
    }

    VkInstance instance;
    VK_CALL(vkCreateInstance(&createInfo, NULL, &instance));
    return instance;
}
```

Listing 2. Funkcja tworząca instancję Vulkan

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, w których często znajduje się wiele takich samych procesorów graficznych współpracujących ze sobą. W kodzie 3 dokonywana jest dokładnie ta operacja, z założeniem, iż wybrane zostanie zawsze pierwsze zwrócone urządzenie fizyczne.

```
static VkPhysicalDevice
findPhysicalDevice(VkInstance instance) {
    u32 deviceCount = 1;
    VkPhysicalDevice device = { 0 }
    vkEnumeratePhysicalDevices(instance, &deviceCount, device);
    return device;
}
```

Listing 3. Funkcja znajdująca obiekt fizycznego urządzenia

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. W funkcji 4 dane te są wpisywane do obiektu typu *VkDeviceQueueCreateInfo*, dla tego programu stworzona zostanie jedna kolejka, z priorytetem równym jeden, ten może przyjąć dowolną wartość zmiennoprzecinkową pomiędzy zero a jeden dla każdej tworzonej kolejki. Stworzone kolejki będą typu, który wskazuje indeks rodziny kolejki, ten został pobrany poprzez enumerację wszystkich kolejek na danym urządzeniu fizycznym przy użyciu *vkGetPhysicalDeviceQueueFamilyProperties* i wybraniu tej, która ma ustawioną flagę *VK_QUEUE_COMPUTE_BIT*.

Następnie użytkownik ma możliwość włączenia wybranych specjalnych funkcji procesora graficznego, przykładowo obsługa typu zmiennoprzecinkowego podwójnej precyzji, która 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. Wspieranie takiej funkcjonalności nie może być dodane przez specyfikację, ponieważ nie wszystkie procesory graficzne będą wspierać to rozszerzenie. Mimo wszystko posiadanie rozszerzeń pozwala interfejsowi *Vulkan* być bardziej elastycznym we wspieraniu wielu pofragmentowanych rozwiązań. Ponieważ program wymaga wykorzystania operacji atomowych dla wybranych algorytmów należy je włączyć poprzez dodanie rozszerzenia *VK_EXT_shader_atomic_float*. Nazwa rozszerzenia zostaje dodana do tablicy włączonych rozszerzeń, a konfiguracja tego rozszerzenia zdefiniowana przez strukturę *VkPhysicalDeviceShaderAtomicFloatFeaturesEXT*, w której włączona jest atomowa operacja dodawania zostaje dodana do pola *pNext*, które służy do rozszerzenia tej struktury przy użyciu listy połączonej. 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.

```
static VKDeviceAndComputeQueue
createDevice(VkPhysicalDevice phyDevice) {
    float queuePriorities = 1.0;
    VkDeviceQueueCreateInfo queueCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,
        .queueCount = 1, .pQueuePriorities = &queuePriorities,
        .queueFamilyIndex = getComputeQueueFamilyIndex(phyDevice),
    };

    const char *atomicExtensionName = "VK_EXT_shader_atomic_float";
    VkPhysicalDeviceShaderAtomicFloatFeaturesEXT atomicDeviceFeature = {
        .sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT,
        .shaderBufferFloat32AtomicAdd = true,
    };

    VkDeviceCreateInfo deviceCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,
        .queueCreateInfoCount = 1, .pQueueCreateInfos = &queueCreateInfo,
        .enabledExtensionCount = 1, .ppEnabledExtensionNames = &atomicExtensionName,
        .pNext = &atomicDeviceFeature,
    };

    VKDeviceAndComputeQueue result = {
        .computeQueueFamilyIndex = queueCreateInfo.queueFamilyIndex,
    };
    VK_CALL(vkCreateDevice(phyDevice, &deviceCreateInfo, NULL, &result.device));
    vkGetDeviceQueue(device, queueCreateInfo.queueFamilyIndex, 0, &result.computeQueue);
    return result;
}
```

Listing 4. Funkcja tworząca logiczny obiekt urządzenia

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. Funkcje odpowiedzialne za stworzenie obu tych obiektów przedstawione są w kodzie 5. 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 liczba.

```
static VkQueryPool
createQueryPool(VkDevice device)
{
    VkQueryPoolCreateInfo queryPoolCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
        .pNext = NULL,
        .queryType = VK_QUERY_TYPE_TIMESTAMP,
        .queryCount = 2
    };

    VkQueryPool queryPool = { 0 };
    VK_CALL(vkCreateQueryPool(device, &queryPoolCreateInfo, NULL, &queryPool));
    return queryPool;
}

static VkCommandPool
createCommandPool(VkDevice device, u32 computeQueueFamilyIndex)
{
    VkCommandPoolCreateInfo commandPoolCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,
        .queueFamilyIndex = computeQueueFamilyIndex,
    };

    VkCommandPool commandPool = { 0 };
    VK_CALL(vkCreateCommandPool(device, &commandPoolCreateInfo, NULL, &commandPool));
    return commandPool;
}
```

Listing 5. Funkcje tworzące pulę komend i pulę kwerend

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 między sobą, na przykład rozmiarem lub liczbą. Zbiór wszystkich obiektów potrzebnych do komunikacji z procesorem graficznym w celu wywołania poszczególnych shaderów dla wybranego typu macierzy, które nie są stanem globalnym, nazwano scenariuszem. Każdy scenariusz zawierał następujące obiekty: pary buforów i pamięci, pulę deskryptorów, ich ułożenie oraz zbiór, definicję potoku, oraz bufor komend. Przykładowa struktura opisująca scenariusz znajduje się w kodzie 6.

```
typedef struct ScenarioBSR
{
    VkDescriptorSetLayout descriptorSetLayout;
    VkDescriptorPool descriptorPool;
    VkDescriptorSet descriptorSet;

    VKBufferAndMemory matFloatHost;
    VKBufferAndMemory matRowOffsetsHost;
    VKBufferAndMemory matColIndiciesHost;
    VKBufferAndMemory inVecHost;
    VKBufferAndMemory outVecHost;

    VKBufferAndMemory matFloatDevice;
    VKBufferAndMemory matRowOffsetsDevice;
    VKBufferAndMemory matColIndiciesDevice;
    VKBufferAndMemory inVecDevice;
    VKBufferAndMemory outVecDevice;

    VKPipelineDefinition pipelineDefinition;
    VkCommandBuffer commandBuffer;
} ScenarioBSR;
```

Listing 6. Struktura opisująca scenariusz dla macierzy BSR

W zastosowanym rozwiązaniu każdy bufor przypisany jest do unikatowego obiektu pamięci, z tego względu zdecydowano zgrupować bufor i pamięć razem, tak jak na kodzie obok.

```
typedef struct VKBufferAndMemory
{
    VkBuffer buffer;
    VkDeviceMemory bufferMemory;
    u32 bufferSize;
} VKBufferAndMemory;
```

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 wglądów do zaalokowanej pamięci, które mogą się zmieniać, 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 określenia, jakie typy pamięci, które posiada urządzenie będą wspierać wszystkie zdefiniowane sposoby

wykorzystywania. Zaalokowanie pamięci wymaga odpytania urządzenia, jakie typy pamięci posiada i w jakich kopcach się znajdują. Wybór typu pamięci sprowadza się do tego, czy istnieje pamięć, która jest w stanie obsłużyć wszystkie wymagania bufora i czy jest wystarczająco blisko pamięci samego rdzenia mikroprocesora graficznego. Po wskazaniu wyrównanej ilości pamięci do zaalokowania oraz tego, który typ pamięci ma zostać wykorzystany zostaje stworzony obiekt pamięci. Ten finalnie może zostać przypisany do bufora. Funkcja odpowiedzialna za wykonanie tych wszystkich operacji znajduje się w kodzie 7. Zakłada ona, że zawsze tylko jedna kolejka będzie miała dostęp do bufora, dlatego *sharingMode* ustawiony jest na wartość wskazującą wyłączony dostęp.

```
static VKBufferAndMemory
createBuffer(VKState *state, u32 bufferSize, VkBufferUsageFlags usageFlags,
             VkMemoryPropertyFlagBits memoryFlags)
{
    VkBufferCreateInfo bufferCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO,
        .size = bufferSize,
        .usage = usageFlags,
        .sharingMode = VK_SHARING_MODE_EXCLUSIVE,
    };
    VkBuffer buffer = { 0 };
    VK_CALL(vkCreateBuffer(state->device, &bufferCreateInfo, NULL, &buffer));

    VkMemoryRequirements memoryReqs;
    vkGetBufferMemoryRequirements(state->device, buffer, &memoryReqs);
    VkMemoryAllocateInfo allocateInfo = {
        .sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,
        .allocationSize = memoryReqs.size,
        .memoryTypeIndex = findMemoryType(state->phyDevice,
                                          memoryReqs.memoryTypeBits, memoryFlags)
    };
    VkDeviceMemory bufferMemory = { 0 };
    VK_CALL(vkAllocateMemory(state->device, &allocateInfo, NULL, &bufferMemory));
    VK_CALL(vkBindBufferMemory(state->device, buffer, bufferMemory, 0));

    VKBufferAndMemory result = {
        .buffer = buffer,
        .bufferMemory = bufferMemory,
        .bufferSize = bufferSize,
    };
    return result;
}
```

Listing 7. Funkcja odpowiedzialna za tworzenie 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. Przykładowa funkcja w kodzie 8 przy użyciu funkcji *vkMapMemory* i *vkUnmapMemory* przenosi zawartość wektora do pamięci bufora przestojowego.

```
static void
inVecToSSBO(VKState *state, Vector vec, VKBufferAndMemory ssbo)
{
    void *mem = NULL;
    vkMapMemory(state->device, ssbo.bufferMemory, 0, ssbo.bufferSize, 0, &mem);
    memcpy(mem, vec.floatdata, vec.len * sizeof(vec.floatdata[0]));
    vkUnmapMemory(state->device, ssbo.bufferMemory);
}
```

Listing 8. Funkcja odpowiedzialna za tworzenie bufora

Zaprzęgnięcie procesora graficznego do faktycznego wykonywania pracy wymaga wysłania bufora z nagranymi 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, aż do zakończenia jego nagrywania. Gotowy bufor 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. Funkcja przedstawiona w kodzie 9 przy wykorzystaniu buforów komend kopiuje dane z bufora przestojowego do bufora znajdującego się w pamięci globalnej procesora graficznego. Tworząc bufor komend możliwe jest określenie dwóch poziomów, bufor główny może wykonać nagrany bufor drugorzędny oraz samemu być przesłanym na kolejkę. Bufor drugorzędny może być jedynie wykonany w kontekście bufora głównego i nie może być sam przesłany na kolejkę. Ponieważ dla tego zastosowania bufor zostaje zwolniony po jego wykonaniu, dodatkowo przesłana flaga w obiekcie struktury *VkCommandBufferBeginInfo*, wskazuje na fakt, że bufor jest przeznaczony do jednorazowego użycia. Używając komendy *vkCmdCopyBuffer*, której argumentem jest obiekt określający cały rozmiar bufora bez żadnego przesunięcia ani w buforze źródłowym, ani w buforze docelowym, dokonywana jest kopia bufora. Nagrany bufor komend zostaje przesłany na kolejkę funkcją *vkQueueSubmit* oraz przy użyciu *vkQueueWaitIdle* program po stronie jednostki centralnej czeka na zakończenie operacji.

```
static void
copyStagingBufferToDevice(VKState *state, VKBufferAndMemory staging, VKBufferAndMemory device)
{
    assert(staging.bufferSize == device.bufferSize);
    VkCommandBufferAllocateInfo allocInfo = {
        .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO,
        .level = VK_COMMAND_BUFFER_LEVEL_PRIMARY,
        .commandPool = state->commandPool,
        .commandBufferCount = 1,
    };
    VkCommandBuffer cmdBuffer = { 0 };
    vkAllocateCommandBuffers(state->device, &allocInfo, &cmdBuffer);

    VkCommandBufferBeginInfo beginInfo = {
        .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
        .flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,
    };
    vkBeginCommandBuffer(cmdBuffer, &beginInfo);
    VkBufferCopy copyRegion = { .size = staging.bufferSize };
    vkCmdCopyBuffer(cmdBuffer, staging.buffer, device.buffer, 1, &copyRegion);
    vkEndCommandBuffer(cmdBuffer);

    VkSubmitInfo submitInfo = {
        .sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
        .commandBufferCount = 1,
        .pCommandBuffers = &cmdBuffer,
    };

    vkQueueSubmit(state->computeQueue, 1, &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(state->computeQueue);

    vkFreeCommandBuffers(state->device, state->commandPool, 1, &cmdBuffer);
}
```

Listing 9. Funkcja kopiująca bufor przestojowy do pamięci globalnego procesora graficznego

Pula deskryptorów pełni taką samą rolę jak pula komend i kwerend, alokacje deskryptorów są lokalizowane do danej puli, ułatwiając zarządzanie. W kodzie 10 przedstawiona jest funkcja, tworząca pulę deskryptorów, z której będzie można zaalokować jeden deskryptor typu *VK_DESCRIPTOR_TYPE_STORAGE_BUFFER*. W konfiguracji tworzenia zostaje przesłana flaga pozwalająca na zwalnianie zaalokowanych deskryptorów z powrotem do puli, bez tej flagi jedyne operacje, jakie są dostępne to wykorzystanie puli do alokacji i jej zrestartowanie. Ponieważ konfiguracja przyjmuje tablicę rozmiarów do zaalokowania, jedna pula może służyć jako globalna pula, z której alokowane są deskryptory różnych typów.

```
static VkDescriptorPool
createDescriptorPool(VkDevice device)
{
    VkDescriptorPoolSize poolSize = {
        .type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER,
        .descriptorCount = 1,
    };

    VkDescriptorPoolCreateInfo createInfo = {
        .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO,
        .maxSets = 1,
        .poolSizeCount = 1,
        .pPoolSizes = &poolSize,
        .flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT,
    };

    VkDescriptorPool result;
    VK_CALL(vkCreateDescriptorPool(device, &createInfo, NULL, &result));

    return result;
}
```

Listing 10. Funkcja tworząca pule deskryptorów

Deskryptor reprezentuje zasób dostępny shaderowi, na przykład bufor lub sampler tekstur. Zdefiniowanie deskryptora wymaga stworzenia obiektu zestawu deskryptorów, a to z kolei wymaga opisania układu deskryptorów w zestawie poprzez stworzenie obiektu układu zestawu deskryptorów. Funkcja 11 tworzy układ, w którym każdy deskryptor będzie następować po sobie, określamy indeks w zestawie, pod którym ten deskryptor będzie dostępny, jego typ oraz liczbę, a gdy ta jest większa niż jeden umożliwia, to stworzenie tablicy obiektów, na przykład, takich jak samplery tekstur w shaderze. Różnica pomiędzy ilością deskryptorów w danym powiązaniu została przedstawiona na rysunku 4.1.

```
// Przykład dla descriptorCount == 1
layout(set = 0, binding = 1)
    buffer type_t {
        float data[];
    } single;

// Użycie
int main {
    single.data[lid] += 1;
}
```

```
// Przykład dla descriptorCount > 1
layout(set = 0, binding = 1)
  buffer type_t {
      float data[];
  } array[];

// Użycie
int main {
    array[lid].data[bid] += 1;
}
```

Rys. 4.1. Różnica pomiędzy liczbą deskryptorów w danym powiązaniu

```
static VkDescriptorSetLayout
createConsecutiveDescriptorSetLayout(VkDevice device, u32 num)
{
    u32 size = num * sizeof(VkDescriptorSetLayoutBinding);
    VkDescriptorSetLayoutBinding *layoutBindings = malloc(size);
    memset(layoutBindings, 0, size);

    for(u32 i = 0; i < num; i++) {
        layoutBindings[i].binding = i;
        layoutBindings[i].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        layoutBindings[i].descriptorCount = 1;
        layoutBindings[i].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
    }

    VkDescriptorSetLayoutCreateInfo createInfo = {
        .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
        .bindingCount = num,
        .pBindings = layoutBindings,
    };

    VkDescriptorSetLayout result = { 0 };
    VK_CALL(vkCreateDescriptorSetLayout(device, &createInfo, NULL, &result));
    free(layoutBindings);

    return result;
}
```

Listing 11. Funkcja tworząca seryjny układ deskryptorów

Na podstawie jednego lub większej liczby układów zestawu deskryptorów funkcja 12 tworzy obiekt zestawu deskryptorów, który obecnie jest tylko sposobem interpretacji podłączonych zasobów, natomiast nie posiada żadnych zasobów powiązanych ze sobą.

```
static VkDescriptorSet createDescriptorSet(VkDevice device,
    VkDescriptorSetLayout layout, VkDescriptorPool pool)
{
    VkDescriptorSetAllocateInfo allocateInfo = {
        .sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO,
        .descriptorPool = pool,
        .descriptorSetCount = 1, .pSetLayouts = &layout
    };
    VkDescriptorSet result = { 0 };
    VK_CALL(vkAllocateDescriptorSets(device, &allocateInfo, &result));
    return result;
}
```

Listing 12. Funkcja tworząca obiekt zestawu deskryptorów

Przypisanie buforów odbywa się poprzez aktualizację zestawu deskryptorów, wymagane informacje to: który zestaw deskryptorów jest modyfikowany, który indeks w tym zestawie oraz liczba deskryptorów w tym indeksie, jakie należy zaktualizować. Sam zasób jest określany przez jeden z trzech różnych wskaźników na obrazy, bufory lub określony zakres bufora, dla buforów określany jest identyfikator bufora, przesunięcie względem początku oraz rozmiar, jaki ma być zmapowany. Liczba deskryptorów określa liczbę elementów znajdujących się pod jednym z tych trzech wskaźników, to który z nich będzie wybrany określa typ deskryptora. Funkcja przypisująca bufory do deskryptorów przy pomocy funkcji *vkUpdateDescriptorSets* przedstawiona jest w kodzie 13, jako parametry wejściowe przyjmuje tablicę buforów, przesunięć w pamięci do nich i przypisuje je do układu deskryptorów stworzonego przez funkcję w kodzie 11.

```
static void
bindDescriptorSetWithBuffers(VKState *state, VkDescriptorSet descriptorSet,
                             VKBufferAndMemory *buffers, u32 *offsets, u32 len)
{
    u32 bufferInfoSize = len * sizeof(VkDescriptorBufferInfo);
    VkDescriptorBufferInfo *bufferInfo = calloc(1, bufferInfoSize);
    for(u32 i = 0; i < len; i++) {
        bufferInfo[i].buffer = buffers[i].buffer;
        bufferInfo[i].offset = offsets[i];
        bufferInfo[i].range = buffers[i].bufferSize - offsets[i];
    }

    u32 writeDescSize = len * sizeof(VkWriteDescriptorSet);
    VkWriteDescriptorSet *setWrite = calloc(1, writeDescSize);
    for(u32 i = 0; i < len; i++) {
        setWrite[i].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
        setWrite[i].dstSet = descriptorSet;
        setWrite[i].dstBinding = i;
        setWrite[i].descriptorCount = 1;
        setWrite[i].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
        setWrite[i].pBufferInfo = &bufferInfo[i];
    }

    vkUpdateDescriptorSets(state->device, len, setWrite, 0, NULL);

    free(bufferInfo);
    free(setWrite);
}
```

Listing 13. Funkcja przypisująca bufory do deskryptorów w zestawie deskryptorów

Tak stworzony i zaktualizowany z przypisanymi buforami zestaw deskryptorów może następnie być wykorzystany podczas tworzenia potoku obliczeniowego. Będzie on uruchamiał dany shader, więc stworzony zostaje moduł shadera poprzez wykorzystanie danych binarnych w formacie SPIR-V. Następnie

stworzony zostaje układu potoku, który wymaga podania układu deskryptorów oraz opcjonalnie zestawu stałych, które mają zostać przesłane do shadera. Ostatnimi informacjami do określenia to struktura opisująca etap potoku, 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 potok obliczeniowy, który hermetyzuje sposób uruchomienia danego shadera i informacji z nim związanych. Funkcja przedstawiona w kodzie 14 jest odpowiedzialna za wszystkie wyżej wymienione kroki, jej rezultatem jest układ potoku zwierający dane o sposobie ułożenia deskryptorów i zestawie stałych oraz sam obiekt opisujący potok, który opisuje etap potoku oraz shader, który zostanie uruchomiony.

```
static VKPipelineDefinition
createComputePipeline(VkDevice device, const char *shaderPath,
                      VkDescriptorSetLayout descriptorSetLayout)
{
    Data spirvData = readEntireFile(shaderPath);
    VkShaderModuleCreateInfo createInfo = {
        .sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO,
        .pCode = (u32 *)spirvData.bytes, .codeSize = spirvData.length,
    };
    VkShaderModule computeShaderModule = { 0 };
    VK_CALL(vkCreateShaderModule(device, &createInfo, NULL, &computeShaderModule));
    free(spirvData.bytes);

    VkPipelineLayoutCreateInfo layoutCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,
        .setLayoutCount = 1, .pSetLayouts = &descriptorSetLayout,
    };
    VkPipelineLayout layout = { 0 };
    VK_CALL(vkCreatePipelineLayout(device, &layoutCreateInfo, NULL, &layout));

    VkPipelineShaderStageCreateInfo shaderStageCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,
        .stage = VK_SHADER_STAGE_COMPUTE_BIT,
        .module = computeShaderModule, .pName = "main",
    };
    VkComputePipelineCreateInfo pipelineCreateInfo = {
        .sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO,
        .stage = shaderStageCreateInfo, .layout = layout,
    };
    VkPipeline pipeline;
    VK_CALL(vkCreateComputePipelines(device, VK_NULL_HANDLE, 1,
                                     &pipelineCreateInfo, NULL, &pipeline));

    VKPipelineDefinition result = { .pipeline = pipeline, .layout = layout };
    return result;
}
```

Listing 14. Funkcja tworząca potok obliczeniowy wykorzystując podany shader

Aby go uruchomić należy stworzyć nowy bufor komend, rozpocząć jego nagrywanie i wywołać następujące komendy: przypisanie potoku, przypisanie zestawu deskryptorów, zrestartowanie puli kwerend, wpisanie obecnego znacznika czasowego do puli kwerend, wywołanie $X \times Y \times Z$ grup inwokujących shader z tego potoku oraz wpisanego drugiego znacznika czasowego po zakończonych inwokacjach shadera, po czym nagrywanie bufora zostaje zakończone. Funkcja 15 tworzy i nagrywa taki bufor komend na podstawie przesłanego obiektu potoku, zestawu deskryptorów oraz liczbie grup, które mają być wywołane podczas uruchomienia. Tak nagrany bufor komend zostaje przesłany na kolejkę obliczeniową urządzenia, celem jego wykonania. Gdy kolejka przejdzie w stan bezczynności wszystkie komendy zostały wykonane, tak samo jak w przypadku funkcji 9. Przy użyciu obiektu puli kwerend pobierane zostają dwa znaczniki czasowe, a różnica pomiędzy nimi opisuje czas potrzebny na wykonanie tego shadera dla użytych danych.

```
static VkCommandBuffer
createCommandBuffer(VKState *state, VKPipelineDefinition *pipeline,
                    VkDescriptorSet *descriptorSet,
                    u32 dispatchX, u32 dispatchY, u32 dispatchZ)
{
    VkCommandBufferAllocateInfo allocateInfo = {
        .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO,
        .commandPool = state->commandPool,
        .level = VK_COMMAND_BUFFER_LEVEL_PRIMARY,
        .commandBufferCount = 1,
    };
    VkCommandBuffer result = { 0 };
    VK_CALL(vkAllocateCommandBuffers(state->device, &allocateInfo, &result));

    VkCommandBufferBeginInfo beginInfo = {
        .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    };
    VK_CALL(vkBeginCommandBuffer(result, &beginInfo));

    vkCmdBindPipeline(result, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline->pipeline);
    vkCmdBindDescriptorSets(result, VK_PIPELINE_BIND_POINT_COMPUTE,
                            pipeline->layout, 0, 1, descriptorSet, 0, NULL);
    vkCmdResetQueryPool(result, state->queryPool, 0, 2);
    vkCmdWriteTimestamp(result, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, state->queryPool, 0);
    vkCmdDispatch(result, dispatchX, dispatchY, dispatchZ);
    vkCmdWriteTimestamp(result, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, state->queryPool, 1);

    VK_CALL(vkEndCommandBuffer(result));
    return result;
}
```

Listing 15. Funkcja bufor komend uruchamiający shader obliczeniowy

4.2. Implementacja mnożenia macierz - wektor dla każdego z formatów

Rozdział ten przedstawia sposób przechowywania w kodzie każdej z macierzy i shader wykorzystujący dany format do obliczenia iloczynu macierz-wektor.

4.2.1. Format COO

Kod 16 przedstawia strukturę odpowiedzialną za przechowywanie danych potrzebnych do opisania 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 16. Struktura definiująca macierz w formacie COO

Shader ukazany w kodzie 17 oblicza wynik mnożenia macierzy w formacie COO z wektorem. Przyjmuje cztery bufory wejściowe i jeden wyjściowy. Pierwsze trzy opisują kolejno: 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 jest 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 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 atomowo dodaje go do wartości wektora wyjściowego, znajdującej się pod indeksem rzędu elementu. Atomowa 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

layout (local_size_x = 32, 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 17. Shader obliczeniowy wykorzystujący macierz COO

4.2.2. Format CSR

Kod 18 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 18. Struktura definiująca macierz w formacie CSR

Shader ukazany w kodzie 19 oblicza wynik mnożenia macierzy w formacie CSR z wektorem. Bufory 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 *prod* sumuje wszystkie iloczyny wartości elementów występujących w tym rzędzie. Ponieważ każdy wątek odpowiedzialny jest za unikatowy rząd, dane miejsce w pamięci wyjściowej jest zapisywane tylko przez jeden wątek. Usuwa to potrzebę bazowania na operacjach atomowych, co przyśpiesza wykonanie shadera.

```
#version 450

layout (local_size_x = 32, 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 prod = 0.0;
        for(uint coli = 0; coli < nzCount; coli++)
        {
            const uint cellOffset = rowOffset + coli;
            prod += inVec[columnIndex[cellOffset]] * floatdata[cellOffset];
        }
        outVec[rowi] = prod;
    }
}
```

Listing 19. Shader obliczeniowy wykorzystujący macierz CSR

4.2.3. Format CSC

Kod 20 przedstawia strukturę odpowiedzialną za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie CSC.

```
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 20. Struktura definiująca macierz w formacie CSC

Shader ukazany w kodzie 21 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, a danym elementem macierzy w tej kolumnie. Ze względu na przechodzenie po kolumnach zamiast po rzędach dana inwokacja shadera nie gwarantuje wyłączności do danej komórki wektora wyjściowego. Uniknięcie wyścigu danych wymaga wykorzystania atomowych operacji dodawania.

```
#version 450
#extension GL_EXT_shader_atomic_float: enable

layout (local_size_x = 32, 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]], prod);
        }
    }
}
```

Listing 21. Shader obliczeniowy wykorzystujący macierz CSC

4.2.4. Format ELL

Kod 22 przedstawia strukturę odpowiedzialną za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie ELL.

Shader ukazany w kodzie 23 oblicza wynik mnożenia macierzy w formacie ELL z wektorem. Bufory wejściowe są podobne jak w przypadku macierzy COO, natomiast shader nie wymaga tablicy określającej rząd danego elementu i dodatkowo w nagłówku przesłana zostaje wartość P. 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 oblicza przesunięcie do tablic *floatdata* i *columnIndices*, pod którym zaczynają się elementy dla tego rzędu. Następnie, iterując w pętli do teoretycznego maksymalnego indeksu P, określany jest indeks elementu w obu tablicach. Jeżeli dla tego elementu wartość w *columnIndices* wskazuje na brak danych, praca tego wątku zostaje przerwana. W przeciwnym przypadku, obliczony zostaje produkt, który jest sumowany

```
// wartość signalizująca brak danych dla tej komórki
#define INVALID_COLUMN 0xffffffff

typedef struct MatrixELL
{
    u32 M, N, elementNum; // wymiar M x N i ilość elementów niezerowych
    u32 P;                // maksymalna liczba elementów niezerowych
    float *floatdata;     // tablica wartości elementów
    u32 *columnIndices;   // tablica kolumn elementów
} MatrixELL;
```

Listing 22. Struktura definiująca macierz w formacie ELL

do zmiennej lokalnej *prod*, która pod koniec zostaje wpisana do wektora wyjściowego. Każdy wątek odpowiedzialny jest za unikatowy rząd, dane miejsce w pamięci wyjściowej jest zapisywane tylko przez jeden wątek bez potrzeby wykorzystania operacji atomowych.

```
#version 450

layout (local_size_x = 32, local_size_y = 1) in;
layout(set = 0, binding = 0) buffer bufA {
    uint M, P, N;
    uint columnIndices[];
};
layout(set = 0, binding = 1) buffer bufAFloat    { float floatdata[]; };
layout(set = 0, binding = 2) buffer inputVector  { float inVec[]; };
layout(set = 0, binding = 3) buffer outputVector { float outVec[]; };

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

    if(rowi < M) {
        const uint rowOffset = rowi * P;

        float prod = 0.0;
        for(uint coli = 0; coli < P; coli++) {
            const uint cellOffset = rowOffset + coli;
            if(columnIndices[cellOffset] == 0xffffffff) { break; }
            prod += inVec[columnIndices[cellOffset]] * floatdata[cellOffset];
        }
        outVec[rowi] = prod;
    }
}
```

Listing 23. Shader obliczeniowy wykorzystujący macierz ELL

4.2.5. Format SELL

Kod 24 przedstawia strukturę odpowiedzialną za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie SELL.

```
typedef struct MatrixSELL
{
    u32 M, N, C;         // wymiar M x N i wysokość paska
    u32 elementNum;      // ilość elementów niezerowych
    float *floatdata;    // tablica wartości elementów
    u32 *columnIndices; // tablica kolumn elementów
    u32 *rowOffsets;     // tablica pierwszego indeksu elementu w pasku
} MatrixSELL;
```

Listing 24. Struktura definiująca macierz w formacie SELL

Shader ukazany w kodzie 25 oblicza wynik mnożenia macierzy w formacie SELL z wektorem. Bufory wejściowe są identyczne jak w przypadku macierzy CSR, dodatkowo w zmiennej C nagłówek zawiera wysokość paska dla tej macierzy. Globalny identyfikator traktowany jest jako indeks rzędu, na którym mają zostać wykonane obliczenia. Jeżeli indeks ten jest większy, niż liczba wszystkich rzędów wątek, ten nie wykonuje żadnej pracy. W przeciwnym wypadku obliczony zostaje indeks paska oraz indeks rzędu wewnątrz paska na podstawie globalnego indeksu rzędu i wartości C, wartość ta służy też do określenia ilości kolumn w tym pasku przy wykorzystaniu tablicy *rowOffsets*, jako różnica przesunięcia następnego paska, a paska obecnego podzielona przez wartość C. Pobrane zostaje również przesunięcie do tablic *floatdata* i *columnIndex*, pod którym zaczynają się elementy dla tego paska. Wartości pasków przechowywane są kolumnowo, aby umożliwić procesorowi graficznemu łączenie dostępów do pamięci pomiędzy wątkami, ze względu na to, pierwszy element należący do danego wątku to przesunięcie w pamięci do obecnego paska plus indeks rzędu wewnątrz paska. Iterując przez wszystkie kolumny w pasku, do zmiennej lokalnej *prod* sumowane zostaje iloczyn elementu macierzy z elementem wektora wejściowego. Aby uniknąć operacji warunkowej sprawdzającej obecność danych w elemencie, pod uwagę brane są wszystkie komórki macierzy wejściowej. W miejscach, gdzie nie ma danych znajduje się zero, które podczas mnożenia nie produkuje wartości wypływających na wynik ostateczny. Każdy wątek odpowiedzialny jest za unikatowy rząd, dane miejsce w pamięci wyjściowej jest zapisywane tylko przez jeden wątek bez potrzeby wykorzystania operacji atomowych.

```
#version 450

layout (local_size_x = 32, local_size_y = 1) in;
layout(set = 0, binding = 0) buffer matHeaderAndColIndex {
    uint M, C, N;
    uint columnIndex[];
};
layout(set = 0, binding = 1) buffer matRowOffsets { uint rowOffsets[]; };
layout(set = 0, binding = 2) buffer matFloat      { float floatdata[]; };
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 sliceIndex = rowi / C;
        const uint rowIndexInSlice = rowi % C;
        const uint columnCount = (rowOffsets[sliceIndex + 1] - rowOffsets[sliceIndex]) / C;
        const uint rowOffset = rowOffsets[sliceIndex] + rowIndexInSlice;

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

Listing 25. Shader obliczeniowy wykorzystujący macierz SELL

4.2.6. Format BSR

Kod 26 przedstawia strukturę odpowiedzialną za przechowywanie danych potrzebnych do opisania macierzy rzadkiej w formacie BSR.

Shader ukazany w kodzie 27 oblicza wynik mnożenia macierzy w formacie BSR z wektorem. Bufory wejściowe są identyczne jak w przypadku macierzy CSR. Globalny identyfikator traktowany jest jako indeks rzędu, na którym mają zostać wykonane obliczenia, na jego podstawie obliczany jest indeks rzędu blokowego i indeks rzędu wewnątrz bloku. Jeżeli indeks rzędu blokowego jest większy, niż liczba wszystkich rzędów blokowych lub indeks rzędu wewnątrz bloku jest większy, niż rozmiar bloku, to wątek ten nie wykonuje żadnej pracy. W przeciwnym wypadku pobiera przesunięcie do tablic *floatdata* i *columnIndex*, pod którym zaczynają się elementy dla tego rzędu blokowego. Na podstawie tablicy *rowOffsets* ustalona zostaje ilość bloków niezerowych w tym rzędzie blokowym jako różnica przesunięcia

```
typedef struct MatrixBSR
{
    u32 MB, NB, nnzb;    // wymiar MB x NB i ilość bloków niezerowych
    u32 elementNum;      // liczba elementów niezerowych
    u32 blockSize;       // rozmiar bloku
    float *floatdata;    // tablica wartości elementów
    u32 *rowOffsets;     // tablica pierwszego indeksu bloku w rzędzie
    u32 *columnIndices; // tablica kolumn bloków
} MatrixBSR;
```

Listing 26. Struktura definiująca macierz w formacie BSR

następnego rzędu blokowego, a rzędu obecnego. Iterując przez wszystkie kolumny blokowe, obliczane jest przesunięcie dla tego bloku i pobierana zostaje blokowa wartość kolumny tego bloku. W mniejszej pętli przechodzącej przez każdą kolumnę w bloku, która znajduje się na przypisanym temu wątkowi rzędzie, do zmiennej lokalnej *prod* sumowany zostaje iloczyn elementów macierzy z elementem wektora wejściowego. Każdy wątek odpowiedzialny jest za unikatowy rząd, dane miejsce w pamięci wyjściowej jest zapisywane tylko przez jeden wątek bez potrzeby wykorzystania operacji atomowych.

```
#version 450

layout (local_size_x = 32, local_size_y = 1) in;
layout(set = 0, binding = 0) buffer bufA {
    uint bs, MB, NB;
    float floatdata[];
};
layout(set = 0, binding = 1) buffer bufAColumnIndex { uint rowOffsets[]; };
layout(set = 0, binding = 2) buffer bufARowOffsets  { uint colIndicies[]; };
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;
    const uint rowInBlockIndex = rowi % bs;
    const uint rowBlockIndex = rowi / bs;

    if(rowInBlockIndex < bs && rowBlockIndex < MB) {
        uint nnzCol = rowOffsets[rowBlockIndex + 1] - rowOffsets[rowBlockIndex];
        float prod = 0.0;
        for(uint coli = 0; coli < nnzCol; coli++) {
            const uint blockOffset = (rowOffset + coli) * bs * bs;
            const uint colbi = colIndicies[rowOffset + coli];
            for(uint cbi = 0; cbi < bs; cbi++) {
                float a = inVec[colbi * bs + cbi];
                float b = floatdata[blockOffset + cbi + rowInBlockIndex * bs];
                prod +=  a * b;
            }
        }
        outVec[rowBlockIndex * bs + rowInBlockIndex] = prod;
    }
}
```

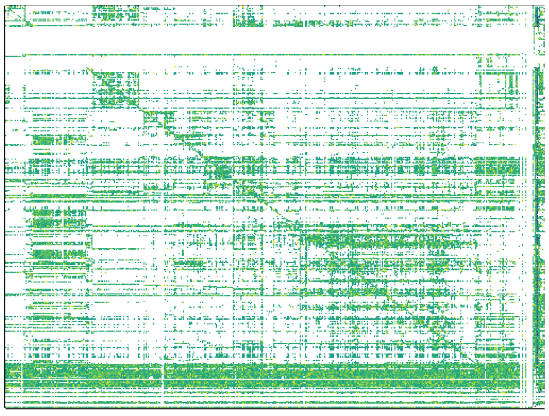
Listing 27. Shader obliczeniowy wykorzystujący macierz BSR

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.

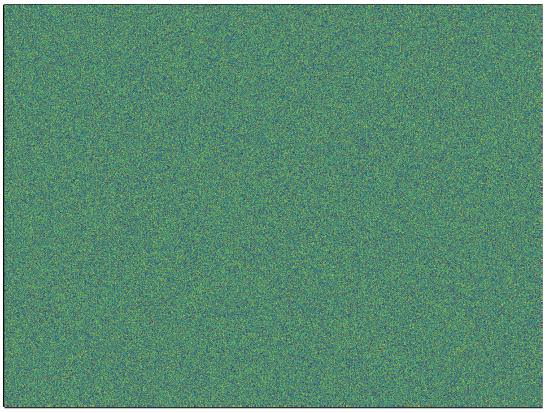
System wykorzystany do przeprowadzenia testów wydajności wyposażony był w procesor graficzny NVIDIA RTX 3060 (GA106) w konfiguracji 130W, działający pod sterownikiem w wersji 528.24. Systemem operacyjnym był Windows 11 w wersji 22H2, kompilator *glslc* [6] w wersji v2021.2 oraz VulkanSDK [7] w wersji 1.2.182.0. Na potrzeby testów wykorzystano pięć macierzy o nazwach: 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.



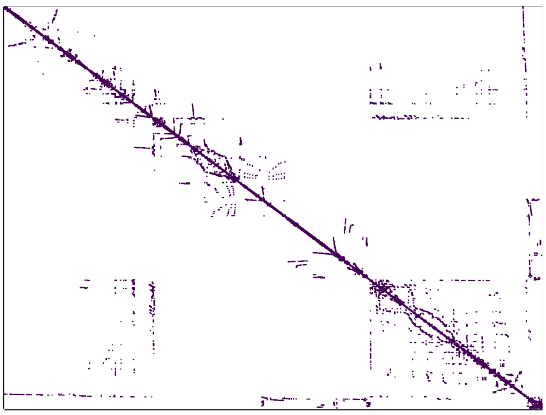
Rys. 5.1. Rozkład w macierzy BEAFLW

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



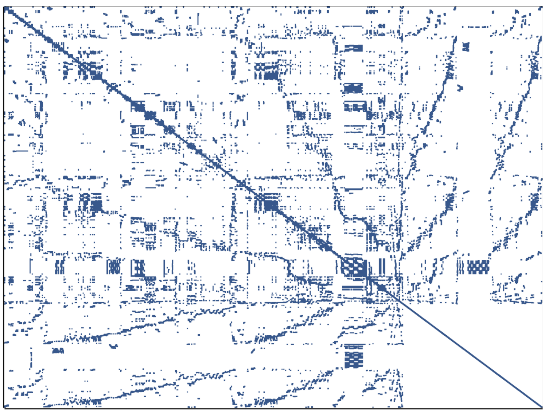
Rys. 5.2. Rozkład w macierzy DENSE2

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.



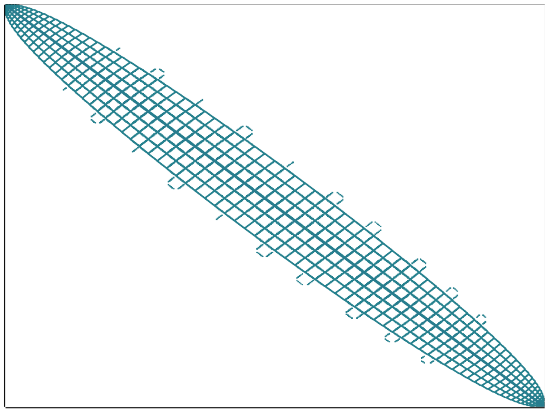
Rys. 5.3. Rozkład w macierzy BCSSTK32

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.



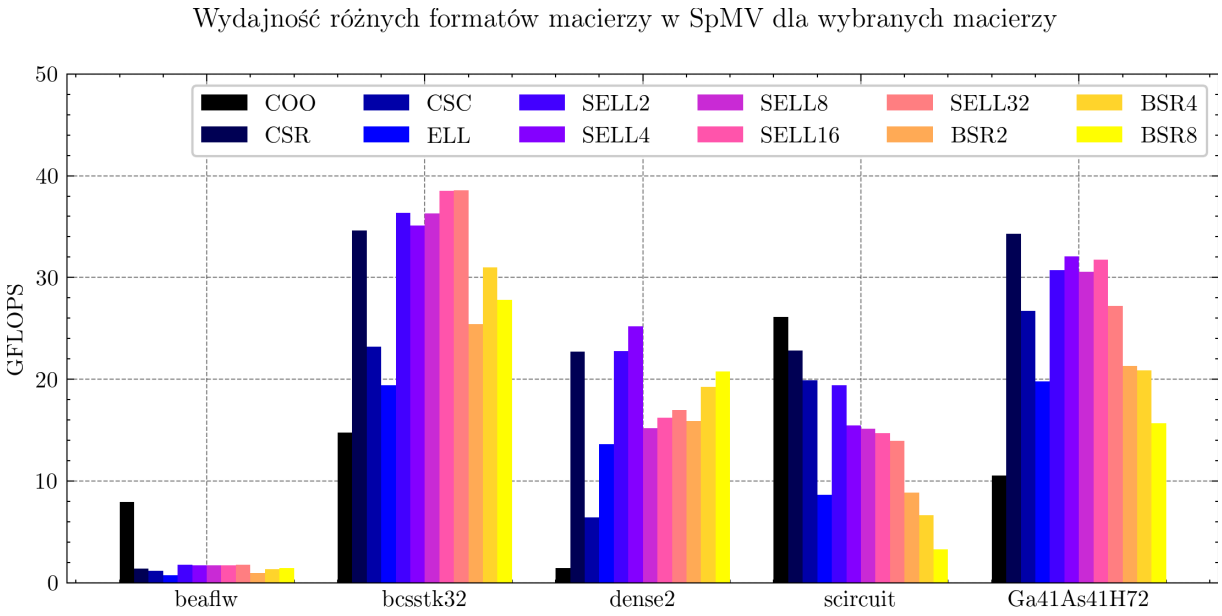
Rys. 5.4. Rozkład w macierzy SCIRCUIT

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 uzyskuje wyższe wyniki przy wzroście parametru C, w niektórych przypadkach, będąc bardziej wydajnym, niż CSR, natomiast w innych przypadkach przy wartości C większej niż 4, wydajność spadała. Wydajność formatu ELL nie wyróżniała się ponad inne formaty, pomimo największego zużycia pamięci. W ponad połowie przypadków format ten był szybszy, niż COO i CSC, natomiast dla macierzy o małym zagęszczeniu lub małej liczbie elementów, format COO uzyskiwał najwyższą wydajność spośród wszystkich formatów. Pomimo mniejszego zużycia pamięci, format BSR uzyskiwał wyniki zawsze gorsze niż formaty SELL i CSR, poza macierzą DENSE2, która wypełnia całkowicie macierz blokami bez elementów zerowych, wraz ze wzrostem rozmiaru bloku wydajność spadała.



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:

$$k = \frac{1}{t \cdot s} \tag{5.1}$$

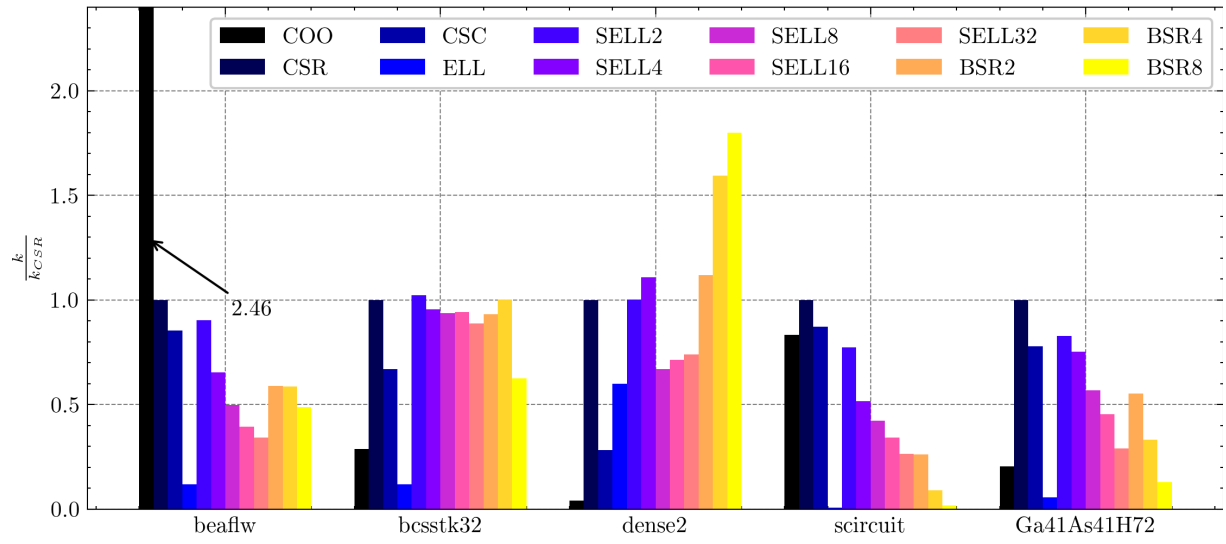
k – stała efektywności

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. Wskutek 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 dwóch przypadkach macierzy, format CSR jest najbardziej efektywnym formatem, dla macierzy *DENSE2* jest prawie dwa razy mniej efektywny, niż BSR dla rozmiaru bloku równego 8, dla macierzy *BEAFLW* jest drugi co do efektywności względem formatu COO, która jest 2.5 raza bardziej efektywna i dla BCSSTK32 jest prawie na równi z wariantami SELL i BSR.

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 [8] 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 atomowej 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 liczba elementów i niejednolite wartości macierzy GA41AS41H72 tylko potęgują ten efekt.

Obliczenia wymagane do wykonania operacji iloczynu macierz-wektor są proste, jedno mnożenie i jedno dodawanie dla każdego elementu macierzy, dlatego mikroprocesor przez większość czasu zawsze będzie czekać na pamięć potrzebną do wykonania zadanych operacji. Mierzony czas w większości algorytmów będzie jedynie czasem potrzebnym na załadowanie danej macierzy z pamięci głównej operacyjnej procesora graficznego do jego rejestrów. Przez to czynnikiem limitującym teoretyczne maksimum wydajności będzie maksymalna przepustowość pamięci głównej. Na wykresie 5.8 ukazano całkowitą przepustowość jako ilość pamięci potrzebnej do reprezentacji danej macierzy podzieloną przez średni czas wykonania algorytmu. Aby określić maksymalną przepustowość pamięci procesora graficznego należy znać częstotliwość taktowania pamięci, szerokość szyny pamięci i rodzaj pamięci, który jest zainstalowany, ponieważ różne typy mogą dostarczyć więcej niż jeden bit danych w każdym cyklu. Używając następującego równania:

$$T = Gf_m(w_b/8) \tag{5.3}$$

T – teoretyczna maksymalna przepustowość pamięci

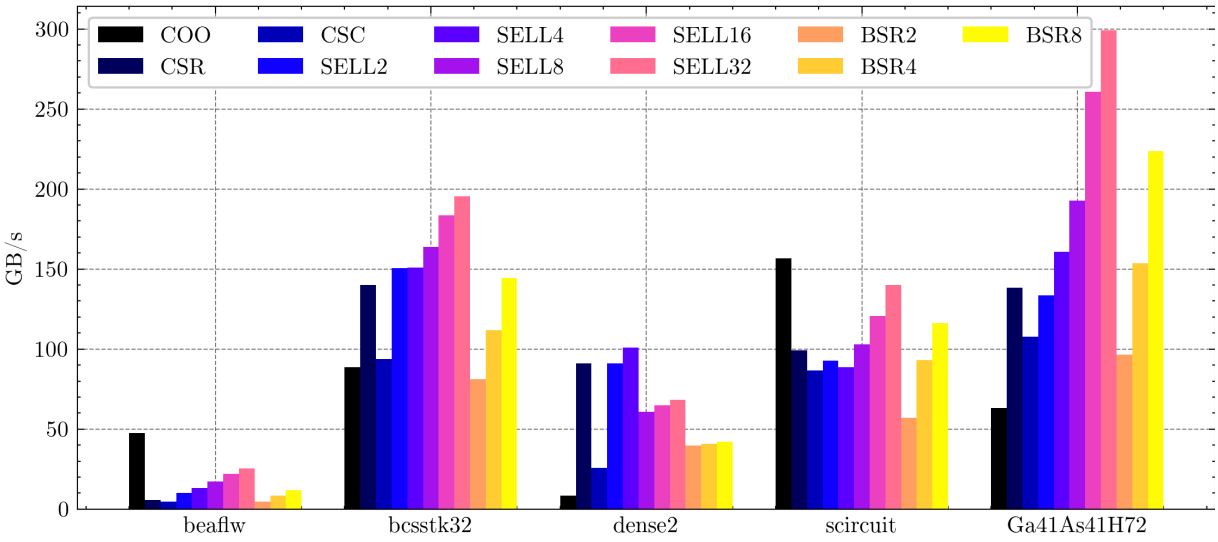
G – ilość bitów przesyłanych w każdym cyklu przez pamięć

f_m – częstotliwość taktowania pamięci

w_b – szerokość szyny w bitach

Na podstawie tego równania określono maksymalną przepustowość pamięci na wartość równą 336GB/s, algorytmy zbliżające się do tej wartości można uznać za optymalne, ponieważ wykorzystują całą dostarczoną im pamięć bezstratnie. W budowaniu wykresu 5.8 pominięto format ELL, gdyż trudno jest określić, jaka część pamięci została załadowana ze względu na fakt, iż obliczenia mogą zostać zakończone, jeżeli dane w danym rzędzie się skończyły.

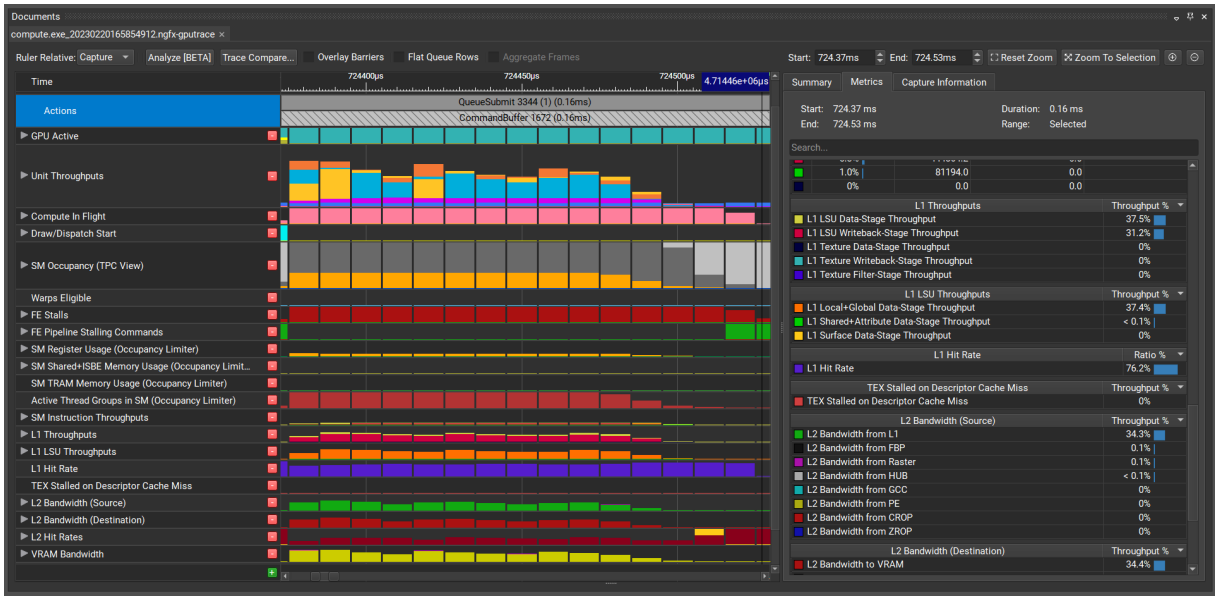
Na wykresie 5.8 można zauważyć, że formatem najbliżej teoretycznego limitu jest SELL, gdy $C = 32$ dla macierzy Ga41As41H72, ten rozmiar paska równy jest rozmiarowi *warp*'u dla procesorów



Rys. 5.8. Porównanie całkowitej przepustowości pamięci

graficznych NVIDIA. Wzrost przepustowości wraz ze wzrostem C dla formatu SELL, można wytłumaczyć łączeniem dostępu do pamięci pomiędzy wątkami wewnątrz pojedynczego *warp*'u. Jeżeli wszystkie wątki wykonują wczytywanie z kolejno po sobie występującej pamięci, kontroler może połączyć wszystkie zapytania w jedno, co zmniejsza narzut na szynę pamięci procesora graficznego. W prawie każdym przypadku CSR posiada drugą najwyższą przepustowość pamięci po SELL, nie przekracza ona natomiast nigdy progu 50%, co wskazuje na pole do optymalizacji, którą udało się osiągnąć w praktyce[9]. Format COO i CSC wykazują w większości najniższe przepustowości, najprawdopodobniej ze względu na kolizję w operacjach atomowych. BSR wykazuje niską przepustowość całkowitą na pewnych macierzach, natomiast wysoką na innych, format ten jest dobrym kandydatem na głębszą analizę wydajnościową przy wykorzystaniu takich narzędzi jak NVIDIA NSight.

NVIDIA NSight to zestaw narzędzi pozwalających na pracowanie z programami graficznymi, gdy uruchomione one zostaną na procesorze graficznym firmy NVIDIA. Dla *Vulkan* istnieje możliwość zebrania danych na temat wykorzystania procesora graficznego przy użyciu *NSight Graphics*[10], używając opcji *GPU Trace*, ta pozwoli nam zebrać takie dane jak te w ilustracji 5.9. Są to między innymi metryki odnoszące się do przepustowości pamięci w każdej z pamięci podręcznych, nietrafienia w pamięć podręczną, wykorzystanie rejestrów i wiele innych. Na ich podstawie możliwe jest dokonanie decyzji, które poprawią dany aspekt rozwiązania w taki sposób, aby rzecz będąca czynnikiem limitującym była zminimalizowana w najbardziej możliwy sposób.



Rys. 5.9. Metryki zebrane z uruchomienia shadera w NSight Graphics

6. Podsumowanie

Praca obrała za cel stworzenie programu komunikującego się z procesorem graficznym przy pomocy interfejsu Vulkan. Przy jego pomocy wykorzystano procesor graficzny do wykonywania arbitralnych obliczeń, które wykorzystano do obliczenia wyniku mnożenia macierzy rzadkiej z wektorem. Operację wykonano przy pomocy wielu różnych formatów przechowywania macierzy rzadkich na przykładowych macierzach rzadkich o szerokim spektrum cech charakterystycznych. Na podstawie uzyskanych wyników można stwierdzić, że format CSR najczęściej będzie optymalnym lub najbliżej optymalnego formatu przechowania macierzy rzadkiej, nie tylko w kontekście czystej wydajności obliczeniowej, lecz również z perspektywy efektywności zużycia pamięci. Wysoka 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 inżynieryjnej najlepsza decyzja, to ta podjęta w odpowiednim kontekście, dla najlepszych wyników należałoby, więc dokonać ewaluacji jak największej liczbie 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. Do takich zastosowań nawet mniej wydajny format BSR będzie w stanie dostarczyć najmniejsze zużycie 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 przyśpieszenia lub wykorzystanie procesora graficznego w urządzeniach mobilnych, aby umożliwić im lepsze reprezentowanie rozszerzonej rzeczywistości poprzez zwiększoną wydajność przetwarzania obrazów.

Bibliografia

[1] *AMD GPU ISA documentation*. *https : // gpuopen . com / amd - isa - documentation/*, dostęp 2023.01.01. GPUOpen.

[2] 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.

[3] M. J. Flynn. „Some Computer Organizations and Their Effectiveness". W: *IEEE Transactions on Computers* C-21 (9 wrz. 1972), s. 948–960.

[4] *Vulkan 1.3.221 Specification*. *https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html*, dostęp 2022.07.14. Khronos Group.

[5] *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.

[6] *Shaderc compiler*. *https://github.com/google/shaderc*, dostęp 2023.01.01. Google.

[7] *VulkanSDK*. *https://www.lunarg.com/vulkan-sdk/*, dostęp 2023.01.01. LunarG.

[8] IEEE Computer Society. „IEEE Standard for Floating-Point Arithmetic". W: *IEEE STD 754-2019* (2019), s. 1–84.

[9] Joseph L. Greathouse i Mayank Daga. „Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format". W: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, s. 769–780. DOI: *10.1109/SC.2014.68*.

[10] *NSight Graphics*. *https://developer.nvidia.com/nsight-graphics*, dostęp 2023.01.01. NVIDIA.