POLITECHNIKA WARSZAWSKA
WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

PRACA DYPLOMOWA MAGISTERSKA

INFORMATYKA

Optymalizator kompresji szeregów czasowych dla GPU

Time Series Compression Optimizer for GPU

Autor:

Karol Dzitkowski

Promotor: dr inż. Krzysztof Kaczmarski

Warszawa, luty 2016

....

podpis promotora podpis autora

Spis treści

Abstrakt

Wiele urządzeń takich jak czujniki, stacje pomiarowe, czy nawet serwery, produkują ogromne ilości danych w postaci szeregów czasowych, które następnie są przetwarzane i składowane do późniejszej analizy. Ogromną rolę w tym procesie stanowi przetwarzanie danych na kartach graficznych w celu przyspieszenia obliczeń. Aby wydajnie korzystać z GPGPU (General Purpose Graphical Processing Unit) przedstawiono szereg rozwiązań, korzystających z kart graficznych jako koprocesory w bazach danych lub nawet bazy danych ze składem po stronie GPU. We wszystkich rozwiązaniach bardzo istotną rolę stanowi kompresja danych. Szeregi czasowe są bardzo szczególnym rodzajem danych, dla których kluczowy jest dobór odpowiedniej kompresji wedle charakterystyki danych szeregu.

Niniejsza praca dyplomowa ma charakter badawczy i przedstawia nowe podejście do kompresji szeregów czasowych po stronie GPU, przy użyciu planera budującego na bieżąco schematy kompresji na podstawie statystyk napływających danych. Przedstawione rozwiązanie kompresuje dane za pomocą lekkich i bezstratnych kompresji w technologii CUDA. Celem jest stworzenie optymalizatora o wysokiej wydajności pod względem uzyskiwanego współczynnika kompresji dla danych o zmiennej charakterystyce.

Początek dokumentu stanowi opis problemu wraz z analizą istniejących rozwiązań i badań, pod kierunkiem kompresji z użyciem architektur SIMD (Single Instruction Multiple Data). Następnie opisana jest używana technologia stworzona przez firmę NVIDIA, oraz zastosowane algorytmy lekkiej kompresji dla GPU. Kolejne rozdziały zawierają opis implementacji algorytmu optymalizatora, wraz ze stworzonym środowiskiem i programem do równoległej kompresji kolumn danych. Na końcu znajdują się wyniki eksperymentów dowodzących przydatność takiego rozwiązania oraz opis możliwych rozszerzeń i usprawnień. Opisywana metoda ma zastosowanie we wszystkich rodzajach kolumnowych składów danych, co będzie przedmiotem dalszych prac badawczych.

Abstract

Many devices such as sensors, measuring stations or even servers produce enormous amounts of data in the form of time series, which are then processed and stored for later analysis. A huge role in this process takes data processing on graphics cards in order to accelerate calculations. To efficiently use the GPGPU (General Purpose Graphical Processing Unit) a number of solutions has been presented, that use the GPU as a coprocessor in a databases. There were also attempts to create a GPU-side databases. It has been known that data compression plays here the crucial role. Time series are special kind of data, for which choosing the right compression according to the characteristics of the data series is essencial.

This paper is a research and presents a new approach to compression of time series on the side of the GPU, using a planner to keep building the compression scheme based on statistics of incoming data, in the incremental manner. The solution compresses columnar data using lightweight and lossless compressions in CUDA technology. The aim of the work is to create an optimizer with high performance in terms of obtained compression ratio for data of variable characteristics.

The beginning of the document is a description of the problem, along with an analysis of existing solutions and research, under the direction of compression using SIMD (Single Instruction Multiple Data) architectures. Further it describes adopted technology developed by NVIDIA and implemented algorithms of light compression for GPU. The following sections describe the implementation of the optimizer algorithm, along with created environment and a program for parallel compression of data columns. At the end are the results of experiments demonstrating the usefulness of such a solution and a description of further work that will be conducted in the topic. This method is applicable in all types of columnar data warehouses, which will be the subject further research.

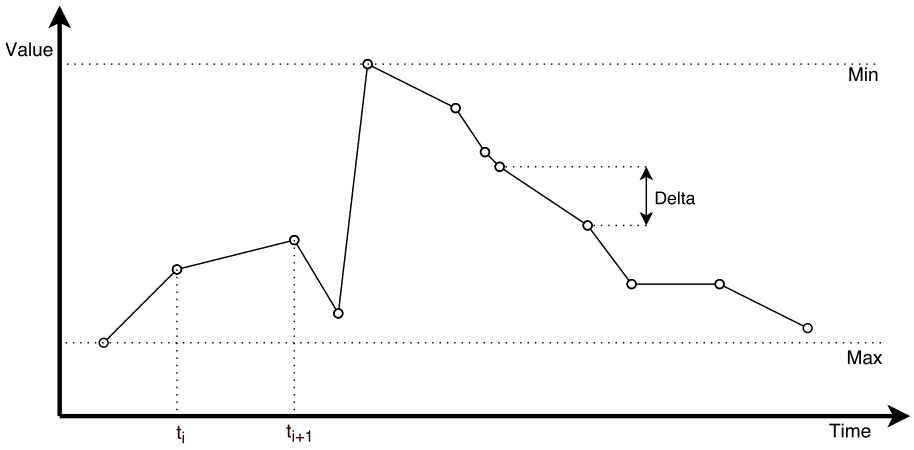
Rozdział 1

Wstęp

Poniższa praca zawiera opis implementacji optymalizatora kompresji szeregów czasowych, bazującego na dynamicznie generowanych statystykach danych. Pomysł opiera się na tworzeniu planów kompresji (kompresja kaskadowa) oraz zbieraniu informacji o parach kodowań w nich występujących - jak dobrze sprawdzają się dla napływających danych. System będzie również dynamicznie je zmieniał - korygował, w zależności od charakterystyki kolejnych paczek danych. W założeniu program ma umożliwić kompresję dużych ilości danych przy wykorzystaniu potencjału obliczeniowego współczesnych kart graficznych.

1.1 Opis problemu

Terabajty danych w postaci szeregów czasowych są przetwarzane i analizowane każdego dnia na całym świecie. Zapytania i agregacje na tak wielkich porcjach danych jest czasochłonne i wymaga dużej ilości zasobów. Aby zmierzyć się z tym problemem, powstały wyspecjalizowane bazy danych, wspierające analizę szeregów czasowych. Ważnym czynnikiem w tych rozwiązaniach jest kompresja oraz użycie procesorów graficznych w celu przyspieszenia obliczeń. Aby przetwarzać dane na GPU bez konieczności ich ciągłego kopiowania poprzez szynę PCI-E, powstają ba-



Rysunek 1.1: Szereg czasowy

zy danych po stronie GPU (najczęściej rozproszone), takie jak MapD[1] [1] lub DDJ[2]. Innymi rozwiązaniami są koprocesory obliczeniowe GPU, wspomagające działanie baz takich jak Cassandra, HBase, TepoDB, OpenTSDB czy PostgreSQL. Charakterystyka danych wielu szeregów wskazuje, że przy odpowiedniej obróbce mogą być kompresowane z bardzo dużym współczynnikiem kompresji, szczególnie jeśli byłoby możliwe kompresowanie za pomocą dynamicznie zmieniających się sekwencji (różnych) algorytmów kompresji i transformacji danych - kodowań. Dla przykładu, jeśli jakiś fragment szeregu jest stały, z nielicznymi wyjątkami, warto byłoby oddzielić wyjątki, a resztę skompresować jako jedną liczbę - uzyskując współczynnik kompresji rzędu długości danych.

Definicja 1.1.1 (Współczynnik kompresji)**.** Stosunek rozmiaru danych wejściowych s_{in} kompresji A, do rozmiaru danych wyjściowych s_{out}, będziemy nazywać współczynnikiem kompresji i ozn. $C(A) = \frac{s_{in}}{s_{out}}$.

1.1.1 Forma szeregu czasowego

Postać szeregu czasowego prezentuje rysunek 1.1. Dane w tej postaci są próbkami pewnych pomiarów w czasie t_i, gdzie czas próbkowania nie musi być stały ($\Delta t \neq$

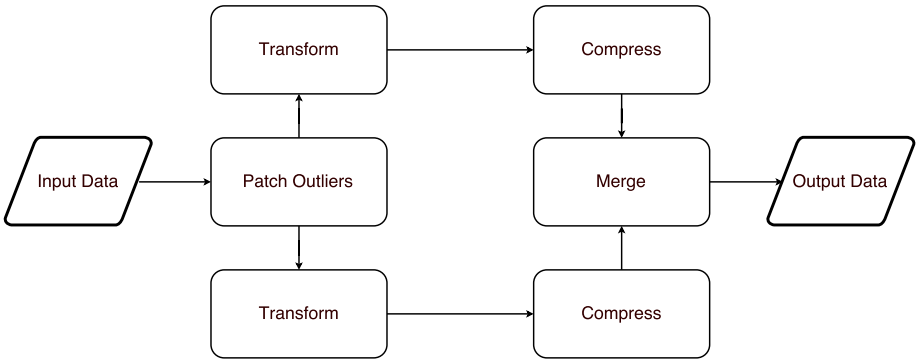
[1]Baza danych GPU z MIT - `http://www.mapd.com/`

[2]Rozproszona baza danych szeregów czasowych na klastrze obliczeniowym GPU, zaproponowana we wcześniejszej pracy - `https://github.com/d-d-j`

$const$). Ponadto zakładamy, że wartości pomiędzy dwoma pomiarami można przybliżyć stosując interpolację liniową, co może być przydatne np. przy dodawaniu dwóch szeregów. Najważniejsze jednak jest to, że jako pomiary pewnej wielkości, dane te najczęściej zmieniają się w charakterystyczny dla nich sposób, z pewnymi odstępstwami w postaci błędów, które nazywać będę z ang. *Outliers*.

1.2 Lekka kompresja z SIMD

Bazy danych przechowujące szeregi czasowe są najczęściej zorientowane kolumnowo oraz stosują metody lekkiej kompresji w celu oszczędności pamięci. W tych przypadkach stosuje się metody lekkiej kompresji, takie jak kodowanie słownikowe, delta lub stałej liczby bitów, zamiast bardziej skomplikowanych i wolniejszych metod, które często zapewniłyby lepszy poziom kompresji. Systemy te ładują swoje dane do pamięci trwałej paczkami, które mogą być kompresowane osobno, przy użyciu różnych algorytmów, zmieniających się dynamicznie w czasie. Takie kolumny wartości numerycznych tego samego typu wydajnie przetwarza się przy użyciu procesorów typu SIMD[3], co daje wielokrotne przyspieszenie w stosunku do tradycyjnych architektur. Okazuje się że większość algorytmów lekkiej kompresji może z dużym po-



Rysunek 1.2: Przykład prostej kompresji kaskadowej

wodzeniem być w ten sposób zrównoleglona [2]. Również dynamiczne generowanie statystyk napływających danych może być przyspieszone z użyciem SIMD, co

[3]Single Instruction Multiple Data (patrz str. 10)

otwiera możliwość implementacji wydajnych systemów, dynamicznie optymalizujących użyte kompresje w celu zwiększenia współczynnika kompresji danych. Dodatkowo użycie kaskadowej kompresji może wielokrotnie wzmocnić poziom kompresji, pod warunkiem stworzenia dobrego planu kompresji, właśnie na podstawie wygenerowanych statystyk. Pomysł takiego schematu pokazany pokazany jest na rysunku 1.2. Ogromna moc obliczeniowa procesorów graficznych może pozwolić wygenerować odpowiedni plan w akceptowalnym czasie. Takie użycie jest możliwe np. w bazach danych po stronie GPU, gdzie jest to niezmiernie ważne z powodu ścisłego limitu pamięci na kartach i ich wysokiego kosztu.

1.3 Zawartość pracy

W tej pracy przedstawię planer kompresji (optymalizator) kompresujący napływające paczki danych. Zaprezentuję nowe podejście do planera budującego drzewa kompresji i uczącego się ich konstrukcji na podstawie statystyk węzłów takich drzew generowanych w czasie rzeczywistym[4]. Przedstawię również zaimplementowane środowisko oraz użyte algorytmy lekkiej kompresji. W ramach tej pracy stworzone zostały 4 biblioteki, wykorzystujące technologię *NVIDIA CUDA*, tworzące framework optymalizatora kompresji oraz program kompresujący kolumny podanego szeregu czasowego w sposób równoległy. Następny podrozdział zawiera opis wcześniejszych prac, a także krótki opis technologii, w tym architektury CUDA. W rozdziale 2 omówię metody lekkiej kompresji, ze szczególnym uwzględnieniem kompresji FL oraz GFC. Rozdział 3 jest w całości poświęcony optymalizatorowi kompresji oraz generowaniu drzew i statystyk. Następnie przedstawię implementację systemu oraz programu konsolowego. Ostatnie rozdziały 5 i 6 to wyniki prac i eksperymentów, a także podsumowanie oraz zakres przyszłych prac i optymalizacji.

[4]jak również statystyk napływających danych

1.4 Powiązane prace

1.4.1 Szeregi czasowe

Szeregi czasowe są typem danych dla których istnieje wiele efektywnych sposobów kompresji, zależnych od ich charakterystyki. W wielu pracach przedstawiono podejścia do tego problemu od strony lekkiej kompresji [3]. Najczęstszymi z nich są kodowanie ekstremami [4], stałej długości bitów [5], czyli tzw. NULL Suppression (NS)[6] oraz proste kodowania słownikowe np. wszystkich unikalnych wartości, które można zakodować pewną założoną z góry liczbą bitów [7].
Dodatkowo do kompresji szeregów stosuje się metody regresji [8]. Autor stosuje Piecewise Regression - regresję odcinkową, polegającą na przybliżaniu kawałków szeregu funkcją, np. wielomianem. Ma to swoją wersję stratną jak i bezstratną, gdzie możemy zapisać różnicę od zadanej funkcji i wynik zapisać na mniejszej liczbie bitów.
Analizuje się także wiele sposobów kompresji liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji. Najczęściej próbuje się zamienić liczbę wymierną (w reprezentacji maszynowej), na całkowitą, stosując skalowanie [9]. Istnieją też bardziej skomplikowane metody na przykład kompresji liczb double algorytmem FPC [10], który kompresuje liniową sekwencję liczb o podwójnej precyzji (IEEE 754), sekwencyjnie przewidując każdą wartość, a następnie wykonując operację XOR z prawdziwą wartością szeregu, po czym usuwane są wiodące zera.

1.4.2 SIMD SSE

Biorąc pod uwagę algorytmy lekkiej kompresji dla szeregów, warto zwrócić uwagę na udane próby optymalizacji z użyciem prostego SIMD jakim są operacje wektorowe SSE na procesorach Intel [2, 11]. W tych pracach pokazano przekład algorytmów kodowania z wyrównaniem do bajtów (Byte-Aligned Coding) oraz do słów (Word-Aligned Coding) i zmierzono wydajność implementacji wektorowej wersji tych kodowań z użyciem SSE. Autorzy zastosowali również binarne pakowanie (Bi-

nary Packing) w formie algorytmu FOR (Frame of Reference) [12] dzieląc dane na bloki o długości 128 elementów (zmiennych całkowitych o długości 32 bitów) i stosując patchowanie. Taki algorytm okazał się najwydajniejszy. Pokazano, że bez spadku jakości kompresji można uzyskać w ten sposób wzrost szybkości kompresji od 2 do 4 razy w stosunku do tradycyjnej implementacji tego algorytmu.

1.4.3 Obliczenia GPU

Dzięki ogromnej mocy obliczeniowej kart graficznych uzyskano znaczący wzrost wydajności wielu algorytmów dających się w mniejszym lub większym stopniu zrównoleglić. Przykładowymi algorytmami o tej właściwości są choćby radix sort [13], hashowanie kukułcze [14], sumy prefixowe [15] i inne zaimplementowane w podstawowych bibliotekach takich jak CUDPP[5] czy Thrust[6].
Najważniejsze są jednak bardzo zadowalające rezultaty zrównoleglania algorytmów używanych w bazach danych takich jak index search [16], wszelkiego rodzaju agregacje i operacje join, scatter i gather [17] oraz obliczanie statystyk danych [18], jak również dopasowywanie wyrażeń regularnych [19]. Dla przykładu, wzrost wydajności oferowany przez algorytmy z biblioteki Thrust, która jest niejako odpowiednikiem *STL*, jest średnio 10-krotny[7] w stosunku do najszybszych wersji CPU. Większość przytoczonych wyżej przykładów również reprezentuje wzrost wydajności o rząd wielkości. W pracach na temat akceleracji baz danych za pomocą technologii CUDA, autorzy otrzymują przyspieszenie $20 - 60$ krotne [20], w przypadku operacji *SELECT WHERE* i *SELECT JOIN* z agregacjami [17]. Powyższe wyniki zwykle są osiągane bez uwzględniania czasu kopiowania danych na GPU, co jak obliczono zajmuje to średnio ok 90% czasu działania algorytmów [21].

[5] CUDA Data Parallel Primitives - `http://cudpp.github.io/`

[6] Parallel algorithms library - `https://developer.nvidia.com/thrust`

[7] `http://developer.download.nvidia.com/compute/cuda/compute-docs/cuda-performance-report.pdf`

1.4.4 Kompresje

Opisane wyżej algorytmy lekkiej kompresji szeregów czasowych, w większości zostały zaimplementowane przez Fang et al.[22] oraz Przymus et al.[21] w ich pracach, gdzie autorzy zaznaczają ogromny wzrost przepustowości takich algorytmów oraz ich wysoką skuteczność w kompresji szeregów. W przypadku pierwszego jest to nawet 56 GB/s dekodowania, a dla drugiego od 2 do 40 GB/s kodowania w zależności od stopnia skomplikowania algorytmu. Większość przewidzianych metod ma swoje odpowiedniki z patchowaniem, w którym elementy niepasujące odkładane są do osobnej tablicy wyjątków. Prace te odnoszą się jednak tylko do liczb całkowitych, a najczęściej całkowitych bez znaku (naturalnych).
Lekką kompresję liczb zmiennopozycyjnych o podwójnej precyzji przestawił w swojej pracy O'Neil et al.[23], w której stosując technologię CUDA i dzieląc dane na odpowiednie bloki, zastosowano wariację algorytmu FOR i osiągnięto bardzo dobre wyniki rzędu 75 Gb/s kodowania oraz aż 90 Gb/s dekodowania (daje to podobne rezultaty jak implementacja algorytmu FL dla liczb naturanlych[5]). Algorytm nazwano GFC, a jego uogólnienie dla liczb o pojedynczej precyzji przedstawię dokładnie w kolejnym rozdziale.
Wzrost wydajności osiągnięto również na tle bardziej skomplikowanych metod (dających często lepsze współczynniki kompresji) jak kodowanie Huffmana[24, 25], gdzie równoległa implementacja na GPU uzyskała *2-5* krotne przyspieszenie, natomiast przepustowość takiej kompresji to dla porównania 300 do 500 MB/s. Próby zoptymalizowania algorytmów takich jak LZSS w pracach Ozsoy et al.[26], skończyły się lekkim (max 2.2x) wzrostem wydajności (w stosunku do wielordzeniowych implementacji CPU), przy uzyskanej przepustowości 1700 Mb/s w konfiguracji z dwoma kartami GPU[27]. Również inni autorzy mają nadzieję na optymalizację z użyciem kodowań słownikowych, takich jak LZW, pobijając wyniki CPU po przepisaniu ich do architektury SIMD[28]. Można dodatkowo spotkać wariacje algorytmu LZSS przepisanego na CUDA, takie jak CANLZSS[29], która według autora przewyższa wydajnością ponad 60 razy seryjną implementację zwykłego LZSS. Kolejna optymalizacja GLZSS, w której zreorganizowano słownik do postaci tablicy haszy,

oraz przyspieszono porównywanie podciągów[8], osiągając dwukrotne przyspieszenie względem poprzednich prac[30].
Porównując szybkość działania oraz współczynniki kompresji uzyskane przez autorów, można dojść do wniosku, że zastosowanie metod lekkich kompresji na GPU dla szeregów czasowych, w miejscach gdzie szczególne znaczenie ma szybka dekompresja, jest uzasadnione.

1.4.5 Planery kompresji

W celu wielokrotnego zwiększenia współczynnika kompresji szeregu czasowego, warto go przetransformować przed kodowaniem lub skompresować wielokrotnie różnymi algorytmami. Powstały systemy planujące serię kodowań - planery kompresji [5], które działają na zasadzie kodowania kaskadowego, czyli ciągu następujących po sobie kodowań tworzących ciąg. Ze względu na to, że kodowanie może posiadać wiele tablic wynikowych, schematem takiej kompresji jest drzewo. Dzięki zastosowaniu procesorów graficznych osiągnięto bardzo dobre wyniki zarówno pod względem współczynnika kompresji, jak i szybkości działania. Dla przykładu przepustowość kodowania 45 GB/s oraz dekodowania 56 GB/s została osiągnięta przez Fang et al. [22]. W tym przypadku posługując się heurystykami wykorzystującymi statystyki danych wejściowych, spośród ogromnej ilości dostępnych schematów kodowania, wyznaczano najlepiej pasujące[9]. Następnie wybierano spośród nich plan spełniający zdefiniowane normy, np. plan o największym współczynniku kompresji. Statystyki na których oparty był algorytm pozyskiwano z informacji o kolumnie w bazie danych. Zanotowano dużo lepszą kompresję niż w przypadku tradycyjnego kodowania pojedynczą metodą dla realistycznych danych.
Kolejnym, podobnym podejściem do planera jest praca Przymus, Kaczmarski [9], w której plan złożony jest z trzech warstw metod następujących po sobie: transformacji, kodowania bazowego i pomocniczego. Cechą charakterystyczną tego rozwiązania jest dynamiczny generator statystyk, który uaktualnia statystyki w momen-

[8] również zrównoleglając je na GPU
[9] np. dla danych posortowanych powinny zaczynać się od RLE itd.

cie tworzenia planu, wykorzystując właściwości poszczególnych metod takiej kompresji[10]. Dodatkowo praca ta implementuje znajdowanie optimum ze względu na dwie sprzeczne zmienne (bi-objective optimizer): szybkość działania i jakość kompresji, stosując optymalność Pareto [31]. Generowanie statystyk dla takiego planera na GPU zapewnia do 70 razy lepszą wydajność w stosunku do analogicznej implementacji na CPU [21].

1.5 Technologie

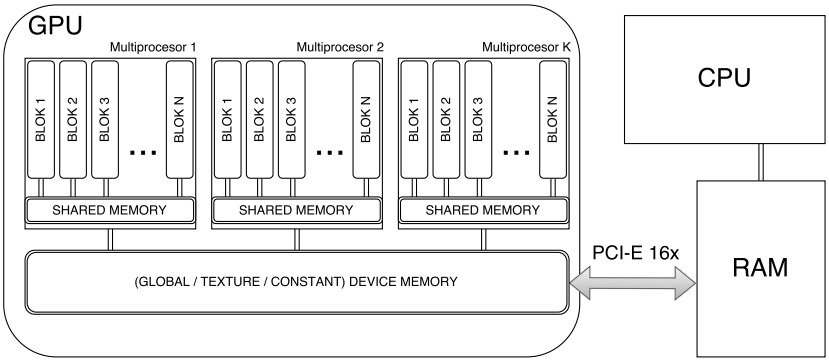
Procesory graficzne stały się znaczącymi i potężnymi koprocesorami obliczeń dla wielu aplikacji i systemów takich jak bazy danych, badania naukowe czy wyszukiwarki www. Nowoczesne GPU posiadają moc obliczeniową o rząd większą niż zwykle, wielordzeniowe procesory CPU, takie jak AMD FX 8XXX czy Intel Core i7. Dla przykładu flagowa konstrukcja firmy NVIDIA - GeForce GTX Titan X osiąga moc 6600 GFLOPS (miliardów operacji zmiennoprzecinkowych na sekundę), przy 336 GB/s przepustowości pamięci, podczas gdy najszybsze procesory takie jak Intel Core i7-5960x osiągają niecałe 180 GFLOPS, przy przepustowości 68 GB/s. Kartom graficznym dorównują tylko inne jednostki typu SIMD, na przykład karty obliczeniowe Xeon Phi. Pomimo tak oszałamiających wyników, programowanie na jednostkach SIMD jest o wiele trudniejsze, jak również ograniczone przepustowością szyny PCI-E, która dla wersji 3.0 x16, wynosi nie więcej niż 16 GB/s. Przemawia to za użyciem kompresji przy przetwarzaniu danych, choćby w celu przyspieszenia kopiowania z i na kartę graficzną.

1.5.1 CUDA

Jedną z wielu zalet architektury kart graficznych jest to, że składają się z kilku lub kilkunastu multiprocesorów (SMs - Streaming Multiprocessors) architektury SIMD. Jest to właściwie architektura typu SIMT, gdzie multiprocesor wykonuje wątki w grupach o liczności 32 zwanymi *warps*. Architektura ta jest zbliżona do SIMD z tą

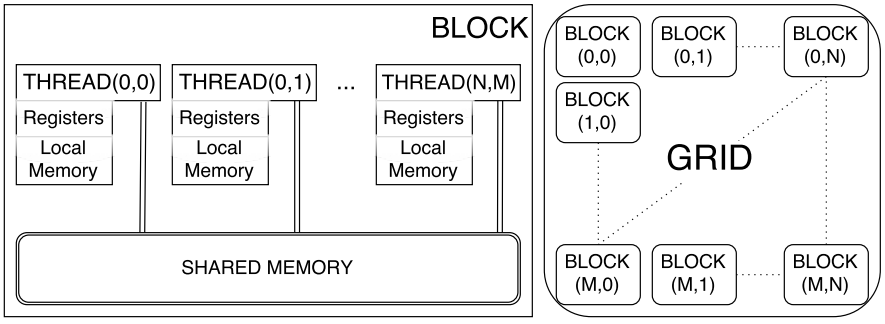
[10] szczególnie w przypadku minimalnej ilości bitów potrzebnych do zapisania każdej liczby

różnicą, że to nie organizacja wektora danych kontroluje jednostki obliczeniowe, a organizacja instrukcji pojedyńczego wątku. Umożliwia on zatem pisanie równolegle wykonywanego kodu dla niezależnych i skalowalnych wątków, jak i dla tych koordynowanych danymi.



Rysunek 1.3: Architektura NVIDIA CUDA

Wszystkie wątki wykonują ten sam kod funkcji, nazywanej kernelem. Ponadto CUDA tworzy abstrakcję bloków wątków, które zorganizowane są w siatkę (GRID) i współdzielą zasoby multiprocesora. Ważna jest również hierarchia pamięci, w której część przydzielana jest wątkom w postaci pamięci lokalnej i rejestrów, oraz pamięci współdzielonej przez wątki z tego samego bloku (Shared Memory) - te pamięci muszą być znane w trakcie kompilacji kernela. Najwolniejsza jest pamięć globalna (Device Memory), wspólna dla wszystkich wątków, wszystkich bloków, na wszystkich multiprocesorach. Dokładny opis tej architektury można znaleźć bezpośrednio na stronie producenta[11].



Rysunek 1.4: Abstrakcja bloków i siatki w CUDA

[11] http://docs.nvidia.com/cuda/

Rozdział 2

Algorytmy lekkiej kompresji

Poniżej przedstawię kodowania użyte w implementacji planera kompresji oraz ich modyfikacje, a następnie opiszę szczegółowo dwa najważniejsze, które zwykle kończą ścieżki kompresji w generowanych planach. Są to kodowania bazujące na pomyśle usuwania zbędnych zer wiodących, dla liczb naturalnych - AFL oraz ułamkowych - GFC. Trzeba zauważyć, że operacja patchowania jest tutaj zaimplementowana odmiennie niż w innych publikacjach, jako osobny rodzaj kodowania, mogący przybierać wiele form, natomiast algorytmy nie mają swoich wersji z patchowaniem.

2.1 Global memory coalescing

Tablice zaalokowane w pamięci urządzenia GPU Nvidia są wyrównane do bloków o wielkości 256 bajtów przez sterownik urządzenia. Urządzenie może dostać się do pamięci przez 32, 64 lub 128 bajtowe transakcje ,które są wyrównane do ich wielkości. Odczytując lub zapisując do pamięci globalnej, ważne jest zatem aby wątki wymagały dostępu zawsze do kolejnych rekordów tablicy, najlepiej wszystkie należące do tego samego *warp*. Dla osiągnięcia takiego rezultatu można zastosować poniższy algorytm obliczania indeksów wejściowych dla danego wątku CUDA.
Oznaczmy jako:

- Σ – ilość przetwarzanych elementów

- ω – ilość wątków w grupie
- κ – ilość elementów w bloku danych
- λ – ilość elementów przetwarzanych przez pojedynczy wątek
- B_{size} – wielkość bloków wątków – (CUDA block size)
- B_{count} – ilość bloków – (CUDA block count)
- w_g – ilość grup w bloku
- $W_{lane}(t) = t_{idx}(mod\ \omega)$ – indeks wątku t w grupie – (warp lane)

Załóżmy że chcemy przetworzyć 1MB danych tj. $1024 \cdot 1024$ elementy, używając bloków z 4 grupami po 32 wątki każdy, czyli 4 warpy. Ponadto chcemy aby bloki danych miały 32 rekordy, a każdy wątek przetwarzał 16 elementów: $\Sigma = 1024 \cdot 1024$, $\omega = 32, \quad \kappa = 32, \quad \lambda = 16, \quad w_g = 4$.
Bardzo łatwo można obliczyć ile wynosi rozmiar pojedynczego bloku B_{size}, oraz ilość wszystkich bloków, potrzebnych do przetworzenia wszystkich elementów wektora wejściowego B_{count}:

$$B_{size} = \omega \cdot w_g$$
$$B_{count} = \frac{\Sigma + B_{size} \cdot \lambda - 1}{B_{size} \cdot \lambda}$$

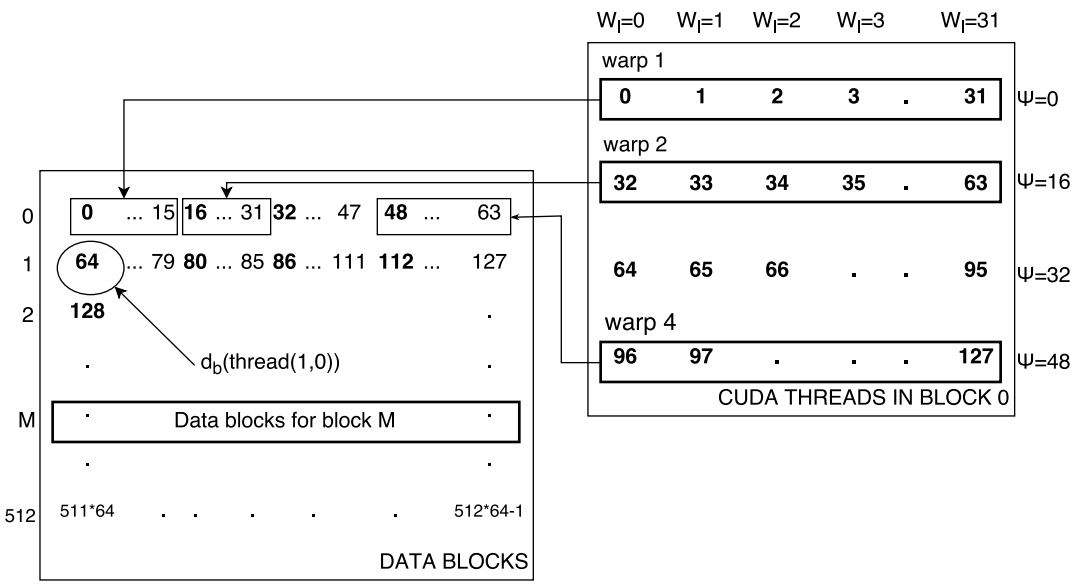
Przyjmując początkowy indeks bloków danych dla wątku t o indeksie $t_{idx}(t)$ z bloku $b_{idx}(t)$ za $\Psi(t) = (t_{idx}(t) - W_{lane}(t)) \cdot \lambda/32$, indeks bloku danych dla danego wątku można obliczyć za pomocą wzoru:

$$d_b(t) = b_{idx}(t) \cdot w_g \cdot \lambda + \Psi(t)$$

co dla naszego przykładu obrazuje rysunek 2.1.

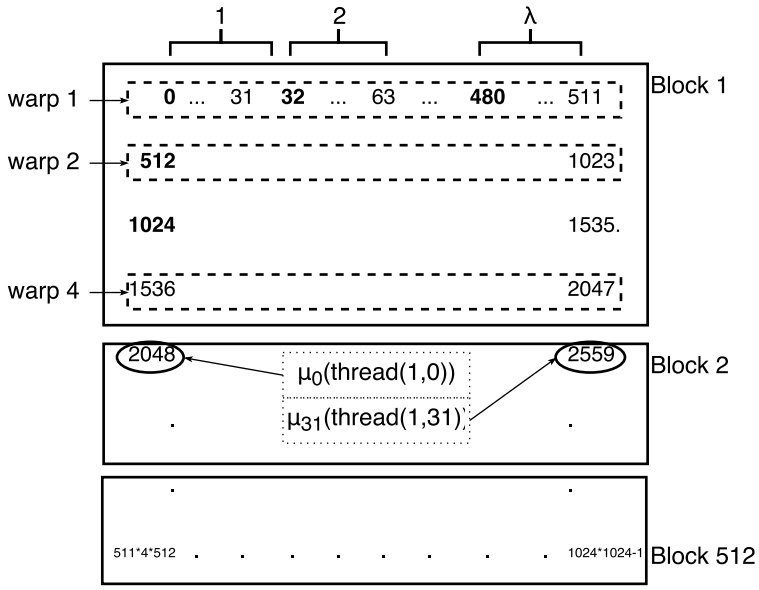
Każdej grupie w bloku CUDA odpowiada tyle samo bloków danych, przy czym bloki danych są wielkości κ elementów i ich numery są potrzebne do obliczenia początkowego indeksu wejściowego dla wątku, który definiujemy jako

$$\mu_0(t) = d_b(t) \cdot \kappa + W_{lane}(t)$$



Rysunek 2.1: Przejście z ułożenia wątków w bloku na bloki danych

Kolejne indeksy wyliczane są jako przesunięcia o ω, czyli $\mu_k(t) = \mu_0(t) + k \cdot \omega$, dla $k \in [1, \kappa - 1]$. Tworzy to podział danych wejściowych, który zgodnie z przykładem obrazuje rysunek 2.2.



Rysunek 2.2: Indeksy danych wejściowych

Dzięki takiemu ułożeniu kolejne wątki z tego samego warp, wykonują odczyt kolejnych rekordów z tablicy źródłowej, ponieważ wątek 0 odczytuje rekord 0, wątek

1 rekord 1, aż wreszcie wątek ω rekord ω. Następnie po przesunięciu o ω wątki ponownie odczytują dane, które są ze sobą sąsiadujące. Dzięki temu odczyt może być połączony (Global Memory Coalescing [32]). Łączny odczyt może być wielokrotnie szybszy niż odczyty z losowych indeksów, co jest kluczowe przy budowie bardzo wydajnych algorytmów przetwarzających dane na GPU. W przypadku CUDA grupa powinna mieć liczność 32, co odpowiada ilości wątków w jednym *warp*.

2.2 Podstawowe algorytmy transformacji szeregów

Delta – Zapisuje różnice pomiędzy kolejnymi danymi w szeregu, rejestrując pierwszą wartość w metadanych. Bardzo dobrze sprawdza się na danych posortowanych bądź o zmianach liniowych o stałej wartości. Algorytm ten został zaimplementowany częściowo z użyciem biblioteki Thrust i funkcji *inclusive_scan* - do dekodowania.

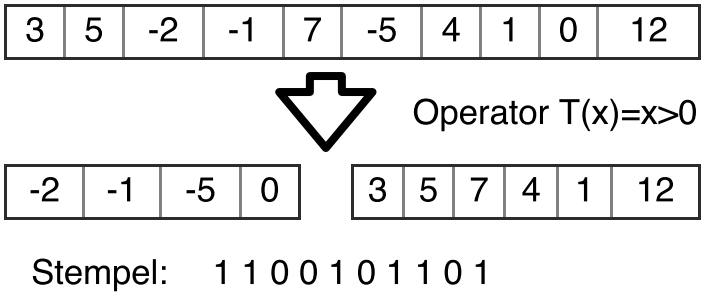
Scale – Bardzo prosta transformacja, odmienna od wielu implementacji o tej samej nazwie, polegająca na odjęciu lub dodaniu (dla liczb ujemnych) najmniejszej dodatniej wartości szeregu. Bardzo dobrze sprawdza się dla dużych wartości szeregu o małej wariancji. Dla przykładu, mając wartości:

$$1234000, ..., 1234500, ..., 1234999$$

dane zostaną zapisane jako $0, 1, ..., 999$ i będą mogły być zredukowane do dużo mniejszej liczby bitów.

FloatToInt – Wersja algorytmu, którą często w literaturze nazywa się mianem *Scale*. Znając maksymalną precyzję danych zmiennopozycyjnych, możliwe jest ich zapisanie w postaci liczb całkowitych, mnożąc przez odpowiednią potęgę liczby 10. Oznacza to, że mając wektor cen w złotówkach, można przemnożyć cenę 99.99 przez 100 otrzymując 9999 i zmienić reprezentację liczb na całkowite. W wyniku takiego działania reprezentacja liczb całkowitych może ulec lepszej kompresji.

Patch – Bardzo ważnym zadaniem w kompresji szeregów czasowych jest usuwanie wartości odstających (ang. *outliers*). Kodowanie to dzieli wektor danych na dwa wektory względem zdefiniowanego operatora, mówiącego np. że jako wartości odstające należy uznać wszystkie wartości przekraczające 90% wartości maksymalnej. W ten sposób może być zdefiniowanych wiele różnych wersji „patchowania", dla przykładu, dzieląc wartości na ujemne i dodatnie.



Rysunek 2.3: Przykład użycia patchowania z operatorem - "większy od zera"

W przeciwieństwie do innych prac algorytm ten nie zapisuje wyjątków wraz z ich pozycjami, umożliwiając lepsze ich skompresowanie w dalszych krokach. Zamiast tego przechowuje w metadanych zapisany bitowo stempel przynależności poszczególnych elementów wektora do pierwszej lub drugiej tablicy wynikowej. Rozmiar takich metadanych wynosi $(N + 32)$ bitów, gdzie N to liczba kompresowanych elementów, która dopisywana jest na początku stempla. Do zapisu tej liczby używana jest zmienna 32 bitowa, ponieważ $10^6 < N < 10^9$.

2.3 Algorytmy kompresji szeregów

RLE – kodowanie długości serii (Run-Length-Encoding) to sposób kompresji polegający na zapisie ciągów takich samych wartości, jako wartość i długość tego ciągu. W tym przypadku obie te wartości trafiają do 2 różnych tablic, które następnie mogą być kompresowane osobno. Szczególnie dobrze sprawdza się w przypadku posortowanych danych, lub często się powtarzających. Dla przykładu wektor $5, 5, 5, 5, 1, 1, 1, 1, 17, 17, 17, 17$ zostanie skompresowany do $5, 1, 17$

oraz $4, 4, 4$. Data technika kompresji jest opłacalna jeśli średnia długość ciągów przekracza 2 ($D_{sr} > 2$).

Dana metoda została zaimplementowana z użyciem biblioteki Thrust, stosując między innymi metodę *reduce_by_key*.

Dict – Kolejną grupą kompresji są kodowania bazujące na pomyśle słownikowym ($Dict_K$). Wykorzystują one informację o liczności poszczególnych wartości w wektorze. Kompresja słownikowa wykorzystuje K najczęściej występujących wartości i zapisuje ich tablicę w metadanych. Następnie wartości z wektora danych są kodowane indeksami w tej tablicy, przy użyciu jak najmniejszej liczby bitów $bit_{cnt} = log_2(K)$. Reszta niepasujących wartości zapisywana jest do osobnej tablicy, co działa podobnie jak w kodowaniu *Patch*.

Unique, Const – Zoptymalizowane wersje kodowania $Dict_K$ dla $K = 1$ (*Const*), oraz $K = N_u$ (*Unique*), gdzie N_u jest liczbą unikalnych wartości w całym kompresowanym wektorze. Dla przykładu, jeśli wszystkie wartości są równe, z nielicznymi wyjątkami, kodowanie *Const*, zapisze najczęstszą wartość w metadanych i stworzy tablicę wyjątków, uzyskując bardzo wysoki stopień kompresji.

2.3.1 Kodowanie AFL

Ten rodzaj kodowania (Aligned Fixed-Length Encoding) nazywany jest kodowaniem o stałej długości z wyrównaniem. Ogólny pomysł algorytmu jest bardzo prosty i bazuje na algorytmie NS (null suppression), czyli usuwaniu wiodących zer z liczb i zapisywanie ich na mniejszej ilości bitów. W tym przypadku ilość bitów na których zapisujemy liczby jest znana z góry przed rozpoczęciem kodowania i nie ulega zmianie. Dla uproszczenia przyjmijmy, że kompresujemy liczby naturalne o długości 32 bitów, np. *unsigned int*. Wyrównanie polega na grupowaniu wykonawczych wątków w grupy o pewnej liczności. W naszym przypadku będzie to liczba 32, czyli liczba wątków należących do jednego *warpa*, z uwagi na wykorzystanie tzw. *łącznego dostępu do pamięci globalnej CUDA*. Wtedy liczba kompresowanych elementów musi

być wielokrotnością długości kodowanego słowa pomnożonej przez 32, czyli 1024. Kodowanie przebiega następująco:

1. Obliczana jest najmniejsza liczba bitów potrzebna do zakodowania każdego ze słów w wektorze, nazwijmy go W i oznaczmy liczbę bitów jako σ:

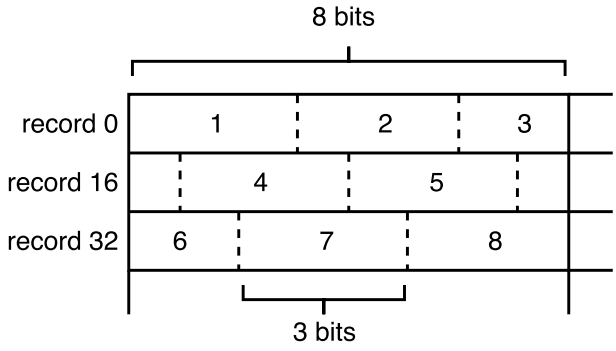
$$w = max(W)$$

$$\sigma = \begin{cases} log_2(w) + 1 & \text{dla } w > 0 \\ 32 & \text{dla } w \leq 0 \end{cases}$$

2. Następnie wektor elementów jest dopełniany, aby jego długość była wielokrotnością 1024, a liczba dopełnionych elementów wraz z potrzebną do zakodowania każdej wartości liczbą bitów σ zapisywana jest do metadanych.

3. Dla każdego wątku t CUDA ozn. $t = thread(b_{idx}, t_{idx})$, co oznacza wątek o indeksie t_{idx} z bloku b_{idx}, wyznaczany jest początkowy indeks danych wejściowych i wyjściowych dla tego wątku, zgodnie z algorytmem przedstawionym w poprzedniej sekcji. Wielkość bloków wątków ustalmy na $B_{size} = 32 \cdot 8$ co sprawi, że każdy blok będzie się składał z 8 warpów, a grupy będą miały liczność 32 ($\omega = 32$, $w_g = 8$). Ponadto, chcemy aby bloki danych były wielkości 32 elementów ($\kappa = 32$). Mając dane wejściowe o indeksach $\mu_0, \mu_1, ..., \mu_{\kappa-1}$ wątek zapisuje σ dolnych bitów każdej z liczb spod tych indeksów, do rekordów tablicy wynikowej. Wielkości bloków są tak dobrane, aby każdy wątek kodował κ liczb używając σ bitów i otrzymując w ten sposób σ liczb o wielkości κ. Załóżmy, że $\kappa = 8$ oraz $\sigma = 3$, wtedy 2.4 obrazuje ułożenie danych wejściowych o wielkości 8 bitów, skompresowanych do wielkości 3 bitów w wektorze wynikowym, przyjmując 16 ilość wątków w grupie ($\omega = 16$). Jak widać skompresowane dane idealnie mieszczą się w 3 rekordach tablicy wynikowej.

   Początkowy indeks tablicy wynikowej pod który wątek ma zapisać skompresowane dane, wyliczany jest jako $\nu_0(t) = d_b(t) \cdot \sigma + W_{lane}(t)$. Do rekordu zapisywanych jest możliwie najwięcej bitów (tak jak pokazuje 2.4), po czym jeśli



Rysunek 2.4: AFL - zapis w tablicy wyjściowej

brakło miejsca dalsze bity przenoszone są do następnego rekordu, którego indeks wyliczany jest jako $\nu_k(t) = \nu_0(t) + k \cdot \omega$, gdzie $k \in [1, \sigma - 1]$. Tutaj również wątki posługują się łącznym dostępem do pamięci.

4. Po skompresowaniu, dane powinny mieć dokładnie $\sigma * \Sigma$ bitów długości, nie licząc metadanych, które można zapisać w dwóch bajtach.

Tak zaimplementowany algorytm okazuje się być wielokrotnie szybszy[1] od tradycyjnej implementacji *Fixed-Length encoding* na CUDA, który jest równoważny tej kompresji z $\omega = 1$.

2.3.2 Kodowanie GFC

GFC to zaproponowana przez Burtscher et al. [23] modyfikacja algorytmu *pFPC* kompresji liczb zmiennoprzecinkowych o podwójnej precyzji, który jest równoległą wersją algorytmu *FPC* zaimplementowaną na procesory graficzne. Zamiast operacji XOR na prawdziwej i przewidzianej wartości kompresowanej liczby, metoda ta używa zwykłego odejmowania i dodatkowo zapamiętuje znak, a kompresuje wartość absolutną tej różnicy. Algorytm ten również stosuje wyrównanie, używając łączny dostęp do pamięci, za to w przeciwieństwie do algorytmu *AFL* nie wymaga aby ilość kompresowanych liczb przez pojedynczy wątek była podzielna przez 32. Może być to na przykład 15. Blok danych będzie miał ponownie 32 wartości, ponieważ każ-

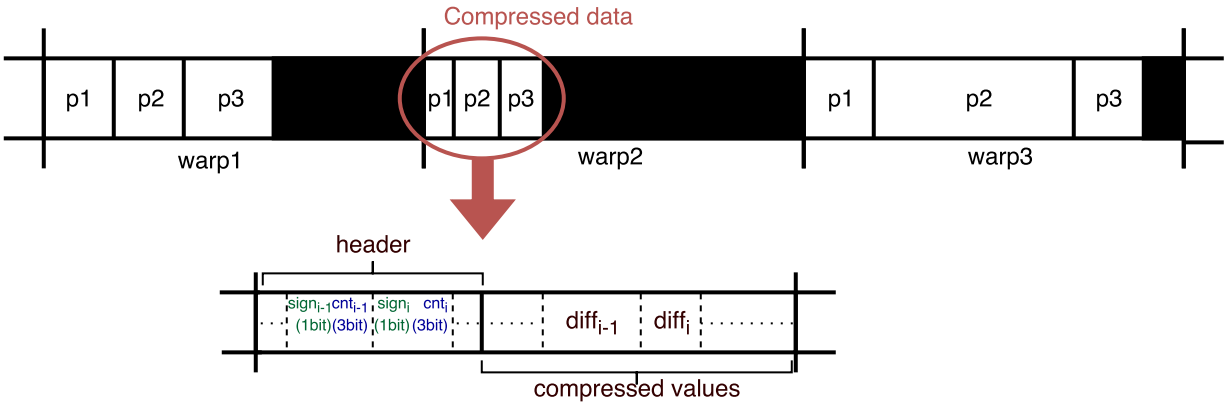
[1] patrz rozdział 5.7.1 str. 62

dy *warp* musi przetworzyć 32 wartości, aby uzyskać σ liczb wynikowych. W tym algorytmie liczba σ jest wyznaczana na bieżąco dla każdej liczby osobno i wyrażona jest w bajtach. Opiszę tutaj wersję algorytmu dla liczb o pojedynczej precyzji (32-bit) typu np. float. Zatem σ może przybierać wartości 1, 2, 3 lub 4 i można zapisać ją na 2 bajtach. Ilość bajtów na których zostanie zapisana liczba x może być przedstawiona w uproszczeniu jako: $\sigma(x) = log_2(x)/8 + 1$. Poniżej zamieszczam prosty pseudokod algorytmu wykonywanego przez każdy wątek w kernelu CUDA:

Algorithm 1: Pseudokod algorytmu kompresji GFC dla wątku t

```
/* WEJŚCIE: data - wektor do skompresowania                    */
/* WYJŚCIE: compr - skompr. dane, offsets - rozm. paczek       */
1  last = 0, i = 0;
   while i < λ do
2      diff = data[μ_i(t)] - last;
3      sign = bit znaku liczby diff;
4      diff = abs(diff);
5      minByte = σ(diff);
6      size = wykonaj sumę prefixową na minByte w warpie;
7      save(compr, diff, minByte) ;     // zapisz minByte bajtów liczby
       diff do tablicy wynikowej
8      saveMeta(compr, minByte, sign) ; // zapisz minByte oraz sign do
       tablicy wynikowej
9      off = off + size + 16;
10     beg = beg + 32;
11     last = data[beg − 1];
   if warp_idx == 31 then
12     offsets[warp] = off;
```

Poszczególne *warp-y* pracują oddzielnie, kompresując dane do osobnych paczek. Dane z całego *warp* pakowane są w jedno miejsce, począwszy od wyliczonego wcześniej przesunięcia. W każdej iteracji, *warp* pakuje 32 liczby tworząc z nich podpaczkę, poprzedzoną małym nagłówkiem zawierającym informacje o znakach i liczbie bitów na których jest zapisana każda liczba. Rozmiar tych informacji to 4 bity, więc dane w pesymistycznym wypadku mogą rozrosnąć się o $l_w/2$ bajtów. Podpaczki zapisywane są jedna za drugą. Ponieważ rozmiar skompresowanej paczki nie jest znany w momencie uruchomienia algorytmu, potrzebna jest osobna tablica wynikowa, w której przechowywane są wynikowe wielkości paczek. Ponadto trzeba zaalokować tablicę wynikową o wielkości $c_s = (l_W + 1) * \frac{9}{2}$ bajtów miejsca.



Rysunek 2.5: wynik działania algorytmu *GFC* - tablica wyjściowa

Pomimo podobnego podziału na bloki danych, powyższy rodzaj kompresji znacząco różni się od metody *AFL*. Ilość stworzonych paczek danych przez ten algorytm wynosi $p_n = w_g * B_n$. Paczki skompresowane przez poszczególne *warp-y* nie przylegają do siebie, co widać na rysunku 2.5, przez to minusem tej metody jest to, że po kompresji należy przekopiować wszystkie p_n paczek, o różnych rozmiarach do tablicy wynikowej. Kopiowanie to jest odpowiednio szybkie dla małej ilości paczek o większych rozmiarach. Wtedy zachodzi zależność, że kiedy p_n rośnie, wydajność kopiowania maleje, natomiast wydajność samego algorytmu wzrasta. Jeśli zmniejszymy p_n automatycznie musimy zwiększyć λ, bo B_n zależy odwrotnie proporcjonalnie od λ, a im większe λ, tym wolniejszy jest sam algorytm (pojedyncze wątki wykonu-

ją więcej pracy). Warto sprawdzić zatem, czy napisanie dedykowanego algorytmu kopiowania (jako kernel CUDA), może przyspieszyć takie kopiowania danych na GPU, względem tradycyjnych wywołań *cudaMemcpy* w pętli. Próby zrównoleglenia tych kopiowań na różnych *stream-ach* na mojej karcie graficznej (GeForce GT 640) skończyły się gorszą wydajnością niż seria kopiowań na *stream 0*.

Rozdział 3

Optymalizator kompresji

W tym rodziale opiszę algorytm nazwany roboczo *Online Stat Compression Planner*, który będzie uczył się budować jak najlepsze drzewo kompresji na podstawie napływających danych, jednocześnie je kompresując. Ponadto algorytm ten przeszukując przestrzeń ciekawych drzew kompresji i testując je na małym fragmencie danych, będzie miał szansę znaleźć optymalne drzewo już na początku swojego działania. Jednak w momencie zmiany charakterystyki danych, algorytm powinien zareagować mutując drzewo, zmieniając jego węzły, aby dostosować go do nowych danych. Na uwagę zasługuje fakt, że wszystkie dane na których operują optymalizator oraz drzewo, zawsze znajdują się na GPU, w celu minimalizacji liczby kopiowań przez szynę PCI-E. Dodatkowo w oparciu o *Shared Pointer* z biblioteki *Boost* zaimplementowano inteligentne wskaźniki na wektory danych w globalnej pamięci GPU. Takie wskaźniki przechowujące tablicę bajtów będę dalej nazywał SCBP[1].

Definicja 3.0.1 (Paczka danych)**.** Optymalizator kompresując dane $D = D_1, D_2, ..., D_N$ otrzymuje je w częściach D_i o zdefiniowanym maksymalnym rozmiarze, nazywanych paczkami.

[1]SCBP - inteligentny wskaźnik na tablicę bajtów w globalnej pamięci GPU (Shared Cuda Byte Pointer)

3.1 Kodowanie

Wszystkie algorytmy kompresji zostały zaimplementowane jako klasy dziedziczące po abstrakcyjnej klasie kodowania (*Encoding*), implementując metody kompresji i dekompresji dla wszystkich typów wspieranych przez system (optymalizator). Dla nas najważniejsze są dwie metody:

Encode – przyjmuje dane jako SCBP oraz typ danych[2] do kompresji. Metoda ta kompresuje dane i zwraca wektor wskaźników SCBP długości 2 lub 3, z których pierwszy jest zawsze metadanymi.

Decode – przyjmuje w zasadzie to co zwraca *Encode* oraz typ danych, a zwraca zdekodowane dane w postaci SCBP.

Obiekt każdego kodowania może być stworzony za pomocą fabryki, podając jego typ oraz typ danych jakie ma kompresować. Aby kodowania, a raczej ich wynik był możliwy do zserializowania i w następstwie pozwalał odtworzyć schemat użyty do kompresji, przed metadanymi wyniku dodawany jest header w postaci pokazanej na rysunku 3.1, nazwijmy go EH.

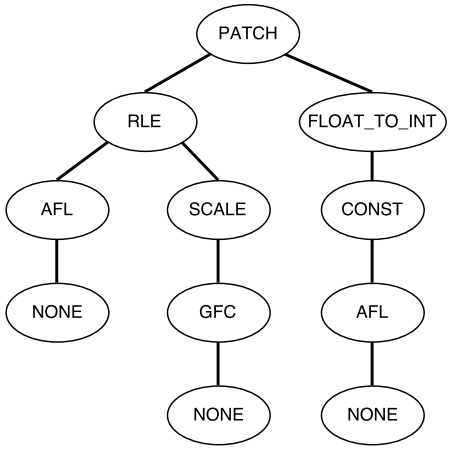
Rysunek 3.1: Nagłówek metadanych wyniku kodowania

Ponadto kodowanie umożliwia sprawdzenie jaki jest jego typ wynikowy oraz ile skompresowanych części zwraca, np. metody *PATCH* lub *DICT* zwracają po 2 wyniki.

3.2 Drzewo kompresji

Drzewo kompresji jest implementacją idei kompresji kaskadowej. Węzłowi drzewa odpowiada kompresja danego typu. Węzeł ma tyle dzieci ile wyników zwraca kodowanie które reprezentuje. Dodatkowy typ kodowania został dodany aby oznaczyć liście takiego drzewa. Przykładowe drzewo zostało przedstawione na rysunku 3.2.

[2] wartość słownikowa 4.1.3, patrz str. 44



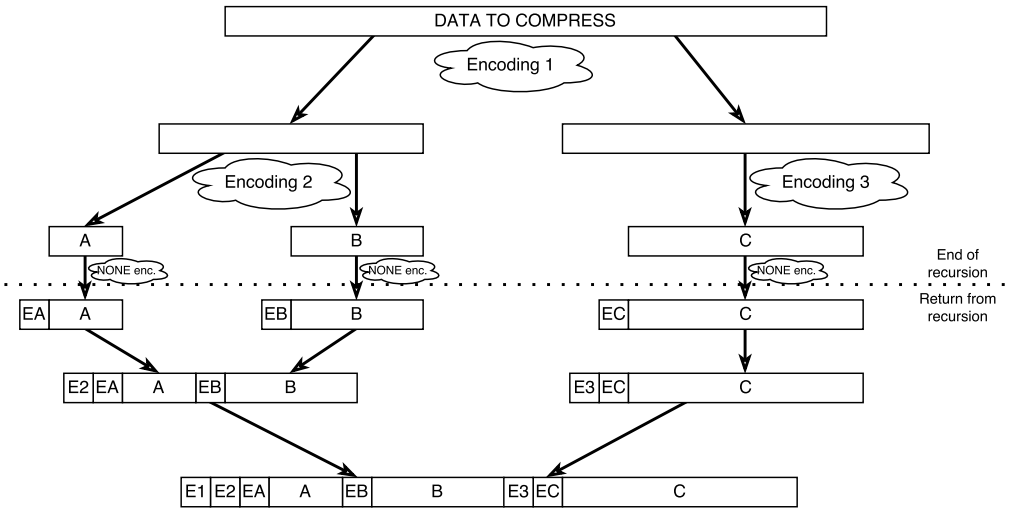
Rysunek 3.2: Schemat drzewa kompresji

Drzewo może być także zapisane jako ciąg typów kodowań (który dalej będę nazywał *TreePath*, na przykład pokazane drzewo można przedstawić w postaci pre-order jako:

PATCH, RLE, AFL, NONE, SCALE, GFC, NONE, FLOAT_TO_INT, CONST, AFL, NONE

Schemat ten jest o tyle ważny, że w ten sposób będą ułożone fragmenty skompresowanych danych przez algorytm kompresji drzewem.

3.2.1 Algorytm kompresji drzewem



Rysunek 3.3: Schemat przebiegu algorytmu kompresji drzewem

Algorytm kompresji drzewem jest rekurencyjny począwszy od korzenia drzewa i na wejście dostaje tylko dane do skompresowania w postaci SCBP. Ogólna idea polega na kompresowaniu danych odpowiednim koderem reprezentowanym przez wierzchołek drzewa i przekazywaniu wyników dzieciom tego węzła. Działanie kompresji opisuje algorytm 2.

Algorithm 2: Pseudokod algorytmu kompresji drzewem

Input: DATA – dane do skompresowania (GPU)
Output: RES – skompresowane dane (GPU), wynik w postaci wektora SCBP
Metoda $Node \mapsto Compress()$

```
1  X ← węzeł na którym wywołano metodę;
2  K ← pobierz koder typu reprezentowanego przez węzeł X z fabryki;
3  COMPR ← zakoduj DATA używając kodera K;
4  EH ← stwórz odpowiedni nagłówek kodowania;
5  dołącz EH na początku wektora COMPR;
   if X jest liściem then
6      dopisz COMPR do RES;
   else
       foreach D dziecka X do
7          CHILD_RES ← wywołaj metodę Compress na D;
8          dopisz CHILD_RES na końcu wektora RES;
9  uaktualnij compression ratio uzyskane przez aktualne poddrzewo;
10 return RES;
```

Metoda $Tree \mapsto Compress()$

```
1  ROOT ← pobierz korzeń drzewa;
2  VEC ← wykonaj metodę Compress(DATA) na ROOT;
   if ustawiony wskaźnik na statystyki then
3      zaktualizuj statystyki; ;          // opisane w 2 fazie optymalizatora
4  RES ← połącz wektor wyników VEC w jeden ciąg pamięci;
5  return RES;
```

Po skompresowaniu danych za pomocą korzenia drzewa uzyskujemy wektor kawałków skompresowanych danych oraz nagłówków. Ostatnią czynnością w algorytmie jest połączenie tych danych w jeden ciąg wynikowy (pojedyncza tablica zaalokowana na GPU). Na rysunku 3.3 widać prosty przebieg działania algorytmu. Jako *E#* oznaczono nagłówki doczepiane z różnych kodowań. Jak widać, nagłówki odzwierciedlają strukturę drzewa i są tożsame z *TreePath*, pomijając kawałki skompresowanych danych (*A*, *B*, *C*). Dzięki temu można odtworzyć drzewo do dekompresji.

3.2.2 Algorytm dekompresji drzewem

Algorytm dekompresji drzewem jest bardziej skomplikowany niż kompresja i składa się z dwóch faz: rekonstrukcji struktury drzewa i właściwej dekompresji drzewem. Ogólny schemat całości algorytmu pokazany jest na rysunku 3.4.

Rekonstrukcja drzewa

Kolejny algorytm rekurencyjny (patrz algorytm 3), który na wejściu dostaje SCBP ze skompresowanymi danymi oraz offset - referencja do parametru określającego położenie już odczytanego fragmentu danych, tj. od jakiego bajtu należy czytać dalsze dane wejściowe. Idea algorytmu polega na odczytywaniu kolejnych nagłówków *EH* i budowaniu na ich podstawie drzewa, odpowiadającego drzewu użytemu do kompresji.

Algorithm 3: Pseudokod algorytmu rekonstrukcji drzewa

Input: DATA – skompresowane dane (GPU), OFFSET – przesunięcie (REF)
Output: NODE – zrekonstruowany węzeł drzewa
Metoda $Tree \mapsto DecompressNodes()$

```
1  EH ← odczytaj nagłówek z wejścia;
2  OFFSET+ = SIZEOF(EH);
3  NODE ← stwórz węzeł typu wskazywanego przez EH;
4  MET_SIZE ← pobierz rozmiar metadanych z EH;
5  MET ← odczytaj metadane o rozmiarze MET_SIZE;
6  OFFSET+ = MET_SIZE;
7  ustaw metadane w NODE na MET;
   if NODE ma typ NONE, czyli jest liściem then
8      SIZE ← odczytaj z MET rozmiar skompresowanych danych;
9      DATA ← odczytaj z wejścia SIZE bajtów;
10     ustaw dane w węźle NODE na DATA;
11     OFFSET+ = SIZE;
   else
12     K ← pobierz koder o typie wskazywanym przez EH;
13     N ← pobierz ilość wyników zwracanych przez koder K;
       for i=0 to N do
14         CHILD_NODE ← DecompressNodes(DATA, OFFSET);
15         dodaj CHILD_NODE jako dziecko NODE;
16 return NODE;
```

Dekompresja

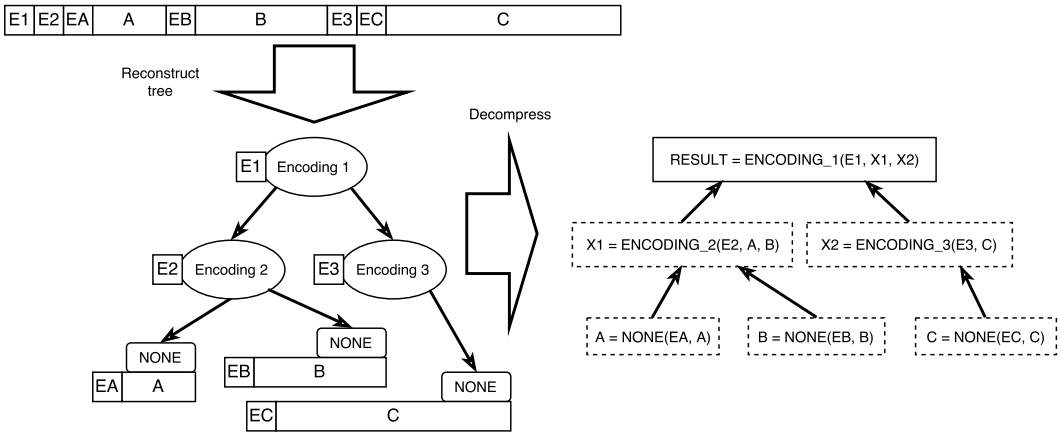
Po odtworzeniu drzewa na korzeniu wywoływana jest metoda *Decompress*, która nie przyjmuje żadnych parametrów, za to zwraca zdekodowane dane w postaci SCBP. Pokazuje to algorytm 4.

Algorithm 4: Pseudokod algorytmu dekompresji drzewem

Input:
Output: RES – zdekodowane dane (GPU)
Metoda $Node \mapsto Decompress()$

```
1  X ← węzeł na którym wywołano metodę;
2  K ← pobierz koder typu reprezentowanego przez węzeł X z fabryki;
3  MET ← pobierz metadane zapisane w węźle X;
4  W ← stwórz wektor wskaźników SCBP i dodaj do niego MET;
   if X jest liściem then
5      DATA ← pobierz dane zapisane w węźle X;
6      dodaj DATA na końcu wektora W;
   else
       foreach D dziecko X do
7          CHILD_RES ← wykonaj metodę Decompress() na X;
8          dodaj CHILD_RES na końcu wektora W;
9  RES ← użyj kodera K do zdekodowania wektora W;
10 return RES;
```

Podsumowując, jest to implementacja kaskadowej kompresji (rekurencyjnej z użyciem drzew jako reprezentacji). Dodatkowo drzewa te posiadają wskaźnik na statystyki drzewa i mogą je uaktualniać, co będzie szerzej opisane w części poświęconej 2 fazie algorytmu optymalizatora.



Rysunek 3.4: Schemat przebiegu algorytmu dekompresji drzewem

3.3 Statystyki

Warunkiem podejmowania dobrych decyzji przez optymalizator jest znajomość charakterystyki danych na których operuje. Główne zastosowanie statystyk w tym systemie, opiera się na przesiewaniu wszystkich możliwych drzew do tych najistotniejszych, które mogą uzyskać dobry współczynnik kompresji. Ponadto, niektóre statystyki wykorzystywane są przez same kodowania, aby odpowiednio działać. W tym systemie zaimplementowano statystyki takie jak:

1. *IsFloatingPoint* – czy dane mają typ zmiennoprzecinkowy czy całkowity
2. *Sorted* – czy dane są posortowane (obojętnie czy rosnąco czy malejąco)
3. *Min* i *Max* – wartość minimalna i maksymalna w zbiorze danych
4. *Precision* – maksymalna dokładność (precyzja) danych, czyli ilość miejsc po przecinku
5. *MinBitLength* – minimalna liczba bitów na których można zakodować każdą liczbę ze zbioru
6. *Size* – rozmiar danych
7. *RleMetric(N)* – statystyka dla kodowania RLE opisana dokładnie w następnej podsekcji
8. *Dictionary Counter* – histogram wszystkich unikalnych wartości ze zbioru
9. *Mean* – średnia arytmetyczna liczb w zbiorze

W systemie w bardzo łatwy sposób można implementować dodatkowe statystyki i udostępniać je do optymalizatora w celu implementacji dodatkowych reguł.

3.3.1 Metryka RLE

Definicja 3.3.1. Metryką RLE będziemy nazywać średnią z długości maksymalnych ciągów równych liczb, nie dłuższych niż N, w danych D i ozn. $\mathrm{RLE}_M(D, N)$, gdzie

$N \geq 2$. Oznaczmy jako l_i^N długość ciągu równych liczb, począwszy od indeksu i, nie dłuższego niż N, oraz długość danych D jako k. Wtedy

$$\text{RLE}_M(D, N) = \frac{\sum_{i=0}^{k} l_i^N}{k}$$

Dla przykładu, jeśli weźmiemy $N = 2$ oraz dane D: 1112234444555555, wtedy odczytane ciągi równych liczb nie dłuższe niż 2 wynoszą: 2212112221222221, z czego średnia jest równa $\frac{27}{16} \approx 1.7$. Natomiast podstawiając $N = 4$: 3212114321444321, średnia wyniesie $\frac{38}{16} \approx 2.4$. Daje to wyobrażenie o średniej długości ciągów równych liczb w danych, a ponadto jest bardzo wydajne obliczeniowo. Wyliczanie takiej statystyki jest trywialnie równoległe, ponieważ można w każdym wątku wyliczać ciąg począwszy od innego indeksu, niezależnie. Pseudokod wyliczania ciągu dla pojedynczego wątku CUDA pokazuje algorytm 5. W ostatnim kroku wystarczy obliczyć średnią, korzystając na przykład z gotowej funkcji z biblioteki Thrust. W tej pracy przyjmuję $N = 2$, ponieważ można udowodnić, że:

Lemat 3.3.1. Jeśli $\text{RLE}_M(D, 2) > 1.5$, to wykorzystanie kodowania RLE zmniejszy rozmiar danych D.

Dowód. Kodowanie ciągu równych liczb odbywa się za pomocą pary wartości, więc aby rozmiar danych się zmniejszył, ciągi średnio muszą mieć długość większą niż 2. Przyjmując $n_1, n_2, ..., n_m$ jako długości ciągów, oraz n_{sr} jako średnią długość tych ciągów, $\frac{\sum_{i=1}^{m} n_i}{m} = n_{sr} > 2$ implikuje współczynnik kompresji większy od 1.0.

$$\sum_{i=1}^{m} n_i = k \implies \frac{k}{m} > 2$$

Zatem warunkiem koniecznym i dostatecznym jest, aby $m < \frac{k}{2}$. Można zauważyć, że ciąg o długości n_i zostanie przez algorytm zapisany jako ciąg $n_i - 1$ dwójek, oraz jedynkę, zatem wzór na metrykę można zapisać w zależności od n_i, jako:

$$\text{RLE}_M(D, 2) = \frac{\sum_{i=1}^{m}(2 * n_i - 1)}{k} = \frac{2(n_1 + ... + n_m) - m}{k}$$

po przekształceniu:

$$\text{RLE}_M(D, 2) = 2 - \frac{m}{k} \implies (2 - \text{RLE}_M(D, 2)) \cdot k = m < \frac{k}{2}$$

Z tego już wprost wynika, że $\mathrm{RLE}_M(D, 2) > 1.5$

□

W implementacji algorytmu, aby uniknąć sprawdzania warunku końca tablicy, warto jest zrezygnować z obliczania $N - 1$ ostatnich wartości, które mają znikomy wpływ na wynik, szczególnie dla dużego rozmiaru danych.

Algorithm 5: Obliczanie metryki RLE

Input: DATA – wektor danych wejściowych;
N – argument metryki RLE (maks. długość sprawdzanego ciągu);
IDX – globalny indeks wątku (unikalny)
Output: LENGTHS – długości ciągów równych liczb
`/* Wykonaj równolegle dla każdego wątku CUDA */`
Funkcja `GetRunLenN()`

```
1  if IDX >= length(DATA) − N then
2      return
3  num ← 1, out ← 1, last ← 1;
4  for i = 1 to N do
5      out ← DATA[IDX] == DATA[IDX + i];
6      num+ = out & last;
7      last ← out;
8  LENGTHS[IDX] ← num;
```

3.3.2 Precyzja

Kolejną ważną statystyką dla liczb zmiennoprzecinkowych jest precyzja, która odnosi się do tego, czy warto zastosować zmianę wartości na typ całkowitoliczbowy, stosując przemnożenie[3] przez odpowiednią potęgę liczby 10. Analogicznie do metryki RLE, algorytm wyliczania precyzji jest trywialnie równoległy, a obliczanie pre-

[3] patrz kodowanie *FLOAT_TO_INT* str. 15

cyzji pokazuje algorytm 6.

Algorithm 6: Obliczanie precyzji

Input: VALUE – wartość wymierna; MAX_PREC – maksymalna precyzja

Output: PREC – precyzja

Funkcja `GetPrecision()`

1 $E \leftarrow 1$;

2 $MP \leftarrow 10^{MAX\_PREC}$;

3 **while** $(\frac{round(VALUE \cdot E)}{E} \neq VALUE)$ && $(E < MP)$ **do**

4 $E \leftarrow E \cdot 10$;

5 **return** $\frac{log_f(E)}{logf(10)}$;

Pomiary w rzeczywistości mają konkretną dokładność i często z góry wiadomo jaka jest precyzja poszczególnych kolumn szeregu czasowego. Te informacje, jeśli dostępne, mogą być przekazane jako parametr w nagłówku szeregu i ustawiane jako domyślne zamiast wyliczania jej na bieżąco za pomocą powyższego algorytmu.

3.4 Optymalizator

Algorytm optymalizatora kompresji maksymalizuje współczynnik kompresji, jednocześnie starając się wybrać jak najmniejsze i najniższe drzewo kompresji. Głównym pomysłem jest tworzenie statystyk krawędzi drzewa kompresji o kształcie pełnego drzewa binarnego (z tego względu kodowania mogą zwracać co najwyżej 2 wyniki). Nic nie stoi jednak na przeszkodzie, aby algorytm ten uogólnić na kodowania zwracające dowolnie dużą ilość wyników.

Definicja 3.4.1 (Optymalne drzewo)**.** Drzewo kompresji będziemy nazywać optymalnym, jeśli dla aktualnie przetwarzanych danych, wykazuje najwyższy współczynnik kompresji, wśród wszystkich znanych i zbadanych drzew. Drzewo wybrane jako optymalne to drzewo, które wierzymy, że jest optymalne wedle posiadanych informacji.

Algorytm składa się z 3 faz, a jego zarys wygląda następująco:

1. Dla małego fragmentu danych generujemy kandydujące drzewa kompresji (patrz def. 3.4.2), zapisując statystyki krawędzi w drzewie binarnym, po czym wybieramy drzewo o najlepszym współczynniku kompresji z niewielką poprawką od wysokości drzewa.

2. Kompresujemy odpowiednio dużą paczkę danych używając drzewa ustawionego jako optymalne i odpowiednio aktualizujemy statystyki krawędzi podczas kompresji.

3. Jeśli współczynnik kompresji dla drzewa ustawionego jako optymalne pogorszył się, zmieniamy drzewo wymieniając odpowiednie poddrzewo na lepsze, według statystyk krawędzi.

3.4.1 Faza 1 - generowanie

Ten fragment opisuje rekurencyjny algorytm generowania kandydujących drzew na postawie statystyk. Faza ta działa na małej części danych wejściowych rzędu 1% rozmiaru paczki, ponieważ może być relatywnie czasochłonna względem całości algorytmu.

Definicja 3.4.2 (Drzewo kandydujące)**.** Drzewo kompresji, którego każdy węzeł został wybrany jako kontynuacja węzła poprzedniego, wedle określonych reguł (heurystyki).

Reguły tworzone są na podstawie statystyk danych, które dany węzeł ma dostać do kompresji, typu poprzedzającej go kompresji oraz typu danych. Znany jest także aktualny poziom drzewa na którym ma się znaleźć nowy węzeł. Metoda zwracająca kontynuacje została nazwana roboczo *GetContinuations* i zwraca listę typów kodowań, wybranych przez reguły[4]. Kilka przykładów możliwych reguł:

- Jeśli poprzednikiem kodowania nie było *FLOAT_TO_INT*, dane mają niską precyzję i niewielką wartość maksymalną, a ponadto typ danych to *double*, to

[4]w ogólności, jest to heurystyka określająca zbiór drzew kandydujących dla konkretnych danych

dodaj do listy kontynuacji typ kodowania *FLOAT_TO_INT*, ponieważ jest zasadne.

- Jeśli poprzednim kodowaniem było *GFC* albo *AFL*, to nie kompresuj już więcej i jako jedyną możliwą kontynuację zwróć NONE. Jeśli nie, a typ danych to *float* lub *double* to dodaj *GFC* do listy możliwych kontynuacji, w p.p. dodaj *AFL*.

Algorytm kompresuje dane w trakcie generowania drzew oraz jednocześnie oblicza i uaktualnia statystyki krawędzi w drzewie binarnym (o tym w fazie 2). Tę fazę optymalizatora implementuje metoda *FullStatisticsUpdate*, której pseudokod pokazany jest jako algorytm 7.

Algorithm 7: Faza 1 algorytmu optymalizatora

Input: DATA – mała paczka danych;
ET – typ kompresji, DT – typ danych;
LEVEL – aktualny poziom drzewa
Output: RES – lista interesujących drzew z ich wynikami
Funkcja `FullStatisticsUpdate()`

1 $STAT \leftarrow$ policz statystyki dla $DATA$;
2 $RES \leftarrow$ pusty wektor drzew;
3 $CONT \leftarrow$ wywołaj $GetContinuations(ET, DT, STAT, LEVEL)$ `// pobierz możliwe kontynuacje`
foreach *C from* $CONT$ **do**
4 $T \leftarrow$ stwórz drzewo o jednym węźle reprezentującym kodowanie C z typem danych DT;
5 $COMPR \leftarrow$ zakoduj dane $DATA$ używając T;
6 $DT \leftarrow$ typ danych zwracany przez kodowanie C;
7 zaktualizuj w T uzyskany współczynnik kompresji;
if *liczba wyników zwracana przez* C *jest równa 1* **then**
8 $PART1 \leftarrow$ wywołaj $FullStatisticsUpdate(COMPR[1], C, DT, LEVEL + 1)$;
9 $PART1 \leftarrow$ wywołaj $CrossTrees(T, PART1, length(DATA), length(COMPR[0])$;
else `// jest równa 2`
10 $PART1 \leftarrow$ wywołaj $FullStatisticsUpdate(COMPR[1], C, DT, LEVEL + 1)$;
11 $PART2 \leftarrow$ wywołaj $FullStatisticsUpdate(COMPR[2], C, DT, LEVEL + 1)$;
12 $PART1 \leftarrow$ wywołaj $CrossTrees(T, PART1, PART2, length(DATA), length(COMPR[0])$;
if $PART1$ *jest pusty* **then**
13 dodaj T do $PART1$;
else
14 dopisz $PART1$ na końcu wektora RES;
15 **return** RES;

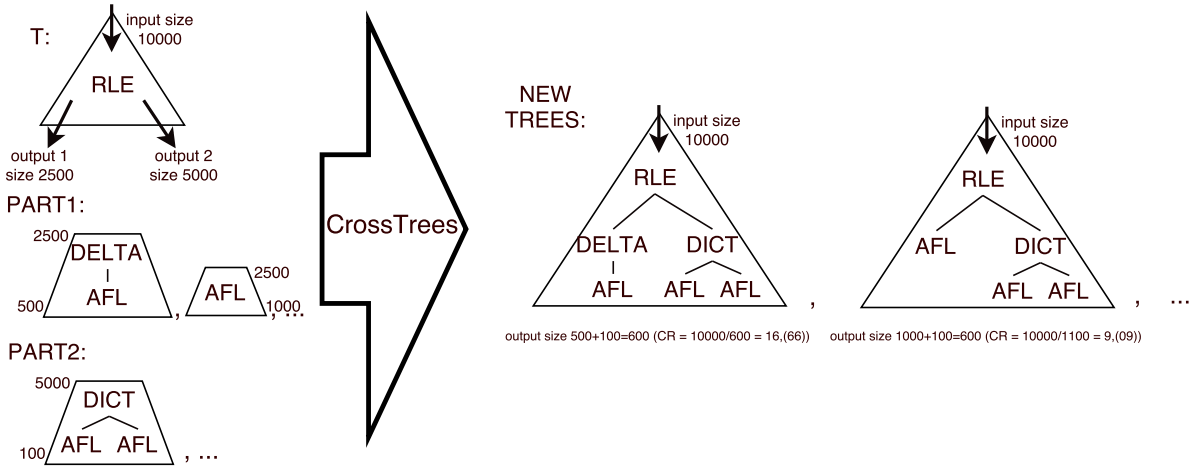
Metody *CrossTrees* tworzą wszystkie możliwe kombinacje, w których T jest wierzchołkiem, a drzewa z kolejnych dwóch argumentów (wektory drzew), jego poddrze-

wami. Dodatkowo uaktualniany jest współczynnik kompresji połączonych drzew. Przykładem działania tej metody jest rysunek 3.5 oraz poniższy schemat:

$$PART1 = \{T_1^A, T_2^A, ..., T_N^A\}, \quad PART2 = \{T_1^B, T_2^B, ..., T_M^B\}$$

$$CrossTrees(T, PART1, PART2) = \{T|T_1^A|T_1^B, T|T_1^A|T_2^B, ..., T|T_2^A|T_1^B, ..., T|T_N^A|T_M^B\}$$

gdzie napis $A|B|C$ oznacza drzewo o wierzchołku A oraz poddrzewach B i C, będących jego dziećmi.

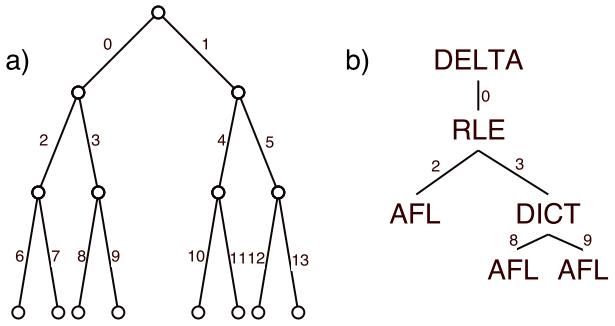


Rysunek 3.5: Przykład działania metody CrossTrees

W skrócie, wyliczamy statystyki danych i na ich podstawie przewidujemy możliwe węzły potomne, konstruując drzewo kandydujące, a następnie kompresujemy dane kodowaniem węzła i rekurencyjnie wyznaczamy kolejne drzewa kandydujące. Następnie łączymy wyniki we wszystkie możliwe kombinacje, ale zachowując hierarchię i dołączamy do możliwych rozwiązań. Dodatkowo, wiemy jak dobrze każde skonstruowane drzewo zachowuje się dla danych wejściowych. Finalnie drzewo o najlepszym wyniku (współczynnik kompresji oraz poprawka za wysokość drzewa) jest optymalne i automatycznie jest wybierane jako drzewo optymalne dla późniejszych faz algorytmu.

3.4.2 Faza 2 - kompresja

W tej fazie pełna paczka danych jest kompresowana przez drzewo wybrane jako optymalne, ale co ważniejsze uaktualniane są statystyki krawędzi drzewa binarnego. Opiszę jak wyglądają te statystyki i jak są uaktualniane. Ważne jest również, że drzewo wybrane jako optymalne pamięta ostatnią kopię statystyk, która jest tworzona w momencie ustawienia nowego drzewa jako optymalne. Krawędzie w drzewie

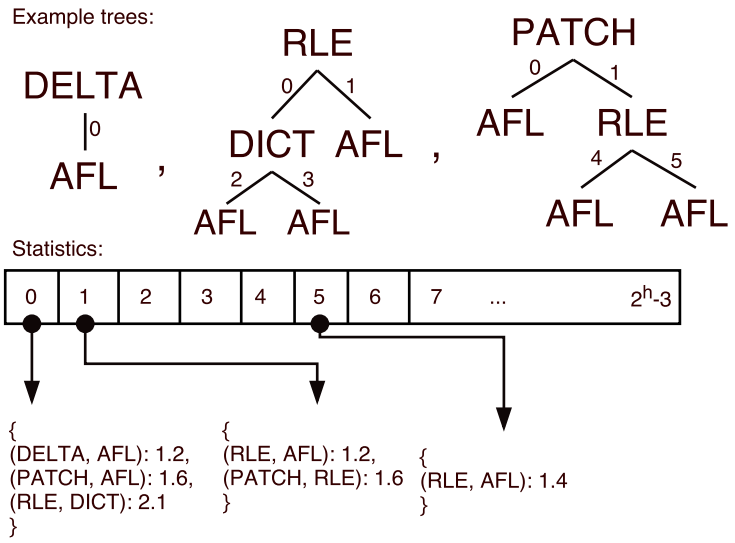


Rysunek 3.6: Przykład numerowania krawędzi w a) drzewie binarnym b) drzewie kompresji

kompresji odpowiadają parom kodowań na pewnym miejscu w odpowiadającym mu pełnym drzewie binarnym, co pokazuje rysunek 3.6. Z założenia, chcemy analizować pary kodowań, ponieważ każdy wcześniejszy etap kompresji wpływa na następny i niejako przygotowuje dla niego dane. Często okazuje się, że *AFL* działa dla jakiś danych bardzo słabo, ale poprzedzone kodowaniem *DELTA* lub *PATCH* osiąga bardzo dobre wyniki, zaś poprzedzone kodowaniem *DICT* ma wynik jeszcze słabszy. W takiej sytuacji oczekuje się, że dwie wcześniejsze konfiguracje będą miały lepszą, większą co do wartości statystykę, na danym miejscu w drzewie binarnym, niż opcja z kodowaniem *DICT*.

Każdej parze następujących po sobie kodowań da się przypisać krawędź w pełnym drzewie binarnym o tej samej, lub większej wysokości. Wynika to z tego, że każde kodowanie może mieć co najwyżej dwójkę następców (dzieci). Ponadto jeśli krawędź ma indeks i, to łatwo policzyć indeksy krawędzi odchodzących od niej jako $2 \cdot (i + 1)$ oraz $2 \cdot (i + 1) + 1$. Program konstruuje tablicę statystyk (patrz rys. 3.7),

w której pod indeksem λ będą przechowywane statystyki par kompresji na miejscu krawędzi λ w drzewie binarnym.



Rysunek 3.7: Tablica statystyk krawędzi

Definicja 3.4.3 (Współczynnik krawędzi drzewa kompresji)**.** Współczynnikiem krawędzi e stworzonej z pary kodowań (A, B) w drzewie kompresji ozn. $C_e = \frac{C(A)+C(B)}{2}$, nazywamy średnią arytmetyczną współczynników kompresji osiągniętych przez poddrzewo definiowane przez węzeł A i poddrzewo definiowane przez B.

Dla danej pary kodowań na wybranym miejscu w drzewie, statystyka jest liczona jako średnia arytmetyczna, ze starej wartości statystyki (inicjowana wartością 1.0) oraz uzyskanego współczynnika krawędzi drzewa kompresji:

$$\alpha' = \frac{\alpha + C_e}{2}$$

Dla przykładu, jeśli krawędź 1 z parą kompresji (A, B) uzyskała $C_e = 2$ a potem 3, to statystyka dla tej pary na tej krawędzi będzie wynosiła $\alpha' = \frac{\frac{1+2}{2}+3}{2} = 2,25$. Taka statystyka sprawia, że nigdy nie będzie większa niż najlepszy współczynnik kompresji uzyskany przez tą parę oraz, że statystyka ta dąży do tego współczynnika (w granicy), jeśli jest stały.

Dowód. Załóżmy, że $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ są kolejnymi współczynnikami krawędzi drzewa, wtedy statystyka α' w granicy będzie wynosiła:

$$\lim_{n\to\infty} \frac{\frac{\frac{\frac{1+\alpha_1}{2}+\alpha_2}{2}+...}{...}+\alpha_n}{2} = \frac{1}{2^n} + \frac{\alpha_1}{2^n} + \frac{\alpha_2}{2^{n-1}} + ... + \frac{\alpha_{n-1}}{2^2} + \frac{\alpha_n}{2^1} = *$$

Biorąc $\alpha = const$, z granicy sumy ciągu geometrycznego wynika, że suma ta zbiega do α^-.

$$* = \lim_{n\to\infty} \frac{1}{2^n} + \alpha \cdot \sum_{i=1}^{n} \frac{1}{2^i} = \alpha^-$$

□

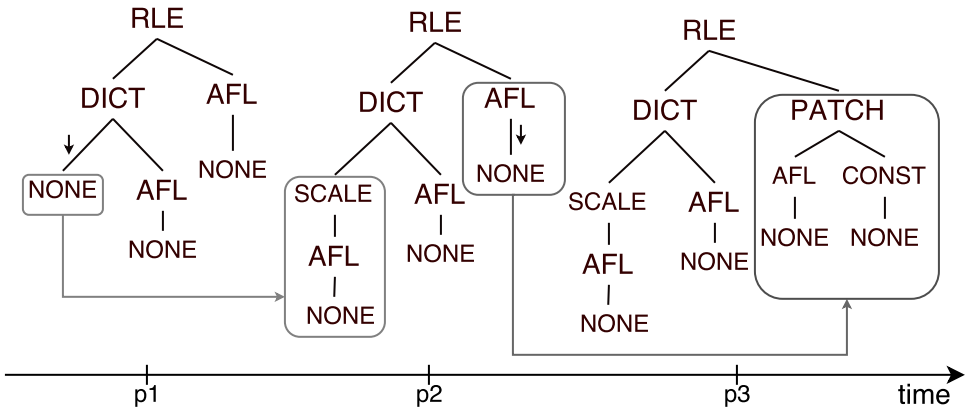
Statystyki krawędzi uaktualniane są w ten sposób podczas trwania każdej kompresji i w momencie jej ukończenia powinny być aktualne.

3.4.3 Faza 3 - poprawa

Po dokonaniu kompresji na całej paczce danych (> 1M elementów), jesteśmy w stanie stwierdzić, jak dobrze dane drzewo (wybrane jako optymalne) się sprawuje. Jeśli jakość kompresji wzrosła lub pozostała bez zmian, algorytm pozostawia stare drzewo pomimo, że może już nie być optymalne[5]. W przeciwnym przypadku należy drzewo poprawić, stosując nową wiedzę, uzyskaną w formie statystyk krawędzi z nowych danych. Aby to osiągnąć algorytm sprawdza statystyki krawędzi i wybiera krawędź o najmniejszym indeksie dla której statystyka się pogorszyła (drzewo wybrane jako optymalne trzyma kopię statystyk oraz współczynnika kompresji z momentu wybrania go na optymalne). Można też w tym momencie wybierać krawędź dla której statystyka pogorszyła się najbardziej np. procentowo. Wybraną krawędź kasujemy (usuwamy ją z drzewa razem z poddrzewem). Na miejscu usuniętej krawędzi, zachłannie, wstawiamy parę kompresji o największym współczynniku α dla

[5]podyktowane jest to faktem, że gdybyśmy chcieli w takim wypadku poszukiwać drzewa optymalnego, musiałaby być uruchomiona faza 1, która jest bardzo kosztowna obliczeniowo

tej krawędzi. Podobnie wstawiamy jej dzieci, aż do limitu wysokości drzewa. Proces ten dokładniej opisują algorytmy 9 i 8, z prostym przykładem w formie rysunku 3.8.



Rysunek 3.8: Przykład mutacji drzewa dla kompresji 3 kolejnych paczek danych

Wymieniając krawędź, algorytm w zasadzie wymienia poddrzewo w grafie, zaczynając od początku lub końca krawędzi (oszczędzając początek). Na rysunku 3.8 widzimy oba przypadki. Na przykład, w pierwszej *mutacji* tylko jedna odnoga *DICT* została zmieniona, podczas gdy w przejściu drugim wymianie podlega cała krawędź *AFL-NONE*.

Algorithm 8: Faza 3 - poprawianie drzewa (Replace)

Input: NODE – wymieniany węzeł drzewa, STATS – statystyki krawędzi;
$EDGE_{NO}$ – numer wymienionej krawędzi, MAX_H – maksymalna wysokość drzewa;
Funkcja `Replace()`

```
1  if EDGE_NO > 2^MAX_H − 3 then return;
2  K ← kodowanie węzła NODE RES_CNT ← pobierz ilość wyników zwracanych przez K;
   for i = 0 to RES_CNT do
4      BEST, BEST_VAL ← pobierz ze STATS parę zaczynającą się na K o największej statystyce na pozycji
       EDGE_NO + i w drzewie binarnym oraz jej wartość;
       if BEST_VAL > 1.0 then
5          CHLD ← węzeł reprezentujący kodowanie BEST.TO;
6          dodaj CHLD jako dziecko NODE;
7          wywołaj Replace(CHLD, STATS, 2 ∗ (EDGE_NO + i) + 2;
       else
8          dodaj do NODE dziecko z kodowaniem NONE;
   return
```

Algorithm 9: Faza 3 - poprawianie drzewa (TryCorrectTree)

Input: OPT – drzewo wybrane jako optymalne, $STAT_{new}$ – aktualne statystyki;

`/* X.TO i X.FROM, X.NO to węzeł końcowy, początkowy oraz numer krawędzi X */`

Funkcja `TryCorrectTree()`

1 $CR_{new} \leftarrow$ pobierz aktualny współczynnik kompresji drzewa OPT;
2 $CR_{old} \leftarrow$ pobierz historyczny współczynnik kompresji OPT;
if $CR_{new} \geq CR_{old}$ **then**
3 **return** *false*;
4 $STAT_{OLD} \leftarrow$ pobierz historyczne statystyki z OPT;
5 $EDGES \leftarrow$ pobierz krawędzie drzewa OPT;
foreach E *from* $EDGES$ **do**
6 $VAL_{OLD} \leftarrow$ wartość statystyki krawędzi E z $STAT_{OLD}$;
7 $BEST \leftarrow$ pobierz wartość najlepszej statystyki na miejscu E z $STAT_{NEW}$;
if $BEST > VAL_{OLD}$ **then**
8 $NEW, VAL_{NEW} \leftarrow$ pobierz ze statystyk $STAT_{NEW}$ parę na miejscu E zaczynającą się tą samą kompresją, o największej statystyce oraz jej wartość;
if $VAL_{NEW} > VAL_{OLD}$ **then**
9 wymień E.TO na NEW.TO, kasując całe poddrzewo od E.TO;
10 wykonaj $Replace(NEW.TO, STAT_{NEW}, 2 * (E.NO + 1))$;
else
11 $NEW \leftarrow$ pobierz ze statystyk $STAT_{NEW}$ parę na miejscu E o największej statystyce;
12 wymień E.FROM na NEW.FROM, kasując całe poddrzewo od E.FROM;
if *E.NO jest parzyste* **then**
13 $NO \leftarrow E.NO - 1$;
else
14 $NO \leftarrow E.NO$;
15 wykonaj $Replace(NEW.FROM, STAT_{NEW}, NO)$;
16 wyjdź z pętli;
17 **return** *true*;

Podsumowując, program mutuje drzewo kompresji dopóki współczynnik kompresji nie zacznie się poprawiać. Pojedyncza mutacja może zmienić jeden węzeł, ale może także wymienić całe drzewo, co dzieje się bardzo często. Wymiany są dokonywane tylko wtedy, gdy według statystyk nastąpi poprawa kompresji. W ten sposób nawet jeśli charakterystyka danych się zmieni, algorytm powinien dostosować drzewo do nowych warunków.

3.4.4 Algorytm optymalizatora kompresji

Algorithm 10: Optymalizacja kompresji

Input: DATA – dane do skompresowania, DT – typ danych;
Output: RES – skompresowane dane
Funkcja `CompressData()`

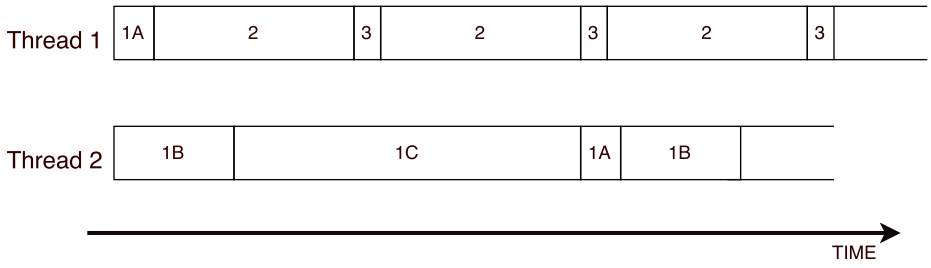
```
    if warto przejrzeć ponownie interesujące drzewa then                    /* heurystyka */
1       SAMPLE ← pobierz kawałek danych z DATA;
2       TREES ← wywołaj FullStatisticsUpdate(SAMPLE, NONE, DT, 0);
3       BEST ← pobierz drzewo z najlepszym wynikiem;
4       ustaw BEST jako optymalne drzewo
5   RES ← wykonaj kompresję optymalnym drzewem;
6   UPDATE ← wywołaj TryCorrectTree na optymalnym drzewie;
    if UPDATE is true then
7       ustaw ponownie optymalne drzewo;
8   return RES
```

Przebieg działania kompresji z użyciem optymalizatora pokazuje algorytm 8. Co pewien czas próbujemy przejrzeć wszystkie interesujące drzewa uaktualniając statystyki i wybierając inne optymalne drzewo. Staramy się je poprawiać i jeśli się udało, to ponownie zapisujemy je jako optymalne, tworząc nową kopię statystyk (historycznych).

3.4.5 Usprawnienie

Ilość drzew do sprawdzenia w fazie 1 może lawinowo rosnąc względem ilości różnych kompresji zaimplementowanych w systemie, w zależności od zastosowanych reguł. Warto zaimplementować zbiory reguł produkujących mniej lub więcej takich drzew, aby lepiej przeszukiwać przestrzeń rozwiązań[6]. Jednak przeszukiwanie dużej ilości drzew jest, pomimo zastosowania GPU, zbyt czasochłonne jak na planowane zastosowania. Możliwym rozwiązaniem (jeszcze nie zaimplementowanym ale planowanym), jest niezależne i równoległe wykonywanie fazy 1 dla różnych zbiorów reguł, równolegle do faz 2 i 3, w sposób pokazany na rysunku 3.9. Na tym rysunku numery 1,2,3 stanowią numer fazy, a A,B,C zbiory reguł w których A są najbardziej restrykcyjne i produkują mało drzew, zaś C bardzo dużo.

[6] patrz tabela 5.4



Rysunek 3.9: Zrównoleglenie fazy 1 w optymalizatorze

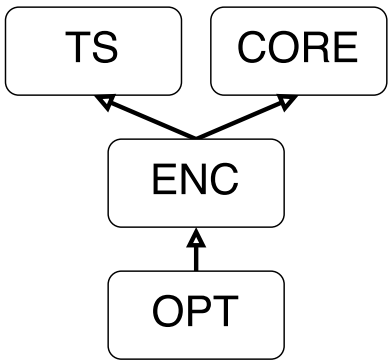
W ten sposób oba wątki mogą pracować nawet na 2 oddzielnych urządzeniach, wielokrotnie zwiększając nie tylko wydajność ale i współczynnik kompresji z racji przeszukania większej ilości opcji. Algorytm powinien także lepiej dostosowywać się do zmian danych. Taka równoległość jest możliwa, ponieważ tablica statystyk krawędzi została zaimplementowana w taki sposób, aby umożliwić równoległy dostęp bez obaw o niespójność danych.

Rozdział 4

System kompresji

Projekt składa się z 4 bibliotek, których zależności wewnętrzne pokazane są na rysunku 4.1. Dwie z nich, *CORE* oraz *ENC*, wykorzystują technologię CUDA i muszą być kompilowane za pomocą *nvcc*. Wszystkie korzystają z biblioteki *Boost* oraz bibliotek do testowania i benchmarków (*GTEST*, *Google Benchmark*). Poniżej znajduje się także opis poszczególnych części projektu.

4.1 Biblioteki



Rysunek 4.1: Biblioteki

4.1.1 TS - TIME SERIES

Biblioteka definiująca strukturę szeregu czasowego, udostępniająca metody do odczytu i zapisu szeregów z plików binarnych oraz tekstowych o kolumnach rozdzielonych separatorem, na przykład *CSV*. Ponadto umożliwia definicję szeregu czasowego poprzez plik nagłówkowy o strukturze wierszy: ([nazwa kolumny],[typ kolumny],[precyzja]), w której każdy wiersz definiuje osobną kolumnę. Przykładowa treść pliku nagłówkowego została umieszczona na listingu 4.1.

Listing 4.1: Przykładowy plik opisu szeregu czasowego

```
timestamp,time,0
CORE_VOLTAGE,float,6
CPU_TEMP,float,6
GPU_TEMP,float,6
```

4.1.2 CORE

Tutaj zaimplementowane są wszystkie podstawowe elementy takie jak logowanie, konfiguracja, inteligentny wskaźnik na pamięć CUDA (zaimplementowany w oparciu o *Shared Pointer* z biblioteki *Boost*), operacje na wektorach danych po stronie GPU takie jak histogramy, wyliczanie statystyk itp., a także bazowe klasy testów i benchmarków wraz z generatorem danych. Zawiera także „scheduler" równoległych zadań kompresji.

4.1.3 ENC - ENCODINGS

Encodings to biblioteka mieszczące wszystkie zaimplementowane w ramach tego projektu algorytmy kodowania i transformacji zaimplementowane na CUDA, potrafiące kodować i dekodować dane wszystkich typów obsługiwanych przez ten system (*char, short, int, unsigned int, long, float, double*).

4.1.4 OPT - OPTIMIZER

Właściwa biblioteka dla tego projektu mieszcząca optymalizator kompresji, a także definicję i implementację drzewa kompresji, jak również drzewa optymalnego (drzewa aktualnie używanego przez kompresor, które umożliwia pewne mutowanie tego drzewa, co zostanie opisane w następnym rozdziale).

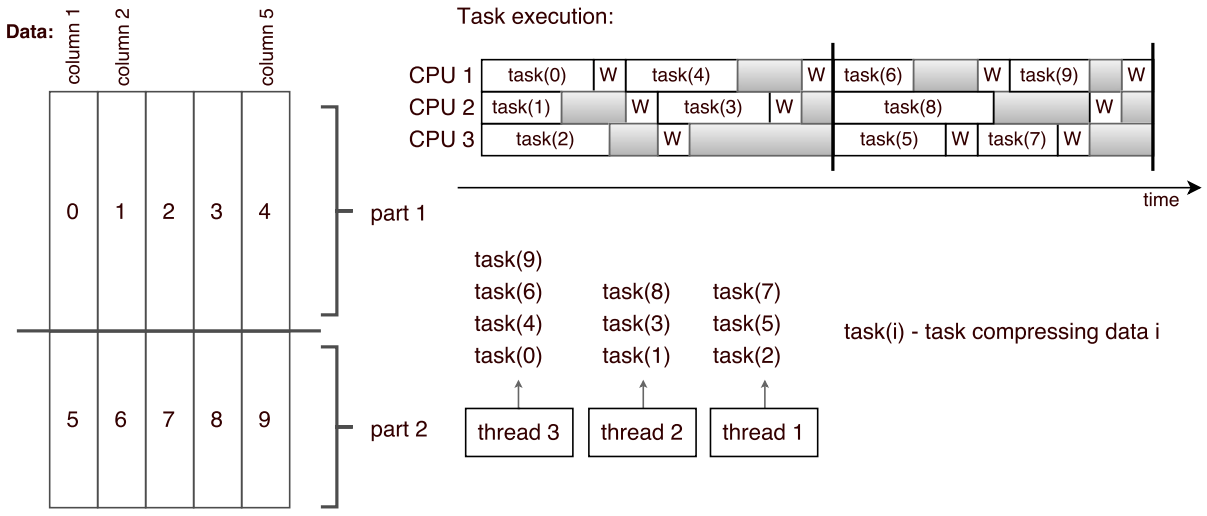
4.2 Program

Przykładowy program wynikowy, który używając optymalizatora kompresji, wielowątkowo i równolegle kompresuje wiele kolumn szeregu czasowego, podanego jako plik wejściowy. Plik wejściowy jest zaczytywany paczkami i w tym samym czasie części już skompresowane mogą być zapisywane do podanego pliku wyjściowego. Dokładny opis tego algorytmu znajduje się poniżej (Patrz *Równoległy kompresor*). Argumenty programu:

- *–compress* lub *-c*: opcja kompresji (nastąpi kompresja pliku wejściowego)
- *–decompress* lub *-d*: opcja dekompresji (nastąpi dekompresja pliku wejściowego)
- *–header* lub *-h [<ścieżka>]*: podanie ścieżki do pliku zawierającego opis szeregu jak wyżej 4.1.
- *–input* lub *-i [<ścieżka>]*: podanie ścieżki do pliku wejściowego
- *–output* lub *-o [<ścieżka>]*: podanie ścieżki do pliku wyjściowego
- *–generate* lub *-g*: – wygeneruj przykładowe pliki danych
- *–padding* lub *-p [<różnica>]*: w przypadku gdy wiersze szeregu w pliku binarnym są wyrównane do jakieś wielkości, podajemy w ten sposób różnicę względem ich rzeczywistej wielkości, np. jeśli dane zajmują 12 bajtów, a są wyrównane do 16, powinniśmy uruchomić program z opcją *-p 4*.

4.3 Równoległa kompresja kolumn danych

Szeregi czasowe zwykle składają się z wielu kolumn, które mają różne charakterystyki, więc mogą być kompresowane niezależnie i równolegle. Oprócz możliwości wykorzystania do tego celu wielu urządzeń GPU jednocześnie, można wykorzystać *CUDA streams* aby wykonywać jednocześnie wiele *kerneli* oraz kopiowań danych na tej samej karcie. W nowych wersjach CUDA (> 7.0) każdy wątek CPU może otrzymać osobny i niezależny *stream* „domyślny" 0, nie powodujący niejawnej synchronizacji. Aby tak się stało, należy skompilować program z flagą –default-stream per-thread lub zdefiniować CUDA_API_PER_THREAD_DEFAULT_STREAM na początku programu. Dzięki tej opcji, wykonanie wielu *kerneli* oraz kopiowań pamięci będzie mogło być zrównoleglone, pomimo wywołania na domyślnym *stream* 0. Zdecydowanie upraszcza to implementację rozwiązania od strony zarządzania wykonaniem funkcji na *stream'ach CUDA*.



Rysunek 4.2: Wykonanie zadań kompresji przez wątki

4.3.1 Opis działania algorytmu

W programie równoległa kompresja opiera się na puli wątków zaimplementowanej jako *Task Scheduler* w bibliotece *CORE*, w oparciu o bibliotekę *BOOST*. Sama kom-

presja i dekompresja polega na uruchamianiu zadań kompresji lub dekompresji kawałków danych z użyciem instancji optymalizatora, unikalnego dla danej kolumny. W zależności od konfiguracji zadania mogą wykorzystywać jeden lub wiele dostępnych GPU.

Kompresja

Równoległa kompresja kolumn jest opisana przez pseudokod w algorytmie 7.

Algorithm 11: Całość kompresji optymalizatorem

Input: $FILE_{IN}$ – plik wejściowy, $FILE_{OUT}$ – plik wyjściowy;
Metoda $ParallelCompressor \rightarrow Compress()$

```
1  ID ← 1;
   repeat
2      TS ← odczytaj paczkę danych z pliku FILE_IN;
       if niezainizjalizowany then
           /* inicjalizuje kompresor tworzy optymalizator dla każdej kolumny
              wejściowych danych oraz tworzy pulę wątków                    */
3          wywołaj init()
4      COL_N ← pobierz ilość kolumn z TS;
       for i = 0 to N do
5          TASK ← stwórz zadanie kompresji z optymalizatorem dla kolumny i oraz danymi TS[i] (i-ta
           kolumna szeregu TS) o numerze ID;
6          dodaj TASK do schedulera oraz powiększ ID o 1;
7      zaczekaj na wykonanie wszystkich zadań;
   until koniec pliku FILE_IN;
```

W tym przypadku ważne jest jak działa samo zadanie kompresji, przy czym kluczowym elementem jest synchronizacja zapisu do pliku wynikowego, który to przedstawia rysunek 4.2. Zadanie to przebiega następująco:

1. Otrzymane dane kopiowane są do pamięci globalnej GPU
2. Dane są kompresowane przy użyciu podanego optymalizatora
3. Dane są zapisywane z powrotem do pamięci RAM
4. Następnie przy użyciu metod synchronizacji wątki zapisują dane do pliku wyjściowego.
    - wątek czeka aż wszystkie wątki o mniejszych numerach zakończą pracę

- wątek zapisuje dane poprzedzając je ich rozmiarem

5. Oznacz zadanie jako ukończone
6. Jeśli jakikolwiek z powyższych pkt. się nie powiedzie oznacz zadanie jako zakończone porażką

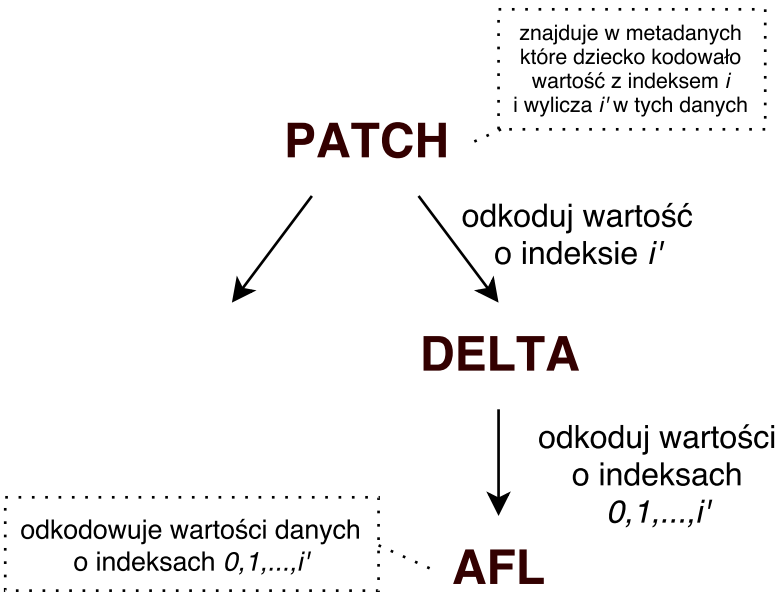
Dekompresja

Dekompresja przebiega na tyle prosto, że ograniczę się do krótkiego opisu bez pseudokodu i obrazków.

1. Kawałek po kawałku, zgodnie z zapisanymi rozmiarami czytamy skompresowane dane kolumn.
2. Dla każdego kawałka tworzone jest zadanie dekompresujące (posiadające wskaźnik na wspólną instancję szeregu czasowego), które:
   (a) Kopiuje dane do pamięci globalnej GPU
   (b) Dekompresuje je
   (c) Dopisuje na końcu kolumny, którą obsługuje w szeregu czasowym
3. Po zdekodowaniu liczby kawałków równej liczbie kolumn czekamy aż wątki zakończą pracę i zapisujemy szereg na wyjściem.
4. Czyścimy szereg czasowy aby nie zapisywać kolejny raz tych samych danych
5. Jeśli to nie koniec danych do dekompresji, wracamy do punktu pierwszego

4.3.2 Random Access

Kompresja danych za pomocą zaimplementowanych kodowań, z użyciem drzew i równoległej kompresji kolumn, umożliwia dostęp do poszczególnych rekordów skompresowanych danych, bez rozpakowywania całości. Może się to jednak wiązać z rozpakowaniem części lub całej paczki danych, zatem umożliwia losowy dostęp do danych w ograniczonym zakresie.



Rysunek 4.3: Schemat dostępu do wartości o indeksie i w paczce

Ponieważ każda paczka danych może być dekompresowana oddzielnie, znajomość przedziałów indeksów danych występujących w każdej paczce umożliwiłaby losowy dostęp do danych, dekompresując pojedynczą paczkę. Takie indeksy wraz z adresami paczek mogłyby być przechowywane na przykład w postaci B+drzew. Aby nie dekompresować całej paczki, niezbędna byłaby, implementacja alternatywnych algorytmów dekodowania, które czytałyby jedną lub wiele wartości skompresowanych danych, w celu odtworzenia wartości spod indeksu i w paczce, np. kodowanie *DELTA* musiałoby czytać wartości z indeksami $0, 1, ..., i$, zaś *AFL* tylko i. Z drugiej strony, drzewiasta struktura kodowań, wymusza konieczność wyliczania indeksu danej wartości na każdym etapie dekompresji w drzewie, co mogłoby spowodować duży spadek wydajności. Jest to jednak temat na kolejną pracę naukową i nie jest częścią powyższej pracy.

Rozdział 5

Wyniki

W tym rozdziale opiszę uzyskane wyniki pod względem współczynników kompresji dla różnych danych, jak również wygenerowane schematy kompresji. Później porównam wydajność całości jak i poszczególnych elementów systemu, z istniejącymi rozwiązaniami.

5.1 Platforma testowa

Wszystkie testy wykonano na komputerze stacjonarnym pod systemem *Ubuntu 14.04.3 LTS*, z zainstalowanymi sterownikami *NVIDIA 352.68* oraz *CUDA 7.5*. Parametry komputera:

- Procesor: AMD FX(tm)-8350 Eight-Core Processor
- Karta graficzna: 2 x GeForce GT 640
- Pamięć RAM: 16 GB (1600 Mhz)
- Dysk: SSD 40 GB
- Płyta główna: 970A-UD3 - Gigabyte

5.2 Dane

5.2.1 Rzeczywiste

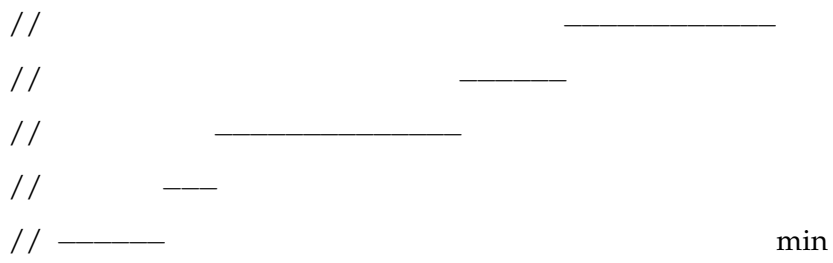
Dane rzeczywiste pochodzą z dwóch źródeł. Pierwsze pozyskano z pomiarów działającego komputera MAC, wykonane za pomocą prostego programu OsxSystemDataLogger[1], które zawierają pola takie jak czas, napięcie na procesorze oraz temperaturę procesora i GPU. Dane te dalej będę nazywał *INFO*. Drugie pochodzą z *New York Stock Exchange* i są danymi historycznymi wszystkich zdarzeń w czasie rzeczywistym w formie *NYSE OpenBook*[2] zawierającymi ponad 20 kolumn, a w tym czas, wolumen, cenę, stronę, symbol i inne. Dane te dalej będę nazywał *NYSE*.

5.2.2 Wygenerowane

Czas

Liczby całkowite o długości $64bit$ monotonicznie rosnące od zadanej wartości startowej, o losową wielkość z zadanego przedziału, z pewnym prawdopodobieństwem. Dla prawdopodobieństwa 0 czas się nie zmienia, natomiast dla prawdopodobieństwa 1, nie będzie dwóch tych samych liczb w wygenerowanym ciągu.

Listing 5.1: Time pattern



Pattern A

Liczby dowolnego typu i wielkości, z przedziału od zadanego minimum(v_{min}) do maksimum(v_{max}). Wartość generowanych punktów zmienia się co len kroków oraz

[1] https://github.com/dzitkowskik/osx-system-data-logger

[2] http://www.nyxdata.com/Data-Products/NYSE-OpenBook

przed każdą zmianą generowany jest pojedynczy punkt o wartości v_{max}.

Listing 5.2: Pattern A

```
// *       *       *       *       *       * max
//                 _______
//           _______         _______
// _______                             _______ min
// <-len->
```

Pattern B

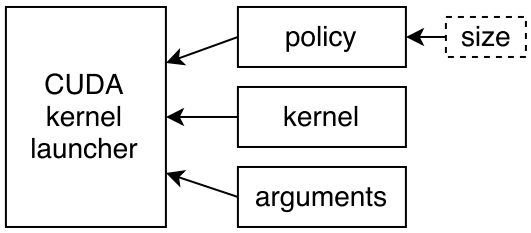
Dane B składają się z dwóch części położonych na przemian co len kroków, z których pierwsza składa się z naprzemian położonych wartości bliskich v_{max} i v_{min}. Druga część składa się z wartości malejących od v_{max} do v_{min} co ustaloną wartość, a następnie rosnąca z powrotem.

Listing 5.3: Pattern B

```
//   pattern 1     |       pattern 2       |      pattern 1
// * * * * * * * * *                         * * * * * * * * * max-rand(0,5)
//  # # # # # # # #    *                   *  # # # # # # # #  max-rand(0,5)
//                       *               *
//                         *           *
//                           *       *
//                             *   *
//  * * * * * * * * *            *   *          * * * * * * * * min+rand(0,5)
//   # # # # # # # #               *              # # # # # # #  min+rand(0,5)
// <------len------> <----------len----------> <-----len----->
```

5.3 Ustawienia siatki CUDA (grid)

Definicja 5.3.1 (Kernel Policy)**.** Polityka kernela definiuje w jaki sposób tworzyć siatkę wątków podzielonych na bloki w CUDA dla podanego rozmiaru danych oraz którego *stream* należy użyć.



Rysunek 5.1: Schemat wywoływania kernela

Pomimo zaimplementowania wygodnego do testów systemu uruchamiania kerneli za pomocą *Kernel Policy*, nie starczyło czasu na przetestowanie optymalnych ustawień polityki dla każdego algorytmu. Dlatego też, podczas testów wszystkie autorskie kernele, były wywoływane z użyciem domyślnej polityki tworzącej bloki o rozmiarze 1024 wątków, co może negatywnie rzutować na wydajność systemu.

5.4 Badanie współczynnika kompresji

Wyniki tego pomiaru otrzymano kompresując pliki *CSV* z danymi rzeczywistymi omówionymi wcześniej za pomocą zaimplementowanego równoległego kompresora oraz kompresorów *7z, zip, bz2* dostępnych w systemie *Linux Ubuntu*. Dane *GEN1* powstały z wygenerowanych kolumn Czasu oraz *Pattern A i B*, opisanych wyżej, dla danych całkowitych oraz zmiennoprzecinkowych pojedynczej precyzji, natomiast *BROW*, to dzienne wartości udziału w rynku przeglądarek[3].

Tabela 5.1: Współczynnik kompresji dla plików CSV z danymi w porównaniu ze standardowymi kompresorami

	optymalizator		bz2		zip		7z	
	ratio	czas	ratio	czas	ratio	czas	ratio	czas
INFO	27.26	2.84s	20.29	3.88s	22.60	0.66s	30.31	5.64s
NYSE	8.99	37.40s	6.60	1m 19.62s	5.64	32.45s	7.42	6m 23.58s
GEN1	17.43	4.86s	8.33	5.34s	7.95	2.92s	10.91	36.05s
BROW	5.12	1.43s	4.51	0.02s	4.15	0.02s	5.40	0.05s

[3] `http://www.economicswebinstitute.org/data/browsermarket01.xls`

Z pomiarów (patrz tabela 5.1) wynika, że dla większości danych optymalizator zachowuje się lepiej niż dostępne w systemie kompresje, które poza tym są o wiele wolniejsze dla dużych danych, co widać dla danych *NYSE* o rozmiarze *720MB*. W dwóch przypadkach jednak *7z* okazał się nieco lepszy, natomiast obie kompresje mają porównywalne wyniki. W pomiarach kompresje korzystają ze wszystkich 8 rdzeni, a kompresor *gzip* okazał się najszybszy, dając porównywalny wynik do optymalizatora. Algorytm korzystający z *CUDA* traci niestety dużą część czasu na inicjalizację oraz transfer danych przez szynę *PCI-E*. Wyniki te uzyskano dla przykładowego programu, używającego równoległego kompresora opisanego wcześniej. Czas liczony jest łącznie z czytaniem danych z dysku, co zajmuje większość czasu.

5.5 Wygenerowane schematy

Kolejny test polegał na prześledzeniu działania optymalizatora dla dużej ilości danych w wariancie *Pattern B*. Patrząc na dane możemy domyślać się jak powinna wyglądać dobra kompresja dla obu części. W pierwszym przypadku, algorytm powinien podzielić dane na górne i dolne operacją *PATCH*, następnie transformować obie części operacjami *SCALE* i *DELTA*, a na koniec skompresować kodowaniem *AFL* lub *GFC*. W drugim spodziewamy się kombinacji *DELTA* z *RLE* lub *CONST*. Przebieg działania algorytmu dla tych danych, pokazuje tabela 5.2.

Całkowity poziom kompresji wyniósł lekko ponad 4, co biorąc pod uwagę pojedyncze statyczne scenariusze, jest dobrym wynikiem. Dla porównania skompresowanie całych danych za pomocą sugerowanych na początku scenariuszy wynoszą odpowiednio 2.4 oraz 1.3 co przedstawia listing 5.4.

Listing 5.4: Pattern B - wyniki przykładowych schematów

```
patch,delta,scale,afl,none,delta,scale,afl,none:
compression ratio = 2.43919

delta,rle,afl,none,afl,none:
compression ratio = 1.3374
```

Tabela 5.2: Działanie optymalizatora dla kolejnych paczek danych Pattern B

Lp.	schemat kompresji	ratio	pattern	zmiana
1	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
2	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
3	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
4	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
5	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
6	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
7	scale,patch,afl,none,scale,afl,none	2.32829	2	tak
8	scale,dict,afl,none,scale,afl,none	2.32829	2	tak
9	scale,patch,afl,none,scale,afl,none	2.32829	2	tak
10	delta,rle,delta,none,delta,none	250.000	2	tak
11	delta,scale,rle,none,none	263.158	2	nie
12	delta,scale,rle,none,none	263.158	2	nie
13	delta,scale,rle,none,none	1.00000	1	tak
14	delta,rle,delta,afl,none,delta,afl,none	1.00000	1	tak
15	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
16	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
17	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
18	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
19	scale,patch,afl,none,scale,afl,none	2.32829	2	tak
20	delta,rle,delta,none,delta,none	250.000	2	tak
21	delta,rle,constData,none,delta,none	243.902	2	tak
22	delta,rle,patch,afl,none,afl,none,delta,none	169.492	2	tak
23	delta,rle,delta,afl,none,delta,none	227.273	2	nie
24	delta,rle,delta,afl,none,delta,none	227.273	2	tak
25	scale,patch,afl,none,scale,afl,none	7.71010	1	nie
26	scale,patch,afl,none,scale,afl,none	7.71010	1	nie

Tabela 5.2 pokazuje jak algorytm dostosowuje się do zmieniającej się charakterystyki danych oraz jak mutuje schematy aby mieć szansę otrzymania lepszego rozwiązania, np. zmiana pomiędzy iteracją 10 a 11. Warto zauważyć, że na końcu po powrocie do dawnej charakterystyki algorytm stosuje ponownie ten sam schemat co na początku, który był optymalny.
Dla danych *Pattern A* algorytm stosuje jedną strategię i jej nie zmienia, ponieważ charakterystyka szeregu jest stała. Strategia ta osiąga współczynnik kompresji 46.8384 i wygląda następująco:
$rle, patch, rle, none, none, rle, none, none, patch, delta, afl, none, rle, none, none$
Analogiczne zachowanie otrzymujemy dla wygenerowanych punktów czasu[4], gdzie algorytm używa kodowania *RLE* oraz *AFL*, stosując również skalowanie oraz deltę. W ten sposób dla czasu osiągany współczynnik kompresji jest rzędu 100.

5.5.1 Pojedyncze schematy

Wykonano porównanie współczynnika kompresji, osiąganego dla danych o zmiennej (okresowo) charakterystyce, przez optymalizator, w stosunku do optymalnego drzewa kompresji dla pierwszej paczki danych. Pomiary przeprowadzono dla optymalizatora wykonującego fazę 1 algorytmu raz na dziesięć paczek danych.

Tabela 5.3: Porównanie optymalizatora do pojedynczych schematów

Dane	COpt	CSch	Schemat	Zysk
B	1.80	0.54	scale,patch,afl,n,scale,afl,n	3.33
A+B	4.71	0.67	dict,scale,dict,n,n,rle,const,n,delta,afl,n	7.03
Time+A	9.75	0.39	patch,rle,const,n,delta,afl,n,const,n	25.00
A+B+MaxPrec	2.00	0.40	rle,patch,const,n,const,n,patch,n,const,n	5.00
Time+Cons+Rand	1.37	0.60	rle,scale,afl,n,delta,dict,afl,n,n	2.28

Wyniki testu przedstawia tabela 5.3, w której:

- Dane - kombinacja paczek danych o różnych charakterystykach (zmiana co 5 paczek)

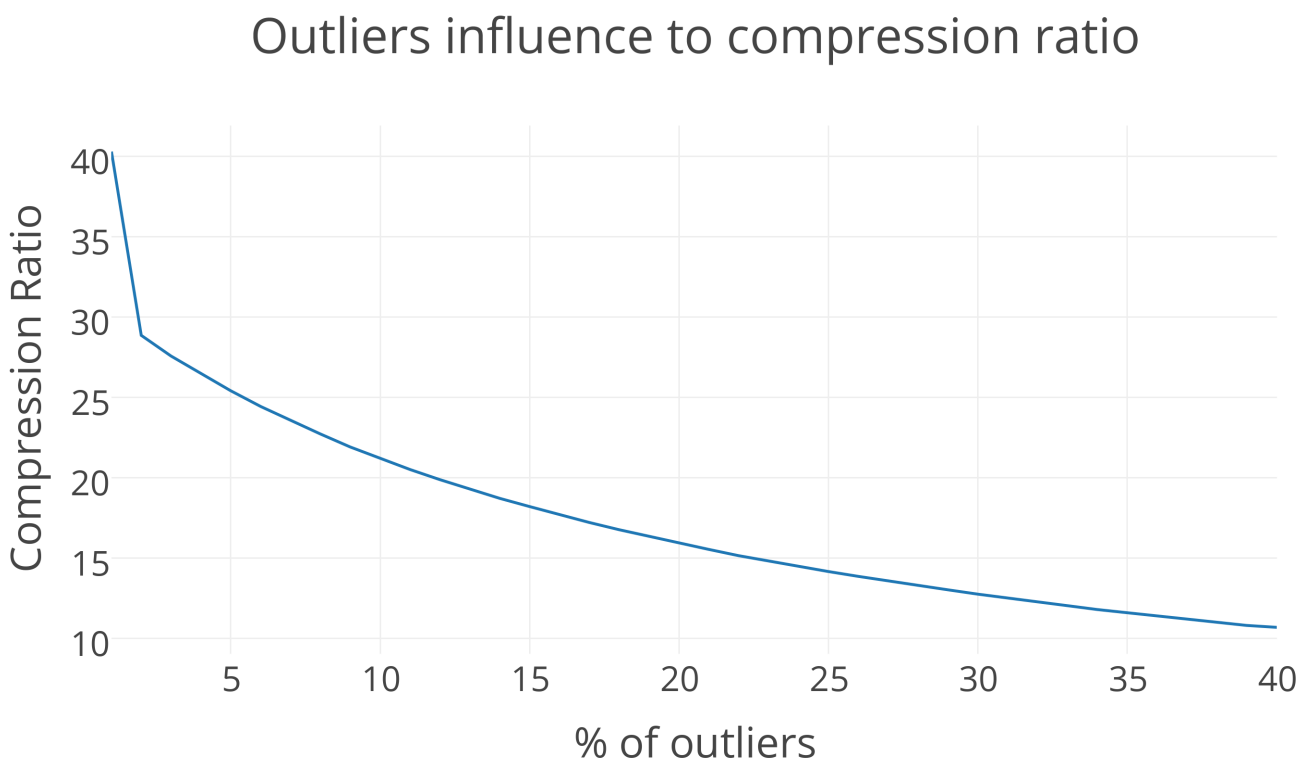
[4]patrz dane czasu - listing 5.1

- COpt - wsp. kompresji osiągnięty przez optymalizator
- CSch - wsp. kompresji osiągnięty przez optymalne drzewo dla 1 paczki w kompresji całości danych
- Zysk - stosunek COpt do CSch, czyli wzrost współczynnika kompresji przy użyciu optymalizatora

Średni wzrost współczynnika kompresji przy zastosowaniu optymalizatora zamiast pojedynczego drzewa kompresji wynosi 8.53. Rozmiar danych został zmniejszony średnio około 4 krotnie, co świadczy o tym, że algorytm dobrze radzi sobie z danymi o zmiennej charakterystyce.

5.6 Dane obarczone błędami

W większości danych pomiarowych trafiają się błędne odczyty lub zaburzenia, które nie pasują do ogólnej charakterystyki danych. Zbadano wpływ takich wartości, zwanych z angielskiego *outliers*, na jakość kompresji danych (patrz rys. 5.2). Badane dane są wygenerowanymi punktami czasu, z dużą powtarzalnością (jak w *Pattern A*), z pewnym procentowym udziałem wartości odstających w losowych miejscach.

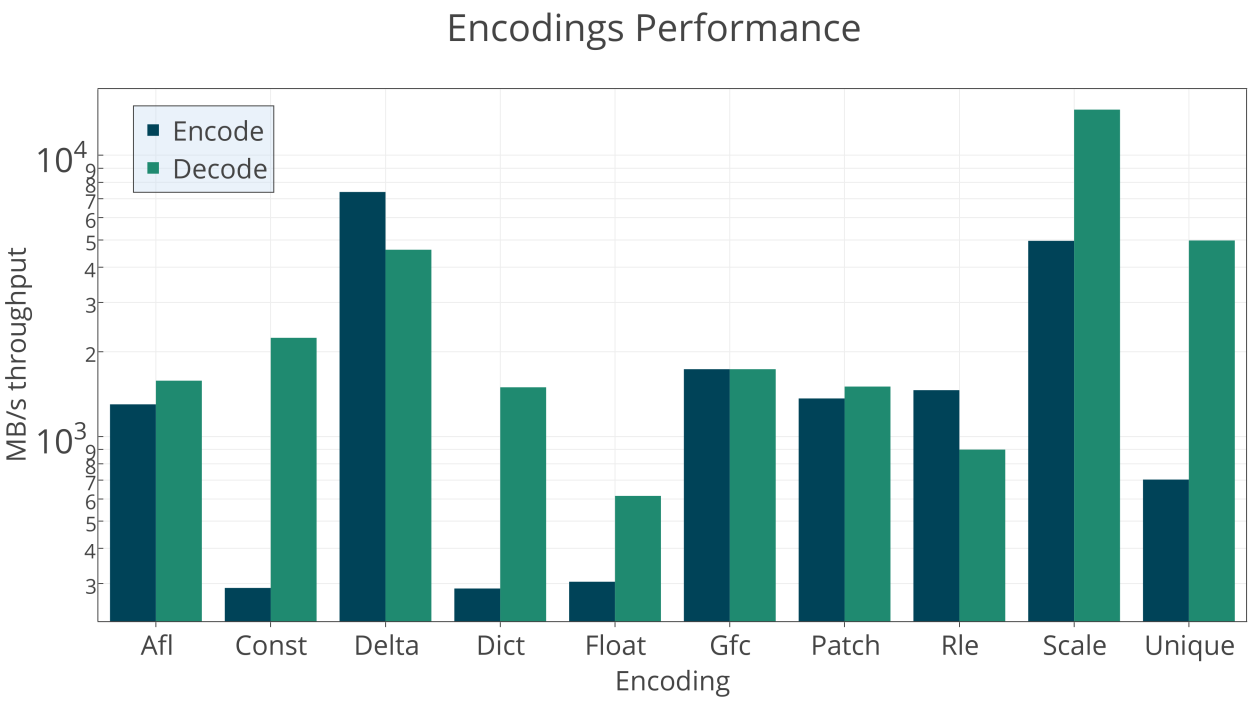


Rysunek 5.2: wsp. kompresji vs % outliers

W rezultacie przy braku outlier'ów optymalizator używał kodowania *RLE*, natomiast po przekroczeniu paru procent błędnych odczytów, zmienił drzewo kompresji na: $patch, delta, rle, none, none, scale, afl, none$ w celu oddzielenia wartości niepasujących od właściwego szeregu i skompresowania ich za pomocą *AFL*. Wraz ze wzrostem liczby outlier'ów poziom kompresji spada liniowo, lekko się spłaszczając by ostatecznie ustalić się na poziomie $C \approx 5$, używając głównie *AFL* i *CONST*.

5.7 Wydajność

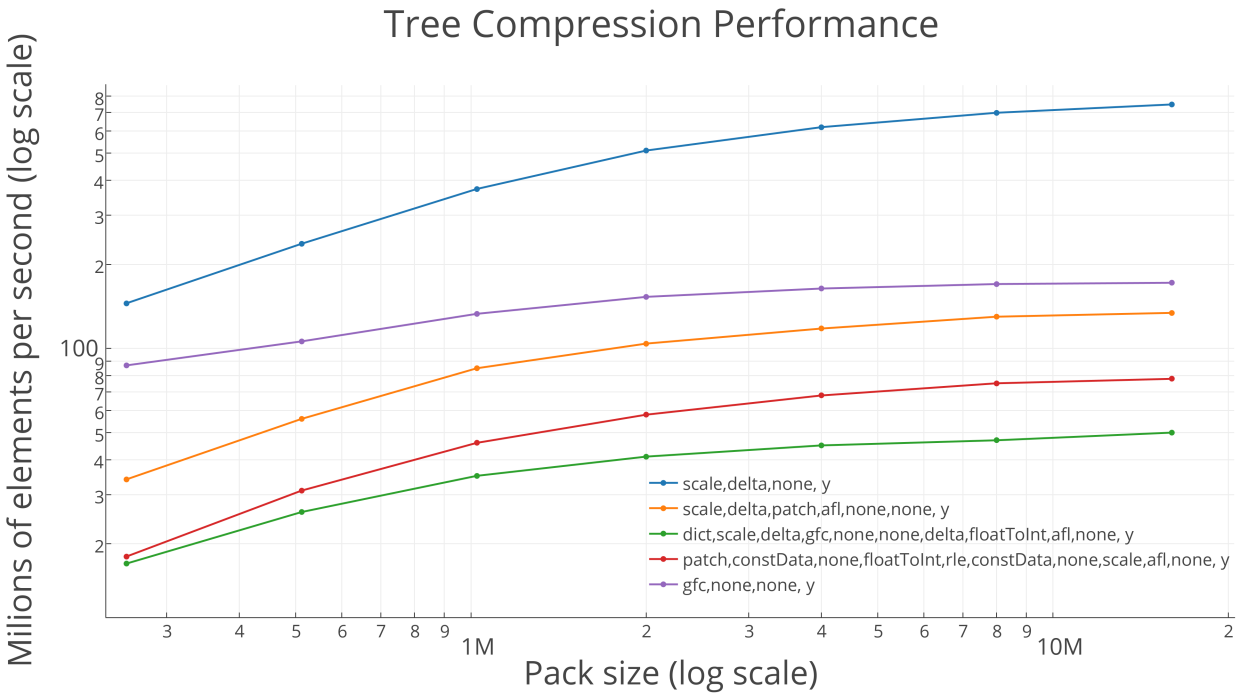
Wydajność kaskadowej kompresji jest limitowana przede wszystkim przez szybkość najwolniejszej kompresji użytej w planie. Niestety, nie wszystkie użyte kompresje zostały zaimplementowane optymalnie, co ma znaczący wpływ na osiągi całości. Będzie to jednak tematem innej pracy i zgodnie z wieloma publikacjami, możliwa jest wydajniejsza implementacja użytych algorytmów [24, 21, 22, 23]. Ponadto, wydajność ta w dużej mierze zależy od użytej karty, która w tym przypadku jest relatywnie niska w porównaniu do drogich kart z wyższej półki. Dla porównania wydajność analogicznej implementacji algorytmu *GFC* uzyskana przez autorów wynosiła około $10GB/s$, natomiast na mojej karcie tylko niecałe $2GB/s$. Przepustowość poszczególnych kodowań i dekodowań umieszczona jest na rysunku 5.3. Można przy-



Rysunek 5.3: Wydajność kodowań

puszczać, że znaczący wpływ na spadek wydajności kompresji ma użycie w drzewie kodowań bazujących na słowniku lub precyzji i te metody powinny być optymalizowane w pierwszej kolejności. Z drugiej strony, użycie transformacji nie powinno wpływać na wydajność drzewa, ponieważ są one znacznie wydajniejsze od reszty

kodowań. Wydajność wybranych drzew kompresji w zależności od wielkości paczki danych kompresowanych przez pojedyncze drzewo, pokazano na rysunku 5.4. Jak można się domyśleć, czym większe i bardziej skomplikowane drzewo tym wolniejszy algorytm, dlatego też warto wybierać niższe drzewo, jeśli nie spowoduje to znacznego spadku jakości. Z uwagi na architekturę GPU, opłaca się za jednym zamachem przetwarzać jak najwięcej danych, pojedynczym drzewem, zatem powinno się dzielić przychodzące dane na paczki o wielkości większej niż milion elementów. Ponadto, bardzo znaczący wpływ na wydajność algorytmu ma wykonanie fazy 1,



Rysunek 5.4: Wydajność kompresji drzewami

której wydajność plasuje się na poziomie jedynie $5MB/s$, ponieważ sprawdza dużą ilość drzew[5]. Może się to wydawać niewiele, jednak faza ta wykonywana jest na bardzo małej liczbie elementów lub równolegle do właściwej kompresji. Jeśli faza ta wykonywana jest rzadko, przestaje wpływać na właściwą wydajność kompresji.

W przeważającej liczbie kodowań prędkość dekompresji jest znacząco większa niż prędkość kompresji[6]. Przy dekodowaniu z danych optymalizatorem, nie ma po-

[5]patrz tabela 5.4
[6]patrz rysunek 5.3

Tabela 5.4: Ilość drzew kompresji generowana dla 2 różnych heurystyk: A - szerokiej, z luźnymi regułami i generującą dużą ilość drzew oraz B - wąską, mocno ograniczoną i generującą małą ilość drzew

Data	Heurystyka A		Heurystyka B	
	Ilość kand. drzew	ratio	Ilość kand. drzew	ratio
NYSE time	97404	176.991	3787	88.691
Float Prec 3	2686	4.03918	1040	1.645
Random Int	1104	2.28206	110	1.000
Int consecutive	38427	625	2963	625.000
Int Pattern A	87973	236.686	494	196.078
Float Pattern A	87370	220.994	1413	196.078
Int Pattern B	1346	7.857	203	2.383
Float Pattern B	3654	4.2328	1837	3.336

trzeby wyliczania żadnych statystyk ani sprawdzania wielu drzew. Wydajność dekompresji powinna być zatem dużo wyższa, co potwierdzają testy pokazane na listingu 5.5.

Listing 5.5: Benchmark optymalizatora

```
Pattern A (type INT):
Pack size = 32M
Scheme = rle[266],patch[13],const[22],none[1],const[23],none,
patch[9],delta[9],afl[9],none,const[23],none
Ratio = 266.407
Compression throughput = 431.267 MB/s
Decompression throughput = 1.28074 GB/s

Pattern B (type INT):
Pack size = 32M
Scheme = scale[7],patch[7],afl[10],none,scale[10],afl[10],none
Ratio = 7.99996
```

```
Compression throughput = 554.785MB/s
Decompression throughput = 853.789MB/s

Time (type LONG):
Pack size = 16M
Scheme = delta[33],patch[33],rle[1014],none,none,scale[63],afl[63],none
Ratio = 33.77
Compression throughput = 257.235MB/s
Decompression throughput = 1.12897GB/s
```

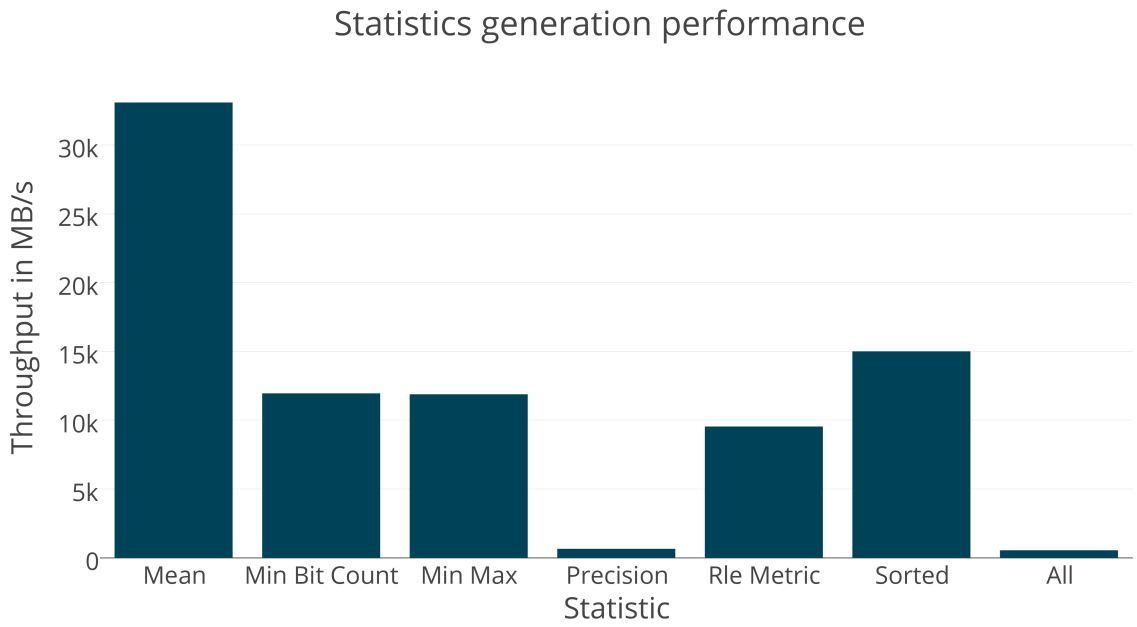
Wartości w nawiasach odpowiadają uciętym do jedności wartościom współczynnika kompresji uzyskanego w poszczególnych węzłach drzewa. Kompresowane dane w powyższym teście były podzielone na 25 paczek po „*Pack size*" elementów, a faza 1 wykonana została 2 do 3 razy w trakcie trwania testu. Biorąc pod uwagę fakt, że wydajność danych drzew jest porównywalna do szybkości optymalizatora, który je wywnioskował, można stwierdzić, że algorytm poprawiania i zgadywania drzew, nie spowalnia znacząco samego procesu kompresji drzewem.

5.7.1 Memory coalescing

Zbadano również wpływ użycia łącznego dostępu do pamięci globalnej, opisanego w rozdziale 2.1, na wydajność implementacji kodowania Fixed-Length, ze „z góry" znanym σ, na danych Pattern A. Wersja z wyrównaniem - AFL, używająca tej techniki ($\omega = 32$) uzyskała przepustowość 20 GB/s, natomiast zwykły FL ($\omega = 1$) osiągnął jedyne 2.5 GB/s, co daje wzrost wydajności o prawie rząd wielkości.

5.7.2 Generowanie statystyk

Podczas pierwszej fazy algorytmu, jak również w czasie trwania niektórych kodowań, wyliczane są statystyki aktualnych danych. Wyliczanie metryki *RLE* zamiast prawdziwej, maksymalnej lub średniej długości ciągów, poskutkowało bardzo wydajną implementacją na GPU. Wyliczanie precyzji nie zostało niestety odpowiednio zoptymalizowane pod kątem obliczeń na *CUDA* i jego wydajność ogranicza sumaryczną przepustowość obliczania statystyk do około *500 MB/s*. Ponadto w testach



Rysunek 5.5: Wydajność wybranych statystyk

na danych typu całkowitego *Pattern A*, maksymalna przepustowość implementacji histogramu oraz statystyki *Dictionary Counter* wyniosły odpowiednio *5.63227 GB/s* oraz *790.123 MB/s*.

Rozdział 6

Podsumowanie

Podsumowując, udało się zaimplementować działający prototyp dynamicznego optymalizatora kompresji szeregów czasowych, który potrafi dostosować się do zmiennej charakterystyki napływających danych. Kompresor w dobrym tempie koduje dane zachowując wysoki poziom kompresji, dorównujący, a dla właściwych szeregów czasowych nawet przewyższający, niektóre istniejące rozwiązania. Algorytm wykazuje wyższy poziom kompresji dla realistycznych danych, niż szeroko stosowane kompresory 7z czy zip. Jednak jest to tylko prototyp i dla rozwiązań w bazach danych GPU uzyskana wydajność kompresji nie jest wystarczająca. Z drugiej strony, patrząc na wydajność odpowiednio zoptymalizowanych odpowiedników wdrożonych kompresji można uznać, że możliwe jest ulepszenie tego rozwiązania w taki sposób, by uzyskało wymaganą wydajność na poziomie kilkudziesięciu Gb/s.

6.1 Konkluzja

Używając kaskadowej kompresji specyficznej dla konkretnego szeregu czasowego można osiągnąć dużo lepsze rezultaty niż stosując metody słownikowe, które często wykazują niską wydajność. Uzyskano ponad ośmiokrotny wzrost jakości kompresji[1] dla danych o zmiennej charakterystyce względem kompresji pojedynczym schematem. Wskazane zostały słabe punkty dotychczasowej implementacji, których wyeli-

[1] patrz tabela 5.3

minowanie powinno prowadzić do wyrównania wydajności z istniejącymi rozwiązaniami kaskadowej kompresji na GPU. Co więcej można uznać, że zaproponowane rozwiązanie ma szansę z powodzeniem zostać wdrożone jako kompresor w bazie danych po stronie GPU, który na bieżąco będzie kompresował i dekompresował napływające dane.

6.2 Przyszłe prace

Projekt nadal wymaga włożenia dużego nakładu pracy w celu optymalizacji wydajności, a także implementacji wielu algorytmów kodowania przydatnych do kompresji szeregów. Dalsze prace będą miały na celu implementację kompresji za pomocą regresji częściowej na GPU. Warto będzie użyć wersji niektórych algorytmów z wbudowanym *patchowaniem* w celu przyspieszenia obliczeń. Ważnym elementem, który będzie w przyszłości również zaimplementowany jest równoległe i niezależne wykonanie fazy 1 algorytmu optymalizatora, która zajmuje większość czasu wykonania. Dodatkowo jest planowane udostępnienie rozwiązania w formie *Open Source*, wraz z przykładami oraz gotowym środowiskiem uruchomieniowym w formie dockera[2]. Ostatecznie projekt mógłby być wykorzystany w bazie danych szeregów czasowych, po stronie GPU.

[2]An open platform for distributed applications for developers and sysadmins: `https://www.docker.com/`

Bibliografia

[1] T Mostak. An overview of mapd (massively parallel database). White Paper, Massachusetts Institute of Technology, April 2013.

[2] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems (TOIS)*, 33(3):15, 2015.

[3] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.

[4] Eugene Fink and Harith Suman Gandhi. Compression of time series by extracting major extrema. *Journal of Experimental & Theoretical Artificial Intelligence*, 23(2):255–270, 2011.

[5] Piotr Przymus and Krzysztof Kaczmarski. Dynamic compression strategy for time series database using gpu. In *New Trends in Databases and Information Systems*, pages 235–244. Springer, 2014.

[6] Miguel C Ferreira. *Compression and query execution within column oriented databases*. PhD thesis, Massachusetts Institute of Technology, 2005.

[7] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[8] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[9] Piotr Przymus and Krzysztof Kaczmarski. Improving efficiency of data intensive applications on gpu using lightweight compression. In *On the Move to Meaningful Internet Systems: OTM 2012 Workshops*, pages 3–12. Springer, 2012.

[10] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *Computers, IEEE Transactions on*, 58(1):18–31, 2009.

[11] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[12] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 370–379. IEEE, 1998.

[13] Garland Michael Satish Nadathur, Harris Mark. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing*, IEEE International Symposium, pages 1–10. IEEE, 2009.

[14] Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (TOG)*, 28(5):154, 2009.

[15] John D. Owens Mark Harris, Shubhabrata Sengupta. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–873, August 2007.

[16] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[17] M Pietroń, Pawel Russek, and Kazimierz Wiatr. Accelerating select where and select join queries on a gpu. *Computer Science*, 14(2):243–252, 2013.

[18] Piotr Przymus and Krzysztof Kaczmarski. Time series queries processing with gpu support. In *New Trends in Databases and Information Systems*, pages 53–60. Springer, 2014.

[19] Shin Morishima and Hiroki Matsutani. Performance evaluations of document-oriented databases using gpu and cache structure. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3, pages 108–115. IEEE, 2015.

[20] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[21] Piotr Przymus and Krzysztof Kaczmarski. Compression planner for time series database with gpu support. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 36–63. Springer, 2014.

[22] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[23] Molly A O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 7. ACM, 2011.

[24] Robert Louis Cloud, Matthew L Curry, H Lee Ward, Anthony Skjellum, and Purushotham Bangalore. Accelerating lossless data compression with gpus. *arXiv preprint arXiv:1107.1525*, 2011.

[25] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[26] Adnan Ozsoy, Martin Swany, and Anamika Chauhan. Pipelined parallel lzss for streaming data compression on gpgpus. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 37–44. IEEE, 2012.

[27] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Optimizing lzss compression on gpgpus. *Future Generation Computer Systems*, 30:170–178, 2014.

[28] K Shyni and Manoj Kumar KV. Lossless lzw data compression algorithm on cuda. *IOSR Journal of Computer Engineering (IOSR-JCE)*, 13(1):122–127, 2013.

[29] Anders Lehrmann Vrønning Nicolaisen. Algorithms for compression on gpus. 2013.

[30] Yuan Zu and Bei Hua. Glzss: Lzss lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 46. ACM, 2014.

[31] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[32] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. Cuda-lite: Reducing gpu programming complexity. In *Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 2008.

Warszawa, dnia .....

Oświadczenie

Oświadczam, że pracę magisterską pod tytułem: „Optymalizator kompresji szeregów czasowych na GPU”, której promotorem jest dr inż. Krzysztof Kaczmarski, wykonałem/am samodzielnie, co poświadczam własnoręcznym podpisem.

.....