Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

RÓWNOLEGŁE ODKRYWANIE REGUŁ ASOCJACYJNYH ZAIMPLEMENTOWANE NA PROCESORY GRAFICZNE

inż. Tomasz Kujawa

Promotor
dr inż. Witold Andrzejwski

Poznań, 2011

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wstęp

Proces informatyzacji przedsiębiorstw, rozpoczęty kilka dekad temu, wprowadził światową gospodarkę na nowe, dotąd nieznane tory rozwoju. Skrócenie procesu produkcyjnego, wprowadzenie kontroli komputerowych, czy też skomputeryzowanych maszyn skróciło i ułatwiło produkcję, a także zarządzanie procesami w firmach i przedsiębiorstwach. Przed ludźmi stanęły możliwości, ale także wyzwania, z którymi nigdy wcześniej nikt nie musiał sobie radzić. Zmiany, jakie nastąpiły przez ostatnie trzy dekady są nieodwracalne i zmuszają programistów do tworzenia nowych aplikacji, które będą w stanie sprostać stawianym im wymaganiom.

Informatyzacja firm, instytucji oraz innych jednostek organizacyjnych powinna realizować dwa podstawowe cele. Z jednej strony powinna ona usprawniać pracę pojedynczego pracownika poprzez automatyzację realizowanych przez niego rutynowych zadań. Dzięki wykorzystaniu możliwości komputerów działania te powinny być wykonywane szybciej i w sposób bardziej niezawodny. Z drugiej strony celem informatyzacji jest wpływanie na działanie całych firm w wyniku wspomagania decyzji kadry zarządzającej przedsiębiorstwami. Szybka analiza bazująca na pełnej i aktualnej informacji o stanie firmy może ułatwić kadrze zarządzającej podejmowanie trafnych i szybkich decyzji o strategicznym znaczeniu dla rozwoju danego przedsiębiorstwa.

Wprowadzenie komputerów do właściwie każdej przestrzeni ludzkiego życia wpłynęło na wyprodukowanie olbrzymich ilości danych. Reprezentowane są one w sposób umożliwiający ich składowanie i przetwarzanie komputerowe przez aplikacje analityczne. W chwili obecnej ludzkość jest świadkiem eksplozji w produkcji danych produkowanych przez różnego rodzaju systemy komputerowe. Analiza tych danych przynieść może wymierne korzyści nie tylko w kwestiach finansowych, ale również poznawczych. Dzięki analizie zebranych w przeszłości informacji możliwe jest lepsze dopasowanie planów w przyszłości - na tej podstawie planowane mogą być np. akcje marketingowe, czy też promocje w supermarketach spożywczych. Wykorzystanie wiedzy uzyskanej w ten sposób jest niezwykle szerokie i może być użyte w każdym obszarze działalności firmy.

Odkrycie zależności pomiędzy zgromadzonymi danymi bez zastosowania narzędzi informatycznych jest procesem bardzo skomplikowanym i wymagającym do realizacji dużo czasu. Przy obecnej złożoności większości systemów oraz rozmiarom danych produkowanych przez te systemy, koszt czasowy jest na tyle duży, że ręczna analiza tych danych stała się niemożliwa. Dlatego też tworzone są narzędzia umożliwiające odkrywanie prawidłowości w dużych zbiorach danych, by człowiek na tej podstawie mógł podejmować decyzje i wyciągać wnioski.

Dział informatyki, który zajmuje się odkrywaniem ukrytych dla człowieka prawidłowości i reguł w danych nazywa się eksploracją danych (ang. *Data Mining*, w literaturze spotkać można również określenie drążenie danych, ekstrakcja danych, pozyskiwanie wiedzy, czy też wydobywanie danych [EN05]), który jest jednym z etapów procesu *odkrywania wiedzy z baz danych* (*KDD*,

ang. *Knowledge Discovery in Databases*). Proces odkrywania wiedzy w bazach danych obejmują zwykle działania bardziej złożone niż tylko eksploracja danych. Są to między innymi selekcja danych, transformacja lub kodowanie danych, czy też raportowanie i prezentowanie odkrytych informacji [EN05]. Eksploracja danych to proces odkrywania wiedzy w postaci nowych, użytecznych, poprawnych i zrozumiałych wzorców w bardzo dużych wolumenach danych [FPSSU96]. Możliwości stosowania technik eksploracji danych w praktyce, wymagają efektywnych metod przeszukiwania ogromnych plików lub baz danych. Warto przy tym wspomnieć, że tego typu technologie nie są w chwili obecnej dobrze zintegrowane z systemami zarządzania bazami danych.

Eksploracja danych odbywa się najczęściej w środowisku baz lub hurtowni danych, które stanowią doskonałe źródła danych do analizy - głównie ze względu na łatwość dostępu oraz usystematyzowaną strukturę przechowywanych informacji. Ponieważ liczba odkrytych wzorców w wielu przypadkach może być bardzo duża, odkryte wzorce bardzo często zapisuje się w osobnych relacjach bazy lub hurtowni danych. Pozwala to na ich dalsze przetwarzanie w trybie off-line przez użytkowników końcowych. Pojęcie eksploracji zyskuje coraz większą popularność (również w wymiarze marketingowym) i jest wykorzystywane w wielu dziedzinach ludzkiego życia.

Jednym z najczęściej wykorzystywanych modeli wiedzy w eksploracji danych są reguły asocjacyjne. Reguła asocjacyjna ma postać $X \Rightarrow Y$, gdzie X oraz Y są wzajemnie rozłącznymi zbiorami elementów. Przykładem reguły, która mogła zostać odkryta w bazie danych sklepu komputerowego, może być reguła postaci $komputer \wedge myszka \Rightarrow monitor$. Prezentuje ona fakt, że klienci kupujący komputer oraz myszke z dużym prawdopodobieństwem kupią również monitor. W [AIS93] po raz pierwszy sformułowany został problem odkrywania reguł asocjacyjnych wraz z algorytmem Apriori, który jest podstawą wielu algorytmów znajdujących reguły asocjacyjne. Algorytm ten został następnie rozszerzony w pracy [Agr94].

W ostatnich latach pojawiły się nowe możliwości wykorzystania współczesnych komputerów. W roku 2007 firma NVIDIA udostępniła programistom uniwersalną architekturę obliczeniową CUDA (ang. *Compute Unified Device Architecture*), który umożliwia wykorzystanie mocy obliczeniowej *procesorów graficznych* (*GPU*, ang. *Graphics Processing Unit*), bądź innych procesorów wielordzeniowych, do rozwiązywania ogólnych problemów obliczeniowych w sposób znacząco wydajniejszy niż w przypadku tradycyjnych, sekwencyjnych procesorów [Cor07a]. Choć w grach komputerowych moc obliczeniową jednostek graficznych można wykorzystać do obliczeń fizyki, to CUDA idzie jeszcze dalej, umożliwiając przyspieszenie obliczeń w takich dziedzinach, jak biologia, fizyka, kryptografia, bioinformatyka oraz innych naukach. Specjalnie dla potrzeb tego segmentu NVidia opracowała kartę graficzną o nazwie *Tesla* [Cor07b]. Układy te są pierwszymi układami produkowanymi na masową skalę, które przeznaczone zostały do pracy *obliczeniach ogólnego przeznaczenia na układach GPU* (*GPGPU*, ang. *General-Purpose Computing on Graphics Processing Units*), czyli segmencie do tej pory zarezerwowanym dla klasycznych procesorów obliczeniowych.

Do tej pory bardzo małe jest zainteresowanie wykorzystaniem tej technologii w procesie odkrywania wiedzy, a w szczególności znajdowania reguł asocjacyjnych. Wyniki przeprowadzonych eksperymentów pozwalają przypuszczać, że algorytm wykorzystujący możliwości procesorów wielordzeniowych, a w szczególności GPU, będzie wyraźnie szybszy od klasycznych algorytmów eksploracji danych zaimplementowany na tradycyjnych procesorach.

1.1 Cel i zakres pracy

Celem pracy jest zaprojektowanie i zaimplementowanie algorytmu odkrywającego reguły asocjacyjne, który będzie wykorzystywał możliwośći współczesnych kart graficznych dzięki wykorzystaniu technologii CUDA oraz porównanie zaprojektowanego i zaimplementowanego algorytmu do

innych, podstawowych algorytmów odkrywania reguł asocjacyjnych. W ramach pracy dokonane zostanie również zebranie wiedzy dotyczącej algorytmów eksploracji reguł asocjacyjnych.

Rozdział 1 - wstęp.. Tutaj dalszy opis struktury pracy - zrobiony na koniec, gdy wszystko dalej będzie już znane.

Rozdział 2

Podstawy teoretyczne

W rozdziale tym przedstawiony zostanie przegląd literatury, który stanowi podstawy wiedzy na temat eksploracji danych, a w szczególności problemu odkrywania reguł asocjacyjnych w dużych zbiorach danych. Zebrana wiedza posłużyła autorowi do opracowania algorytmu wykorzystującego możliwości współczesnych kart graficznych.

2.1 Definicje

2.1.1 Model teoretyczny

Niech $I = \{i_1, i_2, ..., i_m\}$ będzie *zbiorem elementów* o liczności $|I| = m$. *Transakcją* T nazwano dowolny, niepusty podzbiór $X \subseteq I$ zbioru elementów. Bazą danych DB nazwano dowolny zbiór par (id, X), gdzie X jest transakcją, a id jest dowolną wartością unikalną w ramach bazy danych nazywaną *identyfikatorem transakcji*. Bez utraty ogólności założono iż $id \in \mathbb{N}$.

W tabeli 2.1 zaperezentowany został przykładowy zbiór elementów oraz zestaw transakcji. Na podstawie tych danych obliczane będą wartości wprowadzanych kolejno definicji.

Wsparciem (ang. *support*) $sup(X)$ transakcji X, w bazie danych nazwano częstość wystąpień transakcji w bazie danych. Formalnie przedstawia to wzór 2.1.

$$sup(X) = \frac{|\{id : (id, Y) \in DB \wedge X \subseteq Y\}|}{|DB|} \tag{2.1}$$

Łatwo zauważyć, że jeśli poziom ten jest niski, to oznacza to, że elementy zbioru X w transakcjach rzadko występują razem.

Dla przykładowego zbioru $X \subseteq I = \{\text{milk}\}$ wartość $sup(X)$ została obliczona w przykładzie 2.2, ponieważ zbiór X jest podzbiorem dwóch transakcji - 1 = { bread, milk, butter, beer } oraz 4 = {

I	{ beer, bread, butter, diapers, jam, juice, milk, water }
Transakcje	1: { bread, milk, butter, beer } 2: { bread, butter, water, jam, beer } 3: { beer, diapers, bread, butter, jam } 4: { butter, milk, juice } 5: { diapers, beer, juice, water }

Tablica 2.1: Przykładowe dane

butter, milk, juice }.

$$sup(X) = sup(\{\text{milk}\}) = \frac{|\{1,4\}|}{|DB|} = \\ = \frac{2}{5} \tag{2.2}$$

Definicja 1. Niech będą dane dwie transakcje X i Y takie, że $X \cap Y = \emptyset$ oraz $Y \neq \emptyset$. *Regułą asocjacyjną* R nazwano implikację postaci $X \Rightarrow Y$.

Poziom ufności (ang. *confidence*) jest miarą określającą jakość reguły asocjacyjnej [EN05].

Definicja 2. Poziom ufności ($conf$) reguły asocjacyjnej $R: X \Rightarrow Y$ jest równy

$$conf(X \Rightarrow Y) = \frac{sup(X \cup Y)}{sup(X)} \tag{2.3}$$

Z definicji 2 wynika, że poziom ufności może być interpretowany, jako estymacja prawdopodobieństwa w transakcji zbioru Y pod warunkiem wystąpienia w niej również zbioru X - co oznaczone jest poprzez $P(Y|X)$. Formalnie zapisane jest to za pomocą wzoru 2.4 [Mue95].

$$conf(X \Rightarrow Y) = p(Y \subseteq T | X \subseteq T) = \frac{p(Y \subseteq T \wedge X \subseteq T)}{p(X \subseteq T)} = \frac{sup(X \cup Y)}{sup(X)} \tag{2.4}$$

Reguły asocjacyjne zazwyczaj powinny spełniać pewne wymagania zdefiniowane przez użytkownika - minimalne wsparcie oraz minimalny poziom ufności, oznaczane odpowiednio $minsup$ oraz $minconf$. Wyznaczają one dla aplikacji progi, jakie powinny spełniać zbiory oraz reguły, aby były brane pod uwagę w trakcie analizy. Motywacją za tymi minimalnymi wartościami jest fakt, by do analizy brane były tylko te zbiory, które pojawiają się w DB wystarczającą liczbę razy.

Przykładową regułą asocjacyjną znalezioną dla danych przedstawionych w tabeli 2.1 może być reguła {butter} $\Rightarrow$ {bread, beer}, która zostałaby znaleziona dla $minsup = 0,6$. Taka reguła posiada współczynnik pewności równy $\frac{3}{4} = 75\%$, obliczony w przykładzie 2.5.

$$\begin{aligned} conf(X \Rightarrow Y) &= conf(\{\text{butter}\} \Rightarrow \{\text{bread, beer}\}) = \\ &= \frac{sup(\{\text{butter}\} \cup \{\text{bread, beer}\})}{sup(\{\text{butter}\})} = \\ &= \frac{sup(\{\text{butter, bread, beer}\})}{sup(\{\text{butter}\})} = \\ &= \frac{3}{4} = 75\% \end{aligned} \tag{2.5}$$

Definicja 3. Zbiorem częstym $X \subseteq I$ nazywamy taki zbiór, który spełnia zależność $sup(X) \geq minsup$.

Generowanie reguł asocjacyjnych zazwyczaj sprowadza się do dwóch, niezależnych kroków:

1. Minimalne wsparcie jest używane do odnalezienia wszystkich zbiorów częstych w bazie danych DB.

2. Znalezione zbiory często oraz minimalny poziom ufności są używane do wygenerownia reguł asocjacyjnych.

W naiwnym podejściu znalezienie wszystkich zbiorów częstych w bazie danych DB jest zadaniem wymagającym przeszukania wszystkich możliwych kombinacji bez powtórzeń ze zbioru I.

Zbiór X nazwywamy k-zbiorem, jeśli $|X| = k$, tzn. zbior X ma k elementów. Zbiór możliwych zbiorów elementów ma liczność równą $2^n - 1$ (wszystkie zbiory, poza zbiorem pustym, który nie jest w tym wypadku zbiorem sensownym w znaczeniu poddania go analizie), czyli zbiór ten jest $(2^n - 1)$-zbiorem.

Warto zauważyć, że dla każdego zbioru częstego Y, każdy jego podzbiór X jest również zbiorem częstym [AIS93]. Korzystając z tej właściwości wsparcia możliwe jest w sposób efektywny znalezienie wszystkich zbiorów częstych w zadanej bazie danych - z tej zależności korzysta algorytm apriori opisany w rozdziale 2.2.1. Dodatkowo, wszystkie reguły zbudowane na podstawie zbioru częstego Y muszą spełniać warunek minimalnego wsparcia, ponieważ spełnia ten warunek zbiór Y, a suma zbiorów reguły jest zbiorem wyjściowym Y.

2.2 Aktualna wiedza

W rozdziale tym zebrana została oraz opracowana dotychczasowa wiedza (ang. *state-of-the-art*) na temat algorytmów odkrywania reguł asocjacyjnych. Przedstawione zostaną dwa podstawowe algorytmy wykorzystywane w tym procesie: Apriori oraz FP-growth. W chwili obecnej te dwa algorytmy stanowią podstawę, na której budowane są nowe algorytmy, wykorzystujące możliwości współczesnych algorytmów - bazujących na FPGrowth: [LQ05, ZY08] lub na algorytmie Apriori: [PCY95, YC06].

2.2.1 Algorytm Apriori

Pierwszy algorytm odkrywający reguły asocjacyjne został przedstawiony w roku 1994 w pracy [AS94], który jest rozszerzeniem algorytmu zaprezentowanego przez autorów w [AIS93]. Niżej przedstawione zostaną szczegóły działania tego algorytmu nazwanego algorytmem *Apriori*.

Zagadnienie odkrywania reguł asocjacyjnych można podzielić na dwa etapy [AIS93]:

1. Odkrywanie zbiorów częstych, których wartość wsparcia jest wyższa od wartości *minsup*.

2. Generowanie reguł asocjacyjnych na podstawie znalezionych zbiorów częstych. Reguła $X \Rightarrow Y$ jest wynikiem działania algorytmu dla zbioru $Z = X \cup Y$, jeżeli spełnia ona nierówność $conf(X \Rightarrow Y) \geq minconf$. Ponieważ zbiór $Z = X \cup Y$ jest zbiorem częstym, to reguła spełnia również warunek przekraczania minimalnego wsparcia.

   Na tym etapie możliwe jest tworzenie reguł, w których w zbiorze *poprzedników* (X z oznaczeń z definicji 1) jest wiele elementów oraz jeden w *następniku* (zbiór Y z definicji 1) [AIS93] lub dopuszczana jest możliwość wielu elementów również w następniku [AS94]. W niniejszej pracy analizowany jest sposób generowania reguł, w którym oba zbiory mogą być zbiorami wieloelementowymi.

W kolejnych podrozdziałach przedstawione zostaną etapy tworzące razem algorytmy Apriori.

Generowanie zbiorów częstych

W celu wyznaczenia zbiorów częstych algorytm dokonuje analizy bazy danych DB, by w kolejnych iteracjach generować rodziny coraz to liczniejszych zbiorów, będących zbiorami częstymi dla zadanej wartości *minsup*. Algorytm zaczyna od znalezienia wszystkich zbiorów jednoelementowych, które są zbiorami częstymi. W każdym kolejnym kroku generowane są zbiory częste na podstawie zbiorów wygenerowanych w kroku poprzednim. Proces ten jest kontynuowany do momentu aż nie zostaną znalezione żadne zbiory częste.

L_k	Zbiór zawierający k-zbiory. Każdy zbiór zawarty w L_k zawiera dwa pola: i) zbiór oraz ii) wartość *support*.
C_k	Zbiór k-zbiorów kandydatów (potencjalnych zbiorów częstych). Każdy zbiór zawarty w C_k zawiera dwa pola: i) zbiór oraz ii) wartość *support*.

Tablica 2.2: Oznaczenie w opisach algorytmów

Algorytm generuje zbiory kandydatów jedynie na podstawie zbiorów częstych odkrytych w kroku poprzednim - co ważne generowanie ich odbywa się bez wielokrotnego przeglądania bazy danych transakcji. Intuicja podpowiada, że każdy podzbiór zbioru częstego jest zbiorem częstym. Zatem, każdy zbiór częsty zawierający k elementów może być wygenerowany na podstawie połączenia dwóch zbiorów posiadających $k-1$ elementów, a na koniec kasując te zbiory, których jakikolwiek podzbiór nie jest częsty [AS94].

Tabela 2.2 zawiera spis oznaczeń używanych w opisie algorytmu.

Procedura Apriori Frequent Set Generaion przedstawia pseudokod realizujący opisywany w tym rozdziale algorytm generowania zbiorów częstych.

Apriori Frequent Set Generaion

```
1  L_1 ← {1-zbiory częste }
2  for (k = 2; L_{k-1} ≠ ∅; k++)
3       do C_k ← aprioriGen(L_{k-1})
4          for each trasakcja t ∈ DB
5               do C_t ← subset(C_k, t)
6                  for each kandydat c ∈ C_t
7                       do c.count++
8          L_k ← {c ∈ C_k | c.count ≥ minsup}
9  Answer ← ⋃_k L_k
```

Procedura aprioriGen Procedura *aprioriGen* reprezentuje proces twórzenia zbiorów $k-$ elementowych kandydatów na podstawie zbiorów wejściowych $(k-1)$-elementowych. Procedura ta jest podzielona na dwa etapy: łączenia oraz przycinania.

Jak łatwo zauważyć wynikiem działania Join Step są zbiory k-elementowe, które powstały na podstawie zbiorów wejściowych L_{k-1}, a ich zawartość różni się tylko jednym elementem - ostatnim. Ważnym faktem jest to, iż elementy w zbiorach są uporządkowane leksykograficzne, co wykorzystywane jest w tej procedurze.

Join Step

```
1  insert into C_k
2  select p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}
3  from L_{k-1} p, L_{k-1} q
4  where p.item_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}
```

Warto zauważyć, że Join Step jest ekwiwalentem rozszerania zbioru L_{k-1} każdym elementem zbioru elementów I, a następnie kasowania tych $(k-1)$-zbiorów otrzymanych przez usuwanie $(k-1)$ elementu, które nie są w L_{k-1}.

Warunek $\text{p.item}_{k-1} < \text{q.item}_{k-1}$ zapewnia, że nie będą generowane duplikaty. Dlatego też po etapie łączenia zachodzi zależność $C_k \supseteq L_k$.

Następnym krokiem jest PRUNE STEP, w którym usuwane są wszystkie elementy $c \in C_k$, którego jakikolwiek podzbiór $(k-1)$-elementowy zbioru c nie należy do L_{k-1}.

PRUNE STEP

```
1  for each zbiór c ∈ C_k
2        do
3            for each (k − 1)-podzbiór s zbioru c
4                do if s ∉ L_{k−1}
5                       then delete c z C_k
```

Celem operacji przycinania (ang. *prune*) jest ograniczenie rozmiaru zbioru C_k przed sprawdzeniem wsparcia dla kandydatów w bazie danych DB. W tym celu wykorzystywana jest właściwość, z której wynika, że jeśli jakiś $(k-1)$-podzbiór danego kandydata ($c \in C_k$) nie występuje w L_{k-1}, to kandydatk c nie jest zbiorem częstym i powinien być usunięty z C_k.

Przykład działania procedury aprioriGen Niech zbiór L_3 będzie równy $\{\{1,2,3\},\{1,2,4\},\{1,3,4\},\{1,3,5\},\{2,3,4\}\}$. Po działaniu procedury JOIN STEP wartość zbioru C_4 będzie zatem równa $\{\{1,2,3,4\},\{1,3,4,5\}\}$. W następnym kroku, czyli wewnątrz procedury PRUNE STEP usunięty zostanie z C_4 element $\{1,3,4,5\}$ (patrz linia 5), ponieważ element $\{1,4,5\}$ nie należy do L_3 (patrz linia 4). Dlatego też $C_4 = \{\{1,2,3,4\}\}$.

Generowanie reguł asocjacyjnych

Po zakończeniu pierwszego etapu algorytm przystępuje do drugiego, czyli do budowania reguł asocjacyjnych na podstawie odkrytych zbiorów. Podobnie, jak w [AS94] algorytm będący przedmiotem analizy niniejszej pracy, generuje wszystkie możliwe reguły asocjacyjne dla zadanego zbioru. Mniej ogólny sposób generowania reguł został przedstawiony w pracy [AIS93], jednakże podjęto decyzję, że jest to sposób zbyt mało użyteczny w środowisku produkcyjnym.

Aby wygenerować reguły, dla każdego zbioru częstego l znajdowane są niepuste podzbiory - podzbiór taki oznaczony jest jako a. Dla takich oznaczeń wygenerowna zostanie reguła $a \Rightarrow (l-a)$, jeżeli spełniona jest nierówność $\frac{support(l)}{support(a)} \geq minconf$. Warto zauważyć, że dla każdego zbioru częstego generowane są wszystkie możliwe niepuste podzbiory - zapewnia to, że odkryte zostaną wszystkie możliwe reguły.

Istnieje możliwość poprawienia tego podejścia poprzez generowanie podzbiorów dużego zbioru w sposób rekurencyjnego wywołania *najpierw w głąb* (ang. *depth-first fashion*). Na przykład, dla zbioru $\{A,B,C,D\}$, najpierw rozważany jest podzbiór $\{A,B,C\}$, potem $\{A,B\}$ itd. Wtedy, jeśli podzbiór a zbioru l nie jest źródłem reguły asocjacyjnj, podzbiór a nie musi być rozważany w generowaniu reguł ze zbioru l. Dla przykładu, jeśli reguła $\{A,B,C\} \Rightarrow \{D\}$ nie posiada wystarczającego współczynnika $conf$, wówczas nie ma potrzeby sprawdzać reguły $\{A,B\} \Rightarrow \{C,D\}$. Nie jest pominięta żadna możliwa reguła, ponieważ wartość *sup* dowolnego $\tilde{a} \in a$ nie może być większe niż wartość *sup* dla zbioru a. Dlatego też, wartość $conf$ dla reguły $\tilde{a} \Rightarrow (l-\tilde{a})$ nie może być większa niż poziom ufności $a \Rightarrow (l-a)$. Stąd, jeżeli a nie zwrócił reguły zawierającej wszystkich elementów z l w a, jako poprzednika, nie dokona tego również $\tilde{a}$ [**?**].

Procedura GENERATE FREQUENT ITEMSETS prezentuje generowanie reguł asocjacyjnych na podstawie odkrytych k-zbiorów częstych l_k będących elementami zbioru L_k ($l_k \in L_k$).

GENERATE FREQUENT ITEMSETS
1 **for each** zbiór częsty l_k, $k \geq 2$
2 **do call** genrules(l_k, l_k)

W powyższym algorytmie wykorzystana została funkcja GENRULES, która na podstawie dwóch zbiorów generuje reguły asocjacyjne. Zapis pseudokodu tej funkcji przedstawiony jest poniżej.

GENRULES(l_k: k-zbiór częsty, a_m: m-zbiór częsty)
1 $A \leftarrow \{(m-1)$-zbiór $a_{m-1} | a_{m-1} \subset a_m\}$
2 **for** $a_{m-1} \in A$
3 **do** $conf \leftarrow \frac{support(l_k)}{support(a_{m-1})}$
4 **if** $conf \geq minconf$
5 **then output** reguła $a_{m-1} \Rightarrow (l_k - a_{m-1})$
ufność = $conf$ oraz wsparcie= $support(l_k)$
6 **if** $m - 1 > 1$
7 **then call** genrules(l_k, a_{m-1})
generowanie reguł podzbiorów zbioru a_{m-1}

2.2.2 Algorytm FP-growth

Podstawową wadą algorytmu Apriori jest wysoki koszt przetważania dużych zbiorów danych. Przykładowo, dla 10^4 1-zbiorów częstych, algorytm Apriori wygeneruje około 10^7 2-zbiorów kandydatów, które następnie poddane zostaną weryfikacji, czy są zbiorami częstymi. Poza tym algorytm ten wymaga wielokrotnego odczytywania zawartości bazy danych - w każdym kroku algorytmu należy odczytać całą bazę danych w celu obliczenia wsparcia zbiorów kandydujących.

Wymienione wyżej wady algorytmu Apriori nie występują w algorytmie *FP-growth* przedstawionym w pełni w pracy [HPYM04]. Algorytm ten pozwala wyeliminować konieczność generowania tak dużej liczby kandydujących zbiorów elementów oraz ogranicza liczbę dostępu do bazy danych do absolutnego minimum. Co więcej algorytm ten charakteryzuje się kompletnością, co oznacza, że znajdowane są wszystkie wzorce o określonej częstości.

Algorytm FP-growth można podzielić na trzy podstawowe kroki.

1. W kroku pierwszym generowana jest skompresowana wersja bazy danych DB, mająca postać drzewa częstych wzorców - jako posortowane listy elementów częstych w każdej z transakcji.

2. Drugim krokiem jest transformacja tak skonstruowanego drzewa do postaci *FP-drzewa* (patrz definicja 4).

3. Trzeci krok polega na analizie FP-drzewa celem odnalezienia reguł asocjacyjnych. W kroku tym stosowana jest metoda dziel i zwyciężaj (ang. *divide-and-conquer*) zamiast podejścia Apriori, czyli generowania na każdym poziomie zbioru kandydatów na zbiory częste, a następnie odcinaniu kandydatów nie spełniających kryteriów akceptacji. Takie podejście przekształca problem znajdowania długich reguł w problem szukania krótszych, a następnie konkatenacji wyników.

Przykład budowy FP-drzewa

Niech zbiór transakcji DB będzie zdefiniowany przez tablicę 2.3. Dla przykładu przyjęto, że $minsup = 3$.

TID	Lista elementów	Posortowana lista elementów częstych
100	f, a, c, d, g, i, m, p	f, c, a m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Tablica 2.3: Przykładowy zbiór transakcji

1. Ponieważ w budowie FP-drzewa (patrz definicja 4) będą uczestniczyc jedynie elementy będące elementami częstymi, to pierwszym etapem algorytmu jest jednokrotne przejrzenie zawartości DB oraz zidentyfikowanie zbioru elementów częstych (z wartością f_count dla każdego elementu, przechowujący liczbę jego wystąpień).

2. Jeśli zbiory elementów częstych, dla każdej z transakcji, mogą być przechowywane w relatywnie małej strukturze - warto jest zachować wynik skanowania, by uniknąć w kolejnych etapach kolejnych, niepotrzebnych i wyraźnie obniżających wydajność algorytmu, skanów bazy danych.

3. Jeśli kilka transakcji posiada te same zbiory elementów częstych, możliwe jest przechowywanie tych zbiorów jako jednego, z licznikiem w ilu transakcjach on występuje (*count*). Czy dwa zbiory są identyczne można w łatwy sposób zweryfikować, jeśli w każdym ze zbiorów są posortowane zgodnie z przyjętym dla wszystkich porządkiem.

4. Możliwe jest również częściowe współzapamiętywanie zbiorów, jeśli prefiksy (czyli części zbiorów posortowanych) są identyczne dla dwóch transakcji. Te same części mogą być scalone do jednego prefiksu, jeśli wartość *count* jest właściwie obliczona. Jeśli elementy częste są posortowane w porządku malejącej liczby wystąpień, istnieje większe prawdopodobieństwo znalezienia większej liczby wspólnych prefiksów.

Dla danych zawartych w tabeli 2.3 przedstawione zostanie teraz działanie algorytmu.

Na początek, skan bazy danych DB tworzy *listę elementów częstych*: $\langle f:4, c:4, a:3, b:3, m:3, p:3\rangle$ (gdzie para $a:0\ldots n$ reprezentuje odpowienio - element a oraz liczbę jego wystąpień), w której elementy posortowane są zgodnie z malejącą liczbą wystąpień. Sposób sortowania jest istotny, ponieważ proces budowy drzewa będzie właśnie oparty na tym porządku. Dla wygody analizy oraz łatwiejszego zrozumiania, pierwszą kolumnę z prawej w tablicy 2.3 tworzą elementy częste z danej transakcji, posortowane właśnie w ten sposób.

Co więcej, korzeń drzewa jest tworzony i oznaczony jako *null*, czyli wartość pusta. FP-drzewo jest tworzone poprzez skanowanie raz jeszcze bazy danych DB i działania opisane przez kolejne kroki poniżej.

1. Skan pierwszej transakcji ($TID = 100$) prowadzi do wprowadzenia w drzewie pierwszej gałęzi: $\langle f:1, c:1, a:1, m:1, p:1\rangle$. Warto zauważyć, że elementy częste z danej transakcji są wylistowane zgodnie z porządkiem w *liście elementów częstych*.

2. Analizując drugą transakcję ($TID = 200$), dla której lista elementów częstych $\{f, c, a, b, m\}$ współdzieli ten sam prefiks $\{f, c, a\}$ z istniejąca już ścieżką $\{f, c, a, m, p\}$, dlatego też liczba wystąpień na tej ścieżce inkrementowana jest o 1, a także tworzony jest nowy element $b:1$ oraz dodawany jako potomek elementu $a:2$, a następny nowy element $m:1$ jest dodawany, jako potomek elementu $b:1$.

3. Trzecia transakcja ($TID = 300$) $\{f, b\}$ współdzieli jedynie pierwszy element $\{f\}$ z f-poddrzewem prefiksowym, wartość *count* dla elementu $f : 2$ jest inkrementowana o 1, a nowy element $b : 1$ jest dodawany, jako bezpośredni potomek elementu $f : 3$.

4. Skan czwartej transakcji ($TID = 400$) prowadzi do utworzenia nowej gałęzi drzewa $\langle c : 1, b : 1, p : 1\rangle$, ponieważ transakcja ta nie współdzieli żadnego fragmentu z istniejącą już ścieżką w FP-drzewie.

5. Ostatnia transakcja ($TID = 500$) posiada identyczną listę elementów częstych, jak transakcja pierwsza ($TID = 100$), dlatego też elementy na tej ścieżce są inkrementowane odpowiednio o 1.

By ułatwić *przechodzenie drzewa* (ang. *tree traversal*) stworzona została również tablica nagłówkowa, w której każdy element posiada wskaźnik do elementu drzewa. Wierzchołki reprezentujące ten sam element, są oprócz tego połączone bezpośrednio między sobą. Po przeskanowaniu wszystkich transakcji, drzewo razem z połączeniami pomiędzy wierzchołkami zostało zaprezentowane na rysunku 2.1.

Definicja 4. FP-drzewo (ang. *frequent-pattern tree*) jest to ukorzeniony, etykietowany w wierzchołkach graf acykliczny spełniający poniższe cechy.

1. Korzeniem drzewa jest jeden element *null*, zbiór poddrzew prefiksowanych elementami (jako dzieci elementu *null*) oraz tablicy nagłówkowej zawierającej wpisy *element* $\rightarrow$ *wskaźnik na element drzewa*.

2. Każdy wierzchołek poddrzewa składa się z trzech elementów: nazwy elementu (ang. *item name*), licznika (ang. *count*) oraz wskaźnika na inny wierzchołek. Nazwa elementu (*itemName*) w sposób jednoznaczny identyfikuje element ze zbioru elementów I, licznik przechowuje liczbę transakcji reprezentowanych przez ścieżkę od *null* do tego elementu, natomiast wskaźnik wskazuje na kolejny wierzchołek w FP-drzewie, którego nazwa jest identyczna do danego.

3. Każdy wpis w *tablicy nagłówkowej* (ang. *frequent-item-header table*) skałada się z dwóch elementów: nazwy elementu oraz wskaźnika na pierwszy element w drzewie posiadający identyczną nazwę.

Rysunek 2.1 przedstawia przykładowe FP-drzewo wraz z tablicą nagłówkową reprezentujący dane z przykładu wprowadzonego i analizowanego wcześniej.

Kompresja bazy danych

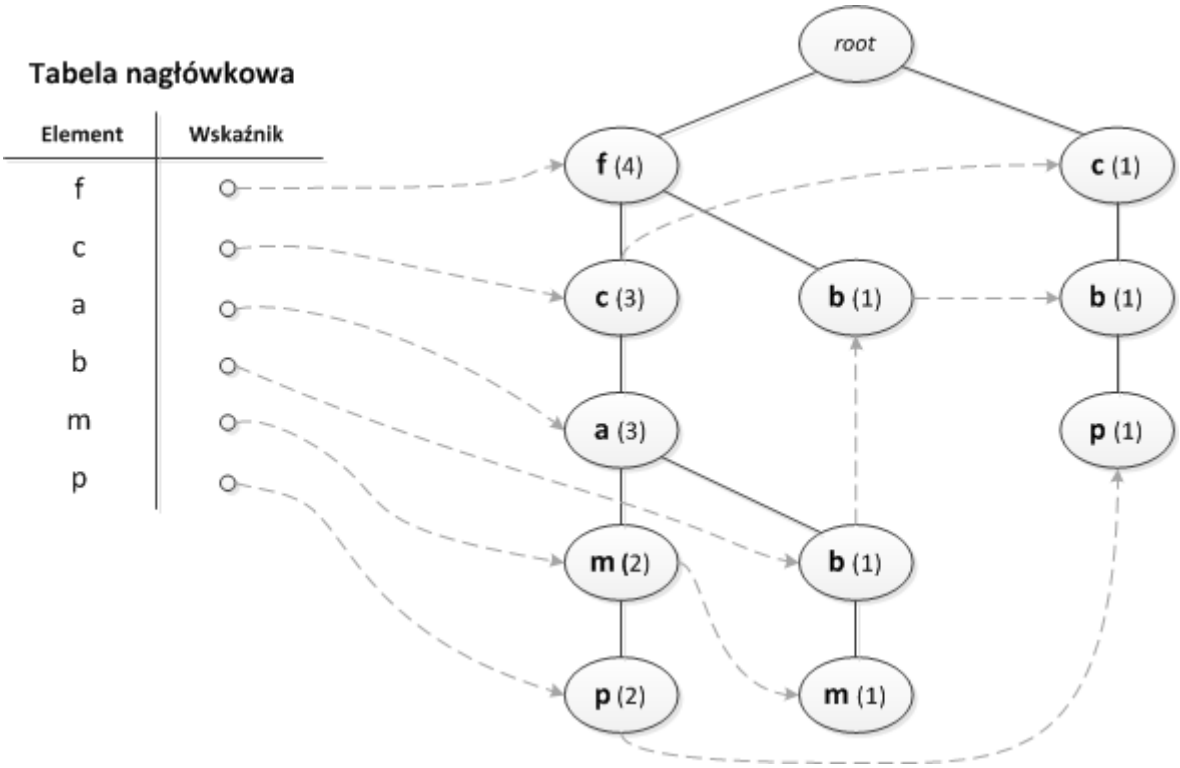
Pierwszy etap polega na znalezieniu wszystkich 1-zbiorów częstych występujących w bazie danych DB. Po ich odnalezieniu (F_1 - zbiór znalezionych 1-zbiorów częstych) z każdej transakcji T usuwany jest ten element, który nie należy do F_1.

W wyniku usunięcia elementów nie tworzących jednoelementowych zbiorów częstych, baza ma zazwyczaj znacznie mniejszy rozmiar niż wyjściowa baza danych. Dodatkowo w tym kroku elementy w każdej transakcji zostają posortowane według malejącej wartości ich wsparcia.

Konstrukcja FP-drzewa

Algorytm 1. *Konstrukcja FP-drzewa.*

Input: Baza danych transakcji DB oraz minimalne wsparcie (*minsup*).



Rysunek 2.1: Ilustracja przykładowego FP-drzewa dla zbioru danych z tablicy 2.3

Output: FP-drzewo utworzone na podstawie zawartości DB
Metoda: Poniżej zostanie opisany proces konstrukcji FP-drzewa.

1. Przeskanowanie bazy danych transakcji DB odbywa się jednokrotnie. Utworzony na tej podstawie zostanie zbiór F, zawierający 1-zbiory częste. Posortowany malejąco zbiór F na podstawie wartości *support* dla każdego elementu tworzy listę $FList$, czyli listę wszystkich elementów tworzących jednoelementowe zbiory częste.

2. Tworzony jest pierwszy element drzewa - korzeniem zostaje (zgodnie z definicją 4) element z etykietą *null*. Dla każdej transakcji w bazie danych DB wykonywane jest, co następuje.

   Wybierane są elementy częste z transakcji, a następnie sortowane zgodnie z kolejnością w $FList$. Niech taka posortowana lista elementów częstych ma postać $[p|P]$, gdzie p jest pierwszym elementem, a P jest pozostałą częścią listy. Następnie wywoływana jest funkcja $insertTree([p|P], T)$, gdzie T jest FP-drzewem.

   Funkcja insertTree Jeśli drzewo T ma dziecko N takie, że $N.itemName = p.itemName$, zwiększana jest wartość $N.count$ o wartość 1; w przeciwnym wypadku tworzony jest nowe element N z wartością $count = 1$, a wskaźnik rodzica ustawiany jest na T oraz wskaźnik sąsiedzctwa elementu ustawiany jest na element z takim samym $itemName$. Jeśli lista P była niepusta, to wywoływana jest funkcja $insertTree(P, T)$ rekurencyjnie, w przeciwnym wypadku kończone jest działanie funkcji.

Eksploracja FP-drzewa

Po utworzeniu FP-drzewa przeprowadzana jest jego analiza w celu znalezienia wszystkich zbiorów częstych. Eksploracja bazuje na obserwacji, że dla każdego 1-zbioru częstego α wszystkie częste

nadzbiory tego zbioru są reprezentowane w FP-drzewie przez ścieżki zawierające wierzchołek (bądź wierzchołki) α.

Analiza rozpoczyna się od znalezienia dla każdego 1-zbioru częstego α wszystkich ścieżek w FP-drzewie, których końcowym wierzchołkiem jest wierzchołek odpowiadający zbiorowi α. Pojedyncza ścieżka, na której końcu znajduje się wierzchołek α w dalszej analizie będzie nazywana *ścieżką prefiksową wzorca* α.

Poniżej zaprezentowany zostanie pseudokod algorytmu przeszukiwania FP-drzewa celem odnalezienia reguł asocjacyjnych.

Algorytm 2. *FP-growth: Przeszukiwanie FP-drzewa celem odnalezienia reguł asocjacyjnych.*

Input: Baza danych transakcji DB reprezentowana przez FP-drzewo zwrócone przez algorytm 1 oraz minimalne wsparcie (*minsup*).

Output: Kompletny zbiór reguł asocjacyjnych.

Metoda: Wywołanie FP-GROWTH($FP - drzewo, null$).

FP-GROWTH(*Tree*, α)

```
1  if Tree zawiera jedną ścieżkę prefiksową
2     then P ← ścieżka prefiksowa drzewa Tree
3          Q ← wieloczęściowa ścieżka z najwyższym elementem zastąpionym przez korzeń null
4          for each kombinacja (β) elementów z P
5               do generuj regułę β ∪ α z support = minimalna wartość support elementów w β
6                  niech freqPatternSet(P) będzie zbiorem wygenerowanych do tej pory reguł
7     else Q ← Tree
8          for each element a_i ∈ Q
9               do generuj regułę β ← a_i ∪ α z support = a_i.support
10                 stwórz warunkową bazę wzorców z β oraz FP-drzewo (Tree_β) dla β
11                 if Tree_β ≠ ∅
12                    then call FP-GROWTH(Tree_β, β)
13                 niech freqPatternSet(Q) będzie zbiorem wygenerowanych do tej pory reguł
14 return freqPatternSet(P) ∪ freqPatternSet(Q) ∪ (freqPatternSet(P) × freqPatternSet(Q))
```

Przykład eksploracji FP-drzewa

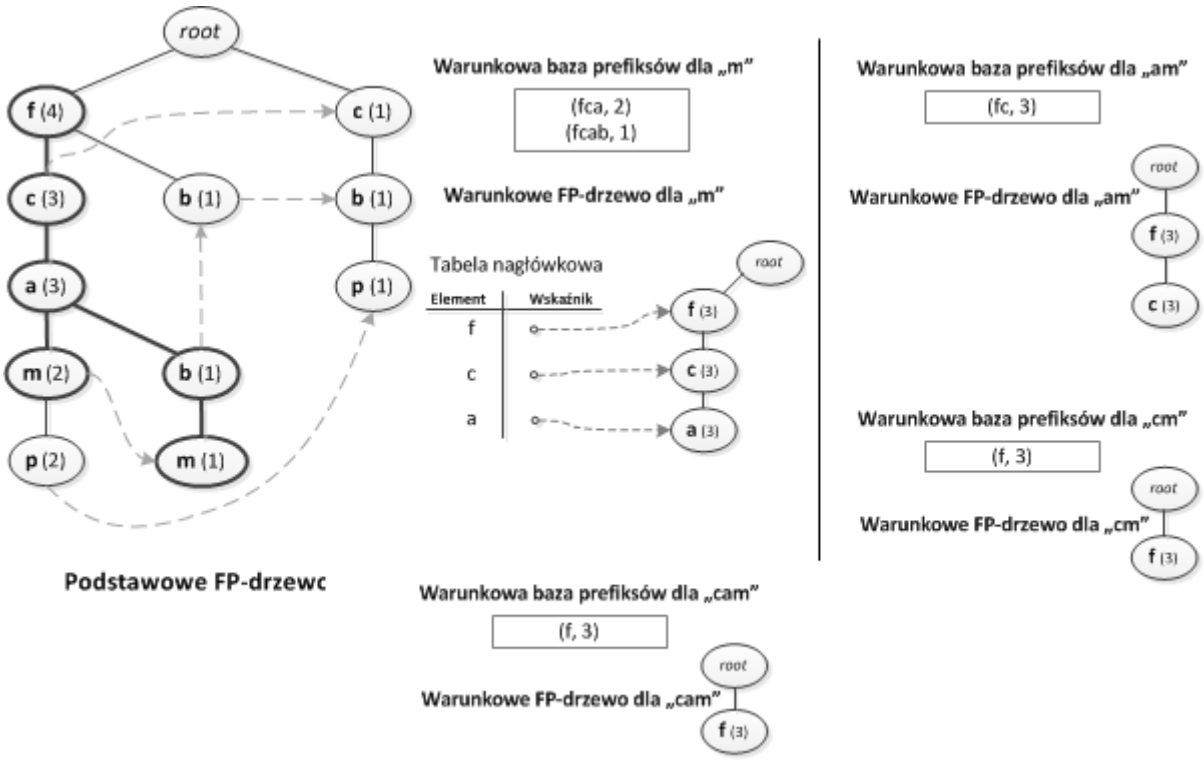
W rozdziale pt. *Przykład budowy FP-drzewa* zaprezentowano sposób budowy FP-drzewa. Opierając się na przykładzie wprowadzonym w tablicy 2.3 przedstawiony zostanie przykład działania eksploracji zbudowanego na tej podstawie FP-drzewa, które zostało zaprezentowane na rysunku 2.1.

Przykład należy rozpocząć od spostrzeżenia, że wszystkie reguły asocjacyjne, zawierające element częsty a_i mogą być zebrane poprzez wyjście ze wskaźnika z tablicy nagłówkowej FP-drzewa, a następnie podążając zgodnie z kolejnymi wskaźnikami na wystąpienia danego elementu w drzewie. W przykładzie analiza zostanie przeprowadzona od końca - tj. podążając od elementu najmniej licznego spośród elementów częstych.

Dla elementu p regułą częstą jest $p : 3$, która posiada dwie ścieżki w drzewie: $\langle f : 4, c : 3, a : 3, m : 2, p : 2\rangle$ oraz $\langle c : 1, b : 1, p : 1\rangle$. Pierwsza ścieżka oznacza, że wartość "(f, c, a, m, p)" występuje w bazie danych DB dwukrotnie. Warto zauważyć, że pierwsza ścieżka oznacza również, że ciąg "(f, c, a)" występuje w bazie dwukrotnie, a "(f)" nawet czterokrotnie. Jednakże występują one jednocześnie z p jedynie dwukrotnie. Dlatego też, do analizy wartości występujących razem z p tylko ścieżka prefiksowana $\langle f : 2, c : 2, a : 2, m : 2\rangle$ (lub zapisane krócej: $\langle fcam : 2\rangle$) jest brana pod

uwagę. Podobnie w przypadku drugiej ścieżki - wynika z niej, że ciąg "(c, b, p)" występuje tylko raz w transakcjach w bazie danych DB, lub też prefiksowana ścieżka dla p ma postać $\langle cb : 1\rangle$. Te dwie ścieżki prefiksowane elementu p (czyli $\{(fcam : 2), (cb : 1)\}$) tworzą bazę wzorca elementu p (ang. *subpattern-base*), która jest nazywana *warunkową bazę wzorców* z p (ang. *conditional pattern base*) - to znaczy bazę wzorca pod warunkiem występowania elementu p [HPYM04]. Konstrukcja FP-drzewa na tej warunkowej bazie wzorców (które nazywane jest *warunkowym FP-drzewem* elementu p ang. *conditional FP-tree*) prowadzi tylko do jednej gałęzi - $(c : 3)$. Stąd też tylko jeden wzorzeć jest dziedziczony - $(cp : 4)$. (Warto zauważyć, że wzorzec jest zbiorem elementów, który jest reprezentowany przez ciąg.) Poszukiwanie częstych wzorców związanych z elementem p kończy działanie.

Dla wierzchołka m jego bezpośrednim wzorcem częstym jest $(m : 3)$, posiada on również dwie ścieżki: $\langle f : 4, c : 3, a : 3, m : 2\rangle$ oraz $\langle f : 4, c : 3, a : 3, b : 1, m : 1\rangle$. W obu tych ścieżkach można zauważyć, że m występuje razem z p, jednakże p w tym miejscu nie musi być brane pod uwagę, ponieważ wszystkie wzorce częste dla tego elementu zostały znalezione wcześniej (opisano to akapit wcześniej). Wykonując działania podobnie, jak w poprzednim akapicie wyznaczona zostanie warunkowa baza wzorców o postaci $\{(fca : 2), (fcab : 1)\}$. Skonstruowane na tej podstawie FP-drzewo, czyli warunkowe FP-drzewo elementu m ($\langle f : 3, c : 3, a : 3\rangle$) posiada jedną ścieżkę częstą, co można zaobserwować na rysunku 2.2. Następnie przeprowadzona jest rekurencyjna eksploracja tego warunkowego drzewa poprzez wywołanie $mine(\langle f : 3, c : 3, a : 3\rangle|m)$.



Rysunek 2.2: Eksploracja warunkowego FP-drzewa dla elementu m (FP-tree $|m$) [HPYM04]

Rysunek 2.2 ilustruje, że wywołanie $mine(\langle f : 3, c : 3, a : 3\rangle|m)$ poddaje przetwarzaniu trzy elementy: $(a), (c), (f)$ sekwencyjnie. Przetwarzanie pierwszego tworzy częsty wzorzec o postaci $(am : 3)$, warunkową bazę wzorców $\{(fc : 3)\}$, a następnie wywoływana jest funkcja $mine(\langle f : 3, c : 3\rangle|am)$; kolejny tworzy częsty wzorzec o postaci $(cm : 3)$, warunkową bazę wzorców $\{(f : 3)\}$, a następnie wywoływana jest funkcja $mine(\langle f : 3\rangle|cm)$; trzecie natomiast tworzy jedynie częsty wzo-

Element	Warunkowa baza wzorców	Warunkowe FP-drzewo
p	$\{(fcam:2),(cb:1)\}$	$\{(c:3)\}\|p$
m	$\{(fca:2),(fcab:1)\}$	$\{(f:3,c:3,a:3)\}\|m$
b	$\{(fca:2),(f:1),(c:1)\}$	$\emptyset$
a	$\{(fc:3)\}$	$\{(f:3,c:3)\}\|a$
c	$\{(f:3)\}$	$\{(f:3)\}\|c$
f	$\emptyset$	$\emptyset$

Tablica 2.4: Wyniki eksploracji FP-drzewa dla przykładu opartego na danych w tablicy 2.3

rzec $(fm:3)$. Dalsze wywołanie metody $mine(\langle f:3,c:3\rangle|am)$ tworzy dwa wzorce $(cam:3)$ oraz $(fam:3)$, warunkową bazę wzorców $\{(f:3)\}$, która prowadzi do wywołania $mine(\langle f:3\rangle|cam)$, czego wynikiem jest najdłuższy wzorzec częsty $(fcam:3)$. Podobnie, wywołanie $mine(\langle f:3\rangle|cm)$ tworzy jednen wzorzec $(fcm:3)$.

Z przeprowadzonej analizy wynika, że zbiór wszystkich wzorców częstych zawierający m ma postać $\{(m:3),(am:3),(cm:3),(fm:3),(cam:3),(fam:3),(fcam:3),(fcm:3)\}$. Wynika z tego, że jedna ścieżka w FP-drzewie może być przeanalizowana poprzez wygenerowanie wszystkich kombinacji (bez powtórzeń) elementów na tej ścieżce.

Podobnie, b tworzy $(b:3)$, posiada również trzy ścieżki: $\langle f:4,c:3,a:3,b:1\rangle$, $\langle f:4,b:1\rangle$ oraz $\langle c:1,b:1\rangle$. Ponieważ warunkowa baza wzorca tego elementu $\{(fca:1),(f:1),(c:1)\}$ nie generuje żadnych elementów częstych, eksploracja po elemencie b kończy działanie.

Element a tworzy jeden element częsty $(a:3)$ oraz bazę wzorca $\{(fc:3)\}$, czyli FP-drzewo posiadające jedną ścieżkę. Wynika z tego, że zbiór częstych wzorców może być wygenerowany poprzez wygenerowanie wszystkich ich kombinacji. Dokonując konkatenacji z $(a:3)$, otrzymany zostanie zbiór $\{(fa:3),(ca:3),(fca:3)\}$. Wierzchołek c generuje $(c:4)$, a także bazę wzorca $\{(f:3)\}$, a zbiór częstych wzorców skojarzonych z $(c:3)$ ma zawartość $\{(fc:3)\}$. Analiza elementu f tworzy jedynie $(f:4)$, lecz bez warunkowych baz wzorców.

W tablicy 2.4 zebrane zostały baze wzorców oraz warunkowe FP-drzewa dla poszczególnych elementów.

Rozdział 3

Uniwersalna architektura procesorów wielordzeniowych

Rozdział ten stanowi wprowadzenie to zagadnienia obliczeń wielordzeniowych. Przedstawiona została w nim pokrótce historia rozwoju oprogramowania oraz sprzętu komputerowego, który podążał za co raz to wyższymi wymaganiami stawianymi przez użytkowników. Opisana zostanie również historia rozwoju *procesora* (*CPU*, ang. *Central Processing Unit*) - głównej jednostki obliczeniowej w praktycznie każdym urządzeniu elektronicznym. Później opisany zostanie rozwój karty graficznej oraz zmiana jej zastosowania na przestrzeni lat.

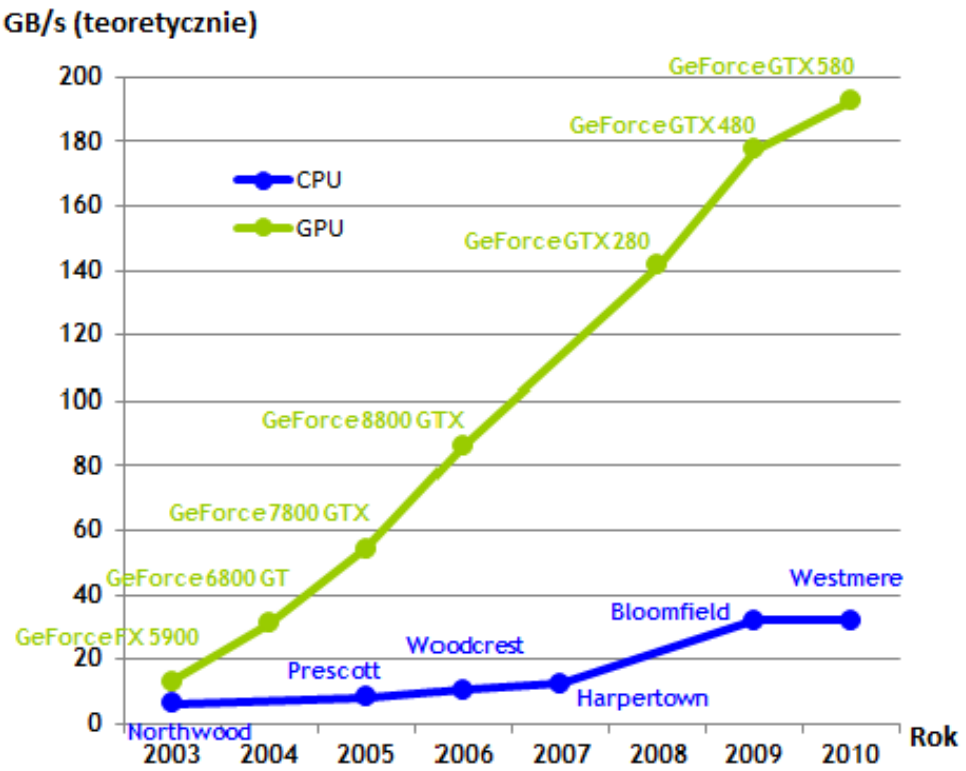
Ostatnim, lecz nie najmniej ważnym elementem tego rozdziału jest opis uniwersalnej architektury obliczeniowej CUDA (ang. *Compute Unified Device Architecture*) wprowadzonej przez firmę nVidia w roku 2007, która umożliwia wykorzystanie mocy obliczeniowej *procesorów graficznych* (*GPU*, ang. *Graphics Processing Unit*), bądź innych procesorów wielordzeniowych, do rozwiązywania ogólnych problemów obliczeniowych w sposób znacząco wydajniejszy niż w przypadku tradycyjnych, sekwencyjnych procesorów [Cor07a].

Na rysunku 3.1 przedstawiony został przyrost w przepustowościach pamięci - odpowiednio dla GPU oraz CPU na przestrzeni lat. Łatwo zauważyć, że w ciągu 7 lat procesor graficzny zyskał około 10-krotną przewagę nad CPU. Dlatego też co raz większe jest zainteresowanie wykorzystaniem kart graficznych w dziedzinach innych niż tylko renderowanie grafiki.

3.1 Czasy przetwarzania równoległego

W poprzednich latach dokonał się znaczący postęp w przechodzeniu przemysłu komputerowego na obliczenia wykonywane równolegle. W roku 2010 większość komputerów konsumenckich była dostarczana do odbiorcy z procesorem zawierającym więcej niż jeden *rdzeń* (ang. *core*). Począwszy od procesorów dwurdzeniowych w laptopach do 8- czy 16-rdzeniowych stacji roboczych - od pewnego czasu obliczenia równoległe nie są już tylko domeną superkomputerów lub mainframe'ów (ang. *main* – główny, *frame* – struktura). Co więcej, urządzenia elektroniczne takie jak telefony komórkowe czy też przenośne odtwarzacze muzyki wyposażane są w procesory wielordzeniowe, co zapewnia im możliwości dalece przekraczające te dostępne dla ich poprzedników.

Wynika z tego, że coraz to więcej programistów będzie musiało radzić sobie z implementacją oprogramowania przeznaczonego na jednostki równoległe, wykorzystywać nowe technologie, które będę pozwalały na dostarczenie nowatorskich rozwiązań dla co raz bardziej wymagającej rzeszy użytkowników. Wiersze poleceń to przeżytek - od dawna komputerem steruje się za pomocą skomplikowanych interfejsów graficznych. To samo tyczy się telefonów komórkowych - w chwili



Rysunek 3.1: Przepustowość pamięci na CPU oraz GPU [Cor11a]

obecnej telefon to tylko jedna z wielu funkcji, jakich może dostarczyć współczesny aparat telefoniczny. Teraz telefony mogą jednocześnie grać muzykę, dostarczać informacji o obecnej lokalizacji przy użyciu modułu nawigacji satelitarnej (*GPS*, ang. *Global Positioning System*) i jednocześnie wyświetlać zdjęcia.

3.1.1 Procesor

Przez około 30 lat jedną z ważniejszych metod udoskonalania centralnych jednostek obliczeniowych, a przez to zwiększania komfortu korzystania z urządzenia przez użytkownika, było zwiększanie prędkości z jaką operował zegar procesora. W latach 80 XX wieku procesor przeznaczony na rynek konsumencki operował z prędkością oscylującą w okolicach 1MHz. Około 30 lat później, w czasach współczesnych, większość komputerów osobistych wyposażona jest w procesory o prędkościach od 1 do 4GHz, czyli obecne jednostki są około 1000 szybsze od wczesnych procesorów. Zwiększanie prędkości, z jaką operuje zegar procesora jest niezawodnym źródłem zwiększania szybkości, należy jednak podkreślić, że nie jest to jedyna metoda na zwiększanie jego wydajności.

Jednakże ograniczenia technologiczne wyznaczają pewne granice, w jakich może wzrastać prędkość zegara procesora. Dlatego też poszukuje się innych, równie niezawodnych źródeł zwiększenia wydajności. Nie można już dłużej polegać jedynie na zwiększaniu prędkości. Z powodu restrykcji na mocy oraz wydzielanym cieple oraz docieraniu do granicy rozmiaru tranzystora, naukowcy oraz producenci rozpoczęli poszukiwanie nowych źródeł i sposobów na zwiększenie możliwości procesorów.

Poza światem konsumentów, czyli w świecie tzw. superkomputerów przez dekady osiągano niezwykle wielkie przyrosty mocy w bardzo podobny sposób. Moc procesora używanego w tych komputerach rosła tak samo szybko, jak w przypadku przyrostów procesorów desktopowych. Jednakże,

poza wielkimi przyrostami mocy obliczeniowej na jednej jednostce, producenci superkomputerów tworzyli komputery, w których solidne przyrosty w wydajności osiągano dzięki zwiększaniu liczby używanych procesorów. Nie jest niczym niezwykłym, że pojedynczy superkomputer składa się z dziesiątek lub setek tysięcy procesorów działających równolegle.

W poszukiwaniu dodatkowych możliwości dla komputerów osobistych, poprawa wydajności w przypadku superkomputerów rodzi pytanie: Dlaczego zamiast zwiększać wydajność pojedynczej jednostki, nie umieścić w komputerze osobistym więcej rdzeni? W wypadku zwiększania liczby rdzeni rozwój jednostek obliczeniowych nie byłby ograniczony przez te same niedogodności, co w przypadku ciągłego zwiększania prędkości zegara procesora.

W roku 2005 wiodący producenci prosesorów zaczęli oferować jednostki z dwoma, zamiast z jednym rdzeniem. W latach następnych kontynuowano tę praktykę, tworząc jednostki trzy-, cztero-, sześcio- oraz ośmio-rdzeniowe. Czasami nazywa się ten okres *rewolucją wielordzeniową* [SK10], ponieważ zmiana podejścia do zwiększania wydajności jednostek w znaczący sposób wpłynęła na ewolucję konsumenckiego rynku komputerów.

W chwili obecnej praktycznie każdy komputer osobisty jest wyposażony w procesor dwurdzeniowy. Nawet na rynku niskobudżetowych komputerów z bardzo niskim zapotrzebowaniem na moc, dokonała się rewolucja wielordzeniowa - już nawet netbooki będą wyposażone w dwa rdzenie [Cor10].

3.2 Wpływ procesorów graficznych na procesy obliczeniowe

Wydawać by się mogło, że użycie procesora graficznego, jako jednostki obliczeniowej dla problemów nie związanych bezpośrednio z przetwarzaniem grafiki jest podejściem zupełnie nowym. Porównując to do klasycznych obliczeń na procesorach jest to w istocie konepcja nowa. Jednakże obliczenia na jednostkach graficznych nie są tak nowe, jak mogłoby się wydawać na pierwszy rzut oka.

3.2.1 Krótka historia kart graficznych

W rozdziale 3.1.1 przedstawiony został rozwój procesora w dwóch płaszczyznach - prędkości oraz liczbie rdzeni. W międzyczasie karty graficzne przeżywały rewolucję rozwojową. Na końcu lat 80 oraz początku 90, wzrost popularności *graficznych interfejsów użytkownika* (*GUI*, ang. *Graphical User Interface*), a w szczególności systemów operacyjnych takich jak Microsoft Windows, wymusił na producentach sprzętu stworzenie nowego typu procesora. We wczesnych latach 90 użytkownicy zaczęli kupować karty graficzne 2D dla swoich komputerów osobistych. Te akceleratory grafiki oferowały wsparcie sprzętowe do operacji bitmapowych by umożliwić wykorzystanie graficznego interfejsu użytkownika.

Mniej więcej w tym samym czasie, w świecie profesjonalnych komputerów, firma o nazwie *Silicon Graphics* popularyzowała w latach 80-tych użycie grafiki trójwymiarowej w wielu dziedzinach, m.in. aplikacje rządowe oraz obronne, aplikacje wspierające naukę oraz wizualizację wyników przeprowadzanych badań naukowych, a także w tworzeniu trójwymiarowych efektów filmowych, czyli rzeczy do tej pory niedostępnych na rynku. W roku 1992 Silicon Graphics stworzyło interfejs programistyczny do swojego sprzętu poprzez wydanie biblioteki IRIS GL, która następnie wyewoluowała do OpenGL (ang. *Open Graphics Library*), czyli specyfikacji uniwersalnego API do generowania grafiki. Firma ta wypuszczając tę bibliotekę chciała, by była ona ustandaryzowaną, niezależną od platformy biblioteką do tworzenia aplikacji trójwymiarowych. Zupełnie, jak w pryz-

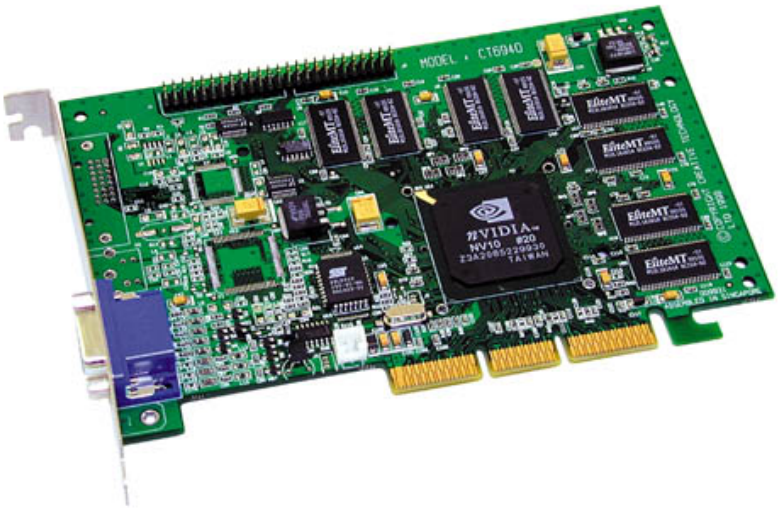
padku licznych rdzeni na procesorach (patrz rozdział 3.1.1) było tylko kwestią czasu, aż aplikacje 3D znajdą się w domach użytkowników na ich prywatnych komputerach.

Rysunek 3.2: Zrzut ekranu z gry DOOM - klasycznej gry z gatunku FPS

Dwa znaczące powody w połowie lat 90-tych spowodowały, że pojawiła się nagła potrzeba stworzenia procesora graficznego potrafiącego tworzyć trójwymiarowe efekty (*3D*, ang. *three-dimensional*). Pierwszym było pojawienie się wciągających gier, w których gracz ogląda świat gry z perspektywy pierwszej osoby, czyli oczami bohatera (*FPP*, ang. *First Person Perspective*), takich jak Doom (patrz rysunek 3.2), Duke Nukem 3D oraz Quake. Przyczyniło się to do rozpoczęcia tworzenia bardziej realistycznych środowisk trójwymiarowych, w których możliwe byłoby tworzenie gier komputerowych. Pomimo tego, że grafika 3D mogła być wykorzystana w praktycznie wszystkich grach komputerowych, to wyjątkowa popularność gier z gatunku FPS (ang. *First Person Shooter*) spowodowała, że większy nacisk położono na prywatne jednostki komputerowe. W tym samym czasie firmy takie jak NVIDIA, ATI Technologies oraz 3dfx Interactive zaczęły produkować akceleratory graficzne, które były w stanie sprostać ciągle rosnącym wymaganiom klientów. Te strategiczne decyzje spowodowały, że grafika trójwymiarowa na dobre zadomowiła się na komputerach stacjonarnych i w nadchodzących latach to ona miała wieść prym wśród technologii komputerowych.

Wypuszczenie w drugiej połowie roku 1999 na rynek karty graficznej NVIDIA GeForce 256 przesunęło możliwości komputerów osobistych jeszcze dalej [Cor99] - wygląd tej karty przedstawiony został na rysunku 3.3. Po raz pierwszy obliczenia świetlne oraz transformacje obliczeniowe możliwe były do wykonania bezpośrednio na procesorze graficznym, co dawało jeszcze większe możliwości tworzenia aplikacji atrakcyjnych wizualnie. Ponieważ operacje te były już integralną częścią *potoku wywołań* (ang. *pipeline*) biblioteki OpenGL, karta GeForce 256 wyznaczyła początek progresywnego zwiększania operacji dostępnych w biliotece OpenGL, które były implementowane na procesorze graficznym.

Uważa się, że wypuszczona w roku 2001 seria kart graficznych GeForce 3 jest jednym z większych przełomów w świecie technologii GPU [Cor01, SK10]. Seria ta była pierwszą w świecie, której chip był kompatybilny z nowym w ówczesnym czasie standardem firmy Microsoft o nazwie DirextX 8.0,



Rysunek 3.3: Karta graficzna NVIDIA GeForce 256

który wymagał by hardware zawierało *programowalne wierzchołki* (ang. *programmable vertex*) oraz *programowalne cieniowanie* (ang. *programmable pixel shading*) w kolejnych fazach przetwarzania. Po raz pierwszy programiści mogli mieć jakąkolwiek kontrolę nad obliczeniami dokonywanymi bezpośrednio na karcie graficznej.

3.2.2 Wczesne obliczenia na kartach graficznych

Wyprodukowanie kart, które posiadały następujące po sobie programowalne fazy (potok wywołań) spowodowały, że wielu naukowców zaczęło wykorzystywać ich możliwości nie tylko poprzez używanie OpenGL czy też DirectX do standardowych zadań. Takie podejście do obliczeń na kartach graficznych we wczesnych latach obliczeń na GPU było niezwykle skomplikowane. Ponieważ standardowe API graficzne takie jak OpenGL, czy też DirectX były jedynymi metodami do interakcji z GPU, to programowanie obliczeń na kartach graficznych nadal sprowadzało się do implementowania przetwarzania graficznego poprzez dotępne metody interfejsu programistycznego karty. Z tego też powodu, wielu naukowców wykonywało swoje obliczenia poprzez wspomniane wcześniej API w taki sposób, by ich problemy sprowadzone zostały do renderowania grafiki, a następnie odpowiedniego przetworzenia.

Zasadniczo każda karta graficzna we wczesnym roku 2000 zaprojektowana była w taki sposób, by produkować kolor dla każdego piksela na ekranie używając programowalnych jednostek arytmetycznych nazywanych *pixel shader*. W ogólności jednostka ta wykorzystuje pozycję (jako parę (x, y)) piksela oraz pewne dodatkowe informacje by obliczyć kolor danego piksela na ekranie. Tymi dodatkowymi informacjami mogą być kolory, wymiary tekstur oraz parametry, które mogą zostać podane, bądź obliczone w trakcie interakcji z użytkownikiem, bądź otoczeniem piksela. Ponieważ obliczenia wykonywane na wejściowych kolorach oraz teksturach były kontrolowane przez programistę, zauważono, że te wejściowe „kolory” mogą w rzeczywistości reprezentować dowolną daną.

Zatem jeśli wejście programu było dowolną daną reprezentowaną jako wartość reprezentującą coś zupełnie innego niż wartość, to programiści mieli możliwość wykorzystywania pixel shaderów, czyli krótkiego programu komputerowego w specjalnym języku, do wykonania na tych danych porządanych obliczeń oraz przekształceń. Rezultatem tych obliczeń był „kolor”, który w da-

nym kontekscie oznaczał coś zupełnie innego niż tylko kolor na ekranie w dosłownym tego słowa znaczeniu. Można więc nazwać to „oszukiwaniem" karty graficznej poprzez poddawanie danych wejściowych przetwarzaniu graficznemu, jakby były to zwyczajne dane potrzebne do wyrenderowania obrazu. Takie podejście odznaczało się wyjątkowym poziomem pomysłowości, ale niestety wykonanie takich obliczeń było wyjątkowo zagmatwane i skomplikowane.

Z powodu wysokiej przepustowości obliczeń arytmetycznych na GPU, początkowe rezultaty takich eksperymentów obiecywały świetlaną przyszłość obliczeń na jednostkach graficznych. Jednakże model programistyczny stosowany do implementacji takich obliczeń był zbyt restrykcyjny dla rzeszy developerów by mógł być wykorzystywany na szeroką skalę. Możliwe było jedynie wykorzysytwanie „kolorów" oraz podmian tekstur, co w wielu przypadkach stanowiło duże ograniczenie przy bardziej skomplikowanych obliczeniach. Było również sporo ograniczeń do do tego jak i gdzie programista mógł zapisywać wyniki do pamięci karty, tak więc algorytmy wykorzystujące rozproszone lokacje nie były możliwe do implementacji na GPU. Co więcej, bardzo trudno było implementować algorytmy korzystające z obliczeń zmiennopozycyjnych (ang. *floating-point*), ponieważ nie można było przewidzieć, jak karta graficzna, jeśli w ogóle, wykona te obliczenia [SK10]. Ostatnim z zasadniczych ograniczeń był fakt braku jakiejkolwiek metody do debugowania kodu uruchamianego na karcie graficznej - na przykład w momencie wyraźnie błędnych obliczeń, zawieszania się komputera czy też w momencie nie zakończenia się programu.

Jeśli wcześniej wymienione ograniczenia nie były przeszkodą, ktokolwieek kto chciał wykorzystać moc obliczeniową karty raficznej do wykonywania obliczeń nie związanych z przetwarzaniem grafiki musiał w dalszym ciągu nauczyć się OpenGL bądź DirectX, gdyż to one pozostawały jedynymi metodami do interakcji z GPU. Nie tylko oznaczało to przechowywanie wyników w teksturach graficznych oraz wykonywania obliczeń poprzez wywoływanie funkcji OpenGL lub DirectX, ale dodatkowo oznaczało to pisanie obliczeń w specjalnych językach programowania graficznego (ang. *shading languages*). Wymaganie by naukowcy dokonywali obliczeń na ograniczonych zasobach oraz specjalnych językach programowania oraz nauczenia się specyfiki jednostek graficznych przed przystąpieniem do wykorzystania możliwości GPU do obliczeń spowodowało, że metoda ta nie przyjęła się, jako wiodąca w świecie naukowców.

3.3 CUDA

Dopiero po około pięciu latach od wydania serii 3 GeForce obliczenia na jednostkach graficznych były gotowe rzeczywiście być implementowane przez programistów z pełnym wykorzystaniem możliwośći kart. W listopadzie 2006 roku NVIDIA wypuściła na rynek pierwszą kartę graficzną wspierającą DirectX 10 - GeForce 8800 GTX [Cor06]. Karta ta była również pierwszą jednostką zbudowaną na architekturze CUDA. Architektura ta zawierałą nowe komponenty zaprojektowane specjalnie pod obliczenia wykonywane na GPU oraz eliminowała wiele ograniczeń, które powstrzymywały programistów przed wykorzystywaniem możliwości karty do obliczeń nie związanych z przetwarzaniem grafiki.

3.3.1 Czym jest architektura CUDA?

W przeciwieństwie do poprzednich generacji kart graficznych, w których jednostki były podzielone na pixel i vertex shader'y, w kartach zgodnych z architektura CUDA zawarta jest zunifikowana linia shaderów (ang. *shader pipeline*). Pozwala to każdej jednostce arytmetyczno-logicznej (ALU, ang. *Arithmetic Logic Unit* lub *Arithmetic and Logical Unit*) na czipie być wykorzystaną przez program wykonujący podstawowe operacje obliczeniowe.

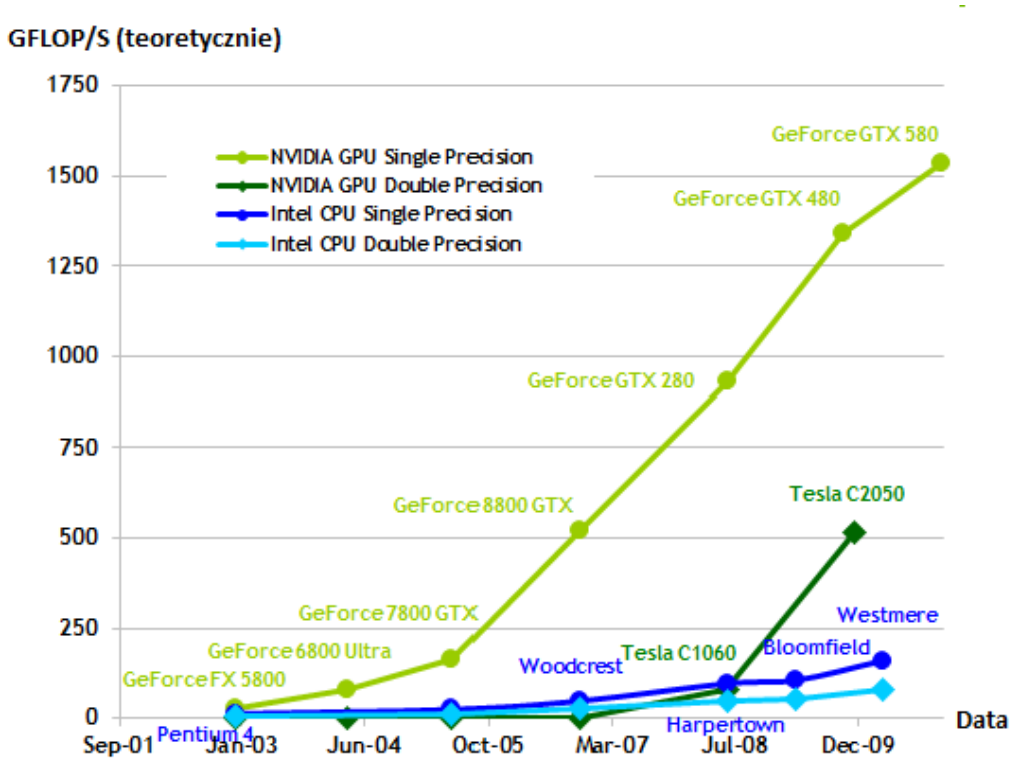
Ponieważ firma NVIDIA zamierzała tę nową rodzinę kart graficznych była wykorzystywana w innych rejonach niż tylko wyświetlanie grafiki, wspomniane wcześniej jednostki ALU są zbudowane w oparciu o wymagania IEEE (ang. *Institute of Electrical and Electronics Engineers*) co do arytmetyki zmiennopozycyjnej. Zostały one również tak zaprojektowane, by zbiór dostępnych operacji był dostosowany do ogólnego użycia niż tylko do wyświetlania grafiki.

Co więcej, jednostki egzekucyjne na GPU zostały wyposażone w możliwość czytania oraz zapisu do pamięci, a także do *pamięci podręcznej* (ang. *cache*) programu, znanej również jako *pamięć współdzielona* (ang. *shared memory*).

Wszystkie te usprawnienia wprowadzone w architekturze CUDA celem rozszerzenia możliwości kart graficznych poza jedynie wyświetlanie grafiki [SK10].

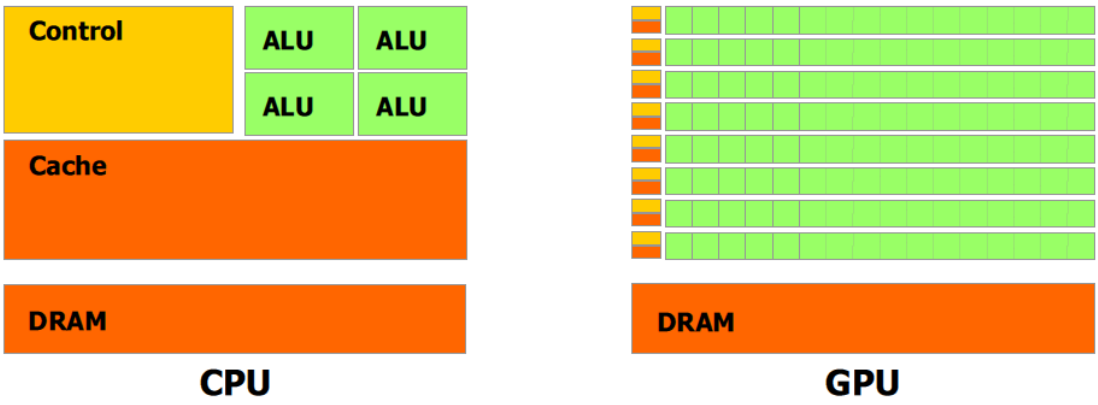
3.3.2 Od wyświetlania grafiki do obliczeń ogólnego przeznaczenia

Jak przedstawiono na rysunkach 3.1 oraz 3.4 procesory graficzne rozwijały się w kierunku przetwarzania równoległego - głównie ze względu wymagań stawianych przez konsumentów na realistyczne grafiki 3D realizowane w czasie rzeczywistym. Z tego też powodu GPU stały się wyspecjalizowanymi jednostkami o wysokim stopniu zrównoleglenia zadań, wielordzeniowymi oraz przetwarzającymi informacje w wielu wątkach.



Rysunek 3.4: Operacje zmiennopozycyjne na sekundę na GPU oraz CPU [Cor11a]

Powodem tak dużej rozbieżności w możliwościach przetwarzania równoległego (zauważalne na rysunku 3.4) pomiędzy CPU a GPU jest fakt wysokiej specjalizacji na intensywne obliczenia w sposób równoległy (czyli to, co jest głównym składnikiem renderowania grafiki) - dlatego też w jednostkach kart grafiki więcej tranzystorów przeznaczonych jest do przetwarzania danych zamiast na cache danych oraz przepływ sterowania - schematycznie zilustrowane to zostało na rysunku 3.5.



Rysunek 3.5: Schematyczne przedstawienie budowy CPU oraz GPU [Cor11a]

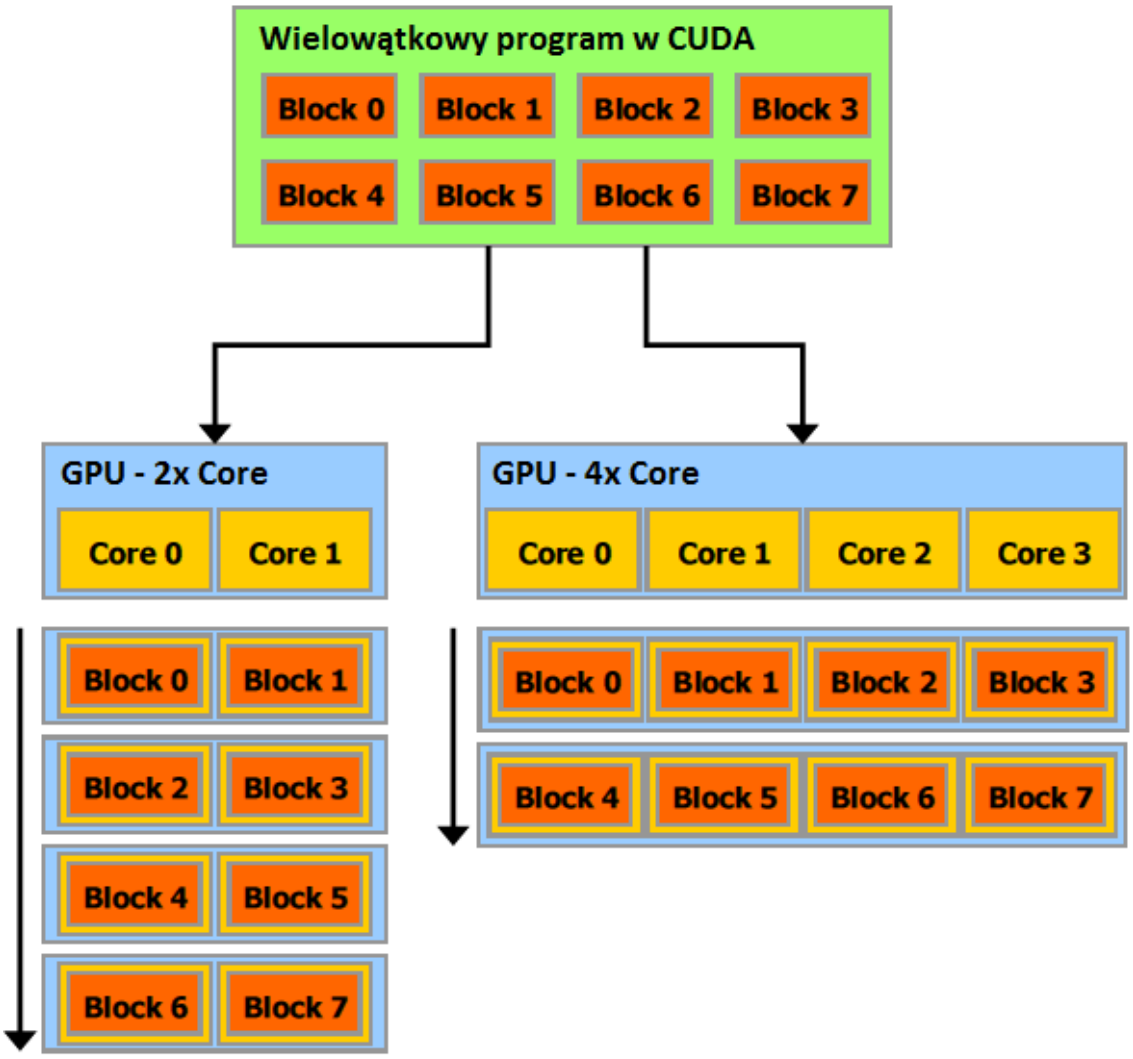
Opisując bardziej szczegółowo, to GPU jest tak zaprojektowane by spełniać wymagania problemów, które mogą być zapisane jako obliczenia równoległę - ten sam program (metoda) jest wykonywany na każdym elemencie danych rónolegle. Ponieważ ten sam program jest uruchomiony na każdym elemencie danyh, to istnieje mniejsze wymaganie w kontroli skomplikowanego przepływu danych. Dodatkowo z powodu wysokiej intensywności arytmetycznej, to opóźnienie dostępu do pamięci może być ukryte przy pomocy obliczeń zamiast dużych części danych przechowywanych w cache jednostki.

Równoległe przetwarzanie danych mapuje elementy danych na równoległe wątki przetwarzające te dane. Wiele aplikacji, które przetwarzają duże zbiory danych może używać podejścia równoległego przetwarzania danych do przyspieszenia obliczeń. W renderowaniu grafiki 3D duże zbiory pikseli oraz wektory są mapowane na równoległe wątki. Podobnie w aplikacjach przetwarzających obraz oraz media, takie jak post-przetwarzanie wyrenderowanych obraów, kodowanie i dekodowaniu video, skalowanie obrazó, czy też rozpoznawanie wzorców może mapować bloki obrazu oraz pikseli na równoległe wątki. W rzeczywistości, wiele algorytmów poza dziedziną przetwarzania obrazów są przyspieszane poprzez równoległe przetwarzanie danych - począwszy od przetwarzania sygnałów czy też symulacji fizycznych do obliczeń finansowych, czy też biologii obliczeniowej.

3.3.3 Skalowalny model programistyczny

Nadejście wielokorowych procesorów oraz multikorowych GPU oznacza, że główne chipy procesorowe są teraz systemami równoległymi. Co więcej równoległość kontynuuje skalowalność zgodnie z prawem Moora - czyli zgodnie z empirycznym prawem, które polega na obserwacji, że ekonomicznie optymalna liczba tranzystorów w układzie scalonym w kolejnych latach posiada trend wykładniczy (podwaja się w niemal równych odcinkach czasu). Wyzwaniem jest stworzenie takiego oprogramowanie, które w niezuważalny sposó będzie skalowało wykorzystywanie równoległości do zmieniającego się środowiska, w którym wzrastać będzie liczba korów procesora, zupełnie jak aplikacje graficzne, które transparentnie skalują swój paralelizm do wielu korów GPU, niezależnie od tego na jakiej karcie graficznej są one uruchamiane, co wiąże sie bezpośrednio z liczbą dostępnych jednostek na procesorze.

Równoległy model programistyczny CUDA jest stworzony by przezwyciężyć wyzwanie, o którym wspomniano wyżej, z jednoczesnym ograniczeniem czasu potrzebnego programistom na nauczenie się nowego podejścia poprzez wykorzystywanie do tworzenia aplikacji standardowych języków programowania, takich jak język C.



Rysunek 3.6: Zilustrowanie automatycznej skalowalności [Cor11a]

W swoim źródle CUDA posiada trzy kluczowe elementy: hierarchię grup wątków, współdzieloną pamięć oraz barierę synchronizacyjną. Elementy te są udostępnione programiście poprzez ograniczony zbiór rozszerzeń języka programowania.

Te abstrakcyjne elementy dostarczają drobnoziarnistą równoległość na poziomie danych oraz wątków, zagnieżdżone w gruboziarnistej równoległości danch oraz zadań (ang. *task*). Takie podejście prowadzi programistę do podziału problemu na większe podproblemy, które mogą być rozwiązane niezależnie przez bloki wątków, a każdy podproblem może być podzielony na mniejsze, które mogą być rozwiązywane wspólnie przez wątki wewnątrz jednego bloku.

Taka dekompozycja zachowuje wszystkie dostępne funkcjonalności języka programowania umożliwiając wątkom kooperować w trakcie rozwiązywania każdego z podproblemów, a w tym samym czasie udostępnia automatyczną skalowalność programu. W rzeczywistości, każdy blok wątków może być zaplanowany na każdym z dostępnych rdzeni na procesorze, w dowolnym porządku, jednocześnie lub sekwencyjnie, tak że skompilowany program w CUDA może być wykonany na dowolnej liczbnie rdzeni procesora, jak zostało to zilustrowane na rysunku 3.6. Jedynie *system uruchomieniowy* (ang. *runtime system*) musi znać faktyczną liczbę jednostek procesora (liczbe rdzeni procesora). Wielowątkowy program jest podzielony na bloki wątkó, które wykonywane

są niezależnie jeden od drugiego, dlatego też GPU z większą liczbą rdzeni będzie automatycznie wykonywać program w krótszym czasie niż karta graficzna z mniejszą liczbą jednostek.

Ten skalowalny model programistyczny umożliwia architekturze CUDA być stosowaną w szerokim spektrum zastosowań: od wysokowydajnych kart graficznych GeForce czy też na profesjonalnych kartach Quadro na komputerach entuzjastów, kartach Tesla przeznaczonych tylko do obliczeń, do kart GeForce przeznaczonych na domowe komputery.

3.3.4 Używanie architektury CUDA

Wysiłek wprowadzenia nowej architektury przez firmę NIDIA, która łączy możliwości klasycznego sposobu generacji grafiki z nowym podejściem do uniwersalnch obliczeń nie oznacza, że wprowadzono jedynie nową architekturę, bez odpowiedniego podejścia dla programowania. Bez względu na to, ile nowych funkcjonalności NVIDIA doda do swoich chipów, niemożliwe stało się używanie ich w dotychczasowy sposób - tj. poprzez OpenGL lub DirectX. Nie tylko wymagałoby to od użytkowników dalszego przekształcania swoich algorytmów do problemów renderowania grafiki (problem opisany w części 3.2.2), ale również w dalszym ciągu użytkownicy (programiści) musieli by programować swoje obliczenia w zorientowanych na renderowanie grafiki środowiskach (tzw. *shading languages*) takich jak GLSL z OpenGL'a czy też HLSL firmy Microsoft.

By osiągnąć możliwie wysoki poziom developerów korzystających z możliwości nowej architektury, NVIDIA rozszeszyła jeden z najbardziej znanych języków programownia - język C, dodając do niego relatywnie mały zbiór nowych słów kluczowych, by programiści mogli korzystać z nowych, specjalnych funkcji dostępnych jedynie na architekturze CUDA. Kilka miesiecy po wprowadzeniu karty GeForce 8800 GTX [Cor06], NVIDIA wprowadziła na rynek darmowy kompilator dla języka CUDA C [Cor11b]. W tym samym momencie, CUDA C stał się pierwszym językiem zaprojektowanym przez firmę produkującą sprzęt graficzny w celu wypełnienia luki w obliczeniach ogólnego przeznaczenia realizowanych na kartach graficznych.

Oprócz języka, w którym tworzone mogą być programy korzystające z mocy GPU, NVIDIA dostarczyła również specjalne *sterowniki* (ang. *drivers*) by wykorzystać pełną moc architektury CUDA. Użytkownicy nie są już więcej zobowiązani do znania interfejsów programowania specyficznych dla procesorów graficznych (takich jak OpenGL czy też DirectX), a co więcej nie są już oni zmuszeni do sprowadzanie swoich problemów do problemu renderowania grafiki.

Literatura

[Agr94] Rakesh Agrawal. Quest: a project on database mining. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, strony 514–, 1994.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, Arun Swami. Mining association rules between sets of items in large databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, strony 207–216, 1993.

[AS94] Rakesh Agrawal, Ramakrishnan Srikant. Fast algorithms for mining association rules. *VLDB*, 1994.

[Cor99] NVIDIA Corporation. Nvidia geforce 256. [on-line] `http://www.nvidia.com/page/geforce256.html`, 1999.

[Cor01] NVIDIA Corporation. Nvidia geforce 3. [on-line] `http://www.nvidia.com/page/geforce3.html`, 2001.

[Cor06] NVIDIA Corporation. Nvidia geforce 8800 gtx. [on-line] `http://www.nvidia.pl/page/geforce_8800.html`, 2006.

[Cor07a] NVIDIA Corporation. Cuda - zone. [on-line] `http://www.nvidia.pl/object/cuda_home_new_pl.html`, 2007.

[Cor07b] NVIDIA Corporation. Nvidia tesla 20. [on-line] `http://www.nvidia.pl/object/cuda_home_new_pl.html`, 2007.

[Cor10] Intel Corporation. New dual-core intel atom processor-based netbooks hit shelves today. [on-line] `http://newsroom.intel.com/community/intel_newsroom/blog/2010/08/23/new-dual-core-intel-atom-processor-based-netbooks-hit-shelves-today`, 2010.

[Cor11a] NVIDIA Corporation. *CUDA C Programming Guide Version 4.0*. NVIDIA Corporation, 2011.

[Cor11b] NVIDIA Corporation. Cuda downloads. [on-line] `http://developer.nvidia.com/cuda-downloads`, 2011.

[EN05] Ramez Elmasri, Shamkand B. NAvathe. *Wprowadzenie do systemów baz danych*. Addison-Wesley, Reading, MA, USA, 2005.

[FPSSU96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, Ramasamy Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[HPYM04] Jiawei Han, Jian Pei, Yiwen Yin, Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 2004.

[LQ05] Yong-Jie LAn, Yong Qiu. Parallel frequent itemsets mining algorithm without intermediate result. *School of Information and Electronic Engineering*, 2005.

[Mue95] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. *University of Maryland at College Park*, 1995.

[PCY95] Jong Soo Park, Ming-Syan Chen, Philip S. Yu. Efficient parallel data mining for association rules. strony 31–36, 1995.

[SK10] Jason Sanders, Edward Kandrot. *CUDA by Example*. Addison-Wesley, Reading, MA, USA, 2010.

[YC06] Yanbin Ye, Chia-Chu Chiang. A parallel apriori algorithm for frequent itemsets mining. strony 87–94, 2006.

[ZY08] Jiayi Zhou, Kun-Ming Yu. Balanced tidset-based parallel fp-tree algorithm for the frequent pattern mining on grid system. *International Conference on Semantics, Knowledge and Grid*, strony 103–108, 2008.

Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Tomasz Kujawa",
    title = "{Równoległe odkrywanie reguł asocjacyjnyh zaimplementowane na procesory graficzne}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2011",
}
```