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. Przemysława Stpiczyńskiego**

Lublin 2022

Spis treści

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

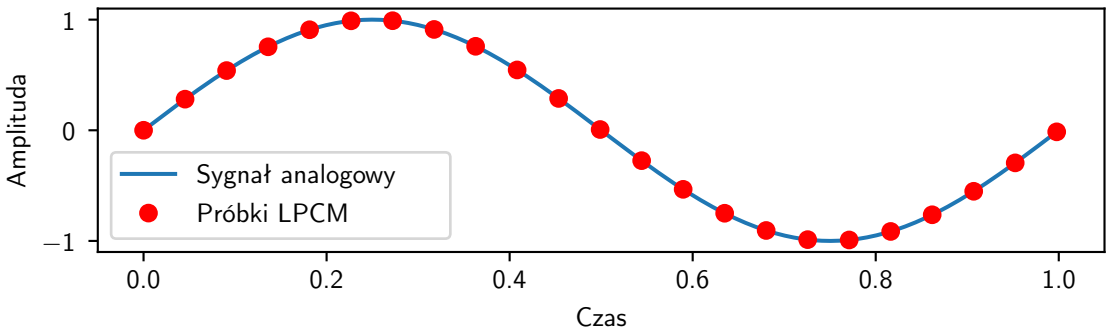
Celem niniejszej pracy jest zaprojektowanie oraz implementacja systemu, pozwalającego na przetestowanie możliwości wykorzystania karty graficznej względem CPU, do przetwarzania sygnałów dźwiękowych. System ma umożliwiać syntezowanie oraz przetwarzanie dźwięku w różnych powszechnie stosowanych formatach (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 zkwantyfikowanego sygnału. Wykorzystuje on skale 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

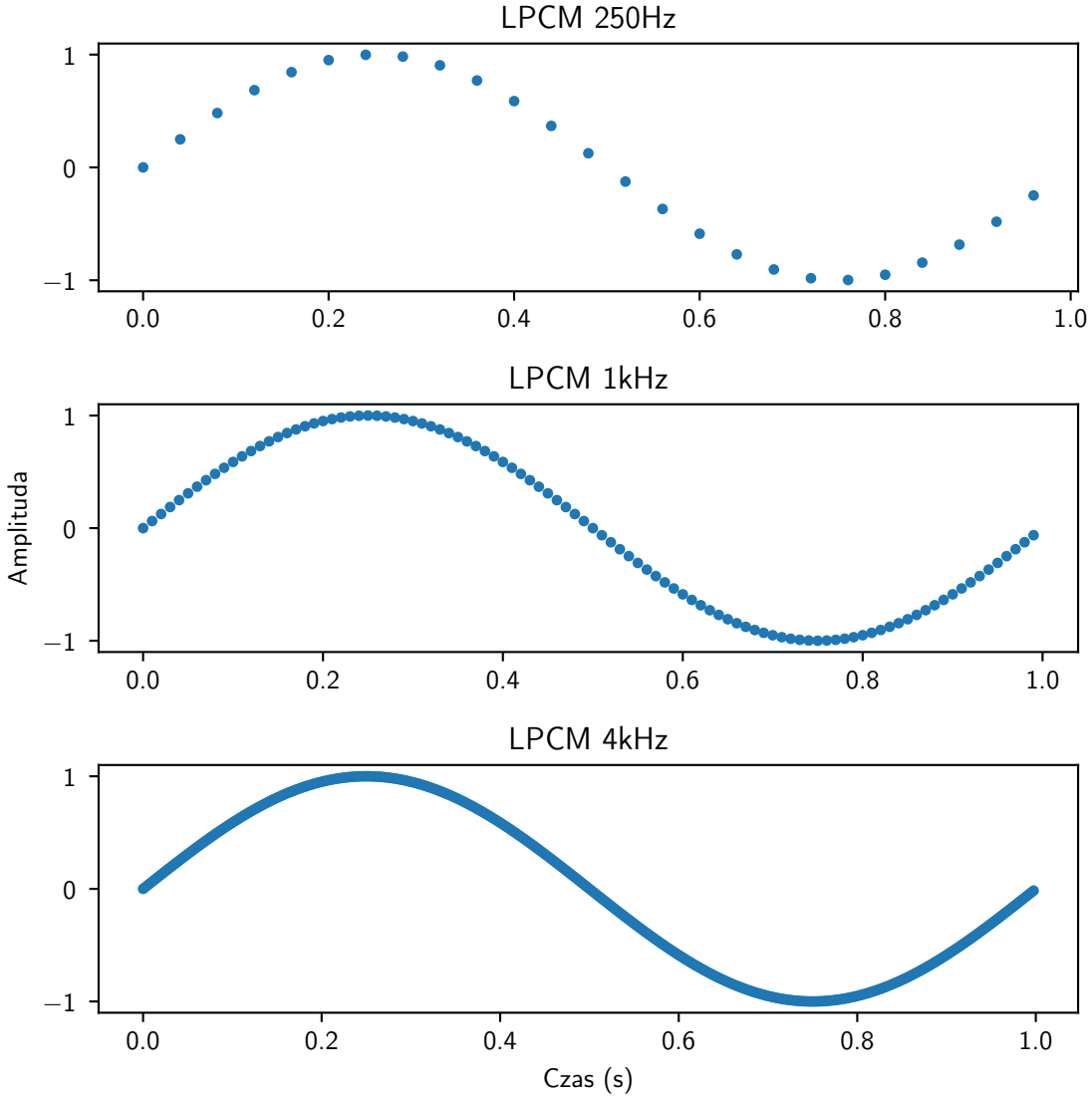
Jakość reprezentacji sygnału zależy do dwóch czynników: częstotliwości próbkowania oraz rozdzielczości pojedynczej próbki. Częstotliwość próbkowania ogranicza odstęp pomiędzy poszczególnymi próbkami. Wyższa częstotliwość próbkowania zmniejsza odstęp pomiędzy próbkami, co pozwala na dokładniejsze odwzorowanie 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 próbkować 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 próbkować go co najmniej z częstotliwością 44kHz, co pokrywa się z ogólno przyjętym standardem standardem stosowanym w przemyśle muzycznym. Przykład różnicy w częstotliwości próbkowania sygnału przedstawia Rysunek 1.2.

1.3 Rozdzielczość próbki

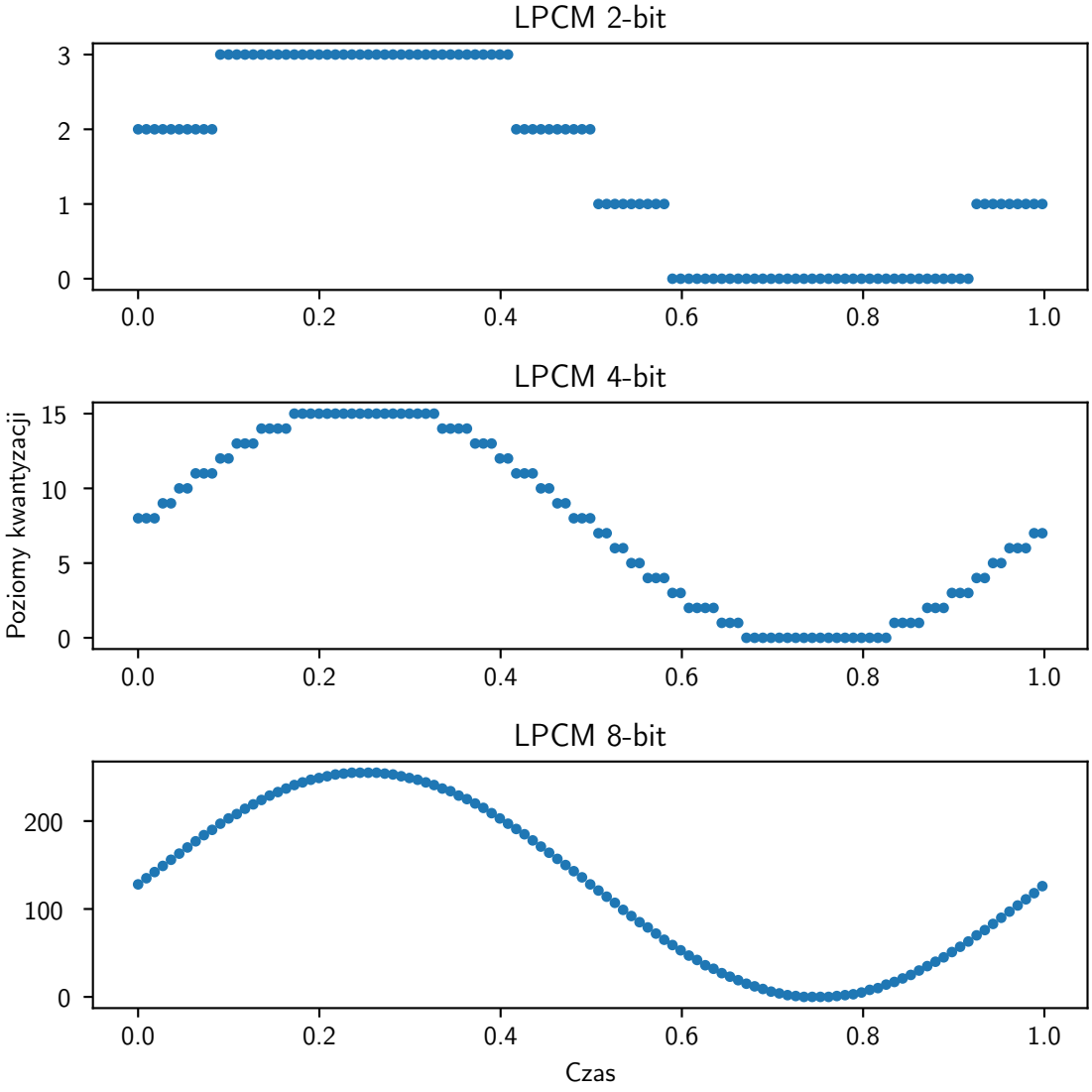
Rozdzielczość próbki, zwana również głębią bitowa / głębią próbki, określa ilość bitów, które są wykorzystywane do jej zapisu. Im większa rozdzielczość, tym mniejszy błąd kwantyzacji oraz większy rozmiar danych do przetworzenia. Determinuje to ilość poziomów kwantyzacji sygnału, która jest równa 2^n, 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 przetwarzania 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	0.176 MB/s	0.352 MB/s
48kHz	0.192 MB/s	0.384 MB/s
96kHz	0.384 MB/s	0.768 MB/s
192kHz	0.768 MB/s	1.536 MB/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. syntezatora. Format MIDI posiada kilka wariantów, rozbudowujących i tak bogaty asortyment możliwości. Nawet w podstawowym wariancie MIDI (General MIDI), pliki MIDI są w stanie przechowywać informacje na różne sposoby. Sposoby te nazywane są formatami. Na rzecz tej prazy 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 syntezować, 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.

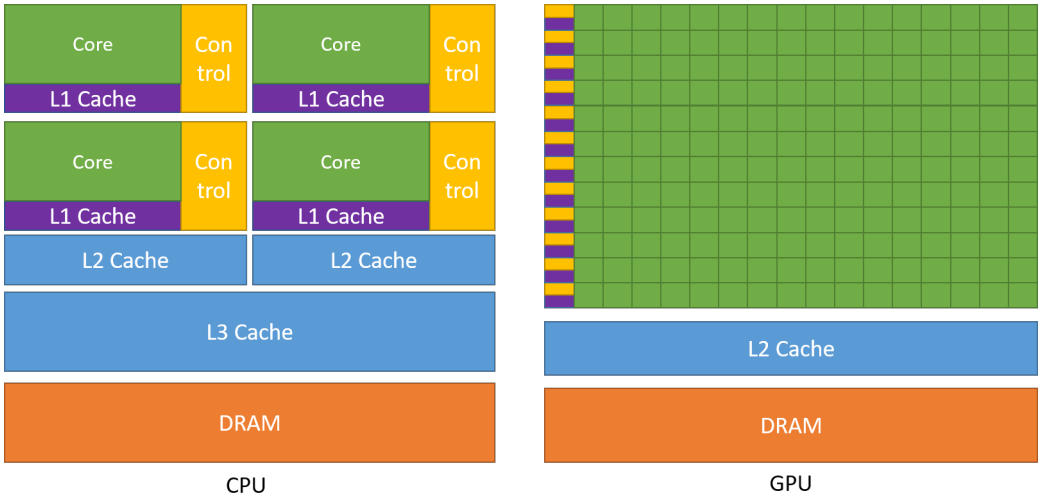
Rozdział 2

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 wielowatkowego 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 równolegle. 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

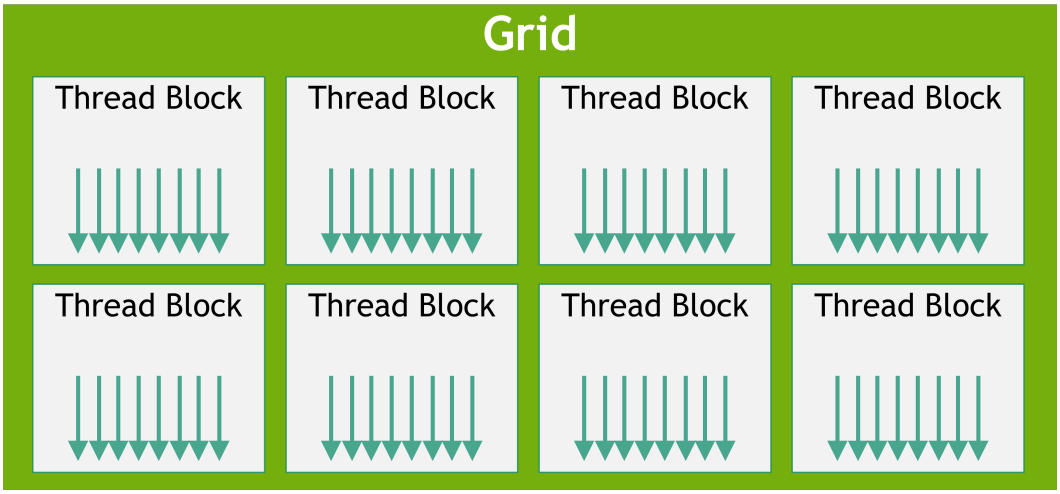


Rysunek 2.1: docs.nvidia.com - "GPU Devotes More Transistors to Data Processing", ilustracja różnicy w architekturze procesora i karty graficznej

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 jedyni 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 tą 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.

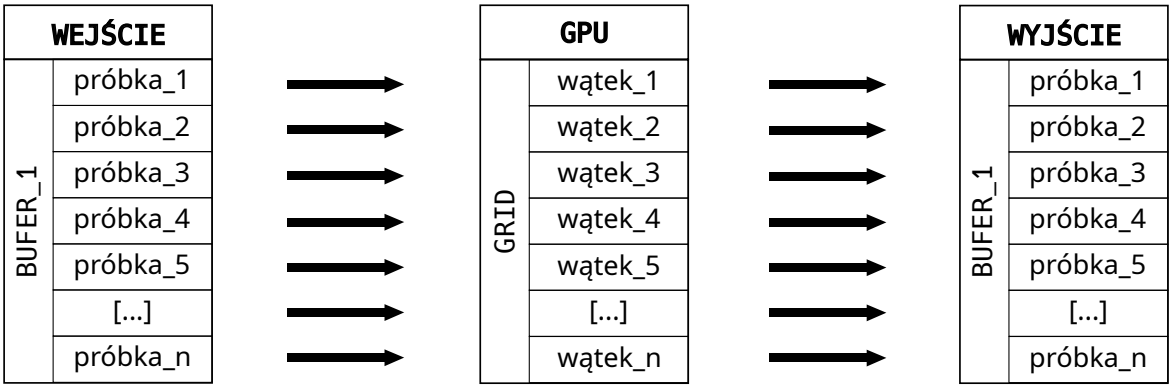
2.3 Zastosowanie dla problematyki przetwarzania sygnałów

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

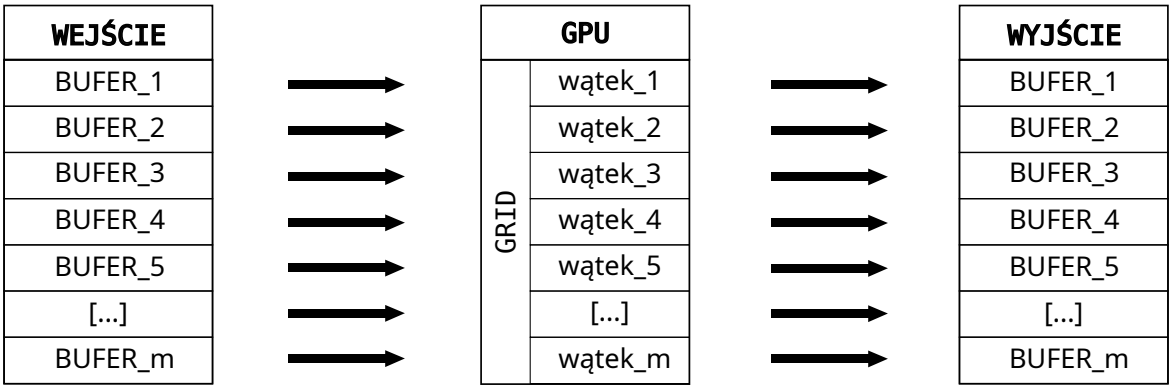
Tabela 2.1: Przeciętna przepustowość przesyłu danych dla danego standardu / urządzenia

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

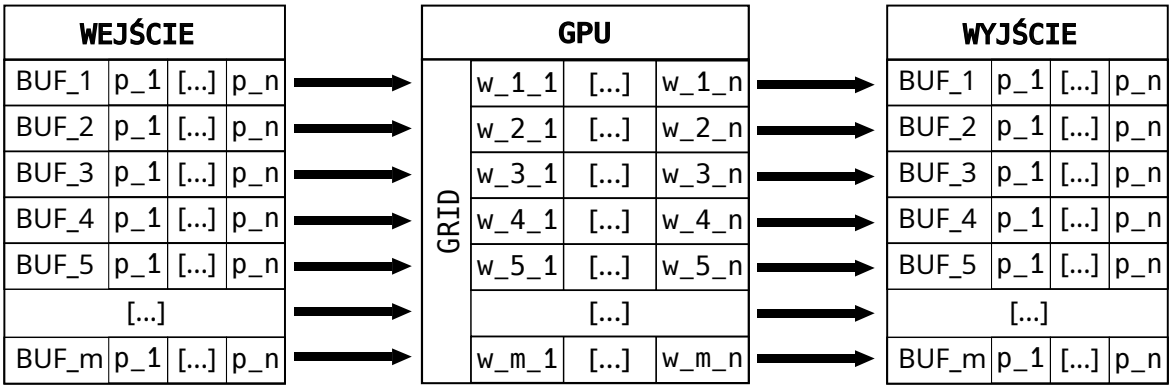
przykładem jest zastosowanie wykorzystanie n wątków GPU do przetworzenia n próbek sygnału zawartych w buforze. Sytuacja przedstawiona na Rysunku 2.3 ilustruje metodę, którą można by bez problemu wykorzystać przy zrównoleglaniu nieskomplikowanych algorytmów. W przypadku algorytmów bardziej złożonych lub 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 koniecznie 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*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. Próbki 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 strumienie wejściowe,
   - zarządza zbiorem syntezatorów,
   - generuje strumienie dźwiękowe przy wykorzystaniu syntezatorów 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,
   - głębia 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 jednoczesne 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 syntezatorów 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 syntezator, przy wykorzystaniu wzorca strategii pozwala na wybór różnych algorytmów generowania dźwięku. Wypełnia bufor audio na podstawie ustawień użytkownika oraz struktur `keyboardTransferBuffer`. Każdy syntezator 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. `OutStreamPulseAudio` - klasa implementująca interfejs `IOutStream`. 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 projetkowego stragegii, 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 syntezatorów, 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

Podsystem danych statystycznych jest zarządzany przez klasę `pipeline::StatisticsService`. Podsystem opiera się o dwa punkty pomiaru czasu w głównej pętli systemowej: na samym początku iteracji (przed wykonaniem jakichkolwiek operacji) oraz po utworzeniu bufora wyjściowego (jeszcze przed przekazaniem jego do serwera PulseAudio). Dzięki temu można dokładnie zmierzyć czas trwania operacji oraz całej pętli programowej. Na podstawie wielkości buforu oraz częstotliwości próbkowania możliwe jest obliczenie przewidywanego czasu obrotu pętli programowej. Porównując to z pomiarami, można między innymi wyznaczyć obciążenie systemu: $O = \frac{t_{\text{pomiar}}}{t_{\text{przewidywany}}}$, gdzie $O > 1$ oznacza, iż czas potrzebny na

wykonanie obliczeń nie pozwala na przetworzenie dźwięku w czasie rzeczywistym. Podsystem operuje na obiektach struktury `pipeline::StatisticsBuffer`. Obiekty te przechowują czasy trwania operacji oraz czasy trwania pętlu w mikrosekundach, jak i procentowe obciążenie systemu. Na podstawie uśrednionych wartości uaktualniana jest struktura `pipeline::Statistics` udostępniona reszcie systemu.

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 syntezatorów
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 syntezatorów
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/generowania dźwięku

Przedstawione w pracy algorytmy należą do najprostszych i najbardziej popularnych. W każdym przypadku zostały one zaimplementowane w możliwie najprostszej formie, wystarczającej do ilustracji ich działania, jak i możliwości zastosowania.

5.1 Generowanie sygnału

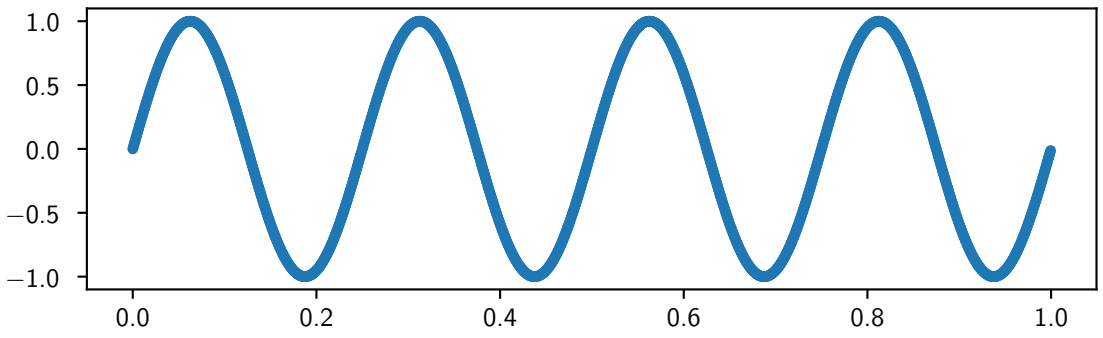
Sygnał dźwiękowy jest generowany przy pomocy syntezatora, który wykorzystuje jeden z poniżej przedstawionych algorytmów, by na podstawie informacji o wciśniętych klawiszach wygenerować odpowiednią sekwencję próbek. Każdemu z klawiszy koresponduje określona częstotliwość, która definiuje długość okresu funkcji sygnału. Algorytmy te są równoznaczne konkretnym funkcjom matematycznym, których dziedziną jest czas, a przeciwdziedziną amplitudy sygnału. Funkcja czasu jest dyskretna, co oznacza, że wartości sygnału są znane tylko w określonych momentach czasu - próbkach. Poniżej zostały przestawione wzory wykorzystane w implementacji poszczególnych funckcji. Dla $F(f, n, f_s)$ będącego funkcją sygnału: f oznacza częstotliwość sygnału, n numer próbki, a f_s częstotliwość próbkowania.

Przedstawione wzory są jedynie przykładami. Istnieją inne możliwości zapisu opisywanych sygnałów.

5.1.1 Sygnał sinusoidalny

$$F_{\sin}(f, n, f_s) = \sin\left(\frac{f \cdot 2\pi \cdot n}{f_s}\right) \tag{5.1}$$

Sygnał sinusoidalny jest najprostszym sygnałem dźwiękowym pod względem matematycznym. Jest to funkcja sinus, której argumentem jest iloczyn częstotliwości sygnału, czasu i 2π. Sygnał sinusoidalny jest sygnałem o najczystszych właściwościach dźwiękowych, co oznacza, że jest to sygnał o najmniejszej zawartości harmonicznych. Te właściwości wyróżniają funkcję sinusoidalną na tle innych. Generując sinus o częstotliwości f Hz, w widmie sygnału nie znajdziemy żadnych innych składowych niż f Hz.

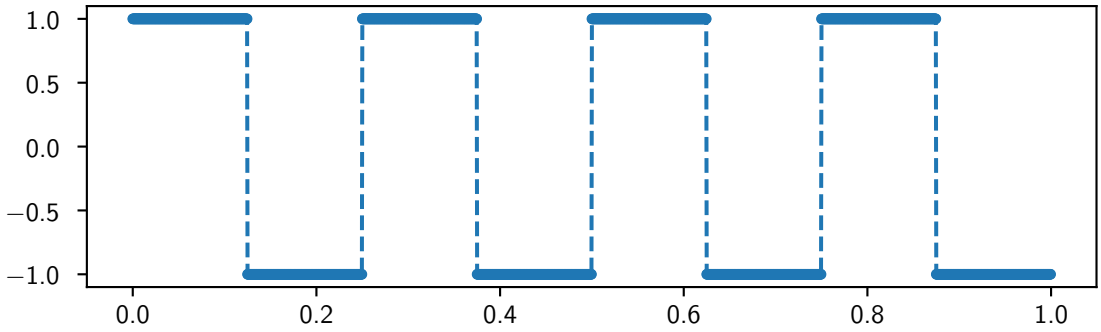


Rysunek 5.1: Graficzna reprezentująca sygnału sinusoidalnego

5.1.2 Sygnał prostokątny

$$F_{\text{sqr}}(f, n, f_s) = \left(\left\lfloor \frac{f \cdot n}{f_s} \right\rfloor \mod 2 \right) \cdot 2 - 1 \tag{5.2}$$

Dziedzina sygnału prostokątnego składa się jedynie z dwóch wartości: -1 i 1. Łatwo zauważyć, że brak wartości pośrednich będzie generować duży przester, który jest przyczyną charakterystycznego brzmienia dla syntezatorów bazujących na sygnałach prostokątnych. W związku z obecnością przesteru, sygnał prostokątny jest sygnałem o najbogatszych harmonicznych, co oznacza, że w widmie sygnału znajdziemy wiele składowych o różnych częstotliwościach.



Rysunek 5.2: Graficzna reprezentująca sygnał prostokątny

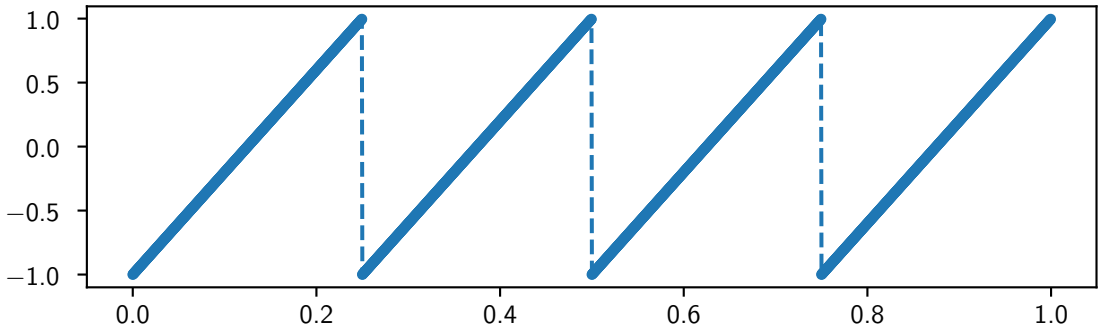
5.1.3 Sygnał piłokształtny

$$A_{\text{saw}}(f, n, f_s) = \frac{f \cdot n}{f_s} \tag{5.3}$$

$$B_{\text{saw}}(f, n, f_s) = A_{\text{saw}}(f, n, f_s) - \lfloor A_{\text{saw}}(f, n, f_s) \rfloor \tag{5.4}$$

$$F_{\text{saw}}(f, n, f_s) = B_{\text{saw}}(f, n, f_s) \cdot 2 - 1 \tag{5.5}$$

Sygnał piłokształtny w związku z obecnością przesteru jest sygnałem o bogatych harmonicznych, podobnie jak sygnał prostokątny. W przeciwieństwie do sygnału prostokątnego, sygnał piłokształtny jest sygnałem o bardziej naturalnym brzmieniu, co sprawia, że jest on często wykorzystywany w syntezatorach. Można wyróżnić dwa rodzaje sygnału piłokształtnego: rosnący i opadający. Sygnał rosnący jest generowany zgodnie z równaniem 5.5.



Rysunek 5.3: Graficzna reprezentująca sygnału piłokształtnego

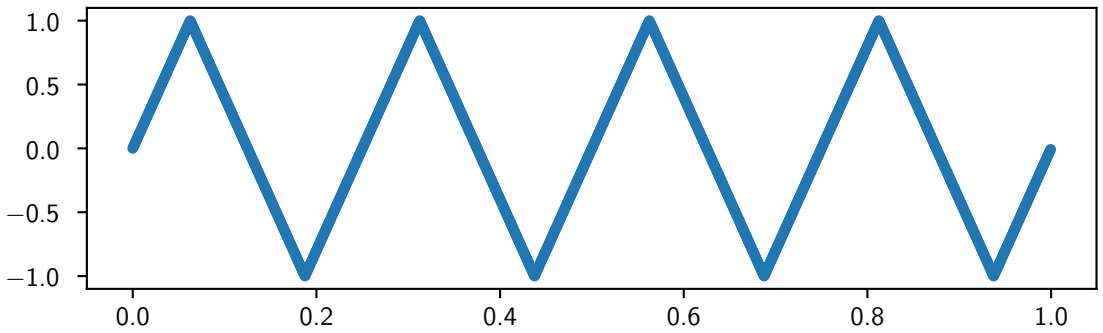
5.1.4 Sygnał trójkątny

$$A_{\text{tri}}(f, n, f_s) = \frac{f \cdot n \cdot 2}{f_s} \tag{5.6}$$

$$B_{\text{tri}}(f, n, f_s) = A_{\text{tri}}(f, n, f_s) - \lfloor A_{\text{tri}}(f, n, f_s) \rfloor \tag{5.7}$$

$$F_{\text{tri}}(f, n, f_s) = \begin{cases} B_{\text{tri}}(f, n, f_s) \cdot 2 - 1; & \text{dla } A_{\text{tri}}(f, n, f_s) \mod 2 = 1 \\ (1 - B_{\text{tri}}(f, n, f_s)) \cdot 2 - 1; & \text{dla } A_{\text{tri}}(f, n, f_s) \mod 2 = 0 \end{cases} \tag{5.8}$$

Sygnał trójkątny w brzmieniu przypomina sygnał sinusoidalny, jednakże w związku z jego gwałtownymi zmianami wartości, wprowadza on niewielką ilość przesteru. Efektem tego jest powstanie sygnału zauważalnie wzbogaconego harmonicznie, ale jednocześnie o bardziej naturalnym brzmieniu niż sygnał prostokątny czy piłokształtny. Przedstawiony wzór (5.8) generuje sygnał trójkątny przy wykorzystaniu na zmianę sygnału piłokształtnego rosnącego i malejącego.



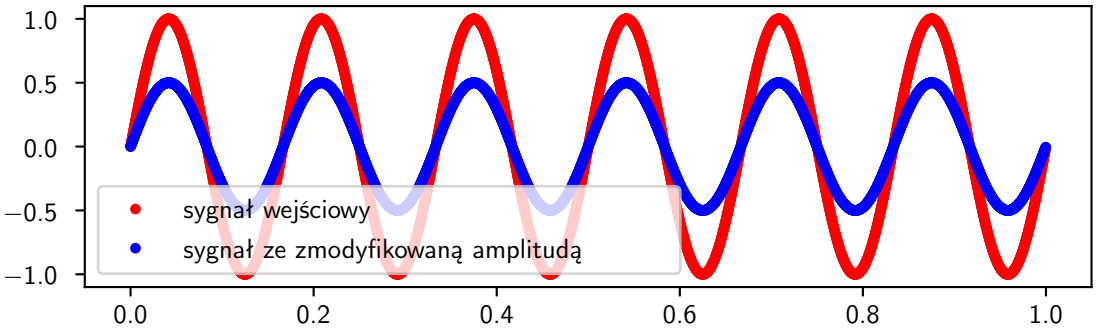
Rysunek 5.4: Graficzna reprezentująca sygnału trójkątny

5.2 Komponenty podstawowe

Komponenty podstawowe reprezentowane poprzez klasy dziedziczące po klasie abstrakcyjnej `AComponent` modyfikują strumień audio do którego są przypisane. Każdy komponent posiada swoje parametry, które mogą być modyfikowane w trakcie działania programu. Poniżej została przedstawiona zasada działania oraz funkcjonalność dla każdego z komponentów.

5.2.1 Volume - głośność

Komponent `Volume` odpowiada za modyfikację amplitudy sygnału audio. Pozwala zarówno na zwiększenie, zmniejszenie głośności sygnału, jak i odwrócenie fazy przy pomocy parametru `vol`. Jest to podstawowy algorytm modyfikacji sygnału audio.



Rysunek 5.5: Reprezentacja komponentu `Volume`

5.2.2 Pan - pozycja dźwięku w przestrzeni

Komponent `Pan` odpowiada za modyfikację pozycji dźwięku w przestrzeni stereofonicznej. Pozwala na względne zwiększenie amplitudy sygnału jednego z kanałów względem drugiego przy pomocy parametru `pan`. Jest to podstawowy algorytm modyfikacji strumieni stereofonicznych.

5.2.3 Distortion - przesterowanie

Komponent `Distortion` implementuje prosty algorytm przesterowania sygnału audio. Posługuje się on następującymi parametrami:

- `gain` - wzmocnienie sygnału przed przesterowaniem,
- `threshold` - próg przesterowania, sygnał powyżej tej wartości zostaje nią zastąpiony,
- `symmetry` - przesunięcie wzmocnienia aplikowanego poprzez parametr `gain` względem wartości 0,
- `vol` - modyfikacja amplitudy sygnału wyjściowego.

Wynikiem działania komponentu jest obcięcie sygnału powyżej wartości `threshold`, co skutkuje powstaniem wielu harmonicznych. Zastosowana w tej pracy implementacja tego algorytmu opiera się na zastosowaniu instrukcji warunkowej, co pozwoli na przetestowanie potencjalnej różnicy w wydajności pomiędzy CPU a GPU dla podobnych operacji.

5.2.4 Echo - efekt powtórzenia

Komponent `Echo` implementuje algorytm powielający sygnał wejściowy i sumujący go z oryginalnym sygnałem wskazaną ilość razy ze wskazanym opóźnieniem. Parametry komponentu to:

- `repeats` - liczba powtórzeń sygnału,
- `delay` - okres czasu po którym zapisany sygnał zostaje ponownie odtworzony,
- `fade` - mnożnik amplitudy sygnału powtarzanego,
- `rvol / lvol` - mnożnik amplitudy sygnału powtarzanego dla odpowiednio prawego i lewego kanału.

Algorytm ten wprowadza wykorzystanie pętli dla wygenerowanie pojedynczej próbki dźwiękowej oraz wymaga zapisu sygnału wejściowego w buforze, co pozwoli na przetestowanie różnicy w wydajności pomiędzy CPU a GPU dla wielokrotnej operacji zapisu i odczytu z pamięci.

5.2.5 Compressor - kompresja

Komponent `Compressor` implementuje algorytm kompresji sygnału audio. W tej dziedzinie kompresja oznacza zmniejszenie różnicy pomiędzy największą a najmniejszą wartością sygnału. W przypadku tej implementacji, algorytm ogranicza amplitudę sygnału powyżej wskazanej wartości, przy jednoczesnej próbie zniwelowania efektu przesteru, który może powstać w wyniku takiej operacji. Parametry komponentu to:

- `threshold` - próg powyżej którego algorytm zostaje wykonany,
- `ratio` - stosunek amplitudy przekraczającej `threshold` sygnału wejściowego do amplitudy przekraczającej `threshold` sygnału wyjściowego. Wartość ta jest ściśle powiązana z parametrem `step`,
- `step` - określa wartość przekroczenia `threshold` sygnału wejściowego, po której wartość `ratio` jest w pełni zastosowana,
- `attack` - czas, potrzebny do zarejestrowania w pełni wzrostu amplitudy sygnału wejściowego,
- `release` - czas, potrzebny do zarejestrowania w pełni spadku amplitudy sygnału wejściowego,
- `vol` - modyfikacja amplitudy sygnału wyjściowego.

Algorytm wykorzystuje zależność czasową pomiędzy kolejnymi próbkami sygnału (parametry `attack` i `release`) do zastosowania kompresji. Pozwoli to na przetestowanie różnicy w wydajności pomiędzy CPU a GPU dla operacji zależnych od siebie w czasie.

5.3 Komponenty zaawansowane

Komponenty zaawansowane reprezentowane poprzez klasy dziedziczące po klasie abstrakcyjnej `AAdvancedComponent` mogą operować na wielu strumieniach audio jednocześnie, tworząc tym samym swój własny strumień wyjściowy. Pozwala to na tworzenie bardziej złożonych operacji na dźwięku, umożliwiając np. miksowanie kilku strumieni audio, czy też tworzenie alternatywnych ścieżek przetwarzania pojedynczego strumienia. Poniżej została przedstawiona zasada działania oraz funkcjonalność dla każdego z komponentów.

5.3.1 copy - kopiowanie strumienia

Komponent `copy` pozwala na kopiowanie strumienia audio. Dzięki temu możliwe jest utworzenie alternatywnych ścieżek przetwarzanie danego sygnału.

5.3.2 sum - łączenie strumieni

Komponent `sum` pozwala na sumowanie `n` strumieni audio. Umożliwia to jednoczesne odtwarzanie sygnałów generowanych przez kilka syntezatorów, lub też miksowanie kilku strumieni wytworzonych przy pomocy komponentu `copy`. W pracy zaimplementowano algorytm sumowania dwóch oraz siedmiu sygnałów.

Rozdział 6

Implementacja GPU

Implementacja GPU jest kluczowym elementem pracy. Powstały system będzie bazował na implementacji CPU, a wprowadzone modyfikacje nie mogą łamać wymagań przedstawionych w trakcie przedstawiania projektu systemu (**Rozdział 3 Projekt systemu**). Może natomiast modyfikować sposób oraz metody działania, wprowadzone w trakcie implementacji CPU, w celu spełnienia wymogów wynikających ze specyfiki programowania GPU. Dzięki tym wytycznym możliwym będzie ocena poziomu trudności dla przeprojektowania systemu dla wykorzystania platformy CUDA. W rozdziale tym zostaną przedstawione najważniejsze zmiany w implementacji dla każdego z podsystemów, które umożliwiły wykorzystanie technologii CUDA. W celu bliższego przyjrzenia się dokonanym zmianom, należy skorzystać z załączonej implementacji systemu, gdzie każda klasa bądź struktura, ktora została zmodyfikowana na tym etapie implementacji, jest oznaczona na końcu swej nazwy ciągiem znaków `_CUDA`.

6.1 Podsystem wejścia

W związku z zauważalnym kosztem przesyłu danych pomiędzy CPU a GPU, jednym z kluczowych elementów implementacji jest sprowadzenie ilości danych przesyłanych pomiędzy urządzeniami do minimum. Jako, iż by móc przetwarzać dane, muszą one być wcześniej dostępne na karcie graficznej. Podsystem wejścia obsługuje urządzenia oraz pliki wejściowe w celu dostarczenia informacji dotyczących konieczności syntezy dźwięków o konkretnych częstotliwościach. Proces ten jest wykonywany za pomocą CPU, a następnie nieprzetworzone dane zostają przeniesione na GPU. Punktem przeniesienia została wybrana struktura `keyboardTransferBuffer`. Jako, iż klasa ta jest relatywnie niewielka, posłuży jako przykład zmian powszechnie wprowadzanych w implementacji GPU. Zmiany te nie zostaną ponownie opisane dla każdej klasy oraz struktury, jako iż są one analogiczne dla każdego z przypadków.

Listing 6.1: Implementacja keyboardTransferBuffer_CUDA

```
1
2  // konstruktor alokuje potrzebne bufory na karcie graficznej
3  // wykorzystuje w tym celu funkcje cudaMalloc
4  keyboardTransferBuffer_CUDA::keyboardTransferBuffer_CUDA(
5      const uint& sampleSize, const unsigned short int& keyCount)
6      : sampleSize(sampleSize), keyCount(keyCount){
7
8      // oznaczenie 'd_' (device_) wskazuje, że dane pole
9      // klasy jest wskaźnikiem na pamięć karty graficznej
10     cudaMalloc((void**)(&d_buffer),
11         keyCount * sampleSize * sizeof(unsigned char));
12
13     cudaMalloc((void**)(&d_input),
14         keyCount * sampleSize * sizeof(unsigned char));
15
16     cudaMalloc((void**)(&d_lastState),
17         keyCount * sizeof(unsigned char));
18 }
19
20 // destruktor zwalnia zaalokowane wcześniej bufory
21 keyboardTransferBuffer_CUDA::~keyboardTransferBuffer_CUDA(){
22     cudaFree(d_buffer);
23     cudaFree(d_input);
24     cudaFree(d_lastState);
25 }
26
27 // niezmodyfikowana metoda pozwalająca na konwersję bufora
28 // dowolnego obiektu implementującego interfejs IKeyboardDoubleBuffer
29 void keyboardTransferBuffer_CUDA::convertBuffer(
30     IKeyboardDoubleBuffer* keyboardBuffer){
31     convertBuffer(keyboardBuffer->getInactiveBuffer());
32 }
33
34 // kernel CUDA odpowiedzialny za konwersję bufora
35 // zadeklarowany jako funkcja globalna
36 __global__ void kernel_convertBuffer(
37     const uint sampleSize, const unsigned short int keyCount,
38     unsigned char* input, unsigned char* buffer,
39     unsigned char* lastState){
40
41     // obliczenie indeksu wątku
42     const uint i = blockIdx.x * blockDim.x + threadIdx.x;
43
44     // sprawdzenie czy indeks wątku mieści się w zakresie
```

```
45      if (i < keyCount){
46          // operacje wykonywane dla pojedynczego wątku są
47          // analogiczne do tych wykonywanych dla pojedynczego
48          // obrotu pętli wersji CPU
49          unsigned char lastStateTemp = lastState[i];
50          for (uint j = 0; j < sampleSize; j++){
51              uint index = i * sampleSize + j;
52              if (input[index] == 255){
53                  lastStateTemp = 0;
54              } else if (input[index] > 0){
55                  lastStateTemp = input[index];
56              }
57              buffer[index] = lastStateTemp;
58          }
59          lastState[i] = lastStateTemp;
60      }
61  }
62
63  void keyboardTransferBuffer_CUDA::convertBuffer(
64      unsigned char* buff[127]){
65
66      // ustalenie rozmiaru bloku
67      static const uint blockSize = 128;
68
69      // przeniesienie danych z pamięci RAM,
70      // przechowywanych w tablicy tablic
71      // na pojedynczy bufor w pamięci karty graficznej
72      for (uint i = 0; i < keyCount; i++){
73          cudaMemcpy(
74              d_input + i * sampleSize,
75              buff[i],
76              sampleSize * sizeof(unsigned char),
77              cudaMemcpyHostToDevice);
78      }
79
80      // obliczenie potrzebnej ilości bloków
81      uint blockCount = (keyCount + blockSize - 1) / blockSize;
82
83      // wywołanie kernelu CUDA
84      kernel_convertBuffer<<<blockCount, blockSize>>>
85          (sampleSize, keyCount, d_input, d_buffer, d_lastState);
86  }
```

Struktura `keyboardTransferBuffer_CUDA` nadal spełnia te same zadania co jej odpowiednik `keyboardTransferBuffer`. Wprowadzone zmiany umożliwiają na

wykorzystanie GPU do wykonania czasochłonnych obliczeń. Przeniesione i przetworzone dane pozostają w pamięci karty graficznej, co pozwala na ich wykorzystanie w dalszych etapach przetwarzania, bez konieczności ich ponownego przesyłania. Tabela 6.1 przedstawia ilość danych przesyłanych pomiędzy CPU a GPU dla pojedynczego urządzenia wejściowego z uwzględnieniem częstotliwości próbkowania oraz standardowej ilości klawiszy dla urządzenia MIDI, wynoszącej 128.

Tabela 6.1: Przesył danych pomiędzy CPU a GPU dla jednego urządzenia wejściowego

częstotliwość próbkowania	wymagana przepustowość
44kHz	5.632 MB/s
48kHz	6.144 MB/s
96kHz	12.288 MB/s
192kHz	24.576 MB/s

Przetwarzane dane posiadają zależność czasową, co nie zezwala na przetworzenie każdej z próbek jednocześnie, jednak dane dzielą się na niezależne fragmenty, gdzie każdy z nich przypisane jest innemu z przycisków klawiatury. Łącząc te dwa fakty, możliwe jest zrównoleglenie obliczeń dla każdego z fragmentów sposobem, który został zilustrowany na grafice 2.4 - Wykorzystanie m wątków do przetworzenia m buforów w sposób liniowy.

Największe zmiany podczas tej fazy implementacyjnej zaszły w klasach odpowiedzialnych za generowanie dźwięku. Każda struktura, aby zostać przeniesiona na GPU, nie może posiadać żadnych metod. W związku z tym wszelkie operacje, które były wcześniej wykonywane w metodach, zostały przeniesione do klas pomocniczych: `NoteBufferHandler_CUDA`, `DynamicsController_CUDA`. Wszelkie operacje, wykonywane na buforach, zostały zastąpione przez odpowiednie kernele CUDA, bądź funkcje CUDA dla np. alokacji pamięci.
Koncepcja klasy abstrakcyjnej `AGenerator`, a tym samym i wszystkich implementujących ją klas, musiała zostać utworzona na nowo. Platforma CUDA nie zezwala na polimorfizm w taki sam sposób, jak to jest możliwe w przypadku jezyka C++. W celu zachowania wzorca projektowego strategii, każda z klas implementujących interfejs `AGenerator` została przeniesiona do klasy `Generator_CUDA`, gdzie polimorfizm został zastąpiony przez tablicę wskaźników na kernele `device` (kernele `device` są wywoływane na karcie graficznej) zaalokowanej w pamięci karty graficznej. W celu wyboru algorytmu generowania sygnału dźwiękowego, przekazywany jest indeks tablicy, przy wykorzystaniu zmiennej typu `enum synthesizer::generator_type` do kernela, wraz ze wskaźnikem na ową tablicę.

Obrany w czasie implementacji CPU algorytm syntezy dźwięku wprowadza zależność pomiędzy kolejnymi próbkami. Znacząco ogranicza to możliwości zrównoleglenia obliczeń, w związku z czym algorytm ten został podzielony na kilka części. Po głębszej analizie udało się wyodrębnić operacje, które można dla określonej grupy przypadków, wykonać przed rozpoczęciem syntezy, a wyniki zapisać w tablicy. Następnie wszelkie operacje, które wymagały zależności czasowej, zostały zamknięte w pierwszym kernelu CUDA, natomiast operacje, które nie wymagały zależności czasowej, zostały zamknięte w drugim. Dzięki temu jedynie operacje zależne czasowo zostają wykonane w przy wykorzystaniu metody opisanej za pomocą 2.4 - Wykorzystanie m wątków do przetworzenia m buforów w sposób liniowy. Operacje wykonane w drugim z opisywanych kerneli pozwalają na wykorzystanie pełni mocy obliczeniowej karty graficznej, jako że korzystają z metody opisanej za pomocą 2.5 - Wykorzystanie m*n wątków do przetworzenia m buforów po n próbek. Listing 6.2 przedstawia fragmenty implementację klasy `Generator_CUDA`, odpowiedzialne za opisane powyżej operacje.

Listing 6.2: Implementacja Generator_CUDA

```
1
2  // przykładowy kernel CUDA odpowiedzialny za
3  // generowanie próbki dźwięku
4  __device__ float kernel_soundSquare(
5      const float phaze, const float multiplier){
6
7      return (int((phaze) / multiplier) & 0x1)*2 - 1;
8  }
9
10 // kernel CUDA inicjalizujący tablicę wskaźników na funkcje
11 __global__ void kernel_initFunctionArray(
12     Generator_CUDA::soundFunctionPointer* functionArray){
13
14     // inicjalizacja tablicy ma zostać przeprowadzona tylko raz
15     if (threadIdx.x == 0 && blockIdx.x == 0){
16         functionArray[SINE] = &kernel_soundSine;
17         functionArray[SQUARE] = &kernel_soundSquare;
18         functionArray[SAWTOOTH] = &kernel_soundSawtooth;
19         functionArray[TRIANGLE] = &kernel_soundTriangle;
20         functionArray[NOISE1] = &kernel_soundNoise1;
21     }
22 }
23
24 // kernel CUDA odpowiedzialny za wykonanie operacji
25 // generowania dźwięku bez zależności czasowej
26 __global__  void kernel_generate(
27     noteBuffer_CUDA* noteBuffer, const uchar* keyState,
```

```
28      const settings_CUDA* settings, const float* dynamicsProfile,
29      const float* releaseProfile, uint* phazeWorkArr,
30      uint* pressSamplessPassedWorkArr,
31      uint* releaseSamplesPassedWorkArr, float* velocityWorkArr,
32      const Generator_CUDA::soundFunctionPointer* soundFunction,
33      synthesizer::generator_type currentGeneratorType){
34
35      // obliczenie indeksu wątku dla dwóch wymiarów
36      uint i = blockIdx.x * blockDim.x + threadIdx.x; // key index
37      uint j = blockIdx.y * blockDim.y + threadIdx.y; // sample index
38      if (i < settings->keyCount && j < settings->sampleSize){
39          uint workArrIndex = j + i*settings->sampleSize;
40
41          // sprawdzenie czy dany klawisz jest wciśnięty
42          // lub czy został puszczony w ciągu określonego czasu
43          if (keyState[workArrIndex] ||
44          releaseSamplesPassedWorkArr[workArrIndex] <
45          settings->release.duration){
46
47              // dynamicsMultiplier odpowiada za zmianę głośności
48              // w zależności od czasu jaki minął od wciśnięcia
49              // bądź puszczenia klawisza
50              float dynamicsMultiplier = settings->dynamicsDuration >
51              pressSamplessPassedWorkArr[workArrIndex] ?
52              dynamicsProfile[pressSamplessPassedWorkArr[workArrIndex]]
53              : settings->fadeTo;
54
55              if (keyState[workArrIndex] == 0){
56                  dynamicsMultiplier *= releaseProfile[
57                      releaseSamplesPassedWorkArr[workArrIndex]];
58              }
59
60              // wywołanie odpowiedniej funkcji generującej dźwięk
61              noteBuffer->buffer[workArrIndex] =
62                  (*soundFunction[currentGeneratorType])
63                  (phazeWorkArr[workArrIndex],
64                  noteBuffer->multiplier[i]) *
65                  settings->volume * velocityWorkArr[workArrIndex] *
66                  dynamicsMultiplier;
67          }
68      }
69  }
```

6.2 Podsystem wyjścia

Podsystem wyjścia jest drugim z podsystemów, który wymaga przesyłu danych pomiędzy CPU a GPU. Jedyną zmianą w implementacji jest zastosowanie technologii CUDA do przetworzenia wychodzącego strumienia audio, na format gotowy do odtworzenia przez serwer dźwięku. Wzorzec strategii został zastosowany do znacznie prostszego algorytmu w przypadku syntezy dźwięku i nie musiał on zostać zmodyfikowany.
W przeciwieństwie do podsystemu wejścia, ilość buforów kopiowanych pomiędzy pamięciami urządzeń nie jest zależna od ilości urządzeń wejściowych i odbywa się jedynie raz. Dla głębi bitowej wynoszącej 32, wartość ta jest zgodna z tabelą 1.2 - Przesył danych dla różnych formatów przy wykorzystaniu float32, jednak jako że jest to sygnał gotowy do odtworzenia lub zapisu, to głębia bitów będzie przyjmować przeważnie wartości 16 lub 24. Zmienia to wymaganą przepustowość odpowiednio o połowę lub o jedną czwartą. Biorąc to pod uwagę można obliczyć dokładną wartość przepustowości wymaganej do działania całego systemu w zależności od obranego formatu audio oraz ilości urządzeń wejściowych, co zostało przedstawione w tabeli 6.2.

Tabela 6.2: Przesył danych pomiędzy CPU a GPU dla całego systemu

	liczba urządzeń wejściowych			
format	1	2	4	8
16b 44kHz stereo	5.808 MB/s	11.44 MB/s	22.704 MB/s	45.232 MB/s
16b 48kHz stereo	6.336 MB/s	12.48 MB/s	24.768 MB/s	49.344 MB/s
16b 96kHz stereo	12.672 MB/s	24.96 MB/s	49.536 MB/s	98.688 MB/s
16b 192kHz stereo	25.344 MB/s	49.92 MB/s	99.072 MB/s	197.376 MB/s
24b 44kHz stereo	5.896 MB/s	11.528 MB/s	22.792 MB/s	45.32 MB/s
24b 192kHz stereo	25.728 MB/s	50.304 MB/s	99.456 MB/s	197.76 MB/s
32b 44kHz stereo	5.984 MB/s	11.616 MB/s	22.88 MB/s	45.408 MB/s
32b 192kHz stereo	26.112 MB/s	50.688 MB/s	99.84 MB/s	198.144 MB/s

6.3 Podsystem komponentów

Każdy z komponentów, których implementacja została opisana w rozdziale **??** **??**, został przeniesiony na GPU w podobny sposób. Struktura `componentSettings` przechowująca ustawienia komponentu, otrzymała, obok wskaźnika na tablicę wartości ustawień, wskaźnik na bufor karty graficznej przechowujący kopie tych wartości. Rozwiązanie te zostało zastosowane również dla innych struktur, pozwala ona na

szybki dostęp do wartości zarówno po stronie CPU, jak i GPU, bez konieczności zbędnego przesyłania danych. Wszelkie operacje, które były wcześniej wykonywane na strumieniach dźwięku, zostały przeniesione do kerneli obliczeniowych. Każdy z komponentów został zaimplementowany przy wykorzystaniu metody opisanej za pomocą 2.3 - Wykorzystanie n wątków do przetworzenia n próbek.
Komponent `Component_Compressor` został wyjątkowo zastapiony komponentem `Component_SimpleCompressor_CUDA`, implementującym prostszy algorytm kompresji, który poprzez pozbycie się zależności czasowej pomiędzy próbkami, pozwala na efektywne wykorzystanie mocy obliczeniowej karty graficznej. Implementacja początkowo przyjętego algorytmu okazała się zbyt trudna, ze względu na operowanie na pojedynczym buforze, co uniemożliwiało zastosowanie podejścia 2.4 - Wykorzystanie m wątków do przetworzenia m buforów w sposób liniowy. Nowo wprowadzony algorytm wykonuje te samo zadanie - kompresuje sygnał dźwiękowy przekraczający określoną amplitudę, co pozawala na ograniczenie głośności sygnału, dodając minimalną ilość przesteru.

6.4 Pozostałe elementy systemu

Reszta systemu nie uległa zmianie w trakcie implementacji GPU. Wszelkie zmiany, które zostały wprowadzone, dotyczyły jedynie sposobu przetwarzania danych dźwiękowych. Zarówno podsystem statystyczny, interfejs użytkownika, jak i główna pętla programowa nie uległy zmianom. Podliczając wszelkie modyfikacje, które zostały wprowadzone w trakcie implementacji GPU, w przybliżeniu jedynie 25% kodu źródłowego uległo zmianie.

Rozdział 7

Wnioski

Przedstawiona implementacja systemu syntezującego i przetwarzającego sygnały dźwiękowe udowadnia, iż jest to możliwe przy użyciu karty graficznej. Problematyka przedstawionego zagadnienia jest wysoce kompatybilna z metodami równoległego przeważania 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 by być równie dobrze wykorzystane do tego rodzaju zadań w zależności od indywidualnych preferencji programisty oraz wymagań stawianych przed projektem. Powstały system bazujący na GPU bezproblemowo dorownuja, a niekiedy nawet przewyższa wydajnością swój odpowiednik powstały na CPU, pomimo iż kwestie 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 dostrzec wynikającą ze zmiany różnicę. // Warto również zwrócić uwagę na wymóg transferu danych pomiędzy CPU a GPU, który w przypadku przenoszenia dokładnych informacji o stanie klawiatury w czasie, skutkował przesyłaniem dużej ilości danych względem pojedynczego strumienia audio. Podejściem rozwiązującym problem mogłoby być zastosowanie algorytmów kompresji tego rodzaju danych. Praktycznym i uniwersalnym rozwiązaniem mogło by się okazać opracowanie odmiany formatu MIDI, przystosowanego do interpretacji przy wykorzystaniu GPU.

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 wzorzec 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 rozwiązanie 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 syntezatorów, 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 wieli 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

Spis tabel

Spis rysunków

Bibliografia