



UMCS

UNIwersytet Marii Curie-Skłodowskiej
w Lublinie

Wydział Matematyki, Fizyki i Informatyki

Kierunek: **informatyka**

Emil Szerafin

nr albumu: 307490

Wykorzystanie GPU do akceleracji generowania i przetwarzania dźwięku

Use of GPU in acceleration of
sound generation and processing

Praca licencjacka

napisana w Katedrze Oprogramowania Systemów Informatycznych
pod kierunkiem **dr hab. Przemysław Stpicyńskiego**

Lublin 2022

Spis treści

Wstęp	4
Cel i zakres pracy	5
1 Cyfrowa reprezentacja dźwięku	6
1.1 Format LPCM	6
1.2 Częstotliwość próbkowania	7
1.3 Rozdzielczość próbki	7
1.4 Rozmiar bufora	7
1.5 Ilość kanałów	10
1.6 Wpływ formatu na czas przetwarzania	11
1.7 Format MIDI	12
1.8 Format WAV	12
2 Technologia CUDA	13
2.1 Model programowania	13
2.2 Host i Device	14
2.3 Zastosowanie dla problematyki przetwarzania sygnałów	15
3 Projekt systemu	19
3.1 Założenia wstępne	19
3.2 Założenia implementacyjne	20
3.3 Architektura systemu	21
3.4 Komunikacja podsystemów	22
4 Implementacja CPU	24
4.1 Podsystem wejścia	24
4.2 Podsystem wyjścia	25
4.3 Podsystem komponentów	26
4.4 Podsystem danych statystycznych	26
4.5 Jądro systemu	26

4.5.1	Główna pętla programowa dla czasu rzeczywistego	27
4.5.2	Główna pętla programowa dla trybu offline	28
4.6	Interfejs użytkownika	31
5	Wybrane algorytmy przetwarzania dźwięku	32
6	Implementacja GPU	33
7	Wnioski	34
8	Możliwości rozwoju	35
8.1	Optymalizacja obecnego rozwiązania	35
8.1.1	Przystosowanie algorytmów do wykorzystania GPU	35
8.1.2	Grupowanie wywołań kerneli	35
8.1.3	Implementacja systemu hybrydowego	36
8.2	Wykorzystanie innych technologii	36
8.2.1	Wykorzystanie rdzeni ray-tracingu	36
8.2.2	Wykorzystanie rdzeni tensorowych	37
8.2.3	Przyszłe technologie	37
	Spis listingów	38
	Spis tabel	39
	Spis rysunków	40
	Bibliografia	41

Wstęp

Przetwarzanie sygnałów dźwiękowych jest dużym obszarem informatyki. Wraz z jego rozwojem, powstawaniem nowych algorytmów, wzrostem standardów dotyczących jakości dźwięku, a także zwiększaniem się ilości przetwarzanych danych, pojawia się coraz większe zapotrzebowanie na moc obliczeniową. W wielu przypadkach, przy zastosowaniu obecnych technik przetwarzania dźwięku, obliczenia w czasie rzeczywistym, stają się nieosiągalne, co mocno utrudnia pracę wielu osobom, a w niektórych przypadkach sprawia, że staje się ona nie możliwa. Wynika z tego, iż ciągły rozwój, a co za tym idzie, ciągły wzrost wymagań oprogramowania, wymusza na użytkownikach okresową wymianę podzespołów na coraz to wydajniejsze. Powstaje pytanie, czy nie istnieje żadne inne rozwiązanie, które pozwoliło by przynajmniej częściowo zniwelować ten problem. W obecny standard komputerów osobistych włączona jest obecność karty graficznej - odmiany procesora, przeznaczonego przede wszystkim do przetwarzania i generowania obrazu. Wraz z rozwojem tej technologii zaczęto zauważać, że karty graficzne posiadają ogromny potencjał obliczeniowy, który można by było wykorzystać w innych dziedzinach informatyki. Wraz z pojawieniem się technologii takich jak CUDA czy OpenCL, rozpowszechniono wykorzystanie karty graficznej do obliczeń ogólnego przeznaczenia. Dziedzina przetwarzania sygnałów dźwiękowych, ze względu na swoją naturę, jest jedną z dziedzin informatyki, która mogłaby skorzystać na wykorzystaniu tych technologii w celu przyspieszenia obliczeń i tym samym umożliwienia szerszemu gronu osób na korzystanie z bardziej zaawansowanych technik przetwarzania dźwięku.

Cel i zakres pracy

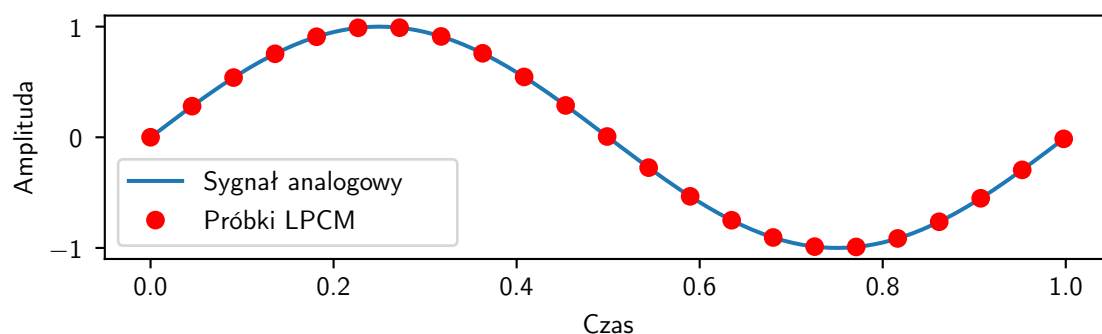
Celem niniejszej pracy jest zaprojektowanie oraz implementacja systemu, pozwalającego na przetestowanie możliwości wykorzystania karty graficznej względem CPU, do przetwarzania sygnałów dźwiękowych. System ma umożliwiać syntezywanie oraz przetwarzanie dźwięku w różnych powszechnie stosowanych formatach (biorąc pod uwagę częstotliwość próbkowania oraz wielkość próbki) zarówno w czasie rzeczywistym, jak i w trybie offline. Tryb offline pozwala na syntezę/przetworzenie sygnału w możliwie najkrótszym dla urządzenia czasie. Na ogół pozwala to na wykonanie skomplikowanych obliczeń, nawet w sytuacji kiedy nie jest możliwe przetworzenie danych w czasie rzeczywistym. W tym przypadku zostanie to wykorzystane w celu porównania czasów wykonania, co nie było by możliwe dla czasu rzeczywistego. System musi udostępniać możliwość gromadzenia danych statystycznych dotyczących wydajności swojej pracy, oraz umożliwiać na przeprowadzanie jednolitych testów wydajnościowych. W swojej uniwersalności powinien pozwalać na wprowadzanie modyfikacji w nieskomplikowany sposób, co umożliwi ekstensywne testowanie algorytmów oraz łączenie ich w bardziej złożonych scenariuszach użycia, jak również zmianę wykorzystanej platformy bez konieczności modyfikacji fundamentów jego działania. Skutkuje to eliminacją możliwie największej liczby czynników zewnętrznych, mogących zaburzyć pomiary wydajności. System powinien przetwarzać dźwięk w formacie PCM, który jest najbardziej powszechnie stosowanym formatem w przemyśle muzycznym oraz być w stanie odczytywać podstawowe sygnały zawarte w MIDI. Biorąc to wszystko pod uwagę, powstały system będzie stanowić podstawę podczas porównania możliwości CPU i GPU dla przedstawionego problemu. Praca ta nie skupia się nadto na optymalizacji powstałych rozwiązań, a na próbie weryfikacji omawianej koncepcji.

Rozdział 1

Cyfrowa reprezentacja dźwięku

1.1 Format LPCM

PCM - pulse-code modulation (modulacja impulsowa) jest popularną metodą reprezentacji sygnałów analogowych w postaci cyfrowej. Polega na próbkowaniu sygnału analogowego w regularnych odstępach czasu, a następnie kwantyzacji wartości próbek. Wartości mogą zostać reprezentowane przy użyciu szerokiej gamy metod. LPCM - linear pulse-code modulation (liniowa modulacja impulsowa) jest jednym z najprostszych sposobów reprezentacji zkwantyzowanego sygnału. Wykorzystuje on skalę liniową, w połączeniu ze stałym punktem odniesienia, względem którego mierzona jest próbka. Reprezentacja wyniku procesu konwersji pomiędzy sygnałem analogowym a sygnałem PCM przedstawia Rysunek 1.1.



Rysunek 1.1: Próbkowanie LPCM FLOAT32

1.2 Częstotliwość próbkowania

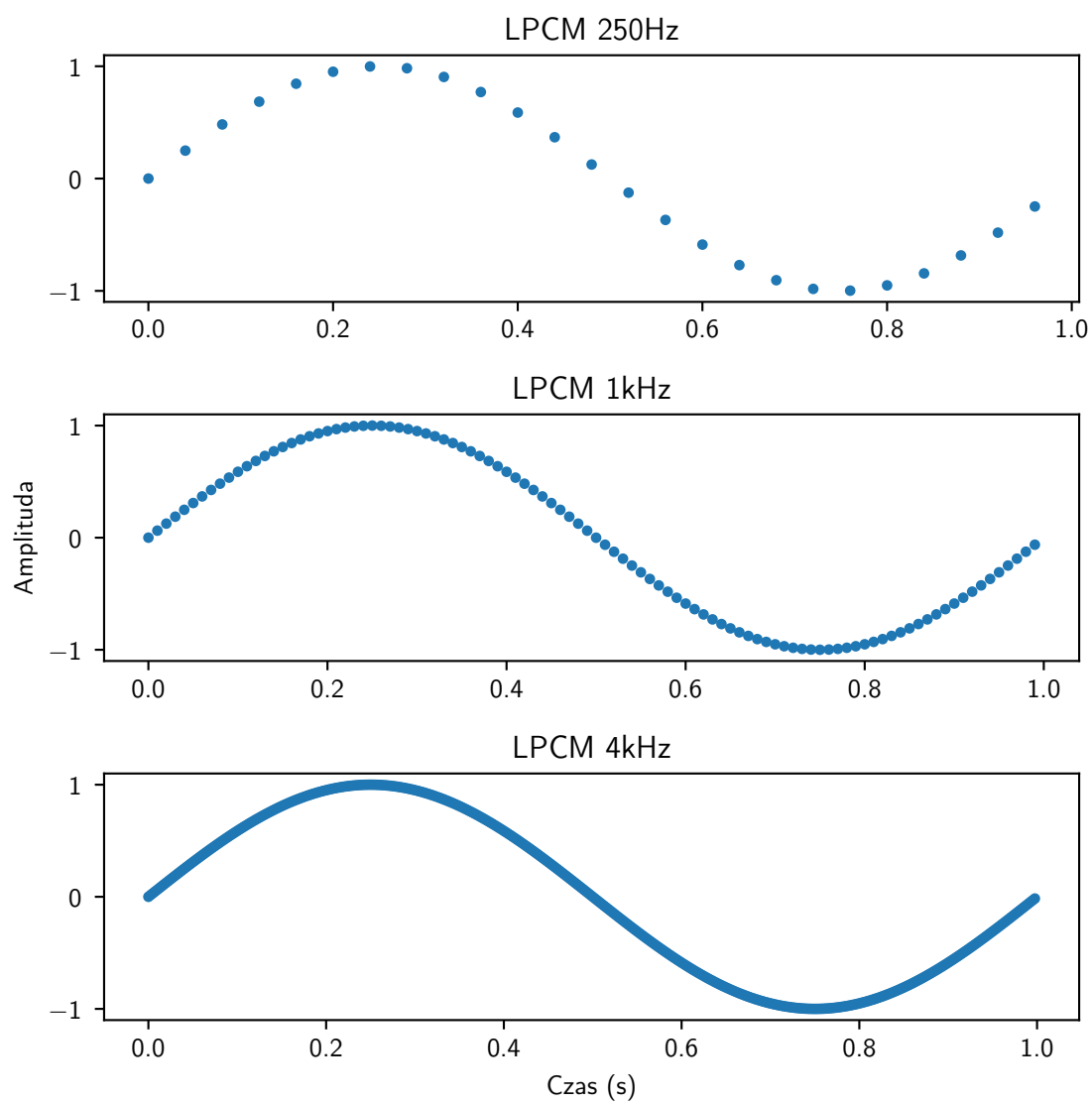
Jakość reprezentacji sygnału zależy do dwóch czynników: częstotliwości próbkowania oraz rozdzielczości pojedynczej próbki. Częstotliwość próbkowania ogranicza odstęp pomiędzy poszczególnymi próbkami. Wyższa częstotliwość próbkowania zmniejsza odstęp pomiędzy próbkami, co pozwala na dokładniejsze odwzorowanie sygnału w dziedzinie czasu, a jednocześnie zwiększa ilość danych, które muszą zostać później przetworzone. Konkretną wartość potrzebną do reprezentacji określonego sygnału można obliczyć przy użyciu twierdzenia Nyquista-Shannona. Głosi ono, iż aby zapisać sygnał o częstotliwości f Hz, należy próbować go co najmniej z częstotliwością $2f$ Hz. W praktyce jednak, aby uniknąć problemów związanych z aliasingiem, zaleca się próbkowanie z częstotliwością co najmniej $2.2f$ Hz. W przypadku słuchu ludzkiego najwyższy słyszalny dźwięk nie będzie przekraczać 20 kHz. Tak więc, aby zapisać sygnał o takiej częstotliwości, należy próbować go co najmniej z częstotliwością 44kHz, co pokrywa się z ogólnie przyjętym standardem stosowanym w przemyśle muzycznym. Przykład różnicy w częstotliwości próbkowania sygnału przedstawia Rysunek 1.2.

1.3 Rozdzielczość próbki

Rozdzielczość próbki, zwana również głębią bitowa / głębią próbki, określa ilość bitów, które są wykorzystywane do jej zapisu. Im większa rozdzielczość, tym mniejszy błąd kwantyzacji oraz większy rozmiar danych do przetworzenia. Determinuje to ilość poziomów kwantyzacji sygnału, która jest równa 2^n , gdzie n to ilość bitów wykorzystywanych do zapisu próbki. W przypadku sygnałów audio najpopularniejszymi wartościami dla głębi są: 16-bit, 24-bit oraz 32-bit. W zależności od sytuacji, wartości te pozwalają na rzetelny zapis sygnału. Wartość ta przeważnie nie posiada bezpośredniego wpływu na czas obliczeń, a jedynie może zwiększyć ilość pamięci koniecznej do zapisu sygnału. Przykład różnicy w rozdzielczości próbki przedstawia Rysunek 1.3.

1.4 Rozmiar bufora

W przeciwieństwie do metod przetwarzania sygnału w sposób analogowy, gdzie sygnał jest ciągły, w przypadku sygnałów cyfrowych, sygnał jest dzielony na porcje o określonej wielkości. Wielkość tej porcji (zazwyczaj jest to wielokrotność liczby 2) określa rozmiar bufora. Bufory pozwalają na przetworzenie większej ilości próbek jednocześnie, co pozwala na zwiększenie wydajności przetwarzania sygnału. Największe znaczenie ma to w przypadku przetwarzania sygnału w czasie rzeczywistym, gdzie każda próbka musi



Rysunek 1.2: Częstotliwość próbkowania LPCM

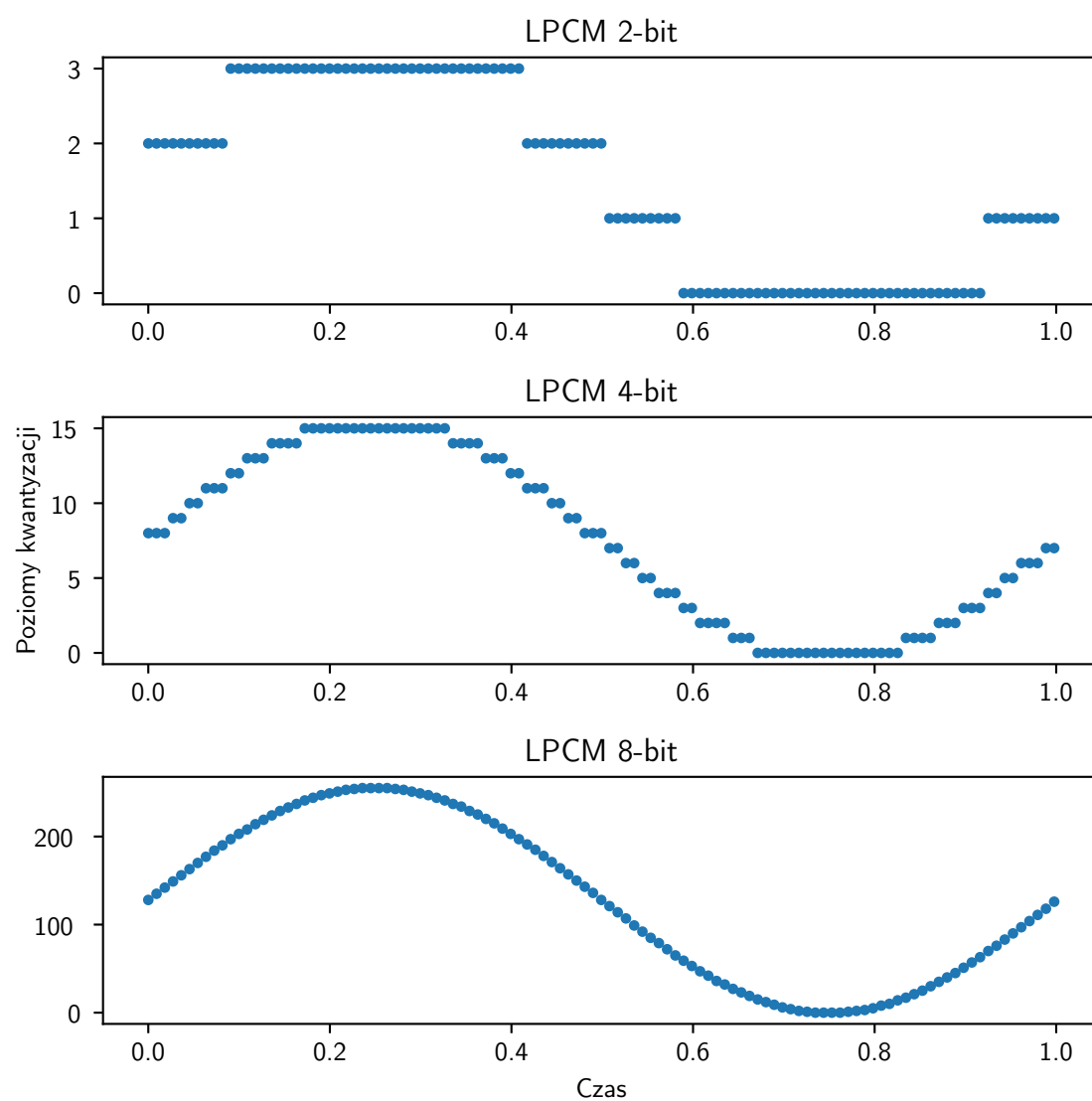
**Rysunek 1.3:** Rozdzielczość próbki LPCM

Tabela 1.1: Rozmiar bufora względem czasu przetworzenia

rozmiar bufora	44kHz	48kHz	96kHz	192kHz
32	0.73ms	0.67ms	0.33ms	0.17ms
64	1.45ms	1.33ms	0.67ms	0.33ms
128	2.91ms	2.67ms	1.33ms	0.67ms
256	5.82ms	5.33ms	2.67ms	1.33ms
512	11.64ms	10.67ms	5.33ms	2.67ms
1024	23.27ms	21.33ms	10.67ms	5.33ms
2048	46.55ms	42.67ms	21.33ms	10.67ms
4096	93.09ms	85.33ms	42.67ms	21.33ms
8192	186.18ms	170.67ms	85.33ms	42.67ms

zostać przetworzona w określonym czasie. Im większy rozmiar bufora, tym więcej czasu zostaje przeznaczone na jego przetworzenie. Wartość ta jest zależna od zastosowania, a także od dostępnych zasobów sprzętowych. W wielu przypadkach pożądany rozmiar bufora jest możliwie najmniejszy, tak aby zminimalizować opóźnienie w przetwarzaniu sygnału. Opóźnienie jest szczególnie niepożądanym efektem ubocznym na przykład w sytuacji w której muzyk potrzebuje słyszeć na bieżąco efekt swojego występu. Nie zawsze jest to jednak możliwe ze względu na ograniczenia sprzętowe. Krótszy czas przetwarzania pozostawia mniejszy margines błędu. Chwilowa niedostępność zasobu sprzętowego może spowodować, że próbka nie zostanie przetworzona w określonym czasie, co prowadzi do powstania niechcianych artefaktów w sygnale. Ilość czasu na przetworzenie bufora można wyliczyć na podstawie wzoru: $t = \frac{N}{f}$, gdzie t to czas przetworzenia bufora, N to ilość próbek w buforze, a f to częstotliwość próbkowania. Przykład zależności czasu przetworzenia od wielkości bufora oraz częstotliwości próbkowania przedstawia Tabela 1.1.

1.5 Ilość kanałów

Ilość kanałów określa ilość niezależnych ścieżek dźwiękowych, które mogą reprezentować sygnał w przestrzeni. Najpopularniejszymi wartościami są 1 (mono) oraz 2 (stereo). O ile istnieje więcej standardów reprezentacji sygnału, to nie są one często wykorzystywane. W przetwarzaniu sygnału zazwyczaj pracuje się używając sygnałów monofonicznych ostatecznie dążąc do uzyskania sygnału stereofonicznego. Wartość ta ma wpływ na ilość danych, które muszą zostać przetworzone, a także nierzadko na ich interpretację, co objawia się w postaci stosowania różnych algorytmów w zależności od

Tabela 1.2: Przesył danych dla różnych formatów przy wykorzystaniu float32

częstotliwość próbkowania	mono	stereo
44kHz	1.76KB/s	3.52KB/s
48kHz	1.92KB/s	3.84KB/s
96kHz	3.84KB/s	7.68KB/s
192kHz	7.68KB/s	15.36KB/s

ilości kanałów.

1.6 Wpływ formatu na czas przetwarzania

Format, w jakim przetwarzany jest sygnał, ma wpływ na jego objętość, a co za tym idzie na czas przetwarzania. Przedstawione powyżej wartości w różnym stopniu przyczyniają się do zwiększenia objętości danych. Najbardziej negatywny wpływ na prędkość obliczeń ma częstotliwość próbkowania. Jest to właściwość uznawana za najważniejszą w kontekście jakości sygnału, a jednocześnie jej zwiększenie pozytywnie wpływa na wielkość opóźnienia (Tabela 1.1). Następną cechą sygnału mającą znaczący wpływ na jego objętość ma ilość kanałów, która to w oczywisty sposób przemnaża liczbę próbek do przetworzenia. Rozdzielczość próbki również ma wpływ na objętość danych, jednak w mniejszym stopniu niż pozostałe cechy. W celu zachowania jakości i zmniejszenia znaczenia błędów kwantyzacji, w trakcie obróbki sygnału najczęściej stosuje się typu float32 i dopiero na sam koniec, w celu zmniejszenia powstałego pliku, konwertuje się sygnał do reprezentacji 24 lub 16 bitowej. Rozmiar bufora nie wpływa na objętość danych. Przeważnie ta wartość służy do uzyskania balansu pomiędzy stabilnością, a wielkością opóźnienia. Nierzadko ma wpływ na wydajność konkretnych algorytmów, jednak nie jest możliwe wyznaczenie stałego współczynnika, bez analizy danego przypadku. Tabela 1.2 przedstawia porównanie potrzebnej przepustowości B/s dla różnych popularnych zestawień parametrów pojedynczego strumienia LPCM. Należy zwrócić uwagę, iż w dziedzinie przetwarzania dźwięku sytuacje w których przetwarza się pojedynczy strumień audio są rzadkością. W większości przypadków przetwarzanych jest wiele strumieni jednocześnie, przy wykorzystaniu sekwencji wielu algorytmów, co zauważalnie zwiększa objętość danych do przetworzenia w raz z potrzebną na to mocą obliczeniową.

1.7 Format MIDI

MIDI - Musical Instrument Digital Interface (Cyfrowy Interfejs Instrumentów Muzycznych) jest standardem komunikacji pomiędzy różnymi urządzeniami muzycznymi, a także służy jako standard zapisu nutowego w postaci cyfrowej. Format ten nie zawiera informacji o dźwięku, a jedynie o sposobie jego syntezy. Dane są zapisane w postaci binarnej, posiadają one zauważalnie mniejsze rozmiary w pamięci w porównaniu z sygnałem PCM. Aby można było odtworzyć dany sygnał w postaci sygnału dźwiękowego, należy go wpierw przetworzyć przy użyciu np. synteźatora. Format MIDI posiada kilka wariantów, rozbudowujących i tak bogaty asortyment możliwości. Nawet w podstawowym wariantcie MIDI (General MIDI), pliki MIDI są w stanie przechowywać informacje na różne sposoby. Sposoby te nazywane są formatami. Na rzecz tej sprawy w zupełności wystarczającym będzie najprostszy w interpretacji format 0. W tym formacie, wszystkie ścieżki są zapisane w jednym strumieniu danych, zawierającym zdarzenia meta (meta events) oraz zdarzenia MIDI (MIDI events). Zdarzenia meta zawierają między innymi informacje o strukturze pliku, takie jak tempo, nazwa utworu, czy informacja o końcu pliku. Zdarzenia MIDI zawierają przede wszystkim informacje o dźwięku, takie jak: numer kanału, numer dźwięku (głosu), siła uderzenia, czy długość trwania dźwięku. Pojedyncze zdarzenie MIDI przekazują informacje takie jak np.: rozpoczęcie dźwięku, zakończenie dźwięku, zmiana siły uderzenia. Dopiero razem tworzą one pełny obraz informacji o dźwięku, który można syntezyować, a następnie odtworzyć.

1.8 Format WAV

WAV - Waveform Audio File Format (Format pliku dźwiękowego) jest formatem plików dźwiękowych, który przechowuje sygnał dźwiękowy w postaci LPCM. Format ten jest jednym z najpopularniejszych formatów plików dźwiękowych, ze względu na swoją prostotę oraz wsparcie w większości systemów operacyjnych. Plik WAV składa się z nagłówka, zawierającego informacje o pliku, takie jak: typ pliku, ilość kanałów, częstotliwość próbkowania, rozdzielczość próbki, oraz ilość próbek. Po nagłówku znajdują się dane, zawierające próbki sygnału dźwiękowego zapisanego przy użyciu PCM. Format ten jest stosunkowo prosty w implementacji, co sprawia, że jest często wykorzystywany w celach edukacyjnych, a także w przypadku prostych aplikacji, które nie wymagają zaawansowanych funkcji.

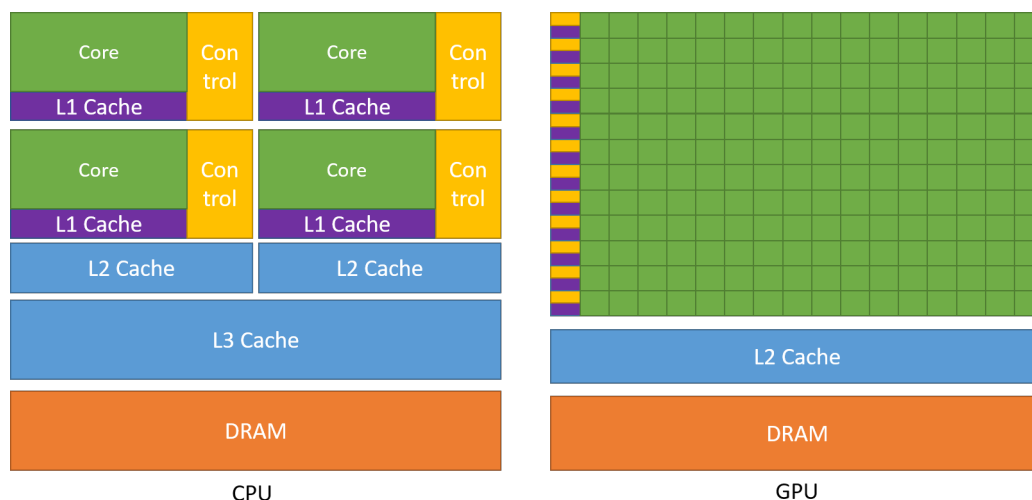
Rozdział 2

Technologia CUDA

CUDA (Compute Unified Device Architecture) to architektura stworzona przez firmę NVIDIA. Pozwala ona na programowanie odpowiednich kart graficznych w sposób umożliwiający wykonywanie obliczeń ogólnego przeznaczenia. Programy napisane z wykorzystaniem technologii CUDA zasadniczo różnią się od standardowych programów, które są wykonywane na zwykłych procesorach. Swoją charakterystyką przypominają one bardziej programy równoległe, które są wykonywane na klastrach obliczeniowych, niż standardowe programy sekwencyjne. CUDA pozwala na wykorzystanie architektury karty graficznej (Rysunek 2.1) w celu wykonania obliczeń w sposób zauważalnie odmienny niż w przypadku procesora.

2.1 Model programowania

W przeciwieństwie do programowania wielowątkowego na CPU, programowanie w technologii CUDA nie polega na zarządzaniu każdym wątkiem oddzielnie, a całym blokiem wątków naraz. Wątki w bloku są grupowane w tzw. *warp*, czyli grupę 32 wątków, które są wykonywane równoległe. Bloki wątków są grupowane w *grid* - siatkę bloków, która jest wykonywana na karcie graficznej. Model programowania w technologii CUDA jest zilustrowany na Rysunku 2.2. Pozwala to wykonywać jednocześnie tę samą operację na wielu elementach danych przy wykorzystaniu rdzeni CUDA. W ten sposób możliwe jest uzyskanie znacznie większej wydajności obliczeń niż w przypadku wykonywania ich sekwencyjnie na procesorze. Architektura GPU narzuca jednak pewne ograniczenia. Możliwość wykonania obliczeń na tysiącach wątków jednocześnie, osiągnięto kosztem możliwości pojedynczego wątku. Wątki na GPU nie mogą komunikować się bezpośrednio między sobą, co w połączeniu z ich ilością, wynosi problem konkurencyjności obliczeń na wyższy poziom abstrakcji niż w przypadku programowania wielowątkowego na CPU. Architektura sprawia również, że operacje warunkowe w kodzie CUDA mogą znacząco

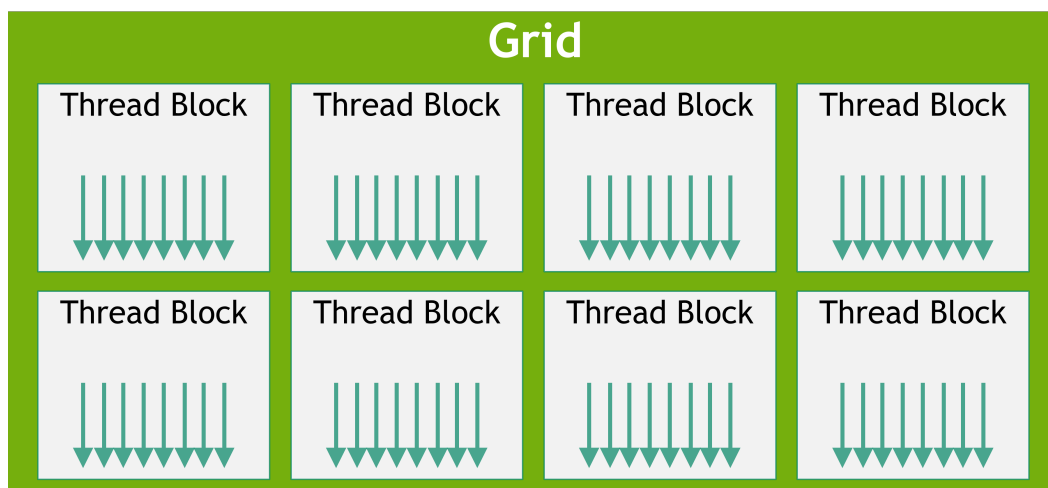


Rysunek 2.1: docs.nvidia.com - "GPU Devotes More Transistors to Data Processing", ilustracja różnicy w architekturze procesora i karty graficznej

obniżyć wydajność urządzenia. W przypadku rozgałęzienia w kodzie CUDA, jeżeli jeden wątek spełni warunek rozgałęzienia, wtedy wszystkie wątki wchodzące w dany warp będą musiały czekać na zakończenie obliczeń przez ten pojedynczy wątek. W pesymistycznym przypadku może to oznaczać, że w danym momencie dysponujemy jedynie 1/32 mocy obliczeniowej karty graficznej.

2.2 Host i Device

Urządzenie GPU można traktować jako oddzielny komputer, posiadający swoje własne podzespoły oraz zasoby. Technologia CUDA właśnie w ten sposób reprezentuje kompatybilną kartę graficzną. Interfejs programistyczny CUDA rozróżnia dwa środowiska: *host* oraz *device*. Host to komputer, na którym uruchamiany jest program, który wykorzystuje technologię CUDA. Device to karta graficzna, na której wykonywane są obliczenia. Najbardziej podstawowym schematem programu wykorzystującego tę technologię jest: przekazanie danych przygotowanych na urządzeniu host do urządzenia device, które wykonuje obliczenia. Po zakończeniu obliczeń, urządzenie host pobiera wyniki z GPU, gdzie możliwa jest ich interpretacja, zapis lub dalsze przetwarzanie. Można zauważyć, iż na moc obliczeniową, wynikającą z połączenia CPU oraz GPU, składa się również szybkość przesyłania danych między tymi dwoma urządzeniami. Tabela 2.1 przedstawia przeciętne możliwości w kategorii kopiowania danych zawartych w konkretnych typach pamięci urządzeń. Należy zauważyć ograniczenia jakie za sobą niesie wykorzystywanie szyny PCIe w przypadku intensywnej komunikacji pomiędzy CPU a GPU. Sugerowanym przez NVIDIA'e rozwiązaniem jest przekazanie jedynie



Rysunek 2.2: docs.nvidia.com - "Grid of Thread Blocks", ilustracja modelu programowania w technologii CUDA

niezbędnych danych w celu wykonania obliczeń na GPU, a następnie zwrócenie jedynie samego wyniku obliczeń. W ten sposób można przynajmniej częściowo zniwelować efekt wąskiego gardła, jaki może wynikać z przepustowości szyny PCIe. Alternatywnym rozwiązaniem jest wykonywanie obliczeń asynchronicznie względem przesyłu danych, co pozwala na wykonanie obu operacji w tym samym czasie i zlikwidowanie czasu oczekiwania na zakończenie przesyłu danych.

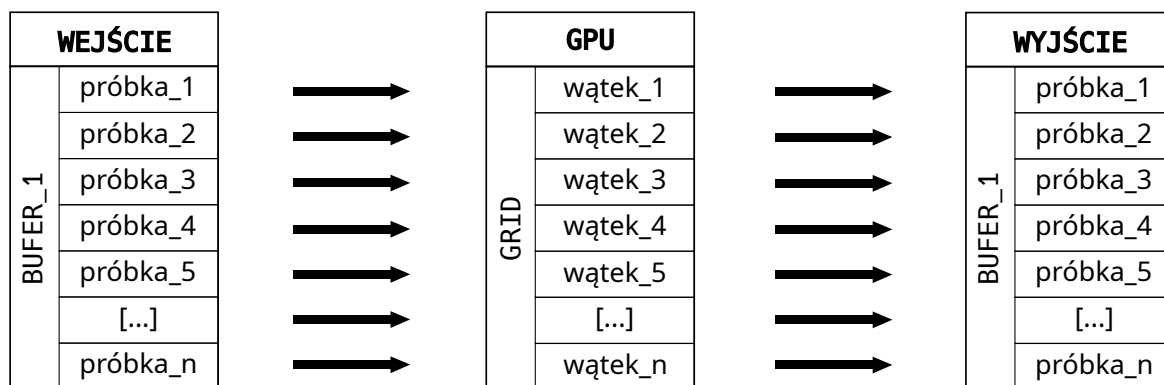
2.3 Zastosowanie dla problematyki przetwarzania sygnałów

Technologia CUDA jest obecnie wykorzystywana do przetwarzania sygnałów cyfrowych. Przy użyciu biblioteki cuFFT możliwe jest wykorzystanie mocy obliczeniowej karty graficznej do przetwarzania sygnałów w dziedzinie częstotliwości. Technologia ta nie jest jednak popularna w przypadku przetwarzania sygnałów dźwiękowych w przypadku w przemyśle rozrywkowym. Nie licząc pojedynczych prób w latach 2000 - 2010, które z powodu małej kompatybilności oprogramowania, zakończyły się niepowodzeniem, nie ma obecnie dostępnych narzędzi, wykorzystujących moc obliczeniową karty graficznej w przemyśle muzycznym. Technologia CUDA oraz karty firmy NVIDIA stały się dużo bardziej rozwinięte i popularne. W związku z tym ponowne podejście do tematu przetwarzania sygnałów dźwiękowych przy użyciu CUDA może przynieść pozytywne rezultaty. Natura popularnego w przemyśle muzycznym formatu PCM sprawia, że przetwarzanie sygnałów dźwiękowych jest zadaniem, które można wykonać przy użyciu algorytmów korzystających z równoległości. Najprostszym

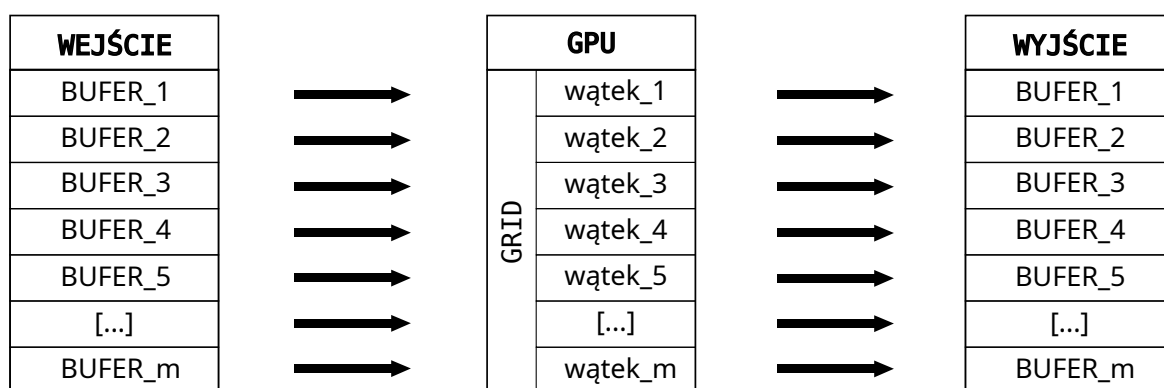
Tabela 2.1: Przeciętna przepustowość przesyłu danych dla danego standardu / urządzenia

urządzenie	typ	przepustowość
CPU	L1 cache	500GB/s - 1TB/s
	L2 cache	200GB/s - 1TB/s
	L3 cache	75GB/s - 400GB/s
RAM	DDR3	10GB/s - 20GB/s
	DDR4	17GB/s - 25GB/s
	DDR5	35GB/s - 50GB/s
GPU	L1 cache	1TB/s - 2TB/s
	L2 cache	500GB/s - 1TB/s
	DRAM	200GB/s - 800GB/s
PCIE	3.0	1GB/s - 16GB/s
	4.0	2GB/s - 32GB/s
	5.0	4GB/s - 64GB/s

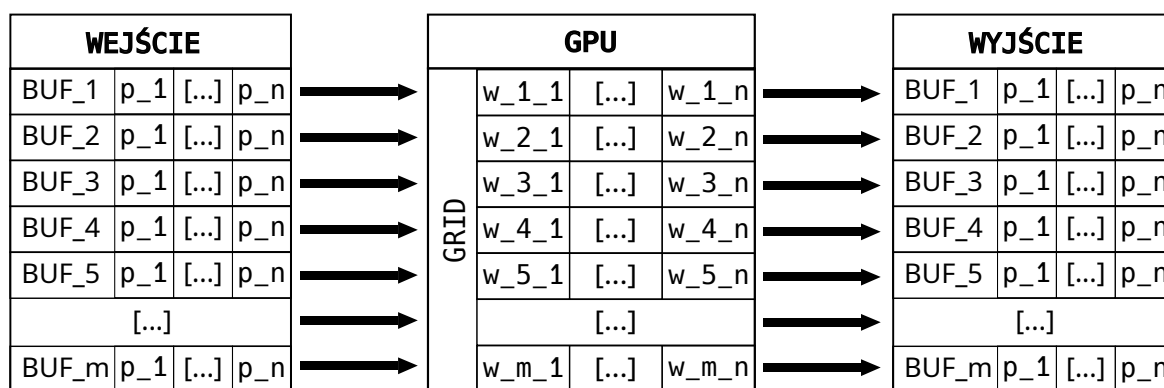
przykładem jest zastosowanie wykorzystanie n wątków GPU do przetworzenia n próbek sygnału zawartych w buforze. Sytuacja przedstawiona na Rysunku 2.3 ilustruje metodę, którą można by bez problemu wykorzystać przy zrównoleglaniu nieskomplikowanych algorytmów. W przypadku algorytmów bardziej złożonych lub wymagających sekwencyjnego przetwarzania danych dla kolejnych próbek, może się stać niemożliwym wykorzystanie tego podejścia. Rozwiązaniem tego problemu może być zastosowanie m wątków do przetworzenia m buforów w sposób liniowy, zakładając, iż dany algorytm zostaje wywołany jednocześnie m-krotnie. Przykład takiego podejścia przedstawiono na Rysunku 2.4. O ile te rozwiązanie nie jest tak samo wydajne, co pierwsze z przedstawionych, nie necessarily musi implementować cały algorytm, a jedynie jego problematyczną część. Idealną sytuacją byłoby wywołanie algorytmu niesekwencyjnego m-krotnie. Taki przypadek pozwalałby na wykorzystanie pełni mocy obliczeniowej karty graficznej. Przykład takiego podejścia przedstawiono na Rysunku 2.5. Dobór podejścia może okazać się kluczowy dla uzyskania zadowalających wyników. Niewykluczone, iż w wielu przypadkach konieczne będzie zastosowanie hybrydowego podejścia, łączącego przedstawione metody.



Rysunek 2.3: Wykorzystanie n wątków do przetworzenia n próbek



Rysunek 2.4: Wykorzystanie m wątków do przetworzenia m buforów w sposób liniowy

Rysunek 2.5: Wykorzystanie $m \cdot n$ wątków do przetworzenia m buforów po n próbek

Listing 2.1: Przykładowe wywołanie kernela CUDA sumującego dwie tablice w języku C++

```
1
2 // kernel sumujący dwie tablice
3 __global__ void sum(float* arrayA, float* arrayB, int size){
4     // obliczenie indeksu wątku
5     int i = blockIdx.x * blockDim.x + threadIdx.x;
6
7     // sumowanie elementów tablicy
8     if (i < size){
9         arrayB[i] = arrayA[i] + arrayB[i];
10    }
11 }
12
13 int main(){
14     // alokacja pamięci na GPU
15     float* arrayA, *arrayB;
16     int size = 1000;
17     cudaMalloc(&arrayA, size * sizeof(float));
18     cudaMalloc(&arrayB, size * sizeof(float));
19
20     // ustawienie rozmiaru bloku i gridu
21     int blockSize = 256;
22     int numBlocks = (size + blockSize - 1) / blockSize;
23
24     // wywołanie kernela
25     sum<<<numBlocks, blockSize>>>(arrayA, arrayB, size);
26
27     // zwolnienie pamięci GPU
28     cudaFree(arrayA);
29     cudaFree(arrayB);
30
31     return 0;
32 }
```

Rozdział 3

Projekt systemu

3.1 Założenia wstępne

Aby zrealizować projekt w sposób składny i efektywny, należy zdefiniować cele i wymagania, które system ma spełniać. Będą one stanowiły podstawę do dalszych prac projektowych.

System musi spełniać następujące wymagania:

1. Możliwość przetwarzania dźwięku w czasie rzeczywistym (online).
2. Możliwość przetwarzania dźwięku w trybie offline (możliwe najszybciej bez jednoczesnego odtwarzania).
3. Pierwsza wersja systemu ma przetwarzać dźwięk jedynie przy użyciu CPU.
4. Łatwy w rozbudowie o przetwarzanie dźwięku przy użyciu GPU, przy zachowaniu minimalnych zmian w kodzie.
5. Uniwersalny podsystem komponentów odpowiedzialnych za przetwarzanie dźwięku - komponent implementuje pojedynczy algorytm przetwarzania dźwięku.
6. Uniwersalny podsystem strumieni dźwięku - strumień może zostać przetworzony przez dowolną liczbę komponentów oraz dowolnie łączony z innymi strumieniami.
7. Możliwość odtwarzania strumienia wyjściowego.
8. Możliwość zapisu strumienia wyjściowego w przypadku przetwarzania offline.
9. Możliwość pełnej kontroli systemu, bez konieczności ingerencji w kod źródłowy.
10. Możliwość syntezy dźwięku - początek strumienia dźwięku.

11. Obsługa urządzeń wejścia, przy pomocy których są generowane dane potrzebne do syntezy dźwięku.
12. Obsługa plików wejściowych, stosowanych zamiennie z urządzeniami wejścia.
13. Podsystem zarządzania plikami wejściowymi - możliwość dodawania, usuwania, rozpoczęcia/wstrzymania odtwarzania, przewijania pliku.
14. Możliwość zmiany formatu strumieni dźwiękowych - zmiana częstotliwości próbkowania, rozmiaru bufora.
15. Podsystem gromadzenia danych statystycznych dotyczących wydajności systemu.
16. Funkcja rzetelnego testowania wydajności systemu - brak czynnika ludzkiego w procesie pomiaru.

3.2 Założenia implementacyjne

Przed rozpoczęciem procesu implementacji należy wpierw określić wykorzystane technologie oraz standardy.

Zbiór założeń implementacyjnych:

1. System zostanie zaimplementowany w języku C++.
2. Elementy systemu odpowiedzialne za przetwarzanie dźwięku zostaną ponownie zaimplementowane na GPU przy użyciu technologii CUDA.
3. Jako formatu przetwarzanego sygnału zostanie użyty format LPCM.
4. Próbkki LPCM będą reprezentowane jako liczby zmiennoprzecinkowe 32-bitowe (typ float w języku C++).
5. Bufory audio będą zawsze grupowane po dwa - sygnał stereo.
6. Interfejs użytkownika zostanie zaimplementowany jako aplikacja konsolowa.
7. Jako urządzenia wejścia posłużą urządzenia MIDI oraz klawiatura komputerowa.
8. Jako format wejściowy zostaną użyte pliki .mid - pliki zapisane w formacie MIDI.
9. Jako format wyjściowy danych dźwiękowych zostaną użyte pliki .wav.
10. W celu odtwarzania dźwięku zostanie użyty system PulseAudio.

11. System nie będzie korzystał z wielowątkowości (w przypadku przetwarzania na CPU) lub asynchronicznych wywołań kerneli (w przypadku przetwarzania na GPU) w celu przetwarzania dźwięku.

3.3 Architektura systemu

System będzie się składał z wielu podsystemów, posiadających odmienne zbiory obowiązków, połączonych ze sobą klasą implementującą wzorzec mediatora oraz fasady.

Dziedziny jakimi będą zajmować się podsystemy:

1. Wejście:

- zarządza urządzeniami wejścia,
- zarządza plikami wejściowymi,
- interpretuje strumień wejściowy,
- zarządza zbiorem syntezyatorów,
- generuje strumień dźwiękowy przy wykorzystaniu syntezyatorów oraz strumieni wejściowych.

2. Wyjście:

- zarządza komunikacją programu z serwerem PulseAudio,
- konwertuje otrzymany strumień dźwiękowy na wybrany przez użytkownika format,
- odtwarza otrzymany strumień dźwiękowy,
- pozwala na zapis otrzymanego strumienia dźwiękowego do pliku.

3. Przetwarzanie sygnału:

- zarządza komponentami implementującymi algorytmy przetwarzania sygnału dźwiękowego,
- pozwala na łączenie strumieni audio,
- pozwala na dodawanie komponentów do strumieni audio,
- generuje listę instrukcji na bazie obecnej konfiguracji w kolejności, w której należy je wykonać w celu uzyskania zdefiniowanego przez użytkownika efektu.

4. Dane statystyczne:

- wykonuje i gromadzi pomiary czasu poszczególnych części pętli programowej,
- pozwala na zapis zgromadzonych i wstępnie przetworzonych danych do pliku.

5. Interface użytkownika:

- zarządza operacjami wejścia-wyjścia w kontekście terminalu,
- wykorzystuje fasadę systemu w celu wykonywania na nim operacji,
- rozpoznaje słowa-klucze (komendy) podawane przez użytkownika i wywołuje odpowiednie metody systemu,
- wykonuje pliki tekstowe zawierające zestawy komend, w roli automatyzacji procesu konfiguracji systemu.

3.4 Komunikacja podsystemów

W związku z koniecznością komunikacji pomiędzy podsystemami oraz wewnątrz klas wchodzących w ich skład, należy zdefiniować ustandaryzowany format dla każdego typu przekazywanych i przechowywanych informacji, z uwzględnieniem koniecznej w przyszłości reimplementacji znaczącej części systemu przy użyciu technologii CUDA.

1. **audioFormatInfo** - struktura przechowująca informacje dotyczące formatu przetwarzanego i odtwarzanego/zapisanego dźwięku takie jak:

- częstotliwość próbkowania,
- rozmiar bufora,
- liczba kanałów wyjściowych,
- głębina bitowa próbki.

2. **IDManager** - klasa-kontener przechowująca obiekty wskazanego typu i nadająca im unikalne identyfikatory, pomaga w zarządzaniu obiektami w systemie, nadając im możliwie najkrótsze identyfikatory, co ułatwia posługiwanie się nimi użytkownikowi.

3. Pozwala na operacje takie jak:

- dodawanie nowego obiektu i jednocześnie przydzielanie mu ID,
- usuwanie obiektu o wskazanym ID,
- pobieranie obiektu o wskazanym ID,
- pobieranie tablicy wszystkich obiektów,

- sprawdzanie czy obiekt o wskazanym ID istnieje.
4. **audioBuffer** - struktura przechowująca próbki dźwiękowe w postaci tablicy bajtów, wykorzystywana w komunikacji pomiędzy systemem a serwerem PulseAudio. Przechowuje:
- tablicę bajtów zawierającą próbki dźwiękowe,
 - ilość próbek w tablicy,
 - wielkość pojedynczej próbki w bajtach.
5. **pipelineAudioBuffer** - struktura przechowująca dwie tablice typu float próbek dźwiękowych, wykorzystywana w wewnętrznej komunikacji pomiędzy komponentami przetwarzania dźwięku, systemem wyjścia oraz systemem wejścia. Przechowuje:
- dwie tablice typu float zawierające próbki dźwiękowe,
 - ilość próbek w tablicach.
6. **audioBufferQueue** - struktura przechowująca pipelineAudioBuffer wraz z informacją o kolejności wykonywania na nim operacji oraz informacja o jego pochodzeniu. Przechowuje:
- obiekt pipelineAudioBuffer,
 - typ podsystemu, który utworzył buffer,
 - ID elementu podsystemu który utworzył buffer,
 - listę ID komponentów przetwarzania dźwięku, które mają zostać na nim wykonane.
7. **keyboardTransferBuffer** - struktura przechowująca informacje o stanie poszczególnych klawiszy, na osi czasu, urządzenia wejściowego. Przechowuje:
- dwuwymiarową tablicę bajtów (gdzie pierwszy wymiar stanowi indeks klawisza, a drugi oś czasu) zawierających informację o stanie klawiszy, wyróżniając 128 stanów zgodnie ze standardem MIDI,
 - jednowymiarową tablicę (wielkości równej ilości klawiszy) zawierającą informację o ostatnim stanie z poprzedniej zawartości bufora,
 - ilość obsługiwanych klawiszy,
 - wielkość bufora w dziedzinie czasu.

Rozdział 4

Implementacja CPU

Implementacja systemu CPU jest pierwszym etapem realizacji projektu. System ten spełnia wymagania postawione w założeniach wstępnych, implementując wiele podsystemów, które są odpowiedzialne za różne aspekty jego funkcjonowania. Poniżej zostaną omówione poszczególne podsystemy, ich zadania oraz najważniejsze klasy, które je implementują.

4.1 Podsystem wejścia

Klasa `pipeline::Input` zarządza urządzeniami wejścia, odtwarzaniem plików MIDI, zbiorem syntezytorów oraz generowaniem próbek dźwiękowych przy ich użyciu. Podsystem operuje między innymi na obiektach następujących klas:

1. **AKeyboardRecorder** - klasa abstrakcyjna reprezentująca urządzenie wejścia, jej implementacje zapisują stan urządzenia na przestrzeni czasu wykorzystując obiekty klas implementujących interfejs `IKeyboardDoubleBuffer`. Klasami dziedziczącymi po `AKeyboardRecorder` są:
 - **KeyboardRecorder_MidiFile** - klasa implementująca odczyt plików MIDI przy wykorzystaniu klasy `MidiFileReader`.
 - **KeyboardRecorder_DevInput** - klasa implementująca odczyt z klawiatury komputerowej przy wykorzystaniu strumieni systemowych znajdujących się w `/dev/input/` w systemie Linux.
 - **KeyboardRecorder_DevSnd** - klasa implementująca odczyt z urządzenia MIDI przy wykorzystaniu strumieni systemowych znajdujących się w `/dev/snd/` w systemie Linux oraz klasy `MidiMessageInterpreter`.
2. **KeyboradManager** - klasa-kontener dziedzicząca po `IDManager`, przechowująca i zarządzająca obiektami klas implementujących `AKeyboardRecorder`.

3. **MidiReaderManager** - klasa zarządzająca odczytem plików MIDI, operuje na obiektach klasy **KeyboardRecorder_MidiFile**.
4. **Synthesizer** - klasa reprezentująca syntezytor, przy wykorzystaniu wzorca strategii pozwala na wybór różnych algorytmów generowania dźwięku. Wypełnia bufor audio na podstawie ustawień użytkownika oraz struktur **keyboardTransferBuffer**. Każdy syntezytor posiada unikalny identyfikator, nadawany przez obiekt klasy **IDManager**, dzięki któremu przyporządkowuje się mu strumień dźwiękowy w postaci struktury **pipelineAudioBuffer**.
5. **MidiFileReader** - klasa zarządzająca odczytem plików MIDI. Jej głównym zadaniem jest obliczenie konkretnego punktu w czasie w którym należy odczytać kolejną wiadomość MIDI na podstawie konfiguracji zawartej w pliku oraz ustawień systemowych. Klasa ta wykorzystuje obiekty klasy **MidiMessageInterpreter** do interpretacji odczytanych wiadomości.

4.2 Podsystem wyjścia

Klasa **pipeline::Output** zarządza komunikacją programu z serwerem **PulseAudio**, konwertuje otrzymany strumień dźwiękowy na wybrany przez użytkownika format oraz pozwala na zapis otrzymanego strumienia dźwiękowego do pliku. Podsystem ten operuje na obiektach następujących klas:

1. **OutputStreamPulseAudio** - klasa implementująca interfejs **IOutputStream**. Klasa ta wykorzystuje bibliotekę **libpulse** do nawiązania połączenia z serwerem **PulseAudio** oraz przesyłania do niego buforów dźwiękowych.
2. **AudioRecorder** - klasa pozwalająca na zapis otrzymanego bufora dźwiękowego do pliku o wskazanej nazwie z rozszerzeniem **.wav**.
3. **AudioBuffer** - struktura reprezentująca bufor dźwiękowy. Przede wszystkim przechowuje próbki dźwiękowe w postaci tablicy bajtów.
4. **ABufferConverter** - klasa abstrakcyjna, której implementacje, dzięki zastosowaniu wzorca projektowego strategii, pozwalają na konwersję bufora dźwiękowego przedstawianego za pomocą struktury **pipelineAudioBuffer** do struktury **AudioBuffer** w wybrany formacie.

4.3 Podsystem komponentów

Podsystem komponentów jest zarządzany równocześnie za pomocą klasy `pipeline::ComponentManager`, jak i klasy `pipeline::ExecutionQueue`. Obie klasy wykorzystują obiekty i zarządzają obiektami tych samych klas w różnych przypadkach. Klasa `pipeline::ComponentManager`, przy użyciu klasy `IDManager`, zarządza komponentami dziedziczącymi po klasie abstrakcyjnej `AComponent`, które implementują algorytmy przetwarzania sygnału dźwiękowego. Klasa `pipeline::ExecutionQueue` zarządza kolejnością wykonywania operacji na strumieniach audio, które są reprezentowane przez obiekty klasy `audioBufferQueue`. Podsystem komponentów operuje między innymi na obiektach następujących klas:

1. `pipelineAudioBuffer` - struktura reprezentująca bufor dźwiękowy, który jest przetwarzany przez komponenty przetwarzania sygnału dźwiękowego.
2. `audioBufferQueue` - struktura reprezentująca strumień audio, który jest przetwarzany przez komponenty przetwarzania sygnału dźwiękowego. Klasa ta przechowuje referencje na obiekty klasy `pipelineAudioBuffer` oraz identyfikatory syntezyatorów, które są podłączone do strumienia.
3. `AComponent` - klasa abstrakcyjna reprezentująca komponent przetwarzania sygnału dźwiękowego. Dziedziczą po niej klasy implementujące konkretne algorytmy przetwarzania sygnału dźwiękowego. Są one dokładniej omówione w następnym rozdziale.
4. `AAdvancedComponent` - klasa abstrakcyjna dziedzicząca po `AComponent`, reprezentująca komponent przetwarzania sygnału dźwiękowego, który posiada swój własny strumień wyjściowy w postaci obiektu klasy `audioBufferQueue`.

4.4 Podsystem danych statystycznych

4.5 Jądro systemu

Klasą z której można zarządzać całym systemem jest klasa `AudioPipelineManager`. Pełni ona rolę fasady dla całego systemu, udostępniając wszelkie jego funkcjonalności rozproszone w różnych podsystemach. Jako, że implementuje ona główną pętlę programową oraz zarządza wątkiem, w którym się ta pętla wykonuje, klasa ta również pełni rolę jądra całego systemu.

4.5.1 Główna pętla programowa dla czasu rzeczywistego

Główna pętla programowa systemu jest pętlą nieskończoną, w której wykonywane są wszystkie operacje związane z przetwarzaniem dźwięku. Pętla uruchamiana jest przy pomocy metody `pipelineThreadFunction(AudioPipelineManager::start)`, w której sprawdzane są warunki niezbędne do rozpoczęcia pracy systemu. Sprawdzane jest: obecny stan flagi `running`, obecność bufora wyjściowego, poprawność uruchomienia wszystkich wątków obsługujących urządzenia wejściowe. Następnie budowana jest kolejka operacji do wykonania, na podstawie grafu połączeń między komponentami przetwarzania dźwięku, a strumieniami audio oraz jednocześnie sprawdzana jest poprawność grafu. Na samym końcu wywoływana jest odpowiednia metoda, zawierająca pętlę programową, w zależności od ustawień systemu (możliwe jest uruchomienie pętli pozwalającej na wyświetlanie spektrum dźwięku, generowanego przy użyciu FFT - Fast Fourier Transform). Główna pętla programowa systemu dla czasu rzeczywistego została przedstawiona oraz dokładnie opisana na listingu 4.1.

Listing 4.1: Główna pętla programowa systemu dla przetwarzania dźwięku w czasie rzeczywistym

```
1 void AudioPipelineManager::pipelineThreadFunction(){
2     // Ustawienie flagi określającej stan systemu
3     running = true;
4
5     // Ustawienie czasu trwania odtworzenia próbki w mikrosekundach
6     ulong sampleTimeLength =
7         audioInfo.sampleSize
8         * long(1000000)
9         / audioInfo.sampleRate;
10
11     // Pobranie referencji na wektor zawierający listę operacji
12     // do wykonania na strumieniach audio
13     const std::vector<audioBufferQueue*>& backwardsExecution =
14         executionQueue.getQueue();
15
16     // Podmiana buforów do których piszą wątki
17     // obsługujące urządzenia wejściowe
18     input.swapActiveBuffers();
19
20     // Ustawienie czasu następnego rozpoczęcia pętli
21     ulong nextLoop =
22         input.getActivationTimestamp()
23         + sampleTimeLength;
24
25     // Inicjalizacja systemu statystyk
```

```
26     statisticsService->firstInvocation();
27
28     // Utworzenie obiektu funkcji wywoływanego
29     // na końcu każdej iteracji pętli
30     std::function<void()> loopWorkEnd = [this]() {
31         this->statisticsService->loopWorkEnd();
32     };
33
34     // Główna pętla programowa
35     while (running){
36         // Oczekiwanie na czas rozpoczęcia pętli
37         std::this_thread::sleep_until(
38             std::chrono::time_point<std::chrono::system_clock>
39                 (std::chrono::nanoseconds((nextLoop)*1000)));
40
41         // Pomiar czasu rozpoczęcia iteracji pętli
42         statisticsService->loopStart();
43
44         // Obliczenie czasu rozpoczęcia następnej iteracji
45         nextLoop += sampleTimeLength;
46
47         // Skopiowanie oraz wstępne przetworzenie danych wejściowych
48         input.cycleBuffers();
49
50         // Wygenerowanie próbek dźwiękowych przy użyciu syntezy
51         // podłączonych do urządzeń wejściowych
52         input.generateSamples(executionQueue.getConnectedSynthIDs());
53
54         // Wykonanie operacji na strumieniach audio
55         for (int i = backwardsExecution.size() - 1; i >= 0; i--){
56             component.applyEffects(backwardsExecution[i]);
57         }
58
59         // Odtworzenie dźwięku i dokonanie pomiaru czasu przed
60         // przekazaniem bufora wyjściowego do serwera PulseAudio
61         output.play(&outputBuffer->buffer, loopWorkEnd);
62     }
63 }
```

4.5.2 Główna pętla programowa dla trybu offline

Główna pętla programowa systemu dla trybu offline różni się od pętli dla czasu rzeczywistego tym, że nie jest ona wykonywana w osobnym wątku, przez co jest to wywołanie blokujące. Pętla ta korzysta jedynie z plików MIDI zastępujących urządzenia

wejściowe. Wygenerowany dźwięk nie jest odtwarzany, a zapisywany do pliku o wskazanej nazwie. Wynikiem tego jest brak instrukcji usypiającej wątek w celu oczekiwania na zakończenie odtwarzania wygenerowanego bufora wyjściowego. Pozwala to na pomiar czasu potrzebnego na wykonanie zadanych operacji. Główna pętla programowa systemu dla trybu offline została przedstawiona oraz dokładnie opisana na listingu 4.2.

Listing 4.2: Główna pętla programowa systemu dla przetwarzania dźwięku w trybie offline

```

1 char AudioPipelineManager::recordMidiFilesOffline(
2     std::string fileName, double& time){
3
4     // Sprawdzenie czy system jest uruchomiony
5     if (running == true){
6         std::fprintf(stderr, "ERR: AudioPipelineManager::
7             recordMidiFilesOffline PIPELINE IS RUNNING\n");
8         return -1;
9     }
10
11    // Budowanie kolejki operacji do wykonania
12    executionQueue.build(
13        componentQueues, outputBuffer, component.components);
14
15    // Sprawdzenie poprawności kolejki operacji
16    if (executionQueue.error() != 0){
17        std::fprintf(stderr, "ERR: AudioPipelineManager::
18            recordMidiFilesOffline PIPELINE IS NOT VALID\n");
19        return -2;
20    }
21
22    // Pobranie referencji na wektor zawierający listę operacji
23    const std::vector<audioBufferQueue*>& backwardsExecution =
24        executionQueue.getQueue();
25
26    // Wyczyszczenie buforów wejściowych oraz buforów komponentów
27    input.clearBuffers();
28    component.clearBuffers();
29
30    // Ustawienie stanu obiektu klasy pipeline::Output
31    // potrzebnego do zapisu wyniku do pliku
32    if (output.startRecording(fileName)){
33        std::fprintf(stderr, "ERR: AudioPipelineManager::
34            recordMidiFilesOffline COULD NOT START RECORDING\n");
35        return -3;
36    }
37
38    std::chrono::_V2::system_clock::time_point timeStart;

```

```
39     std::chrono::_V2::system_clock::time_point timeEnd;
40     time = 0.0;
41
42     double swapTime;
43     double conversionTime;
44
45     // Ustawienie strumieni plików MIDI na ich początek
46     midiReaderManager.rewind();
47
48     // Ustawienie stanu obiektów klasy KeyboardRecorder_MidiFile
49     // potrzebnego do odtworzenia plików MIDI
50     midiReaderManager.play();
51
52     // Główna pętla programowa, której warunkiem zakończenia
53     // jest wyzerowanie się licznika odtwarzanych plików MIDI
54     while (midiReaderManager.getPlayCounter() > 0){
55
56         // Skopiowanie, wstępne przetworzenie danych wejściowych
57         // oraz wykonanie pomiaru czasu trwania operacji
58         input.cycleBuffers(swapTime, conversionTime);
59
60         // Dodanie czasu trwania operacji do ogólnego czasu
61         // wykonania pętli programowej
62         // (zmienna swapTime przechowuje czas wynikający
63         // z operacji odczytu plików MIDI, nie jest on
64         // brany pod uwagę)
65         time += conversionTime;
66
67         // Pomiar czasu rozpoczęcia iteracji pętli
68         timeStart = std::chrono::system_clock::now();
69
70         // Wygenerowanie próbek dźwiękowych przy użyciu syntezy
71         // podłączonych do urządzeń wejściowych
72         input.generateSamples(executionQueue.getConnectedSynthIDs());
73
74         // Wykonanie operacji na strumieniach audio
75         for (int i = backwardsExecution.size() - 1; i >= 0; i--){
76             component.applyEffects(backwardsExecution[i]);
77         }
78
79         // Zapis wyniku do pliku .wav oraz wykonanie pomiaru czasu
80         // końca trwania iteracji pętli
81         output.onlyRecord(&outputBuffer->buffer, timeEnd);
82
83         // Dodanie czasu trwania iteracji do ogólnego czasu
```

```
84         time += std::chrono::duration<double>(  
85             timeEnd - timeStart).count();  
86     }  
87  
88     // Zakończenie zapisu wyniku do pliku .wav  
89     output.stopRecording();  
90  
91     return 0;  
92 }
```

4.6 Interfejs użytkownika

Interfejs użytkownika został zaimplementowany jako aplikacja konsolowa w klasie `SynthUserInterface`. Klasa ta wykorzystuje metody upublicznione w `AudioPipelineManager` w swoich metodach obsługujących poszczególne komendy. `SynthUserInterface` wykorzystuje kontener `std::map` z biblioteki standardowej w celu przechowywania par wartości: nazwy komendy (`std::string`) i wskaźnika na metodę (`void (SynthUserInterface::*)()`). Pozwala to w wydajny sposób na wywołanie odpowiedniej metody w zależności od wprowadzonej przez użytkownika komendy. Interfejs posiada swoją własną pętlę programową, której zadaniem jest oczekiwanie na wprowadzenie komendy przez użytkownika, parsowanie wprowadzonej komendy oraz wywołanie odpowiedniej metody. Klasa `SynthUserInterface` implementuje interfejs `IScriptReaderClient` dzięki czemu obiekt klasy `ScriptReader` może wywołać operacje na obiekcie klasy `SynthUserInterface` i jednocześnie jest on polem klasy na, której operuje. Pozwala to na wywołanie skryptów zarówno z poziomu implementacji systemu, jak i z poziomu interfejsu użytkownika, co automatyzuje proces testowania wydajności systemu, tym samym eliminując czynnik ludzki z procesu pomiaru.

Rozdział 5

Wybrane algorytmy przetwarzania dźwięku

Rozdział 6

Implementacja GPU

Rozdział 7

Wnioski

Przedstawiona implementacja systemu syntezy i przetwarzającego sygnały dźwiękowe udowadnia, iż jest to możliwe przy uruchomieniu karty graficznej. Problematyka przedstawionego zagadnienia jest wysoce kompatybilna z metodami równoległego przetwarzania danych. Wiele algorytmów przetwarzania sygnałów dźwiękowych, które są obecnie wykonywane na CPU, może być bezproblemowo przeniesione na GPU przy użyciu takich platform jak CUDA. Platforma ta zapewnia łatwy dostęp do zasobów karty graficznej, co pozwala na wykorzystanie jej mocy obliczeniowej i przeniesienie kodu na GPU w sposób stosunkowo nieskomplikowany, nieingerujący nadto w strukturę systemu. Jedynym mankamentem okazał się brak pełnego wsparcia dla polimorfizmu w języku CUDA, co wymusiło zmianę sposobu zastosowania wzorca strategii. Na rynku dostępne jest obecnie wiele innych technologii, pozwalających na wykorzystanie mocy obliczeniowej GPU, takich jak OpenCL, OpenACC, czy SYCL. Każda z nich mogła być równie dobrze wykorzystana do tego rodzaju zadań w zależności od indywidualnych preferencji programisty oraz wymagań stawianych przed projektem. Powstały system bazujący na GPU bezproblemowo dorównuje, a niekiedy nawet przewyższa wydajnością swój odpowiednik powstały na CPU, pomimo iż kwestje optymalizacji nie zostały poruszone w tej pracy. Implementacja bardziej złożonych algorytmów, a w szczególności algorytmów sekwencyjnych w sposób optymalny może okazać się jednak trudniejsza i wymagać głębszej analizy problemu.

Rozdział 8

Możliwości rozwoju

8.1 Optymalizacja obecnego rozwiązania

Ideą stojącą za wykorzystanie karty graficznej było zagospodarowanie dodatkowej mocy obliczeniowej w celu umożliwienia przetwarzania większej ilości danych w czasie rzeczywistym.

8.1.1 Przystosowanie algorytmów do wykorzystania GPU

Przedstawiona implementacja kodu w języku CUDA (przystosowana do wykorzystania GPU) przedstawia algorytmy, których zasada działania jest bliźniaczo podobna do implementacji w języku C++ (przystosowanej do wykorzystania CPU). W związku z tym, wiele z nich obarczonych jest koniecznością wykonywania sekwencyjnego, co nie pozwala na pełne wykorzystanie mocy obliczeniowej karty graficznej. Rozwiązaniem może być całkowita rezygnacja z sekwencyjności, co prawdopodobnie wiązałoby się z brakiem możliwości uzyskania identycznych wyników do klasycznych algorytmów. Nie koniecznie jest to wada, w wielu przypadkach ludzka percepcja nie byłaby wystarczająca aby nawet dostrzec różnicę.

8.1.2 Grupowanie wywołań kerneli

Rezygnacja z sekwencyjności w wielu przypadkach nie jest jednak konieczna. Można by zastosować system przypominający relację klient-serwer: klient (instancja wykorzystująca konkretny algorytm) wysyła swoje dane do serwera (jednostki opartej o wzorec singletonu), gdzie są one przetwarzane jednocześnie dla wszystkich możliwych klientów jednocześnie. Algorytm decydujący o możliwym połączeniu wywołań algorytmów mógłby bazować na obecnej implementacji zawartej w klasie ExecutionQueue - algorytmu przechodzenia po drzewie. Różnicą była by konieczność

powiązania ze sobą gałęzi drzewa, które mogą być przetwarzane jednocześnie, a następnie pogrupowanie ich w kolejności pozwalającej na uzyskanie najmniejszej liczby wywołań kerneli. Równocześnie rozwiązanie to pozwalało by w większości przypadków na obliczenia asynchroniczne. Zastosowanie tego rozwiązania przy zachowaniu sekwencyjności było by najskuteczniejsze w przypadków w których dany algorytm jest wykorzystany wielokrotnie w tym samym czasie. Za przykład może posłużyć synteza dźwięku dla wielu synteзаторów, gdzie każdy z nich syntezuje wiele głosów (wciśniętych klawiszy) jednocześnie. Ten samo rozwiązanie mogło by również przynieść korzyści dla algorytmów nie sekwencyjnych, poprzez zwiększenie rozmiaru bloku bądź gridu.

8.1.3 Implementacja systemu hybrydowego

Niezaprzeczalnie zarówno CPU jak i GPU posiadają swoje mocne oraz słabe strony, wynikające z ich architektury. Tę wiedzę można wykorzystać w celu doboru odpowiedniego urządzenia do konkretnego zadania. Skutecznym rozwiązaniem może okazać się system hybrydowy, który posiadał by implementację zarówno na GPU, jak i CPU. W zależności od potrzeb, system mógłby decydować o wyborze jednej z opcji biorąc pod uwagę zarówno zysk w wykorzystaniu zasobów biorący się z konkretnego rozwiązania oraz czas potrzebny na ewentualne przesłanie danych pomiędzy urządzeniami. W tym przypadku skutecznym może okazać się asynchroniczny przesył danych pomiędzy urządzeniami. Pozwoliło by to na równomierne wykorzystanie mocy obliczeniowej obu urządzeń w najopłacalniejszy w danym kontekście sposób przy jednoczesnym zachowaniu możliwości zachowania algorytmów sekwencyjnych, które preferowane są dla implementacji CPU.

8.2 Wykorzystanie innych technologii

Możliwości karty graficznej nie kończą się na przyspieszaniu obliczeń wykonywanych uprzednio na CPU. Nowoczesne karty graficzne posiadają wiele technologii, które mogą być wykorzystane do utworzenia zupełnie nowych algorytmów, które nie mogły by zostać skutecznie zastosowane w przypadku CPU.

8.2.1 Wykorzystanie rdzeni ray-tracingu

Ray-tracing to technika generowania obrazów poprzez śledzenie promieni światła. Jest to technika stosowana w grafice komputerowej, która pozwala na uzyskanie bardzo realistycznych obrazów. W przeciwieństwie do tradycyjnego renderowania,

gdzie obliczenia są wykonywane dla każdego piksela, ray-tracing pozwala na uzyskanie obrazu poprzez śledzenie promieni światła od obserwatora do obiektów na scenie. Można by wykorzystać tę technologię to śledzenia fal dźwiękowych od źródła dźwięku przez obiekty na scenie, aż do obserwatora. Take wykorzystanie tej technologii może pozwolić na uzyskanie bardziej realistycznych efektów, takich jak echa, czy innych zjawisk związanych z propagacją fal dźwiękowych w przestrzeni. Z kolei takie zastosowanie pozwoliło by na dokładne symulowanie akustyki konkretnych pomieszczeń (sal koncertowych, filharmonii, katedr, itp.) oraz ostatecznie takie narzędzie mogło by okazać się przydatne dla architektów, dając im w ten sposób możliwość symulowania akustyki budynków/pomieszczeń jeszcze w fazie projektu.

8.2.2 Wykorzystanie rdzeni tensorowych

Rdzenie tensorowe to jednostki obliczeniowe, które są specjalnie zaprojektowane do wykonywania operacji na tensorach - wielowymiarowych tablic, które mogą przechowywać dane w dowolnym wymiarze. Rdzenie tensorowe są powszechnie wykorzystywane w technologiach związanych z uczeniem maszynowym, takich jak sieci neuronowe. System przetwarzania dźwięku, który pozwala na wykorzystanie karty graficznej, umożliwia na prostą integrację wykorzystania rdzeni tensorowych w algorytmach przetwarzających sygnał dźwiękowy, a co za tym idzie, na wykorzystanie zaawansowanych technik uczenia maszynowego w celu uzyskania lepszych rezultatów niż było to możliwe w przypadku samego CPU.

8.2.3 Przyszłe technologie

Karty graficzne są jednym z najszybciej rozwijających się obszarów technologii komputerowych. Są one obecnie odbiciem rozwoju szeroko pojętej informatyki, zaczynając od przetwarzania dużych zbiorów danych, poprzez branżę rozrywkową, aż po techniki uczenia komputerowego. W związku z tym jest to obszar, w którym można się spodziewać ciągłych innowacji, nowych rozwiązań i wzrostu mocy obliczeniowej. Wykorzystanie karty graficznej do przetwarzania dźwięku to dziedzina, która ma ogromny potencjał, zarówno w celu przyspieszenia obliczeń dla algorytmów obecnie implementowanych na CPU, jak i potencjalnie w celu uzyskania zupełnie nowych sposobów na przetwarzanie sygnałów dźwiękowych.

Spis listingów

2.1	Przykładowe wywołanie kernela CUDA sumującego dwie tablice w języku C++	18
4.1	Główna pętla programowa systemu dla przetwarzania dźwięku w czasie rzeczywistym	27
4.2	Główna pętla programowa systemu dla przetwarzania dźwięku w trybie offline	29

Spis tabel

1.1	Rozmiar bufora względem czasu przetworzenia	10
1.2	Przesył danych dla różnych formatów przy wykorzystaniu float32 . . .	11
2.1	Przeciętna przepustowość przesyłu danych dla danego standardu / urządzenia	16

Spis rysunków

1.1	Próbkowanie LPCM FLOAT32	6
1.2	Częstotliwość próbkowania LPCM	8
1.3	Rozdzielczość próbki LPCM	9
2.1	docs.nvidia.com - "GPU Devotes More Transistors to Data Processing", ilustracja różnicy w architekturze procesora i karty graficznej	14
2.2	docs.nvidia.com - "Grid of Thread Blocks", ilustracja modelu programowania w technologii CUDA	15
2.3	Wykorzystanie n wątków do przetworzenia n próbek	17
2.4	Wykorzystanie m wątków do przetworzenia m buforów w sposób liniowy	17
2.5	Wykorzystanie $m*n$ wątków do przetworzenia m buforów po n próbek .	17

Bibliografia