



POLITECHNIKA WARSZAWSKA

WYDZIAŁ MATEMATYKI  
I NAUK INFORMACYJNYCH



PRACA DYPLOMOWA MAGISTERSKA

INFORMATYKA

# **Optymalizator kompresji szeregów czasowych na GPU**

Autor:

Karol Dzitkowski

Promotor: dr inż. Krzysztof Kaczmarek

Warszawa, luty 2016

.....

podpis promotora

.....

podpis autora

## **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 przedstawiono szereg rozwiązań, korzystających z kart graficznych jako koprocesory w bazach danych lub nawet bazy danych 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. W tej pracy przedstawię nowe podejście do kompresji szeregów czasowych po stronie GPU, przy użyciu planera budującego na bieżąco drzewa kompresji na podstawie statystyk napływających danych. Przedstawione rozwiązanie kompresuje dane za pomocą lekkich i bezstratnych kompresji w technologii CUDA.

## **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 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 essential. In this paper I present a new approach to compression of time series on the side of the GPU, using a planner to keep building the compression tree based on statistics of incoming data. The solution compresses data using lightweight and lossless compression in CUDA technology.

# Rozdział 1

## Wstęp

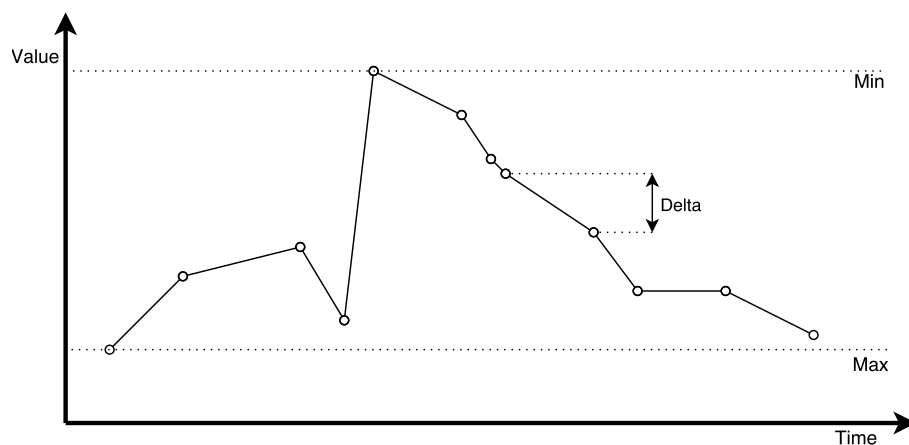
Poniższa praca zawiera opis implementacji optymalizatora kompresji szeregów czasowych, bazującego na dynamicznie generowanych statystykach danych. Pomysł opiera się na tworzeniu drzew kompresji (kompresja kaskadowa) oraz zbieraniu statystyk o krawędziach takich drzew - jak dobrze dana para kompresji sprawdza się dla napływających danych. System będzie również dynamicznie zmieniał - korygował, takie drzewa w zależności od charakterystyki kolejnych paczek danych. W założeniu system ma umożliwić kompresję dużych ilości danych przy wykorzystaniu potencjału obliczeniowego współczesnych kart graficznych.

### 1.1 Procesory graficzne

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 336GB/s przepustowości pamięci, podczas gdy najszybsze procesory takie jak Intel Core i7-5960x osiągają niecałe 180 GFLOPS, przy przepustowości 68GB/s. Kartom graficznym dorównują tylko inne jednostki typu SIMD, na przykład karty oblicze-

niowe Xeon Phi. Pomimo tak oszałamiających wyników, programowanie na jednostkach SIMD jest o wiele trudniejsze, jak również ograniczone przepustowością szyny PCI-E, która wynosi w porywach  $8GB/s$ , co dodatkowo przemawia za użyciem kompresji przy przetwarzaniu szeregów czasowych, choćby w celu przyspieszenia kopiowania danych z i na kartę graficzną w celu wykonania obliczeń.

## 1.2 Szeregi czasowe



Rysunek 1.1: Szereg czasowy

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 bazy danych po stronie GPU (najczęściej rozproszone), takie jak MapD lub DDJ (zaproponowana między innymi przeze mnie w poprzedniej pracy - inżynierskiej). Innymi rozwiązaniami są koprocesory obliczeniowe GPU, wspomagające działanie baz takich jak Cassandra, HBase, TempoDB, OpenTSDB czy PostgreSQL. Charakterystyka danych wielu szeregów wskazuje, że przy odpowiedniej obróbce mogą być

kompresowane z bardzo dużym współczynnikiem, szczególnie jeśli byłoby możliwe kompresowanie za pomocą dynamicznie zmieniających się ciągów (różnych) algorytmów kompresji i transformacji danych. Dla przykładu, jeśli jakiś fragment szeregu jest stały, z nielicznymi wyjątkami, warto byłoby usunąć wyjątki, a resztę skompresować jako jedną liczbę - uzyskując współczynnik kompresji rzędu długości danych.

### 1.3 SIMD i lekka kompresja

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 osobo i być może przy użyciu różnych algorytmów, zmieniających się dynamicznie w czasie. Takie kolumny wartości numerycznych tego samego typu wspólnie przetwarzają się przy użyciu procesorów typu SIMD, co daje wielokrotne przyspieszenie w stosunku do tradycyjnych architektur. Okazuje się że większość algorytmów lekkiej kompresji może z dużym powodzeniem być w ten sposób zrównoleglona. Również dynamiczne generowanie statystyk napływających danych może być przyspieszone z użyciem SIMD, co 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. Ogromna moc obliczeniowa procesorów graficznych może pozwolić wygenerować taki plan w rozsądnym 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.

## 1.4 Zawartość pracy

W tej pracy przedstawię planer kompresji (optymalizator) kompresujący napływające paczki danych. Zaprezentuję nowe podejście, planera budującego drzewa kompresji i uczącego się ich konstrukcji na podstawie na bieżąco generowanych statystyk węzłów takich drzew (jak również statystyk napływających danych). 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 w sposób równoległy kompresujący kolumny podanego szeregu czasowego. Następny podrozdział zawiera opis wcześniejszych prac prowadzonych w tych tematach, a także krótki opis architektury CUDA. W rozdziale 2 omówię stworzony framework oraz 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 wyniki prac i eksperymentów. Ostatni rozdział (piąty) to podsumowanie oraz zakres przyszłych prac i optymalizacji.

## 1.5 Powiązane prace

### 1.5.1 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. Najczęstszymi z nich są kodowanie ekstremami [12], stałej długości bitów [17], czyli tzw. NULL Suppression (NS) oraz proste kodowania słownikowe np. wszystkich unikalnych wartości, które można zakodować pewną założoną z góry liczbą bitów [24].

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 bi-



tów.

Szeregi czasowe to nie tylko liczby całkowitoliczbowe. Analizuje się także wiele sposobów kompresji liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji. Najczęściej próbuje się zamienić liczbę ułamkową na całkowitą stosując skalowanie [23]. Istnieją też bardziej skomplikowane metody na przykład kompresji liczb double algorytmem FPC [9], 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.

### 1.5.2 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 [38] [16]. W tych pracach pokazano przekład algorytmów kodowania z wyrównaniem do bajtów (Byte-Aligned Coding) oraz do słów (Word-Aligned Coding) oraz zmierzono wydajność implementacji wektorowej wersji tych kodowań z użyciem SSE. Autorzy zastosowali również binarne pakowanie (Binary Packing) w formie algorytmu FOR (Frame of Reference) [39] 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 to tradycyjnej implementacji.

### 1.5.3 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[2], hashowanie kukułcze[3, 19], sumy prefixowe[1] i inne zaimplementowane w podsta-

wowych bibliotekach takich jak CUDPP<sup>1</sup> czy Thrust<sup>2</sup>.

Najważniejsze są jednak bardzo zadowalające rezultaty zrównoleglania algorytmów używanych w bazach danych takich jak index search[25], wszelkiego rodzaju agregacje i operacje join, scatter i gather[6] oraz obliczanie statystyk danych[37] jak również dopasowywanie wyrażeń regularnych[30]. Dla przykładu wzrost wydajności oferowany przez algorytmy z biblioteki Thrust, która jest niejako odpowiednikiem Std, jest średnio 10-krotny[41] w stosunku do najszybszych wersji CPU. Większość przytoczonych wyżej przykładów również reprezentuje wzrost wydajności o rząd wielkości. W pracach odnośnie akceleracji baz danych za pomocą technologii CUDA, autorzy otrzymują przyspieszenie 20 – 60 krotne[10], w przypadku operacji *SELECT WHERE* i *SELECT JOIN* z agregacjami[6]. Jednak jest to liczone bez uwzględniania czasu kopiowania danych na GPU. Jak obliczono zajmuje to średnio ok 90% czasu działania algorytmów[13].

#### 1.5.4 Kompresje

Opisane wyżej algorytmy lekkiej kompresji szeregów czasowych, w większości zostały zaimplementowane przez Fang et al.[15] oraz Przymus et al.[13] w ich pracach, gdzie autorzy zaznaczają ogromny wzrost przepustowości takich algorytmów oraz ich wysoką skuteczność w kompresji szeregów. W przypadku pierwszego jest to nawet to 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 zmienopozycyjnych o podwójnej precyzji przestawił w swojej pracy O'Neil et al.[20], 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 algoryt-

---

<sup>1</sup>CUDA Data Parallel Primitives - <http://cudpp.github.io/>

<sup>2</sup>Parallel algorithms library - <https://developer.nvidia.com/thrust>

mu FL dla liczb naturalnych[17]). Algorytm nazwano GFC i jest uogólnienie również 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[5], gdzie równoległa implementacja na GPU uzyskała 2 do 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.[14], 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[31]. Również inni autorzy mają nadzieję na optymalizacje z użyciem kodowań słownikowych takich jak LZW pobiją wyniki CPU jeśli dobrze przepiśże się je do architektury SIMD[27]. Można dodatkowo spotkać wariacje algorytmu LZSS przepisanego na CUDA, takie jak CANLZSS[7], która według autora przewyższa wydajnością ponad 60 razy seryjną implementację zwykłego LZSS. Kolejna optymalizacja GLZSS, w której zreorganizowano słownik to postaci tablicy haszy, oraz przyspieszono porównywanie podciągów (substrings) również zrównoleglając je na GPU, osiągając dwukrotne przyspieszenie względem poprzednich prac[21].

Porównując szybkość działania oraz współczynniki kompresji uzyskane przez autorów, można dojść do wniosku że zastosowanie metod lekkich kompresji dla szeregów czasowych, w miejscach gdzie szczególnie ważna jest szybka dekompresja (bazy danych), jest uzasadnione.

### 1.5.5 Planery kompresji

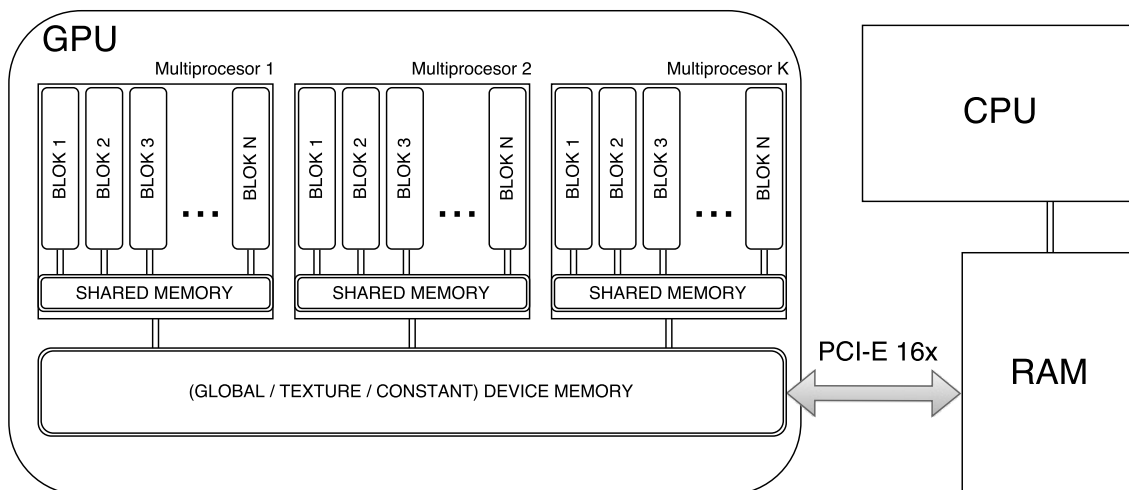
Okazuje się, że aby wielokrotnie zwiększyć współczynnik kompresji szeregu czasowego warto go przetransformować przed kompresją bądź nawet skompresować wielokrotnie różnymi algorytmami. W tym celu powstały planery kompresji które działają na zasadzie kodowania kaskadowego, czyli ciągu następujących po sobie kodowań, tworzących 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 deko-

dowania 56 GB/s została osiągnięta przez Fang et al.[15]. W tym przypadku posługując się heurystykami wykorzystującymi statystyki danych wejściowych, spośród ogromnej ilości dostępnych schematów (> 500 tys.) kodowania (planów), wybierano najlepiej pasujące (np. dla danych posortowanych powinny zaczynać się od RLE itd.). Następnie wybierano spośród nich plan spełniający zdefiniowane normy, np. plan o największym współczynniku kompresji. Statystyki na których oparty był algorytm brane były 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 Przymusa et al.[23], 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 momencie tworzenia planu, wykorzystując właściwości poszczególnych metod takiej kompresji (szczególnie w przypadku minimalnej ilości bitów potrzebnych do zapisania każdej liczby z danego wektora danych). Dodatkowo praca ta implementuje znajdowanie optimum ze względu na dwie sprzeczne zmienne (bi-objective optimizer): szybkość działania i jakość kompresji, stosując optymalność Pareto[42]. Generowanie statystyk dla takiego planera na GPU zapewnia do 70 razy lepszą wydajność w stosunku do analogicznej implementacji na CPU[13].

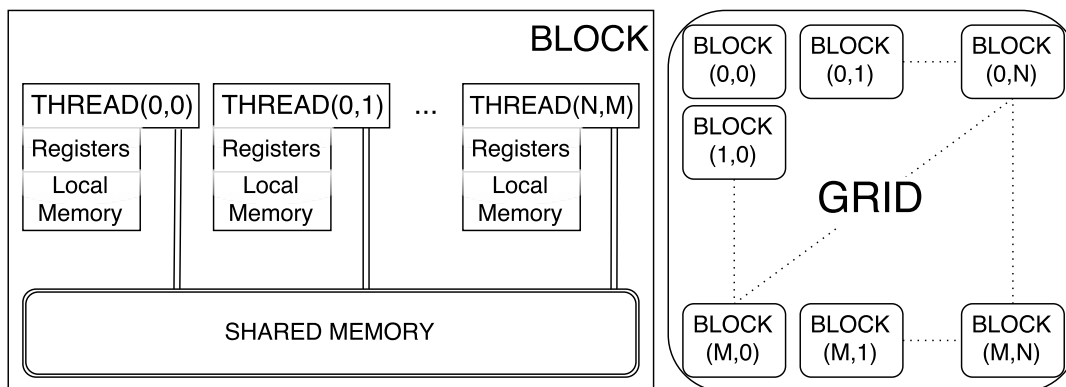
## 1.6 CUDA

Jedną z wielu zalet architektury kart graficznych jest to, że składają się z kilku 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 różnicą, że to nie organizacja wektora danych kontroluje jednostki obliczeniowe, a organizacja instrukcji pojedynczego wątku. Umożliwia on zatem pisanie równolegle wykonywanego kodu dla niezależnych i skalowalnych wątków, jak i dla wątków koordynowanych



Rysunek 1.2: Architektura NVIDIA CUDA

danymi. Wszystkie wątki wykonują ten sam kod funkcji kernela. 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.



Rysunek 1.3: Abstrakcja bloków i siatki w CUDA

# Rozdział 2

## Opis systemu

### 2.1 Kodowania

Przedstawię krótko 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 po-myśle usuwania zbędnych zer wiodących, dla liczb naturalnych - AFL oraz ułam-kowych - GFC. Trzeba zauważyć, że operacja patchowania jest tutaj zaimplemento-wana odmiennie niż w innych publikacjach, jako osobny rodzaj kodowania, mogący przybierać wiele form, natomiast algorytmy nie mają swoich wersji z patchowaniem.

#### 2.1.1 Podstawowe algorytmy transformacji szeregów

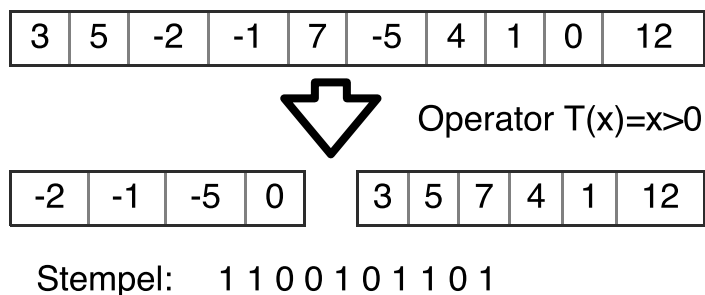
**Delta** – Zapisuje zmiany pomiędzy kolejnymi danymi w szeregu, zapisując pierw-szą wartość w metadanych. Bardzo dobrze sprawdza się na danych posorto-wanych bądź o zmianach liniowych o stałej wartości. Algorytm ten został za-implementowany częściowo z użyciem biblioteki Thrust i funkcji *inclusive\_scan* - do dekodowania.

**Scale** Bardzo prosta transformacja, odmienna od wielu implementacji o tej samej na-zwie, polegająca na odjęciu lub dodaniu (dla liczb ujemnych) najmniejszej do-

datniej 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 zapisane na dużo mniejszej liczbie bitów.

**FloatToInt** – To jest wersja algorytmu którą często w literaturze nazywa się mianem *scale*. Znając maksymalną precyzję danych zmiennopozycyjnych, można zapisać te dane jako liczby całkowite, mnożąc przez odpowiednią potęgę liczby 10. Tak więc mając wektor cen w złotych, można przemnożyć cenę 99.99 przez 100 otrzymując 9999 i zmienić reprezentację liczb na całkowite. Ponieważ reprezentacja liczb całkowitych może dać się lepiej skompresować.

**Patch** – Bardzo ważnym zadaniem w kompresji szeregów czasowych jest usuwanie wartości odstających tzw. *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 wiele różnych wersji patchowania może być zdefiniowanych, może na przykład dzielić wartości na ujemne i dodatnie.



Rysunek 2.1: Przykład użycia patchowania z operatorem - "większy od zera"

W przeciwieństwie do innych prac algorytm ten nie zapisuje wyjątków wraz z ich pozycjami, i umożliwia w ten sposób lepsze ich skompresowanie w dalszych krokach. Zamiast tego trzyma w metadanych zapisany bitowo stempel przynależności poszczególnych elementów wektora to pierwszej lub drugiej tablicy wynikowej. Rozmiar takich metadanych wynosi  $(N + 32)$  bitów, gdzie  $N$  to liczba kompresowanych elementów.

### 2.1.2 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*. Aby tak się stało można zastosować poniższy algorytm obliczania indeksów wejściowych dla danego wątku CUDA.

Oznaczmy jako:

- $\Sigma$  – ilość przetwarzanych elementów
- $\omega$  – ilość wątków w grupie
- $\kappa$  – ilość elementów w bloku danych
- $\lambda$  – 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)

Założmy że chcemy przetworzyć 1MB danych tj.  $1024 * 1024$  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:  $\Sigma = 1024 * 1024$ ,  $\omega = 32$ ,  $\kappa = 32$ ,  $\lambda = 16$ ,  $w_g = 4$ .

Bardzo łatwo można obliczyć ile wynosi rozmiar pojedynczego bloku  $B_{size}$ , oraz ilość wszystkich bloków, potrzebnych do przetworzenia wszystkich elementów wektora wejściowego  $B_{count}$ :

$$B_{size} = \omega * w_g$$

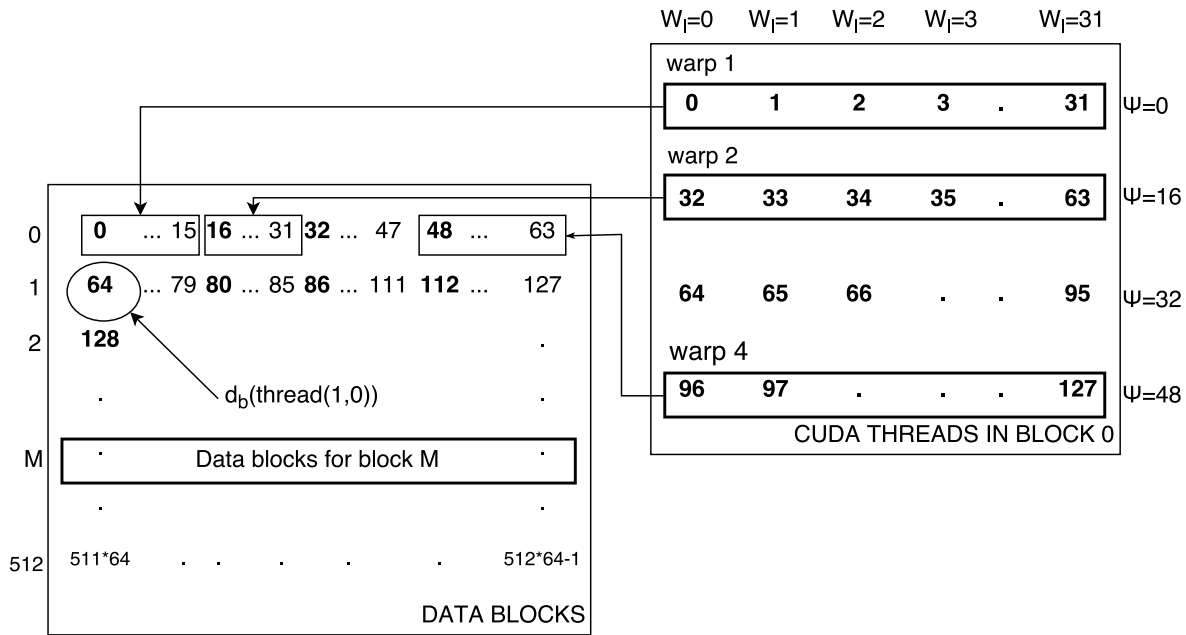


$$B_{count} = \frac{\Sigma + B_{size} * \lambda - 1}{B_{size} * \lambda}$$

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

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

co dla naszego przykładu obrazuje rysunek 2.2.

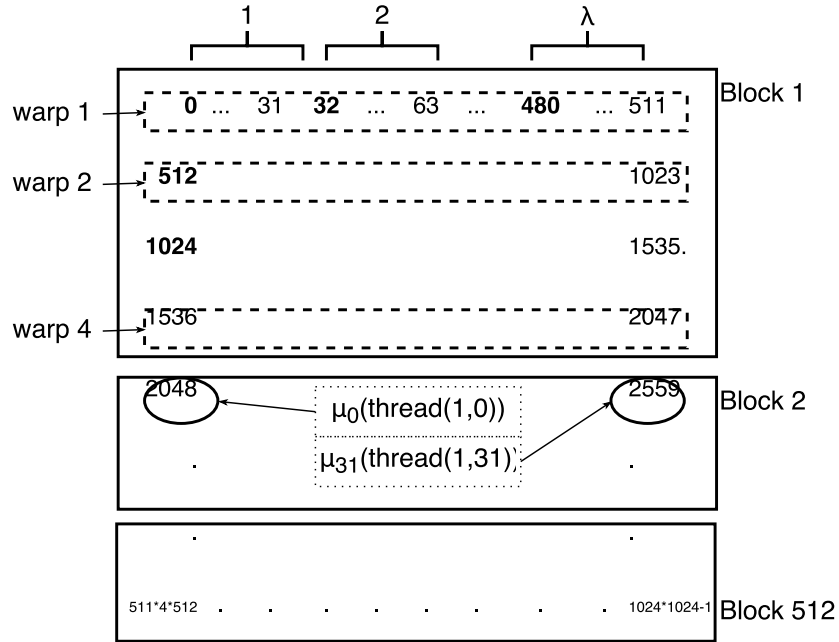


Rysunek 2.2: Przejście z ułożenia wątków w bloku na bloki danych

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

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

Kolejne indeksy wyliczane są jako przesunięcia o  $\omega$ , czyli  $\mu_k(t) = \mu_0(t) + k * \omega$ , dla  $k \in [1, \kappa - 1]$ . Tworzy to podział danych wejściowych, który zgodnie z przykładem obrazuje rysunek 2.3.



Rysunek 2.3: 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  $\omega$  rekord  $\omega$ , następnie po przesunięciu o  $\omega$  wątki ponownie odczytują dane, które są ze sobą sąsiadujące. Dzięki temu odczyt może być połączony (Global Memory Coalescing[43]). Łączny odczyt może być wielokrotnie szybszy, niż odczyty z nie kolejnych indeksów, co jest kluczowe przy budowie bardzo wydajnych algorytmów przetwarzających dane na GPU. W przypadku CUDA grupa powinna mieć licznosc 32, co odpowiada ilości wątków w jednym *warp*.

### 2.1.3 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 kompresje bazujące na pomysłe słownikowym ( $Dict_K$ ). Wykorzystują one informację o liczności poszczególnych wartości w wektorze, tj. które wartości najczęściej się powtarzają. 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** – Zaimplementowane są także lekko 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, zostanie wybrane kodowanie *Const*, które zapisze najczęstszą wartość w metadanych i stworzy tablicę wyjątków, uzyskując bardzo wysoki stopień kompresji.

### AFL

Ten rodzaj kodowania nazywany jest kodowaniem o stałej długości z wyrównaniem. Ogólny pomysł algorytmu jest bardzo prosty i bazuje na algorytmie NS (null sup-

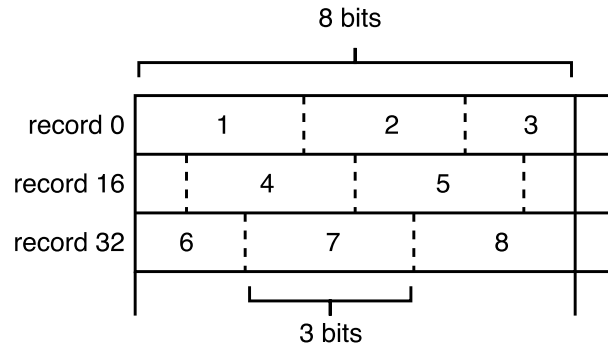
pression), czyli usuwaniu wiodących zer z liczb i zapisywanie ich na mniejszej ilości bitów. W tym przypadku ilość bitów na których zapisujemy liczby jest znana z góry przed rozpoczęciem kodowania i nie ulega zmianie. Dla uproszczenia przyjmijmy, że kompresujemy liczby naturalne o długości 32 bitów, 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. *łącnego 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 wszystkich słów w wektorze, nazwijmy go  $W$  i oznaczmy liczbę bitów jako  $\sigma$ :

$$\sigma = \log_2(\max(W)) + 1$$

2. Następnie wektor elementów jest dopełniany, aby jego długość była wielokrotnością 1024, a liczba dopełnionych elementów wraz z potrzebną do zapisania liczb liczbą bitów  $\sigma$  zapisywana jest do metadanych.
3. Dla każdego wątku  $t$  CUDA ozn.  $t = \text{thread}(b_{idx}, t_{idx})$ , 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  $B_{size} = 32 * 8$  co sprawi, że każdy blok będzie się składał z 8 warpów, a grupy będą miały licznosc 32 ( $\omega = 32$ ,  $w_g = 8$ ). Ponadto chcemy aby bloki danych były wielkości 32 elementów ( $\kappa = 32$ ). Mając dane wejściowe o indeksach  $\mu_0, \mu_1, \dots, \mu_{\kappa-1}$  wątek zapisuje  $\sigma$  dolnych bitów każdej z liczb spod tych indeksów, do rekordów tablicy wynikowej. Wielkości bloków są tak dobrane, aby każdy wątek kodował  $\kappa$  liczb używając  $\sigma$  bitów i otrzymując w ten sposób  $\sigma$  liczb o wielkości  $\kappa$ . 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.



Rysunek 2.4: AFL - zapis w tablicy wyjściowej

Początkowy indeks tablicy wynikowej pod który wątek ma zapisać skompresowane dane, wyliczane są jako  $\nu_0(t) = d_b(t) * \sigma + W_{lane}(t)$ . Do rekordu zapisywanych jest możliwie najwięcej bitów (tak jak pokazuje 2.4), po czym jeśli brakło miejsca dalsze bity przenoszone są do następnego rekordu, którego indeks wyliczany jest jako  $\nu_k(t) = \nu_0(t) + k * \omega$ , 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 około dziesięciokrotnie szybszy od tradycyjnej implementacji Fixed-Length encoding na CUDA, który jest równoważny tej kompresji z  $\omega = 1$ .

## GFC

GFC to zaproponowana przez Burtscher et al.[20] 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 jest wyrównany 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 każdy *warp* musi przetworzyć 32 wartości, aby uzyskać  $\sigma$  liczb wynikowych. W tym algorytmie liczba  $\sigma$  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  $\sigma$  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 wyrażona 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  $t$

---

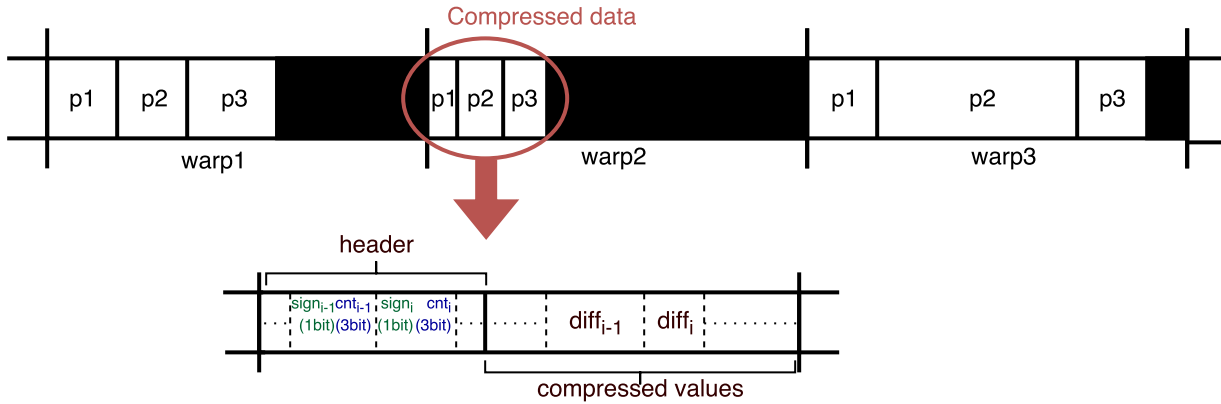
```

/* WEJŚCIE: data - wektor do skompresowania */
/* WYJŚCIE: compr - skompr. dane, offsets - rozm. paczek */
1 last = 0, i = 0;
  while i <  $\lambda$  do
2   diff = data[ $\mu_i(t)$ ] - last;
3   sign = bit znaku liczby diff;
4   diff = abs(diff);
5   minByte =  $\sigma$ (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 warpidx == 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.



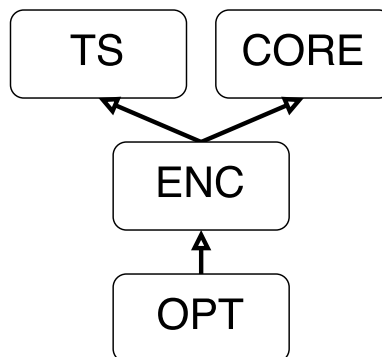
Rysunek 2.5: wynik działania algorytmu GFC - tablica wyjściowa

Kompresja znacząco różni się od AFL pomimo podobnego podziału na bloki danych. 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  $\lambda$ , bo  $B_n$  zależy odwrotnie proporcjonalnie od  $\lambda$ , a im większe  $\lambda$ , tym wolniejszy jest sam algorytm (pojedyncze wątki wykonują 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*.

## 2.2 Biblioteki

Projekt składa się z 4 bibliotek, których zależności wewnętrzne pokazane są na rysunku 2.6. Ponadto biblioteki *CORE* oraz *ENC* wykorzystują technologię CUDA i muszą być kompilowane za pomocą *nvcc*. Wszystkie biblioteki korzystają z biblioteki *Boost* oraz bibliotek do testowania i benchmarków (*GTEST*, *Google Benchmark*). Poniżej znajduje się także opis poszczególnych bibliotek.



Rysunek 2.6: Biblioteki

### 2.2.1 TS

Jest to biblioteka definiująca strukturę szeregu czasowego, oraz 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 2.1.



Listing 2.1: Przykładowy plik opisu szeregu czasowego

```
timestamp , time , 0  
CORE VOLTAGE, float , 6  
CPU TEMP, float , 6  
GPU TEMP, float , 6
```

### 2.2.2 CORE

Tutaj zaimplementowane są wszystkie podstawowe rzeczy 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.

### 2.2.3 ENC

Encodings to biblioteka mieszcząca 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*).

### 2.2.4 OPT

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).

## 2.3 Program

Program wynikowy jest przykładowym programem wynikowym, 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 na końcu rozdziału 3. 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 2.1.
- *-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 jakiejś 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*.

## Rozdział 3

# Optymalizator kompresji

W tym rozdziale opiszę algorytm nazwany roboczo *Online Stat compression Planner*, który będzie uczył się budować jak najlepsze drzewo kompresji na podstawie napływających danych, jednocześnie je kompresując. Zatem wydajność - compression ratio, powinno z czasem rosnać, aż znajdzie się na optymalnym poziomie. Ponadto algorytm ten przeszukując przestrzeń ciekawych drzew kompresji 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 będą dalej nazywał SCBP<sup>1</sup>.

---

<sup>1</sup>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*) i implementują metody kompresji i dekompresji dla wszystkich typów wspieranych przez system (optymalizator). Dla nas najważniejsze są dwie metody:

**Encode** – przyjmuje dane do kompresji jako SCBP oraz typ danych 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, by potem odtworzyć schemat użyty do kompresji, przed metadanymi wyniku dodawany jest header w postaci: [*Typ Kodowania (16 bit)*, *Typ danych (16 bit)*, *Długość metadanych (32 bit)*], nazwijmy go EH<sup>2</sup>. 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

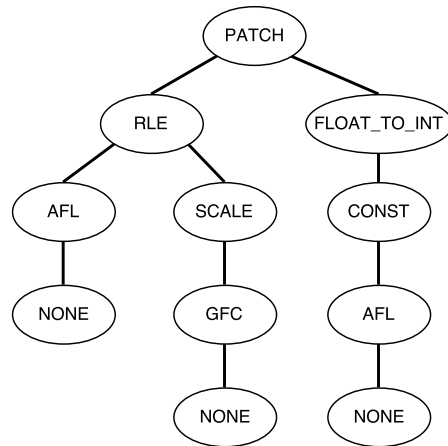
Drzewo kompresji jest implementacją idei kompresji kaskadowej. Węzłowi drzewa odpowiada kompresja danego typu. Węzeł ma tyle dzieci ile wyników zwraca kodowanie które reprezentuje. Dodatkowy typ kodowania został dodany aby oznaczyć liście takiego drzewa. Przykładowe drzewo zostało przedstawione na rysunku 3.1. Drzewo może być także zapisane jako ciąg typów kodowań (który dalej będę nazywał *TreePath*, na przykład pokazane drzewo można przedstawić jako:

*PATCH, RLE, AFL, NONE, SCALE, GFC, NONE, FLOAT\_TO\_INT, CONST, AFL, NONE*

Nie będę tutaj opisywał algorytmu konwersji drzewa z i do takiego opisu, gdyż jest to prosty algorytm rekurencyjny przechodzenia drzewa w głąb. Jednak sam schemat jest o tyle ważny,

---

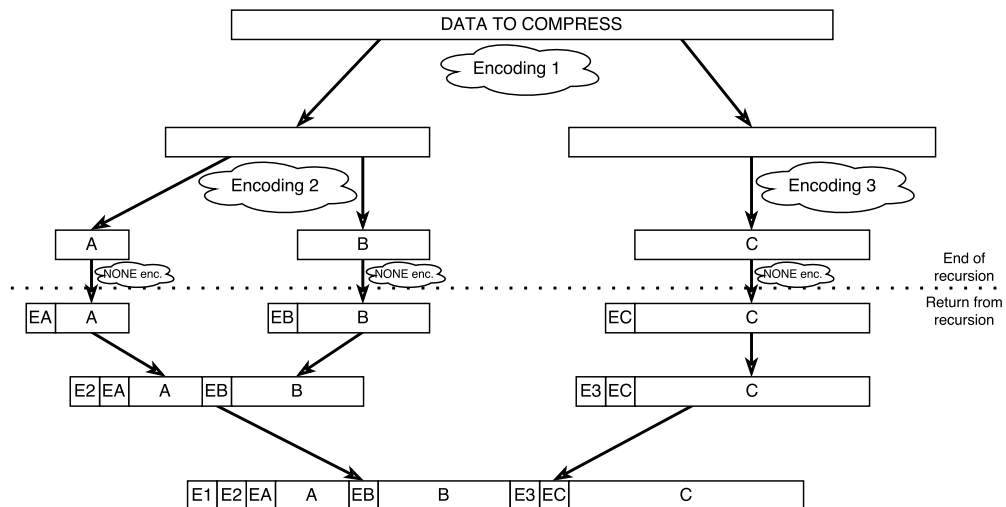
<sup>2</sup>EH - header metadanych wyniku kodowania



Rysunek 3.1: Schemat drzewa kompresji

że w ten sposób będą ułożone fragmenty skompresowanych danych przez algorytm kompresji drzewem.

### 3.2.1 Algorytm kompresji drzewem



Rysunek 3.2: Schemat przebiegu algorytmu kompresji drzewem

Algorytm kompresji drzewem jest rekurencyjny począwszy od korzenia drzewa i za 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   $X \leftarrow$  węzeł na którym wywołano metodę;
2   $K \leftarrow$  pobierz koder typu reprezentowanego przez węzeł  $X$  z fabryki;
3   $COMPR \leftarrow$  zakoduj  $DATA$  używając kodera  $K$ ;
4   $EH \leftarrow$  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          $CHILD\_RES \leftarrow$  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   $ROOT \leftarrow$  pobierz korzeń drzewa;
2   $VEC \leftarrow$  wykonaj metodę  $Compress(DATA)$  na  $ROOT$ ;
   if ustawiony wskaźnik na statystyki then
3     zaktualizuj statystyki; // opisane w 2 fazie optymalizatora
4   $RES \leftarrow$  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 3.2 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 3.3.

#### Rekonstrukcja drzewa

Kolejny algorytm rekurencyjny, 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. Sama przebiega jak pokazuje algorytm 3.

---

#### 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  $\leftarrow$  odczytaj nagłówek z wejścia;
2  | OFFSET + = SIZEOF(EH);
3  | NODE  $\leftarrow$  stwórz węzeł typu wskazywanego przez EH;
4  | MET_SIZE  $\leftarrow$  pobierz rozmiar metadanych z EH;
5  | MET  $\leftarrow$  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  $\leftarrow$  odczytaj z MET rozmiar skompresowanych danych;
9  | | DATA  $\leftarrow$  odczytaj z wejścia SIZE bajtów;
10 | | ustaw dane w węźle NODE na DATA;
11 | | OFFSET + = SIZE;
   | else
12 | | K  $\leftarrow$  pobierz koder o typie wskazywanym przez EH;
13 | | N  $\leftarrow$  pobierz ilość wyników zwracanych przez koder K;
   | | for i=0 to N do
14 | | | CHILD_NODE  $\leftarrow$  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**  $Node \mapsto Decompress()$

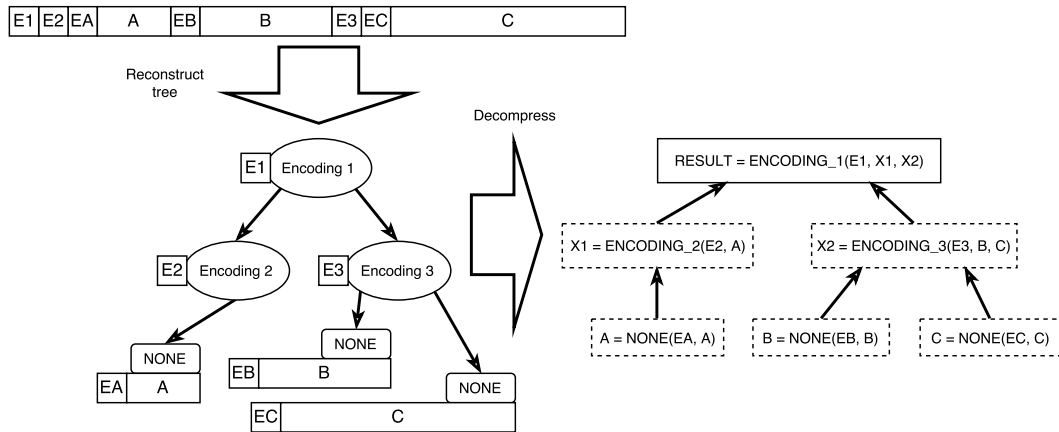
```

1   $X \leftarrow$  węzeł na którym wywołano metodę;
2   $K \leftarrow$  pobierz koder typu reprezentowanego przez węzeł  $X$  z fabryki;
3   $MET \leftarrow$  pobierz metadane zapisane w węźle  $X$ ;
4   $W \leftarrow$  stwórz wektor wskaźników SCBP i dodaj do niego  $MET$ ;
   if  $X$  jest liściem then
5       $DATA \leftarrow$  pobierz dane zapisane w węźle  $X$ ;
6      dodaj  $DATA$  na końcu wektora  $W$ ;
   else
       foreach  $D$  dziecko  $X$  do
7            $CHILD\_RES \leftarrow$  wykonaj metodę  $Decompress()$  na  $X$ ;
8           dodaj  $CHILD\_RES$  na końcu wektora  $W$ ;
9   $RES \leftarrow$  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 3.3: Schemat przebiegu algorytmu dekompresji drzewem



## **3.3 Statystyki**

statystyki

### **3.3.1 Precyzja**

### **3.3.2 Metryka RLE**

## **3.4 Optymalizator**

### **3.4.1 Faza 1 - generowanie**

### **3.4.2 Faza 2 - kompresja**

### **3.4.3 Faza 3 - poprawa**

### **3.4.4 Usprawnienie**

## **3.5 Równoległy kompresor**

### **3.5.1 Opis algorytmu**

Zadanie kompresji

Zadanie dekompresji

## **Rozdział 4**

### **Wyniki**

## **Rozdział 5**

### **Podsumowanie**

# Bibliografia

- [1] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *"Parallel Prefix Sum (Scan) with CUDA"*. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007
- [2] Nadathur Satish, Mark Harris, and Michael Garland. *"Designing Efficient Sorting Algorithms for Manycore GPUs"*. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, May 2009
- [3] Alcantara, Dan A., et al. *"Real-time parallel hashing on the GPU."* ACM Transactions on Graphics (TOG) 28.5 (2009): 154
- [4] Mostak, T., 2013. *"An overview of MapD (massively parallel database)."*, Massachusetts Institute of Technology, Cambridge, MA.
- [5] Cloud RL, Curry ML, Ward HL, Skjellum A, Bangalore P. *"Accelerating lossless data compression with GPUs."* arXiv preprint arXiv:1107.1525. 2011 Jun 21.
- [6] Pietroń, M., Pawel Russek, and Kazimierz Wiatr. *"Accelerating SELECT WHERE and SELECT JOIN queries on a GPU."* Computer Science 14.2 (2013): 243-252.
- [7] Nicolaisen, Anders Lehrmann V?rning. *"Algorithms for Compression on GPUs."* (2013).
- [8] Mensmann, Jörg, Timo Ropinski, and Klaus Hinrichs. *"A GPU-supported lossless compression scheme for rendering time-varying volume data."* 8th IEEE/EG international conference on Volume Graphics, 2–3 May 2010, Norrköping, Sweden. IEEE, 2010.
- [9] Burtscher M, Ratanaworabhan P. *"FPC: A high-speed compressor for double-precision floating-point data."* Computers, IEEE Transactions on. 2009 Jan;58(1):18-31.

- [10] Bakkum, Peter, and Kevin Skadron. *"Accelerating SQL database operations on a GPU with CUDA."* Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. ACM, 2010.
- [11] Ferreira, Miguel C. *"Compression and query execution within column oriented databases."* Diss. Massachusetts Institute of Technology, 2005.
- [12] Fink, Eugene, and Harith Suman Gandhi. *"Compression of time series by extracting major extrema."* Journal of Experimental & Theoretical Artificial Intelligence 23.2 (2011): 255-270.
- [13] Przymus, Piotr, and Krzysztof Kaczmarek. *"Compression Planner for Time Series Database with GPU Support."* Transactions on Large-Scale Data and Knowledge-Centered Systems XV. Springer Berlin Heidelberg, 2014. 36-63.
- [14] Ozsoy, Adnan, Martin Swamy, and Anamika Chauhan. *"Pipelined parallel lzss for streaming data compression on GPGPUs."* Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on. IEEE, 2012.
- [15] Fang, Wenbin, Bingsheng He, and Qiong Luo. *"Database compression on graphics processors."* Proceedings of the VLDB Endowment 3.1-2 (2010): 670-680.
- [16] Lemire, Daniel, and Leonid Boytsov. *"Decoding billions of integers per second through vectorization."* Software: Practice and Experience 45.1 (2015): 1-29.
- [17] Przymus, Piotr, and Krzysztof Kaczmarek. *"Dynamic compression strategy for time series database using GPU."* New Trends in Databases and Information Systems. Springer International Publishing, 2014. 235-244.
- [18] Mani, Ganapathy. *"Data Compression using CUDA programming in GPU."* (2012).
- [19] Alcantara, Dan A., et al. *"Real-time parallel hashing on the GPU."* ACM Transactions on Graphics (TOG) 28.5 (2009): 154.
- [20] O'Neil, Molly A., and Martin Burtscher. *"Floating-point data compression at 75 Gb/s on a GPU."* Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011.

- [21] Zu, Yuan, and Bei Hua. "GLZSS: LZSS lossless data compression can be faster." Proceedings of Workshop on General Purpose Processing Using GPUs. ACM, 2014.
- [22] Silberstein, Mark, et al. "GPUfs: Integrating a file system with GPUs." ACM Transactions on Computer Systems (TOCS) 32.1 (2014): 1.
- [23] Al-Kiswany, Samer, Ammar Gharaibeh, and Matei Ripeanu. "GPUs as storage system accelerators." Parallel and Distributed Systems, IEEE Transactions on 24.8 (2013): 1556-1566.
- [23] Przymus, Piotr, and Krzysztof Kaczmarek. "Improving efficiency of data intensive applications on GPU using lightweight compression." On the Move to Meaningful Internet Systems: OTM 2012 Workshops. Springer Berlin Heidelberg, 2012.
- [24] Abadi, Daniel, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems." Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, 2006.
- [25] Anh, Vo Ngoc, and Alistair Moffat. "Inverted index compression using word-aligned binary codes." Information Retrieval 8.1 (2005): 151-166.
- [26] Eirola, Axel. "Lossless data compression on GPGPU architectures." arXiv preprint arXiv:1109.2348 (2011).
- [27] Shyni, K., and Manoj Kumar KV. "Lossless LZW Data Compression Algorithm on CUDA." IOSR Journal of Computer Engineering (IOSR-JCE) 13.1 (2013): 122-127.
- [28] Mostak, Todd. "An overview of MapD (massively parallel database)." White paper, Massachusetts Institute of Technology, Cambridge, MA (2013).
- [29] Buchsbaum, Adam L., et al. "Engineering the compression of massive tables: an experimental approach." Symposium on Discrete Algorithms: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms. Vol. 9. No. 11. 2000.
- [30] Morishima, Shin, and Hiroki Matsutani. "Performance Evaluations of Document-Oriented Databases Using GPU and Cache Structure." Trustcom/BigDataSE/ISPA, 2015 IEEE. Vol. 3. IEEE, 2015.

- [31] Ozsoy, Adnan, Martin Swamy, and Arun Chauhan. *"Optimizing LZSS compression on GPGPUs."* Future Generation Computer Systems 30 (2014): 170-178.
- [32] Patel, Ritesh A., et al. *"Parallel lossless data compression on the GPU."* IEEE, 2012.
- [33] Polychroniou, Orestis, Arun Raghavan, and Kenneth A. Ross. *"Rethinking SIMD vectorization for in-memory databases."* Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.
- [34] Bhatotia, Pramod, Rodrigo Rodrigues, and Akshat Verma. *"Shredder: GPU-accelerated incremental storage and computation."* FAST. 2012.
- [35] Zukowski, Marcin, et al. *"Super-scalar RAM-CPU cache compression."* Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on. IEEE, 2006.
- [36] Eichinger, Frank, et al. *"A time-series compression technique and its application to the smart grid."* The VLDB Journal 24.2 (2015): 193-218.
- [37] Przymus, Piotr, and Krzysztof Kaczmarek. *"Time series queries processing with gpu support."* New Trends in Databases and Information Systems. Springer International Publishing, 2014. 53-60.
- [38] Zhao, Wayne Xin, et al. *"A General SIMD-based Approach to Accelerating Compression Algorithms."* ACM Transactions on Information Systems (TOIS) 33.3 (2015): 15.
- [39] Goldstein, Jonathan, Raghu Ramakrishnan, and Uri Shaft. *"Compressing relations and indexes."* Data Engineering, 1998. Proceedings., 14th International Conference on. IEEE, 1998.
- [41] <http://developer.download.nvidia.com/compute/cuda/compute-docs/cuda-performance-report.pdf>
- [42] Marler, R. Timothy, and Jasbir S. Arora. *"Survey of multi-objective optimization methods for engineering."* Structural and multidisciplinary optimization 26.6 (2004): 369-395.
- [43] Ueng, Sain-Zee, et al. *"CUDA-lite: Reducing GPU programming complexity."* Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, 2008. 1-15.

- [44] Burtscher, Martin, and Paruj Ratanaworabhan. "*pFPC: A parallel compressor for floating-point data.*" Data Compression Conference, 2009. DCC'09.. IEEE, 2009.



# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Procesory graficzne . . . . .	2
1.2	Szeregi czasowe . . . . .	3
1.3	SIMD i lekka kompresja . . . . .	4
1.4	Zawartość pracy . . . . .	5
1.5	Powiązane prace . . . . .	5
1.5.1	Szeregi czasowe . . . . .	5
1.5.2	SIMD SSE . . . . .	6
1.5.3	Obliczenia GPU . . . . .	6
1.5.4	Kompresje . . . . .	7
1.5.5	Planery kompresji . . . . .	8
1.6	CUDA . . . . .	9
<b>2</b>	<b>Opis systemu</b>	<b>11</b>
2.1	Kodowania . . . . .	11
2.1.1	Podstawowe algorytmy transformacji szeregów . . . . .	11
2.1.2	Global memory coalescing . . . . .	13
2.1.3	Algorytmy kompresji szeregów . . . . .	16
2.2	Biblioteki . . . . .	21
2.2.1	TS . . . . .	21
2.2.2	CORE . . . . .	22
2.2.3	ENC . . . . .	22
2.2.4	OPT . . . . .	22
2.3	Program . . . . .	23

<i>SPIS TREŚCI</i>	39
<b>3 Optymalizator kompresji</b>	<b>24</b>
3.1 Kodowanie . . . . .	25
3.2 Drzewo kompresji . . . . .	25
3.2.1 Algorytm kompresji drzewem . . . . .	26
3.2.2 Algorytm dekompresji drzewem . . . . .	28
3.3 Statystyki . . . . .	30
3.3.1 Precyzja . . . . .	30
3.3.2 Metryka RLE . . . . .	30
3.4 Optymalizator . . . . .	30
3.4.1 Faza 1 - generowanie . . . . .	30
3.4.2 Faza 2 - kompresja . . . . .	30
3.4.3 Faza 3 - poprawa . . . . .	30
3.4.4 Usprawnienie . . . . .	30
3.5 Równoległy kompresor . . . . .	30
3.5.1 Opis algorytmu . . . . .	30
<b>4 Wyniki</b>	<b>31</b>
<b>5 Podsumowanie</b>	<b>32</b>

Warszawa, dnia .....

## Oświadczenie

Oświadczam, że pracę magisterską pod tytułem: „Tytuł pracy”, której promotorem jest prof. dr hab. Jan Wybitny, wykonałem/am samodzielnie, co poświadczam własnoręcznym podpisem.

.....