Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Automatyki i Informatyki Stosowanej

Rok akademicki 2015/2016

Praca dyplomowa magisterska

Adam Rogowiec

Opracowanie równoległej wersji algorytmu estymacji grani funkcji gęstości wielowymiarowej zmiennej losowej i jego implementacja w środowisku CUDA

Opiekun pracy:
dr hab. inż. Marek Nałęcz, prof. nzw. PW

Streszczenie

W niniejszej pracy przedstawiono równoległą implementację w środowisku CUDA, algorytmu estymacji grani funkcji gęstości wielowymiarowej zmiennej losowej, w zastosowaniu do rozwiązania popularnego problemu rekonstrukcji krzywej, z nieuporządkowanego i zaburzonego szumem zbioru punktów. Każda z opracowanych kilku wersji algorytmu pozwala na uzyskanie co najmniej kilkudziesięciokrotnego przyśpieszenia w stosunku do ich wielowątkowego odpowiednika, wykonywanego na CPU. Przebadano również skalowalność zaproponowanego rozwiązania otrzymując liniową zależność czasu pracy od liczby punktów chmury i ich wymiaru.

Słowa kluczowe: *detekcja grani, rekonstrukcja krzywej, GPU, GPGPU, CUDA*

Abstract

Title: *Design of the parallel version of the ridge detection algorithm for the multidimensional random variable density function and its implementation in CUDA technology.*

In this thesis a parallel version of the ridge detection algorithm for the multidimensional random variable density function implemented in CUDA technology is presented and used to tackle popular curve reconstruction from clouds of their noisy and unordered samples problem. There have been a few versions created, each of which gives us at least several dozen times speed up, in comparison to its multithreaded CPU versions. Moreover, the examination of our implementation scalability resulting in linear relationship between its execution time and number of samples, as well as its dimension is presented.

Key words: *ridge detection, curve reconstruction, GPU, GPGPU, CUDA*

Spis treści

1. Wstęp

1.1. Detekcja grani

Dopasowywanie krzywej do nieuporządkowanego zbioru punktów jest zagadnieniem mającym wiele praktycznych zastosowań, m.in. w dziedzinach takich jak grafika komputerowa, modelowanie komputerowe, czy też obrazowanie medyczne. Niejednokrotnie jednak dane wejściowe zawierają pewien szum, związany najczęściej z błędami pomiarowymi. Z taką sytuacją mamy do czynienia np. przy komputerowej tomografii osiowej, pomiarach maszyną współrzędnościową, obrazowaniu metodą rezonansu magnetycznego lub wyznaczaniu trajektorii obiektów z odczytów radaru. W przeszłości wielokrotnie analizowano ten problem, co zaowocowało licznymi rozwiązaniami. Fang i Gossard [10] zrealizowali rekonstrukcję krzywej z danymi punktami końcowymi, przy użyciu nieliniowej minimalizacji funkcji energii sprężystości. Pottmann i Randrup [44] zaproponowali podejście z dziedziny przetwarzania obrazów, polegające na pocienianiu chmury punktów, do momentu otrzymania jej szkieletu, który rekonstruuje krzywą. Problematyczny jest jednak odpowiedni dobór wielkości piksela. Levin [23] zastosował poruszającą się metodę najmniejszych kwadratów, która została później zmodyfikowana przez Lee [22]. Wykorzystał on ważoną, lokalną regresję do wyznaczenia nowych punktów dla każdego punktu chmury, w taki sposób, że gromadzą się one wokół pewnej krzywej. Na koniec wymagany był jeszcze krok decymacji w celu uzyskania zrekonstruowanej krzywej. Goshtasby [13] opublikował rozwiązanie, w którym wyznaczone punkty, lokalnie maksymalizujące określoną funkcję odwrotności odległości, odtwarzały poszukiwaną krzywą. Powstała również procedura określająca i podążająca za najwęższym paskiem, zawierającym punkty obciążone szumem,

autorstwa Cheng'a i Mount'a [24]. Ich praca wyróżnia się dowodem dostarczenia wiarygodnego rozwiązania z prawdopodobieństwem zbieżnym do 1, wraz ze wzrostem liczby punktów.

W niniejszej pracy podjęty został algorytm opracowany przez Marka Rupniewskiego [48]. Ideę jego działania można opisać następująco. Zakładamy, że na wejściu otrzymujemy chmurę punktów rozproszonych wzdłuż pewnej krzywej, w taki sposób, że im bliżej krzywej, tym jest większe prawdopodobieństwo wystąpienia punktu. Pierwszym krokiem jest budowa pewnego zbioru kul, których środkami są punkty chmury. Wyobraźmy sobie pasmo górskie i przyjmijmy, że jego grań wyznacza maksymalne wartości funkcji gęstości prawdopodobieństwa wystąpienia punktów chmury. Następnie odwróćmy do góry nogami nasze pasmo górskie, w taki sposób, że dotychczasowa grań stanie się szlakiem biegnącym dnem doliny, którą wypełniamy kulami. W konsekwencji iteracyjnego usuwania z tej doliny poszczególnych kul, pozostałe będą się przesuwać coraz bliżej dna. Proces jest powtarzany tak długo, aż pozostaną tylko kule znajdujące się bezpośrednio na dnie doliny. W ten sposób wykrywana jest grań, reprezentująca rekonstruowaną krzywą. Niewątpliwie zaletą tego algorytmu jest jego prostota. Ponadto, nie wymaga on wyboru początkowego przybliżenia, czy też określania punktów końcowych rekonstruowanej krzywej. Bez żadnej dodatkowej modyfikacji, możemy zaadoptować go do dowolnej liczby wymiarów. A jego forma sprawia, że naturalnie radzi sobie z dowolną liczbą rozłącznych zarówno otwartych jak i zamkniętych krzywych. Jedynym jego minusem jest słabe radzenie sobie z przecięciami krzywych.

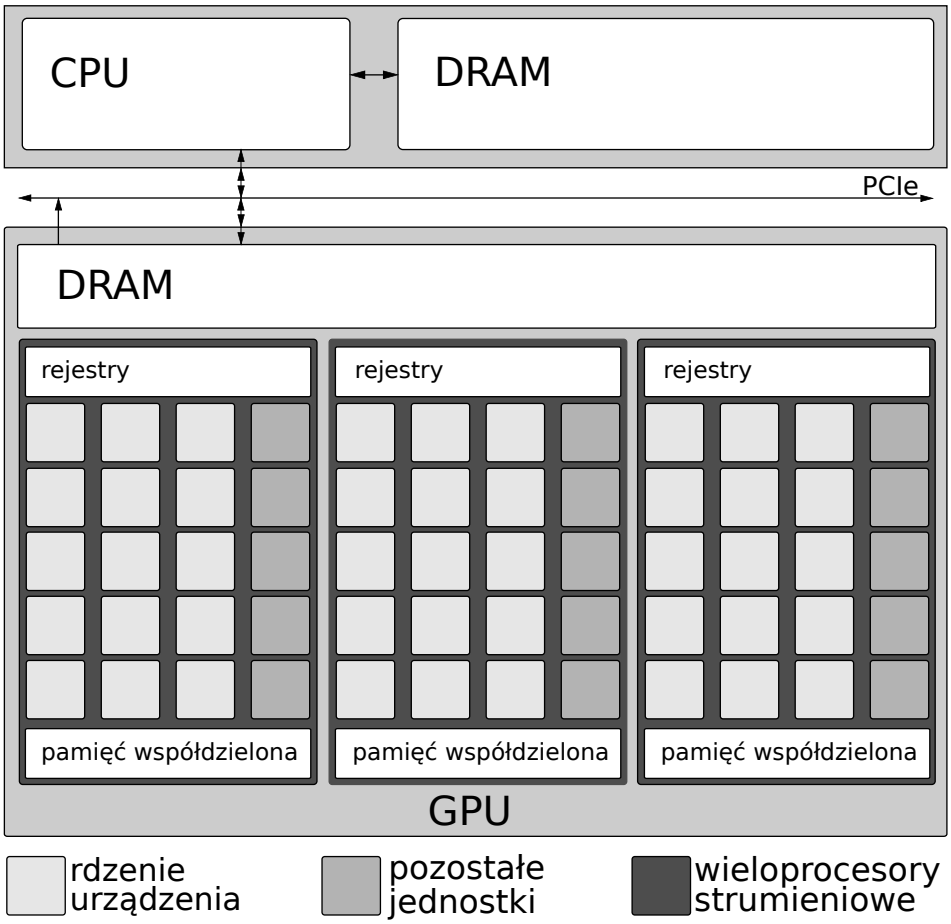
Formalnie, celem detekcji grani jest wydobycie odcinkowo-gładkiej krzywej $\gamma \subset \mathcal{R}^d$ z danego zbioru punktów zaburzonego szumem. Zakładamy przy tym, że punkty należące do wejściowego zbioru są od siebie niezależne, tj. tworzą zbiór $\mathcal{D}$ realizacji niezależnych zmiennych losowych o identycznym rozkładzie z pewną funkcją gęstości prawdopodobieństwa $f\colon \mathcal{R}^d \to \mathcal{R}$, której nośnik jest ulokowany wzdłuż krzywej γ. Odtworzenie krzywej ze zbioru $\mathcal{D}$, polega na sformowaniu uporządkowanego zbiór punktów, tworzących krzywą łamaną, przybliżającą γ.

1.2. Technologia CUDA

Pomimo iż koncepcyjnie algorytm Marka Rupniewskiego jest bardzo prosty, to jednak pociąga za sobą bardzo duży nakład obliczeń, związany z przemieszczaniem się kul w kierunku rekonstruowanej krzywej. Aczkolwiek wart odnotowania jest fakt, że translacja jednej kuli odbywa się niezależnie od pozostałych. W związku z tym, wydaje się, że ten algorytm będzie dobrze pasować do równoległej realizacji na architekturach masywnie wielordzeniowych, w szczególności na kartach graficznych.

Przenoszenie intensywnych obliczeniowo zadań na karty graficzne staje się coraz bardziej popularne, ze względu na bardzo korzystny stosunek ceny do wydajności urządzenia, a także uzyskiwanej wydajności w przeliczeniu na 1 Wat. Ponadto możemy obecnie obserwować rosnący trend wykorzystywania multimedialnych kart graficznych do obliczeń naukowych [53], spowodowany ich dużą dostępnością, niską ceną, a zarazem wydajnością konkurującą ze specjalizowanymi akceleratorami. Pośród dostępnych na rynku technologii, zauważamy wyraźną dominację, dojrzałej już, technologii CUDA [41, 42]. Środowisko CUDA, tworzy wysokopoziomową, abstrakcyjną warstwę, widoczną dla programisty, ukrywającą skomplikowane, niskopoziomowe procedury i szczegóły sprzętowe. Ponadto, dostarcza dobrze zdefiniowane modele pamięci, wykonania i platformy na której może działać, jak też szereg bibliotek i programistycznych narzędzi ułatwiających jego wykorzystanie. W związku z powyższym, wybór tej technologii w niniejszej pracy jest uzasadnioną decyzją. Przejdźmy zatem do jej krótkiego przedstawienia.

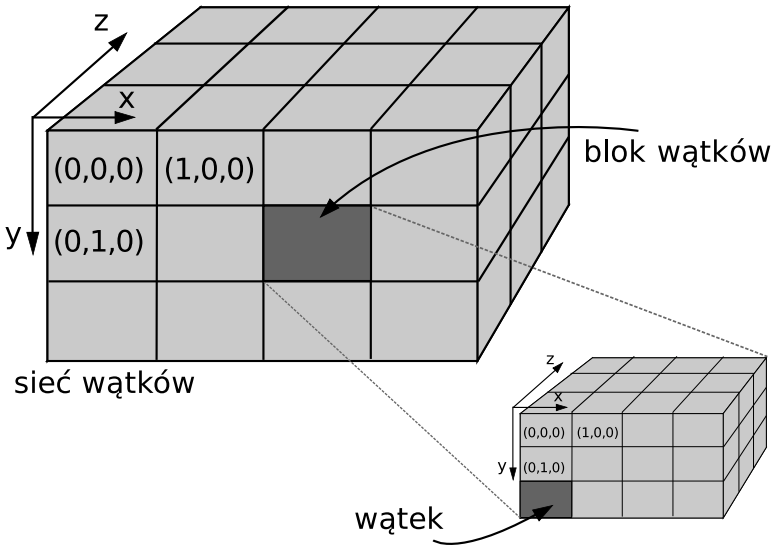
Rysunek 1.1 przedstawia uproszczony model platformy sprzętowej, przyjęty w środowisku CUDA. Można w ramach niego dokonać podziału na dwie główne części: gospodarza i jednego bądź więcej urządzeń. Rola gospodarza jest pełniona przez centralną jednostkę obliczeniową CPU. Natomiast urządzeniami są karty graficzne, które komunikują się z gospodarzem poprzez



Rysunek 1.1: Model platformy sprzętowej.

złącze PCI-Express. Zarówno CPU, jak i GPU, wyposażone są w przydzieloną im pamięć DRAM. Układ GPU charakteryzuje się skomplikowaną hierarchiczną strukturą. W pierwszej kolejności następuje podział na wieloprocesory strumieniowe, z których każdy posiada własną pamięć i zestaw rejestrów. Każdy z wieloprocesorów jest autonomiczną jednostką, niezależną od pozostałych. Idąc głębiej, widzimy wyróżnione jasno-szarym kolorem jednostki arytmetyczno-logiczne, zwane rdzeniami CUDA, a także pozostałą grupę zawierającą m.in jednostki funkcji specjalnych i transferu danych. Warto odnotować, że wspomniane układy pracują równolegle w trybie *Single Instruction, Multiple Data (SIMD)*. Czytelnika zainteresowanego szczegółami architektury kart z rodziny Kepler lub Maxwell odsyłamy do literatury [30–32].

Model wykonania znowu zakłada podział na cześć gospodarza i część urządzenia. Do obowiązków gospodarza należy przede wszystkim przygotowanie



Rysunek 1.2: Hierarchia wątków.

środowiska do pracy, w tym wybór i ewentualna konfiguracja urządzenia, czy też alokacja i przesłanie danych do urządzenia. Ponadto, w głównej mierze, to gospodarz nadzoruje pracę urządzenia, zlecając kolejne zadania do realizacji i synchronizując się z nim w oczekiwaniu na zakończenie obliczeń. Należy tutaj zaznaczyć, że nowsze karty graficzne potrafią same sobie zlecać nowe obliczenia do wykonania. Jest to tak zwana dynamiczna równoległość (ang. *Dynamic Parallelism*). Praca karty graficznej opisywana jest w postaci funkcji jądra. Jego uruchomienie powoduje utworzenie sieci wątków, z których każdy wykonuje ten sam program. Sieć wątków ma hierarchiczną strukturę, zaprezentowaną na rysunku 1.2 i jest zorganizowana tak, by jak najlepiej dopasowywać się do architektury urządzenia. Jej rozmiary użytkownik definiuje poprzez tzw. konfigurację funkcji jądra, podając liczbę bloków wątków w poszczególnych wymiarach, a także wielkość pojedynczego bloku wątków.

Składające się na sieć wątków bloki są rozdzielane pomiędzy poszczególne wieloprocesory karty graficznej. Jednocześnie na jednym takim układzie może rezydować wiele bloków, współbieżnie wykonujących obliczenia. Ponieważ jeden blok wątków może w danej chwili znajdować się tylko na jednym wieloprocesorze, jego wątki mogą się ze sobą komunikować poprzez pamięć współdzieloną, a także synchronizować na specjalnych barierach. Natomiast pomiędzy

dwoma różnymi blokami wątków komunikacja taka nie jest możliwa. W ramach jednego bloku wątków następuje jeszcze podział na nierozłączne, pracujące równolegle i ściśle synchronicznie w trybie SIMD, sploty 32-ch wątków. Wątki wewnątrz tej grupy, na nowszych kartach graficznych, mogą się między sobą komunikować poprzez specjalne instrukcje. Konsekwencją ściśle synchronicznej pracy wątków w ramach splotu, jest zjawisko dywergencji. Ma ono miejsce, gdy w wyniku instrukcji warunkowej nie wszystkie wątki wykonują tę samą operację. W zależności od liczby powstałych w ten sposób grup wątków, następuje wielokrotna serializacja ich pracy, podczas której jedne wątki są aktywowane do pracy, a pozostałe (znajdujące się na innej ścieżce wykonania) czekają na swoją kolej.

Każdy pojedynczy wątek ma dostępny prywatny zestaw rejestrów, do których dostęp jest natychmiastowy. Wyższe poziomy hierarchii pamięci, takie jak pamięć współdzielona i pamięć globalna, nie dają jednak gwarancji, że stan widoczny przez jeden wątek jest też widoczny dla pozostałych wątków. Z tego względu wymagają one instrukcji synchronizacji na odpowiednich barierach. Wszystkie wątki w ramach jednego bloku wątków mają dostęp do tego samego obszaru pamięci współdzielonej, widocznego tylko dla danego bloku. Ponadto wszystkie wątki należące do sieci mają dostęp do tego samego obszaru pamięci globalnej, znajdującej się poza układami wieloprocesorów, w pamięci DRAM urządzenia. Co więcej, spójny obraz pamięci globalnej i przydzielonego obszaru pamięci współdzielonej jest dostępny tylko w ramach jednego bloku wątków.

Dostępy do pamięci globalnej realizowane są w transakcjach 32-, 64- lub 128-bajtowych, gdyż tylko takie odcinki pamięci (których początkowy adres jest wielokrotnością ich wielkości) mogą być czytane lub zapisywane przez odpowiednie układy karty graficznej. Pojedyncza instrukcja dostępu do pamięci globalnej jest grupowana w minimalną liczbę transakcji wymaganą do pobrania wszystkich zażądanych przezeń danych. Na ilość transakcji potrzebną do wykonania danego dostępu mają wpływ takie czynniki jak: wyrównanie w pamięci początkowego adresu, rozmiar typu danych, rozproszenie w pamięci elementów do których sięgają kolejne wątki w ramach splotu, a także konfiguracja sposobu buforowania danych w pamięci podręcznej.

Pamięć współdzielona jest podzielona na tak zwane „banki" w taki sposób, że kolejne jej (w zależności od architektury karty graficznej 32-bitowe lub 64-bitowe) słowa odpowiadają kolejnym bankom. Karty graficzne z generacji Kepler i Maxwell posiadają 32 takie banki. Budowa pamięci współdzielonej zapewnia wydajny dostęp w sytuacji, gdy żadne dwa wątki w ramach splotu nie sięgają do różnych słów w pamięci odpowiadających temu samemu bankowi. Warto zwrócić uwagę, iż wątki mogą odwoływać się do innych adresów w pamięci (np. do słów 8- lub 16-bitowych), pod warunkiem, że te znajdują się w granicach pojedynczego banku. Każdy odmienny wzorzec generuje konflikty banków, powodujące serializację odczytów lub zapisów.

Firma Nvidia wypuszcza na rynek swoje produkty przydzielając je do odpowiedniej generacji, zwanej też „rodziną". Pomiędzy kolejnymi generacjami z reguły występują poważne zmiany architektoniczne, jak również dodawane są nowe funkcjonalności. W celu uporządkowania urządzeń względem ich możliwości, każde z nich ma przydzielony tzw. współczynnik zdolności obliczeniowych (ang. *Compute Capability*). Ma on dwucyfrową postać X.Y, gdzie X określa przynależność do danej generacji, natomiast Y – symbolizuje pod-rodzinę, wprowadzającą mniej istotne zmiany. Każda nowa generacja kart graficznych wnosi ze sobą również nowy zestaw instrukcji procesora. Dlatego też firma Nvidia, chcąc zapewnić zgodność „w przód" kodu napisanego w środowisku CUDA, utworzyła pośrednią reprezentację – tzw. kod wirtualnej architektury ptx (ang. *Parallel Thread Execution*). W ten sposób kompilacja programu została podzielona na dwa etapy: najpierw do pośredniej postaci ptx, a dopiero następnie do reprezentacji binarnej odpowiadającej docelowemu urządzeniu.

Dokładne szczegóły technologii CUDA, można znaleźć w bogatej literaturze: [19, 20, 36, 56], wymieniając zaledwie kilka pozycji traktujących na ten temat.

1.3. Cele i zakres pracy

W niniejszej pracy zostały przyjęte następujące cele:

1. Opracowanie co najmniej dwóch równoległych wersji algorytmu estymacji grani (np. zrównoleglenie względem sfer lub względem punktów chmury).

2. Implementacja opracowanych równoległych wersji algorytmu w środowisku CUDA.
3. Testowanie zaimplementowanych wersji algorytmu dla problemów 2D, 3D i wielowymiarowych.
4. Optymalizacja zaimplementowanych równoległych wersji algorytmu pod względem czasu wykonania na platformach sprzętowych z rodziny Kepler i/lub Maxwell.
5. Badanie wydajności zoptymalizowanych implementacji dla problemów 2D, 3D i wielowymiarowych i dla różnej liczby punktów chmury.
6. Wybór implementacji optymalnej pod względem wydajności.

2. Algorytm Marka Rupniewskiego

Ogólna idea bazowej wersji algorytmu detekcji grani została przedstawiona we wstępie do niniejszej pracy 1.1. W tym podrozdziale poszerzymy i uściślimy ten opis.

$S \leftarrow choose(P, R_1)$
repeat
  $S \leftarrow evolve(S, P, R_1)$
  $S \leftarrow decimate(S, R_2)$
until zbiór S nie zmieni się podczas ostatniej iteracji
$E \leftarrow order(S, R_2)$

Algorytm 1: Algorytm detekcji grani funkcji gęstości wielowymiarowej zmiennej losowej.

Danymi wejściowymi algorytmu jest niepusty zbiór $P \subset \mathcal{R}^d$, a także parametry R_1 i R_2 sterujące jego działaniem. Jak widać (por. algorytm 1) składa się on z czterech następujących po sobie operacji, z czego dwie są powtarzane w pętli. Wynikiem jest zbiór E uporządkowanych par punktów, tworzących krzywą łamaną przybliżającą rekonstruowaną krzywą. W niniejszej pracy zakładamy, że $R_2 = 2R_1$, gdyż takie wartości zostały przyjęte w oryginalnym artykule, aczkolwiek dla jasności będziemy zawsze precyzować o który parametr chodzi.

Choose jest etapem wstępnym przygotowującym początkowy zbiór punktów, określany jako zbiór *wybrańców*. Każde dowolne dwa punkty należące do tego zbioru, są od siebie oddalone o co najmniej R_1. Dzięki temu otrzymujemy równomierne i dokładne pokrycie (w sensie przestrzennym) całej badanej chmury punktów.

Evolve. Krok ewolucji jest jedną z najważniejszych części algorytmu detekcji grani. Ma ona za zadanie przesunięcie *wybrańców* w kierunku rekonstruowanej krzywej. W tym celu, dla każdego punktu ze zbioru S wyznaczamy środek ciężkości obszaru będącego częścią wspólną kuli o środku w danym punkcie

s_i i promieniu R_1 oraz odpowiadającej punktowi s_i komórki diagramu Woronoja utworzonego dla zbioru S. Algorytm działania operacji ewolucji został przedstawiony jako algorytm 2. Przesuwanie *wybrańców* do środków ciężkości

repeat
$Q \leftarrow S$
$S \leftarrow \{avg(P \cap vcell(s, Q, R_1)) \mid s \in Q\}$
until zbiór S nie zmieni się podczas ostatniej iteracji
$vcell(s, S, R_1) = \{p \in \mathcal{R}^d \mid \forall_{o \in S \wedge o \neq s} \ \| s - p \| \leq R_1 \wedge \ \| s - p \| \leq \| o - p \|\}$

Algorytm 2: Algorytm ewolucji. $vcell(s, S, R_1)$ oznacza część wspólną kuli o środku w punkcie s i promieniu R_1 z odpowiadającą punktowi s komórką diagramu Woronoja utworzonego dla zbioru S.

powtarzane jest do momentu, gdy w ciągu ostatniej iteracji położenie punktów ze zbioru S się nie zmieni.

Decimate. Na etapie decymacji ze zbioru S usuwane są punkty zgodnie z przyjętą regułą. Jej postać ma istotny wpływ na wydajność, a także właściwości algorytmu takie jak radzenie sobie z przecięciami krzywej lub innymi osobliwościami. Ponieważ algorytm Marka Rupniewskiego ma zastosowanie w przypadku krzywych gładkich i bez przecięć, zaproponowany został następujący warunek usuwania nadmiarowych *wybrańców*:

$$\#\{q \in S \mid \|p - q\| \leq R_2\} > 3 \vee \#\{q \in S \mid \|p - q\| \leq 2R_2\} < 3 \tag{2.1}$$

Oznacza on, że usuwane są punkty, które mają co najmniej trzech sąsiadów w otoczeniu R_2, lub mają mniej niż dwóch sąsiadów w otoczeniu $2R_2$. Operacja decymacji, podobnie jak ewolucji, również jest powtarzana do póki w ostatniej iteracji zbiór S się nie zmieni (por. alg 2).

repeat
  for all $s \in S$ **do**
    if s spełnia warunek usuwania *wybrańców* **then**
      $|S| \leftarrow |S| - 1$
      $S \leftarrow S \setminus s$
  if $|S| < 3$ **then**
    return
until zbiór S nie zmieni się podczas ostatniej iteracji

Algorytm 3: Algorytm decymacji.

Order czyli ostatnia faza detekcji grani, ma za zadanie uporządkowanie otrzymanego w wyniku poprzednich kroków zbioru *wybrańców* w taki sposób, że każde dwa kolejne punkty na wynikowej krzywej łamanej, sąsiadują ze sobą również w ostatecznym zbiorze E. Następnie, łącząc je odcinkami, otrzymujemy krzywą łamaną przybliżającą rekonstruowaną krzywą. W szczególności zbiór danych wejściowych może składać się z wielu rozłącznych podzbiorów (tworzących krzywe). Ten szczególny przypadek został również uwzględniony w algorytmie zaprojektowanym przez Marka Rupniewskiego przedstawionym jako algorytm 4.

$E \leftarrow \{\{p, q\} \subset S \mid p \neq q \; and \; \| p - q \| \leq R_2\}$
while dopóki istnieją punkty $p, q \in S$ takie, że każdy z nich należy do co najwyżej jednej pary ze zbioru E i $\| p - q \| \leq 2R_2$ **do**
    $E \leftarrow E \cup \{\{p, q\}\}$

Algorytm 4: Porządkowanie otrzymanego zbioru *wybrańców*.

3. Równoległa, siłowa wersja algorytmu

Analizując opisany w rozdziale 2 algorytm, możemy spostrzec dwa rodzaje równoległości danych [12, 25, 45]. Pierwszy z nich dotyczy niezależności obliczeń względem punktów chmury, a drugi — względem *wybrańców*. Zwróćmy jednak uwagę, że ta równoległość występuje jedynie na etapie ewolucji. Pojawia się więc pytanie co z pozostałymi częściami algorytmu? W ramach inicjacji początkowego zbioru *wybrańców*, a także decymacji mamy do wyboru dwa warianty: *konserwatywny* i *zachłanny*. Opcja *konserwatywna* charakteryzuje się dokładnością, brakiem redundancji, ale niestety w głównej mierze wymaga sekwencyjnego wykonywania obliczeń. Natomiast druga możliwość ma zupełnie przeciwne cechy. W przypadku końcowego szeregowania *wybrańców*, bardzo mała ilość danych, a także niewielkie znaczenie (ze względu na ilość obliczeń i jednokrotne wykonanie) sprawiają, że tej funkcji nie opłaca się zrównoleglać.

3.1. Struktury danych

Zanim przejdziemy do szczegółowego omówienia zrównoleglanych elementów algorytmu, zapoznamy się najpierw z zaprojektowaną konfiguracją wykorzystywanych w nich danych. W wersji *siłowej* nie budujemy żadnych specjalnych struktur pośredniczących w dostępie do danych. Przetwarzane przez nas informacje najczęściej mają postać chmury punktów. Dlatego też naturalnym wydaje się przechowywać ich współrzędne w postaci macierzy. Symbolem P oznaczamy zbiór punktów wejściowych. Natomiast symbolem S — zbiór *wybrańców*. Pracując nad wydajną implementacją algorytmu warto jednak mieć na uwadze, że w pamięci urządzenia dane mają układ liniowy, a sposób dostępu do nich ma istotny wpływ na wydajność programu. Punkty w ramach macierzy mogą być uporządkowane wierszowo – np.: jeden wiersz macierzy odpowiada jednemu punktowi, a kolumny kolejnym jego współrzędnym – lub

kolumnowo – odwrotna kompozycja. W niniejszej pracy badane są obydwa sposoby rozmieszczenia informacji. W przypadku kolumnowego układu danych, każdy wiersz takiej macierzy jest odpowiednio wyrównany w pamięci, w celu zapewnienia wydajnego dostępu do danych.

3.2. Wybór reprezentantów

Nasze rozważania oprzemy o podstawową, sekwencyjną wersję, która została przedstawiona w algorytmie 5.

```
1 S ← S ∪ p_0
2 for all p_i ∈ P do
3     if ¬{∃q ∈ S | ‖p_i − q‖ ≤ R_1} then
4         S ← S ∪ p_i
```

Algorytm 5: Sekwencyjny algorytm inicjacji zbioru *wybrańców*.

Na wstępie inicjujemy zbiór wybrańców pierwszym punktem z danych wejściowych. Ponieważ z założenia mają one charakter nieuporządkowany, jest to tak naprawdę losowo wybrany punkt chmury. Następnie bierzemy kolejne punkty należące do zbioru P i sprawdzamy, czy w budowanym właśnie zbiorze S nie znajduje się żaden punkt, którego odległość euklidesowa od aktualnie badanego punktu p_i jest mniejsza niż zadany parametr R_1. Jeśli warunek jest spełniony, dołączamy punkt p_i do zbioru S.

Włączenie danego punktu do zbioru *wybrańców* ma istotny wpływ na wybór kolejnych punktów, gdyż blokuje wszystkie pozostałe punkty chmury znajdujące się w promieniu R_1. Stąd też wynikają dwa wspomniane już podejścia do równoległej implementacji tego etapu. Wariant *konserwatywny* jednocześnie testuje tylko jeden punkt, sekwencyjnie wykonując kolejne operacje tak jak to zostało zobrazowane w poprzednim akapicie. Wymusza to ograniczenie się do zaledwie jednego bloku wątków. Wykorzystujemy technikę *kafelkowania*[1] i potokowego przetwarzania[2] danych ze zbioru P, w celu przyśpieszenia, skromnie wykorzystującej zasoby karty graficznej, funkcji jądra. Ogólny schemat działania funkcji jądra został opisany w algorytmie 6.

[1] Opis znajduje się w rozdziale 6.4
[2] Opis znajduje się w rozdziale 6.5

```
1  Kolektywnie wczytaj pierwszy kafelek danych do pamięci prywatnej wątków
2  for Dla wszystkich pełnych kafelków do
3      Synchronizuj wątki
4      Wczytaj kafelek danych z rejestrów do pamięci współdzielonej
5      Synchronizuj wątki
6      Rozpocznij wczytywanie kolejnego kafelka danych do pamięci prywatnej
       wątków
7      Wykonaj obliczenia na danych w pamięci współdzielonej
8  if Ostatni (niepełny) kafelek then
9      Synchronizuj wątki
10     Wczytaj kafelek danych z rejestrów do pamięci współdzielonej
11     Synchronizuj wątki
12     Wykonaj obliczenia na danych w pamięci współdzielonej
13 Zapisz liczbę wybrańców do pamięci globalnej
```

Algorytm 6: Schemat pracy funkcji jądra inicjującej zbiór wybrańców.

Najbardziej intensywną obliczeniowo częścią ewaluacji warunku, widocznego w linii 3 na algorytmie 5, jest zliczanie sąsiadów danego punktu w ramach zbioru wybrańców. W tym celu należy wyznaczyć odległość pomiędzy każdym punktem zbioru S, a aktualnie sprawdzanym punktem chmury. Oczywiście wykorzystujemy fakt, że każdą odległość możemy liczyć niezależnie od innych. Algorytm 7 przedstawia schemat pracy funkcji zliczającej sąsiadów.

```
1  Wczytaj badany punkt q do rejestrów
2  for Dla każdego kafelka punktów ze zbioru R do
3      smem ← kafelek danych
4      Synchronizuj wątki
5      dist ←‖ q − smem_tx ‖
6      if dist ≤ r then
7          Zlicz ile znaleziono sąsiadów w ramach splotu wątków
8          Wyznacz lidera splotu i atomowo zwiększ licznik sąsiadów
9      Synchronizuj wątki
10     if licznik sasiadow ≥ T then
11         return T
   return licznik sąsiadów
```

Algorytm 7: Schemat zliczania sąsiadów punktu w otoczeniu o zadanym promieniu.

Również w tym przypadku korzystamy z *kafelkowania* danych wejściowych. Kafelek składa się z dokładnie tylu punktów, ile jest uruchomionych wątków w ramach bloku. Taki wybór pozwala uprościć kod wczytywania danych,

a także w prosty sposób rozdziela pracę pomiędzy wątki, z których każdy oblicza jedną odległość. Ponadto, pomimo tylko jednokrotnego korzystania z punktów (ze zbioru S), wczytujemy je do pamięci współdzielonej zachowując porządek jaki miały w pamięci globalnej. W ten sposób, zwłaszcza w przypadku wierszowego układu danych, możemy zapewnić grupowanie odczytów, a także brak konfliktów banków przy zapisach do pamięci współdzielonej i zminimalizować liczbę wykonywanych transakcji dostępu do pamięci globalnej. Następnie, podczas późniejszego obliczania odległości euklidesowej, gdy dane są uporządkowane wierszowo, kolejne wątki sięgają do elementów oddalonych od siebie o liczbę wymiarów punktów. Niestety powoduje to konflikty banków stopnia równego liczbie wymiarów punktów. Jednakże bardziej opłaca się zagwarantować efektywne odczyty z pamięci globalnej, kosztem niewielkich konfliktów banków, gdyż opóźnienia związane z nieefektywnym dostępem do pamięci globalnej są znacznie większe. Kolumnowy układ danych nie wymaga podejmowania powyższych kompromisów, umożliwiając wydajne korzystanie zarówno z pamięci globalnej jak i współdzielonej. Co więcej, zawsze wykorzystujemy fakt synchronicznej pracy wątków w ramach splotu do zmniejszenia liczby wykonywanych operacji atomowych [2]. Korzystając z funkcji `__ballot()`, `__popc()`, `__ffs()` zliczamy ile wątków w ramach splotu znalazło sąsiada i desygnujemy lidera do wykonania aktualizacji licznika sąsiadów, znajdującego się w pamięci współdzielonej. W ten sposób zamiast (maksymalnie) 32 operacji atomowych wykonujemy tylko jedną. Każdorazowo po przetworzeniu kafelka danych synchronizujemy wątki na barierze, by określić czy nie należy zakończyć już pracy. W zamian za oszczędne rozmiary sieci wątków, otrzymujemy bardzo dokładne i przestrzennie równomierne pokrycie całej chmury punktów.

Wersja *zachłanna* podejmuje pewien kompromis pomiędzy szybkością działania, a precyzją ostatecznego wyniku. Mianowicie uruchamiamy nie jeden, a wiele bloków wątków iterujących po *kafelkach* zbioru P. Każdy z nich testuje swój *kafelek* danych wejściowych z aktualnym zbiorem *wybrańców* niezależnie od innych. Oznacza to, że jeden blok nie bierze pod uwagę punktów aktualnie dodawanych do zbioru S przez inny blok. Co więcej, punkty dodawane

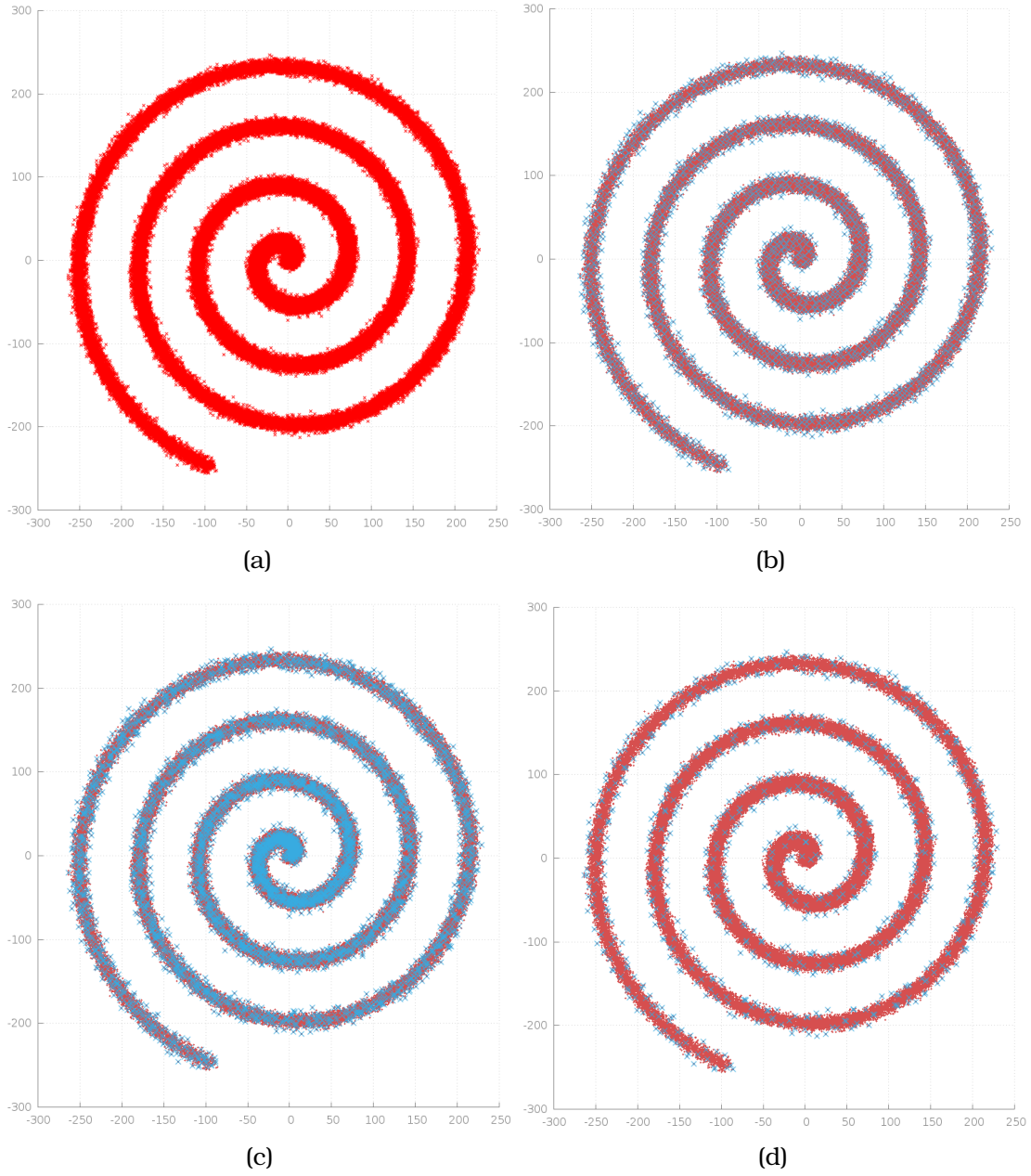
w jednym bloku wątków są niewidoczne z poziomu pozostałych bloków wątków do momentu synchronizacji na barierze porządkującej dostępy do pamięci globalnej. W konsekwencji otrzymujemy znacznie więcej punktów niż w algorytmie *konserwatywnym*, z których wiele nie spełnia postawionego kryterium minimalnej odległości od pozostałych punktów zbioru S. Przykładowe rezultaty zostały pokazane na rysunku 3.1, a czasy działania zanotowane w tabeli 3.1. Algorytm *zachłanny*, co łatwo przewidzieć, jest wielokrotnie szybszy niż jego konkurent. Jednakże bardzo duży początkowy zbiór *wybrańców* negatywnie rzutuje na ostateczny czas pracy całego programu, znacznie go wydłużając. Efekt ten można do pewnego stopnia zniwelować przez dodatkową decymację przed ewolucją. Niestety po zastosowaniu takiego zabiegu, zbiór *wybrańców* zostaje dosłownie zdziesiątkowany, nierównomiernie pokrywając chmurę punktów. Ostatecznie ze względu na niezadowalające wyniki, wersja *zachłanna* nie została wykorzystana w niniejszej pracy.

	wersja konserwatywna	wersja zachłanna	wersja zachłanna z decymacją
czas [ms]	194.27	3.68	35.59
liczba wybrańców	1786	4898	787

Tablica 3.1: Średni czas wykonania i wielkość początkowego zbioru wybrańców dla różnych wersji inicjalizacji. Wyniki dla dwuwymiarowej spirali składającej się ze 100tys. punktów.

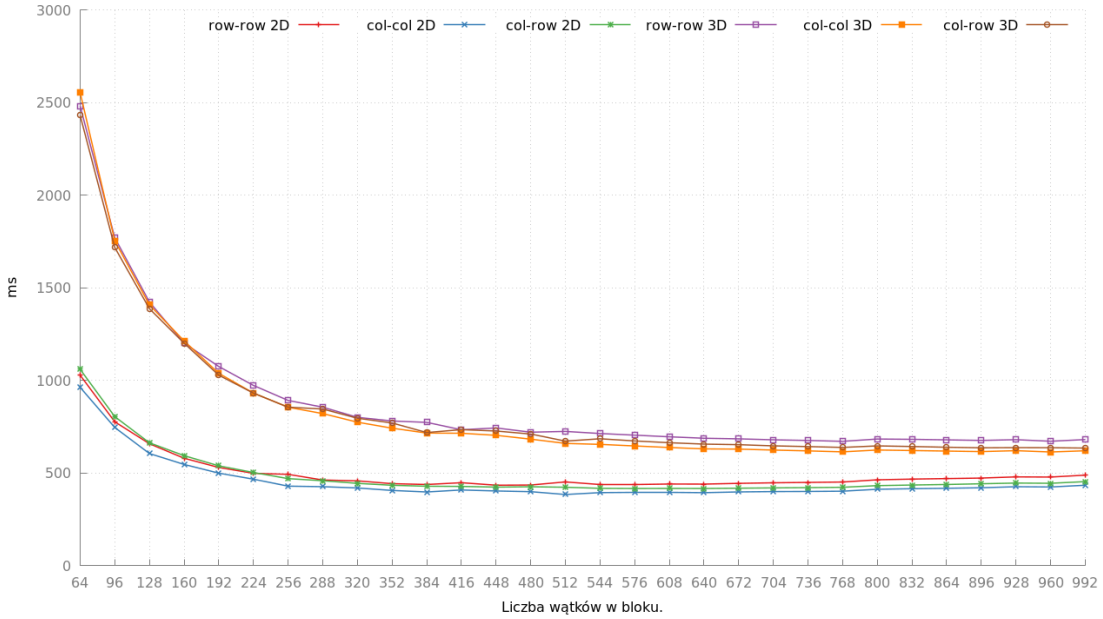
Przed uruchomieniem funkcji jądra, pozostało jeszcze określić jej optymalną konfigurację. Szczegółowy opis procesu doboru wielkości bloku wątków i innych parametrów szablonów funkcji jadra do danej architektury znajduje się w rozdziale 6.1. Tutaj zaprezentujemy tylko wyniki testów wydajności, widoczne na wykresach 3.2 i 3.3. Etykiety mają postać „X-Y (...)", gdzie „X" oznacza układ w pamięci danych wejściowych, natomiast „Y" – danych wyjściowych. „row" symbolizuje porządek wierszowy, a „col" – kolumnowy.

Na karcie graficznej z rodziny Maxwell, wyraźnie widać minimum osiągane dla 512 wątków w bloku, zarówno dla dwóch, jak i trzech wymiarów, toteż w tym przypadku nie ma wątpliwości jaką decyzję podjąć. Sytuacja w przypadku karty GTX TITAN nie jest już tak oczywista. Przy punktach dwuwymiarowych, zauważamy ustabilizowanie się czasu pracy na najniższym poziomie

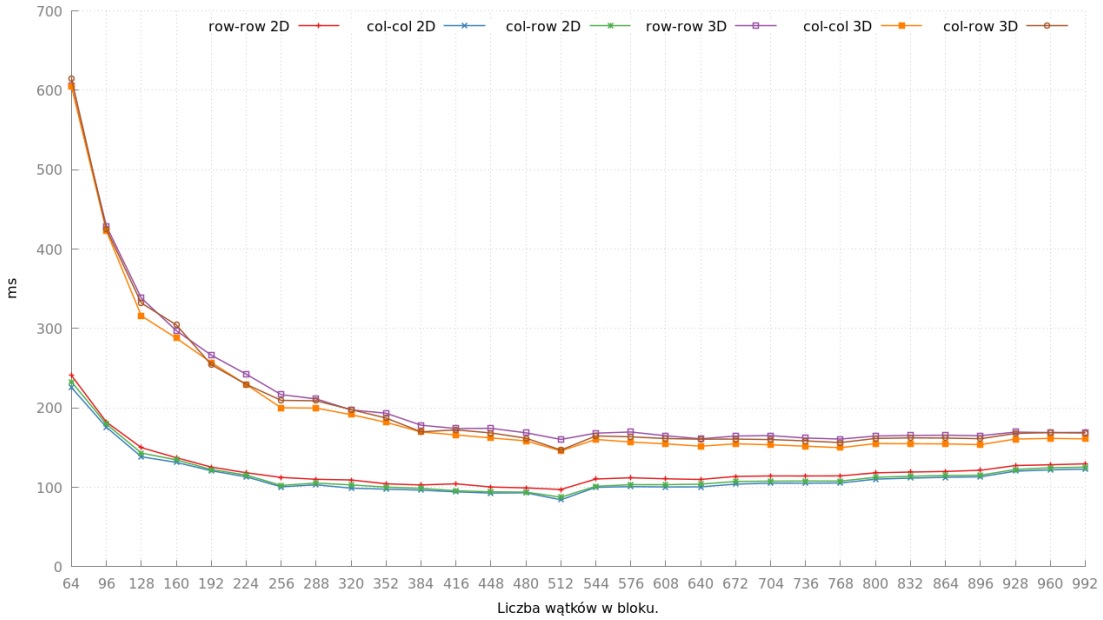


Rysunek 3.1: Rezultaty inicjalizacji zbioru *wybrańców*. Kolorem czerwonym oznaczone są punkty chmury, a niebieskim *wybrańcy*. (a) dane wejściowe, (b) wersja *konserwatywna*, (c) *zachłanna* bez decymacji i (d) z dodatkową decymacją. Spirala składa się ze 100tys. punktów.

w przedziale od 352 do okolic 640 wątków w bloku, z zauważalnymi „wahaniami" dla rozmiaru 512. Biorąc jednak pod uwagę, że aktualna wersja funkcji jądra zużywa znaczną ilość pamięci współdzielonej, rosnącą wraz z wymiarem danych, wybrano opcję 512 wątków na blok. Ponadto zastosowano skalowanie w dół tej wartości, gdy okaże się, że zapotrzebowanie funkcji jądra na zasoby



Rysunek 3.2: Czas inicjalizacji zbioru wybrańców w zależności od liczby uruchomionych wątków w bloku, na karcie GTX TITAN. Zbiór wejściowy stanowi chmura 100 tys. 2-3 wymiarowych punktów, tworząca spiralę.



Rysunek 3.3: Czas inicjalizacji zbioru wybrańców w zależności od liczby uruchomionych wątków w bloku na karcie GTX 750Ti. Zbiór wejściowy stanowi chmura 250 tys. 2-3 wymiarowych punktów, tworząca spiralę.

wieloprocesora jest zbyt wysokie. Dla obydwu testowanych architektur kart, najlepszy okazał się kolumnowy układ danych, zarówno dla punktów chmury, jak i zbioru wybrańców.

3.3. Ewolucja

Kolejnym etapem algorytmu detekcji grani jest ewolucja, skupiająca *wybrańców* wokół rekonstruowanej krzywej. Każdy z punktów w zbiorze *wybrańców* $s \in S$, zostaje przesunięty do środka ciężkości obszaru będącego częścią wspólną kuli o promieniu R_1 i środku w punkcie s i odpowiadającej temu punktowi komórki diagramu Woronoja, utworzonego dla wszystkich *wybrańców*. Najbardziej intensywną obliczeniowo częścią tej operacji jest znalezienie punktów chmury należących do odpowiednich przecięć zbiorów (kuli z komórką diagramu Woronoja). Wymaga to wyznaczenia odległości pomiędzy każdą parą punktów $\{(p, s) : p \in P, s \in S\}$. Na ich podstawie segregujemy punkty $p \in P$ względem przynależności do poszczególnych punktów $s \in S$. Następnie wystarczy już tylko obliczyć środek ciężkości dla każdej tak powstałej grupy punktów i zaktualizować współrzędne *wybrańców*.

Łatwo tutaj zauważyć analogię do problemu znajdowania k najbliższych sąsiadów [9, 50] (k-NN z ang. *k-Nearest Neighbours*), mającego szerokie zastosowania m.in. w statystyce, klasyfikacji, biologii, czy też przetwarzaniu obrazów. Jego opis przedstawia się następująco. Oznaczmy symbolem $\mathcal{R}^d$ zbiór d-wymiarowych punktów odniesienia (ang. *reference points*), a symbolem $\mathcal{Q}^d$ zbiór d-wymiarowych punktów zapytań (ang. *query points*). Następnie dla każdego z punktów $q_i \in Q$ poszukujemy wewnątrz zbioru $\mathcal{R}$, k najbliższych sąsiadów, według przyjętej metryki odległości np. euklidesowej, Manhattan, Pearson'a, Spearman'a itp.

Rozwiązanie zagadnienia k najbliższych sąsiadów w najprostszej formie wykorzystuje metodykę *siłową* (z ang. *brute-force*). Wymaga ona obliczenia odległości pomiędzy każdą parą punktów $\{(r, q) : r \in \mathcal{R}, q \in \mathcal{Q}\}$. Zwróćmy uwagę, że nasz algorytm ewolucji również wykonuje podobne obliczenia. Niestety pomimo swojej prostoty wspomniane podejście pociąga za sobą ogromny nakład obliczeń. Przykładowo, gdy $\|\mathcal{R}^d\| = m$, a $\|\mathcal{Q}^d\| = n$, złożoność czasowa wynosi

$\mathcal{O}(mnd)$ dla samego wyznaczenia macierzy odległości. W konsekwencji bardzo szybko wraz ze wzrostem liczby punktów, a także wymiaru przestrzeni, schemat *siłowy* w sekwencyjnej realizacji staje się niepraktyczny.

Z powodu długiego czasu pracy wersji *siłowej*, zaczęto stosować różnego rodzaju taktyki mające na celu minimalizację liczby kalkulowanych odległości między punktami. W szczególności możemy dokonać jakościowego podziału proponowanych algorytmów na znajdujące rozwiązania dokładne lub przybliżone. Redukcja obliczeń najczęściej odbywa się poprzez budowę rozmaitych (hierarchicznych) struktur drzewiastych, dzielących przestrzeń punktów na mniejsze podzbiory [49]. Z kolei problem dużej liczby wymiarów można rozwiązać dokonując odpowiedniej (np. losowej) projekcji na określoną podprzestrzeń [18], czy też szukając jedynie przybliżonego rozwiązania, stosując np. losowe punkty startowe [4,5], albo permutacje punktów zapytań [43]. Efektywna strategia najczęściej jest sumą dwóch czynników: wydajnej struktury danych i schematu poszukiwań. Steven Skiena [51] przedstawia istotne kwestie dotyczące zastosowania i implementacji algorytmu k-NN, których analiza ułatwia zaprojektowanie programu dostosowanego do naszych potrzeb. Ponadto wymienia przykłady wielu gotowych narzędzi. Jednym z najbardziej znanych jest biblioteka ANN [3] umożliwiająca dokładną i przybliżoną lokalizację najbliższego sąsiada.

Pomimo wysokiej złożoności obliczeniowej, podejście *siłowe* do znajdowania najbliższych sąsiadów idealnie nadaje się do realizacji na kartach graficznych, ze względu na bardzo duży stopień równoległości danych. Jedna z pierwszych implementacji w środowisku CUDA została opublikowana przez Vincenta Garcia i jego zespół [11], osiągając przy tym przyśpieszenie o ponad jeden rząd wielkości w porównaniu do biblioteki ANN [3]. Niestety autorzy bardzo krótko traktują na temat samej implementacji, wspominając jedynie o stosowaniu zgrupowanych odczytów z pamięci globalnej, czy też wykorzystaniu tekstur w przypadku rozproszonych dostępów do pamięci urządzenia. Około dwa lata później ten sam zespół opublikował ulepszoną wersję poszukiwania k-najbliższych sąsiadów [52]. Dzięki przekształceniu wzoru na wyznaczanie odległości euklidesowej i dostosowaniu go do postaci macierzowej, mogli użyć

biblioteki CUBLAS [33]. Jednakże jedynie niewielka część obliczeń mogła zostać w ten sposób przyśpieszona. Warto jeszcze zwrócić uwagę, że autorzy zadbali o wydajny układ danych, przechowując informacje w kolumnowo uporządkowanej macierzy i wyrównując w pamięci jej kolejne wiersze. Zazwyczaj problemem większości dostępnych rozwiązań zagadnienia k-NN jest ich skalowalność. Pod tym względem wyróżnia się rozwiązanie przedstawione w pracy Arefin'a [6], pozwalające pracować na zbiorach o dowolnej liczbie punktów i dowolnej liczbie wymiarów przestrzeni. Możliwe to jest dzięki prostemu zabiegowi podziału wynikowej macierzy odległości na kafelki, które następnie mogą być rozdystrybuowane na wiele urządzeń, gdzie będą kolejno przetwarzane. Ponadto artykuł Arefin'a zawiera bogaty przegląd dotychczasowych rozwiązań w wersji *siłowej* na GPU.

Obliczenia wykonywane w fazie ewolucji algorytmu detekcji grani jedynie częściowo pokrywają się z wyszukiwaniem k-najbliższych sąsiadów. Dokładnie rzecz ujmując, częścią wspólną jest jedynie wyznaczenie macierzy odległości. Dlatego też równoległa implementacja tego etapu detekcji grani została odpowiednio dopasowana do naszych potrzeb. Na wstępie prac pojawiły się dwa schematy zrównoleglenia procesu ewolucji: względem punktów chmury i względem *wybrańców*.

W drugim podejściu automatycznie dokonujemy następującego przydziału obowiązków. Każdy z bloków wątków ma za zadanie przemieszczenie przypisanych mu *wybrańców* w nowo ustalone miejsce. Zaletą powyższego podziału pracy jest możliwość przechowywania w szybkiej pamięci współdzielonej wyników pośrednich (sumy odpowiednich współrzędnych i licznika punktów) podczas obliczania środka ciężkości. Ponadto, każdy wątek mógłby w rejestrach przechowywać współrzędne przypisanych danemu blokowi wybrańców. Opisane rozmieszczenie danych, w pamięci współdzielonej i prywatnych rejestrach wątków, zdecydowanie zmniejsza liczbę dostępów do wolnej pamięci globalnej urządzenia. Jednakże powyższe zrównoleglenie jest sprzeczne z modelem pracy urządzenia w technologii CUDA. Mianowicie, pamiętając o budowie diagramu Woronoja, każdy blok wątków musiałby czekać na pozostałe bloki, by mieć pewność, że dany punkt chmury p należy właśnie do jego komórki. Z tego względu ta wersja została odrzucona.

W przypadku pierwszej koncepcji zrównoleglenia, każdy uruchomiony wątek znajduje najbliższego sąsiada dla jednego lub więcej punktów chmury. Znowu taki scenariusz pracy nie jest bliski idealnego, gdyż powoduje on nierównomierne obciążenie i dywergencję wątków podczas pozostałej części ewolucji (obliczenie środka ciężkości i przesunięcie *wybrańców*). Optymalnym rozwiązaniem okazuje się podział na dwa bezpośrednio następujące po sobie pod-etapy: wyznaczenie najbliższego sąsiada i przesunięcie do środka ciężkości. Pozwala to na dokładne dopasowanie rozmiarów sieci wątków do konkretnego problemu, a tym samym lepsze wykorzystanie zasobów, łatwiejszą i wydajniejszą implementację.

3.3.1. Wersja wielowątkowa na CPU

Ponieważ obecnie praktycznie każdy komputer, a nawet zdecydowana większość nowych telefonów, posiada wielordzeniowe procesory, niesprawiedliwym by było porównywanie mocno zoptymalizowanego algorytmu, wykonywanego na GPU, do jednowątkowej realizacji na CPU. Dlatego też, w ramach niniejszej pracy, powstała również wersja umożliwiająca eksploatację większości dostępnych zasobów współczesnych układów CPU. Niemniej jednak, ograniczono się tutaj jedynie do wykorzystania biblioteki OpenMP [8]. Podyktowane to zostało przede wszystkim wysokim stosunkiem prostoty użycia do uzyskiwanego wzrostu wydajności. W głównej mierze korzystano z dyrektywy `#pragma omp parallel for`. Za jej pośrednictwem uruchamiano tyle wątków ile logicznych rdzeni ma CPU i dokonywano statycznego podziału iteracji pętli. W miejscach gdzie było to konieczne, stosowano również operacje atomowe zapewniające poprawną kolejność dostępów do współdzielonych zasobów.

3.3.2. Implementacja na GPU

W rozdziale 2 widzimy, że obliczenia wykonywane w tej części detekcji grani są powtarzane w pętli. Natomiast z poprzedzających rozważań wiemy już, że w ramach ewolucji wywoływane będą dwie funkcje jądra. Mamy zatem do wyboru dwa warianty nadzorowania pracy: z poziomu gospodarza, lub z poziomu urządzenia. Uruchamianie nowych funkcji jądra z poziomu urządzenia, czyli

tak zwana „dynamiczna równoległość” [36], możliwe jest dla urządzeń posiadających współczynnik zdolności obliczeniowych ≥ 3.5 [31], Oznacza to, że nie wszystkie karty z rodziny Kepler wspierają tę nową funkcjonalność [30]. Toteż obydwie propozycje zostały zaimplementowane w niniejszej pracy. Decyzja, którą z nich uruchomić, jest sterowana przy użyciu odpowiednich definicji makr dostarczonych przez kompilator `nvcc`. Ich wartości są określane na etapie kompilacji poprzez podanie docelowej architektury urządzenia.

Kolejne kroki wykonywane w ramach ewolucji zostały zaprezentowane w algorytmie 8. Pierwszym zadaniem realizowanym przed przystąpieniem do obliczeń jest wyznaczenie konfiguracji funkcji jądra, a dokładnie tylko liczby uruchamianych bloków wątków (linie 4-5). Ich wymiar jest statycznie określany już na etapie kompilacji. Korzystamy z *CUDA occupancy API* [36] do obliczenia ile bloków może jednocześnie rezydować, w danej konfiguracji poszczególnych funkcji jądra, na jednym wieloprocesorze wybranego urządzenia (linie 2-3). Następnie otrzymany wynik jest przemnażany przez liczbę wieloprocesorów i pewną heurystycznie dobraną stałą, zależną od architektury karty graficznej[3]. Ostatnie mnożenie wywodzi się z rekomendacji aktywowania dużej liczby bloków, będącej wielokrotnością liczby wieloprocesorów [35]. Pozwala to na lepsze skalowanie do przyszłych architektur, a przede wszystkim, paradoksalnie, skutkuje to większą wydajnością. Mając na uwadze koszty aktywizacji bloku wątków, w tym przydział zasobów, pobranie parametrów itp., wydawać by się mogło, że wystarczy uruchomić maksymalną liczbę bloków (w danej konfiguracji), mogącą jednocześnie pracować na urządzeniu, by otrzymać maksymalną wydajność. Jednakże ta zależność musi być uwarunkowana również innymi czynnikami, czy mechanizmami, na temat których Nvidia niestety się nie wypowiada. Wspólną cechą konfiguracji obydwu używanych funkcji jądra jest jednowymiarowa sieć i blok wątków. Taki wybór dobrze dopasowuje się do liniowego układu wielowymiarowych danych w pamięci, upraszczając arytmetykę indeksowania. Przed uruchomieniem funkcji jądra zerujemy pomocnicze tablice. Na koniec każdej iteracji synchronizujemy urządzenie, by mieć pewność, że flaga decydująca o kolejnym wykonaniu pętli została zaktualizowana.

[3] Szczegóły procesu wyboru optymalnej konfiguracji funkcji jądra znajdują się w rozdziale 6.1

1 $smCount \leftarrow$ Liczba wieloprocesorów urządzenia
2 $closestSphereSmOccupancy \leftarrow getMaxSmOccupancy()$
3 $shiftSphereSmOccupancy \leftarrow getMaxSmOccupancy()$
4 $csGridSize.x \leftarrow smCount * closestSphereSmOccupancy * FACTOR$
5 $ssGridSize.x \leftarrow smCount * shiftSphereSmOccupancy * FACTOR$
6 **repeat**
7 Inicjuj tablice pomocnicze.
8 closestSphereKernel<<<csGridSize, csBlockSize, 0, stream>>>
9 shiftSphereKernel<<<ssGridSize, ssBlockSize, 0, stream>>>
10 Synchronizuj urządzenie.
11 **until** Zbiór S nie zmieni się podczas ostatniej iteracji

Algorytm 8: Schemat pracy ewolucji.

Pierwsza uruchamiana przez nas funkcja jądra, przedstawiona jako algorytm 9, ma za zadanie przypisanie każdemu punktowi chmury odpowiedniego *wybrańca* zgodnie z warunkiem podanym w algorytmie 2 w linii 5. Ponadto każdy wątek akumuluje współrzędne punktów przypisanych danemu *wybrańcowi* (CordSums) i ich liczbę (SpherePointCount), w celu wyznaczenia środka ciężkości potrzebnego w następnej funkcji jądra. Nasza funkcja jądra, podobnie jak wszystkie inne opracowane w ramach niniejszej pracy, ma postać funkcji szablonowej. Parametrami szablonu są wymiar punktów chmury, typ danych reprezentujących współrzędną punktu, rozmiar bloku wątków, liczba punktów przypadających na jeden wątek, oraz układ w pamięci danych wejściowych (zbiór P) i wyjściowych (zbiór S). Parametryzacja tych wszystkich informacji niesie ze sobą szereg zalet i możliwości, które sukcesywnie będą opisane, przy tylko kilku drobnych wadach. Przede wszystkim statyczne wartości umożliwiają kompilatorowi wykonanie wielu optymalizacji, a także pozwalają na precyzyjne dobranie ich najbardziej wydajnej konfiguracji. Prócz tego, szablon umożliwia wykorzystanie dowolnego, podstawowego, numerycznego typu danych oferowanego przez język C++. Niestety minusem takiego rozwiązania jest uciążliwość, często bardzo długiej, kompilacji programu i słabo czytelne komunikaty błędów i ostrzeżeń kompilatora.

Każdy wątek jest przypisywany do kilku punktów chmury, a w związku z tym przechowuje w rejestrach następujące wartości:

`minSquareDist[ITEMS_PER_THREAD]` – odległość do aktualnie najbliższego *wybrańca* dla danego punktu

```
1  for Dla wszystkich pełnych kafelków w P do
2      for i = 0 to ITEMS_PER_THREAD do
3          point_i ← P_idx
4      for all tile_k : S do
5          smem ← tile_k
6          Synchronizuj wątki
7          for all s ∈ smem do
8              for i = 0 to ITEMS_PER_THREAD do
9                  dist_i ←‖ s − point_i ‖
10                 if dist_i < minSquareDist_i then
11                     minSIdx_i ← globalny indeks wybrańca s
12                     minSquareDist_i ← dist_i
13         Synchronizuj wątki
14     for i = 0 to ITEMS_PER_THREAD do
15         if minSquareDist_i ≤ R_1^2 then
16             SpherePointCount_{minSIdx_i} ++
17             CordSums_{minSIdx_i} += point_i
18 if Ostatni (nie)pełny kafelek then
19     Wykonaj dokładnie to samo co wyżej, ale ze sprawdzaniem zakresu
```

Algorytm 9: Wyznaczanie najbliższego sąsiada danego punktu.

`minSIdx[ITEMS_PER_THREAD]` – indeks w tablicy S aktualnie najbliższego wybrańca dla danego punktu

`points[ITEMS_PER_THREAD * DIM]` – współrzędne punktów przypisanych danemu wątkowi

Przetwarzanie przez jeden wątek kilku punktów, powoduje zwiększenie ziarnistości zadań, opisanej w rozdziale 6.2. Zarazem wpływa to na zmniejszenie liczby dostępów do pamięci globalnej, gdyż więcej punktów chmury jednocześnie współdzieli wczytywane punkty ze zbioru S. Należy zwrócić w tym momencie uwagę, iż jest to implementacja dostosowana do (niewielkiej) liczby wymiarów, pozwalającej na umieszczenie wszystkich wczytywanych punktów chmury w rejestrach. Blok wątków porusza się w pętli, z odpowiednim skokiem, po obydwu przetwarzanych tablicach (P i S), przez co mogą być one dowolnych rozmiarów. Przy takim schemacie pracy należy zawsze sprawdzać warunek przekroczenia zakresu. Na szczęście możemy tego w zdecydowanej większości uniknąć, stosując kafelkowanie opisane w rozdziale 6.4.

Wczytywanie punktów chmury odbywa się ze wzorcem dostępu do pamięci, zależnym od uporządkowania danych, aczkolwiek zawsze używamy kwalifikatorów `const` i `__restrict__`. Instruują one kompilator do wygenerowania instrukcji `LDG` [34], zawsze buforowanej w pamięci podręcznej *tylko do odczytu*[4][5] [36]. W przypadku wierszowego ułożenia współrzędnych, kolejne wątki sięgają do adresów w pamięci oddalonych od siebie o wielokrotność wymiaru punktów. Niestety taka konfiguracja sprzyja niewyrównanym dostępom do pamięci, np. gdy liczba wymiarów nie jest parzysta, a pojedyncza współrzędna reprezentowana jest przez 32-bitowy typ danych. Implikacją powyższych faktów jest nadmiarowa, w stosunku do faktycznych potrzeb, liczba zlecanych transakcji dostępu do pamięci globalnej. Jednakże istnieje duże prawdopodobieństwo, że te „zbędnie" pobrane dane, będą znajdowały się w pamięci podręcznej w momencie, gdy będą nam one już potrzebne, w ten sposób amortyzując dłuższe opóźnienie przy pierwszym dostępie. Oczywiście zależy to od różnych czynników, takich jak architektura urządzenia, liczba aktywnych splotów, czy odległość pomiędzy adresami do których sięgają kolejne wątki. Układ kolumnowy nie stwarza opisywanych wyżej problemów. Odczyty z kolejnych komórek pamięci globalnej, a dodatkowo jeszcze wyrównane w pamięci wiersze macierzy, zapewniają optymalną liczbę transakcji.

Podczas czytania drugiej wejściowej tablicy, czyli zbioru *wybrańców*, również buforujemy odczyty w pamięci podręcznej *tylko do odczytu* za sprawą kwalifikatorów `const` i `__restrict__`. Tym razem jednak dane zapisywane są w pamięci współdzielonej. Niezależnie od porządku macierzy, kolejne wątki sięgają do następujących po sobie komórek pamięci globalnej, a także pamięci współdzielonej, gwarantując tym samym zgrupowane odczyty i brak konfliktów banków.

Napisane w ramach tej pracy funkcje jądra intensywnie korzystają z techniki rozwijania pętli. Jest to możliwe głównie za sprawą parametrów szablonu, które już na etapie kompilacji określają liczbę iteracji. Vasily Volkov, w swojej słynnej prezentacji [55], udowodnił, że poprzez zwiększenie równoległości

[4] Zwanej również buforem tekstur.

[5] W przypadku architektury 3.X jest to oddzielny bufor *tylko do odczytu*/tekstur. Natomiast dla urządzeń o architekturze 5.X jest to zunifikowana pamięć podręczna L1/tekstur

na poziomie instrukcji (ang. *ILP*), możemy ukrywać opóźnienia operacji arytmetycznych, a zarazem otrzymać większą efektywność przy mniejszej liczbie wątków. Od tamtego czasu nawet poradniki firmy Nvidia informują, że sama równoległość na poziomie wątków (ang. *TLP*), może być nie wystarczająca do osiągnięcia maksymalnej wydajności [36, 39]. W ramach funkcji jądra szukającej *wybrańców*, rozwijane są wszystkie pętle iterujące po punktach przypisanych danemu wątkowi (linie 2,8,14 w algorytmie 9), podobnie jak i inne instrukcje iterujące po wymiarach danego punktu (linie 3,9 i 17).

W linii 17 ma miejsce również jeszcze pewna reorganizacja kolejności obliczeń. Mianowicie instrukcje, które zazwyczaj wykonywalibyśmy w jednej pętli zostały rozdzielone na dwie części jak to widać na listingu umieszczonym na rysunku 3.4. Taka sztuczka znowu zwiększa poziom ILP, jednakże wymaga użycia dodatkowych rejestrów [19].

```
#pragma unroll
for (int d = 0; d < DIM; ++d)
{
  dist =
    smem[p*DIM+d]-pt[j*DIM+d];
  sqDist += dist*dist;
}
```

```
#pragma unroll
for (int d = 0; d < DIM; ++d)
  dist[d] =
    smem[p*DIM+d]-pt[j*DIM+d];

#pragma unroll
for (int d = 0; d < DIM; ++d)
  sqDist += dist[d]*dist[d];
```

Rysunek 3.4: Przykład reorganizacji kodu zmniejszający zależność następujących po sobie instrukcji i zwiększający ILP

Należy jeszcze zwrócić uwagę, że aktualizacja danych w liniach 16 i 17 odbywa się atomowo, ponieważ każdy blok wątków może sięgać do tych samych adresów w pamięci.

Przyjrzyjmy się jeszcze stosunkowi obliczeń do dostępów do pamięci globalnej, tak zwany współczynnik *CGMA* z ang. *Compute to Global Memory Access ratio*. W wersji podstawowej bez żadnych optymalizacji ma on następującą wartość:

$$CGMA_1 = \frac{3 * DIM * |P| * |S| + |P| + DIM * |P|}{DIM * |P| * |S| + |P| + 2 * DIM * |P|} \tag{3.1}$$

$$= \frac{DIM * (3 * |S| + 1) + 1}{DIM * (|S| + 2) + 1} \tag{3.2}$$

$$\simeq \frac{3 * |S|}{|S|} = 3 \tag{3.3}$$

Po zastosowaniu schematu obliczeń opisanego powyżej, otrzymuje on postać:

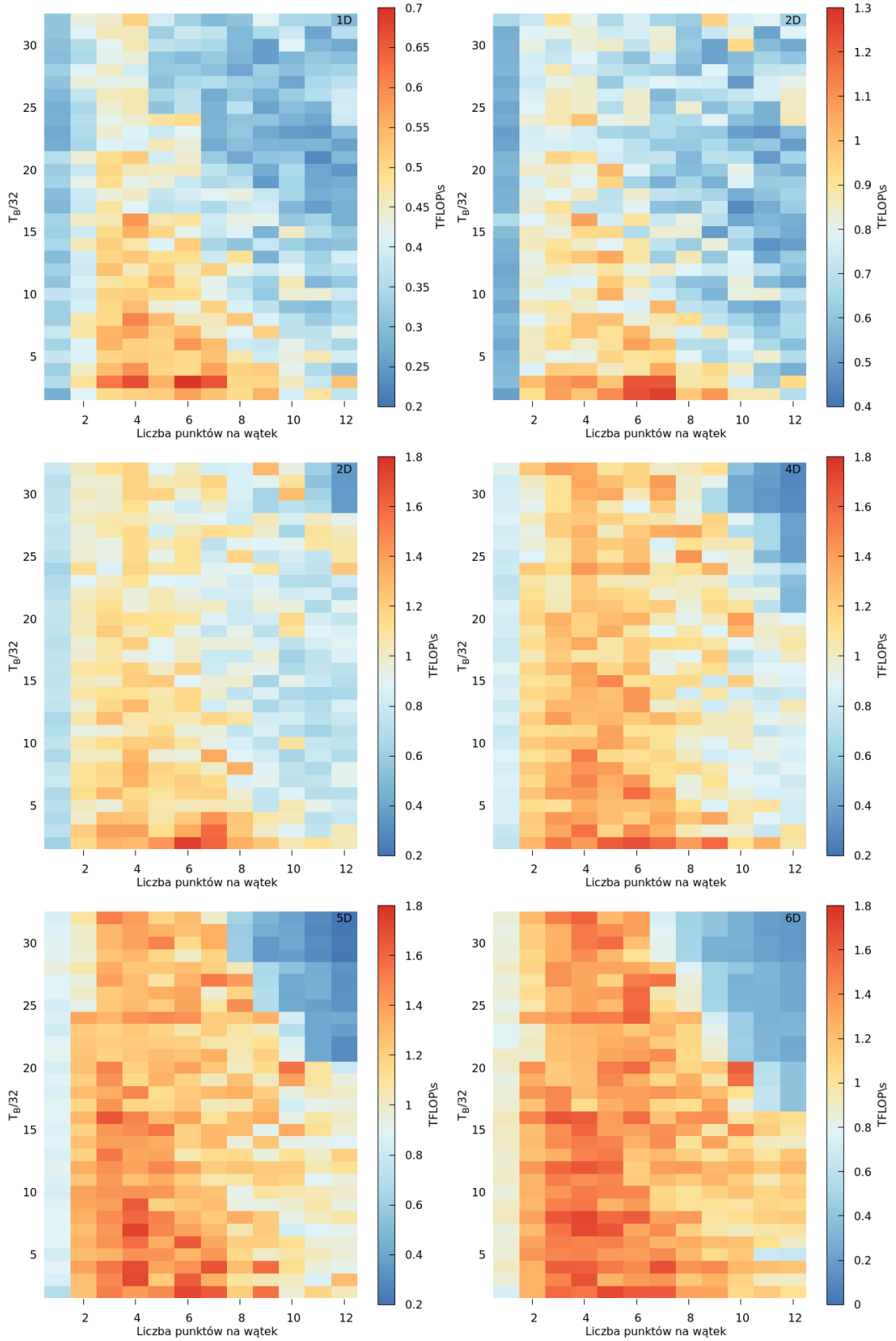
$$CGMA_2 = \frac{3 * DIM * |P| * |S| + |P| + DIM * |P|}{DIM * |P| * |S| \div IPT \div BS + |P| + DIM * |P|} \tag{3.4}$$

$$= \frac{DIM * (3 * |S| + 1) + 1}{DIM * (|S| \div IPT \div BS + 1) + 1} \tag{3.5}$$

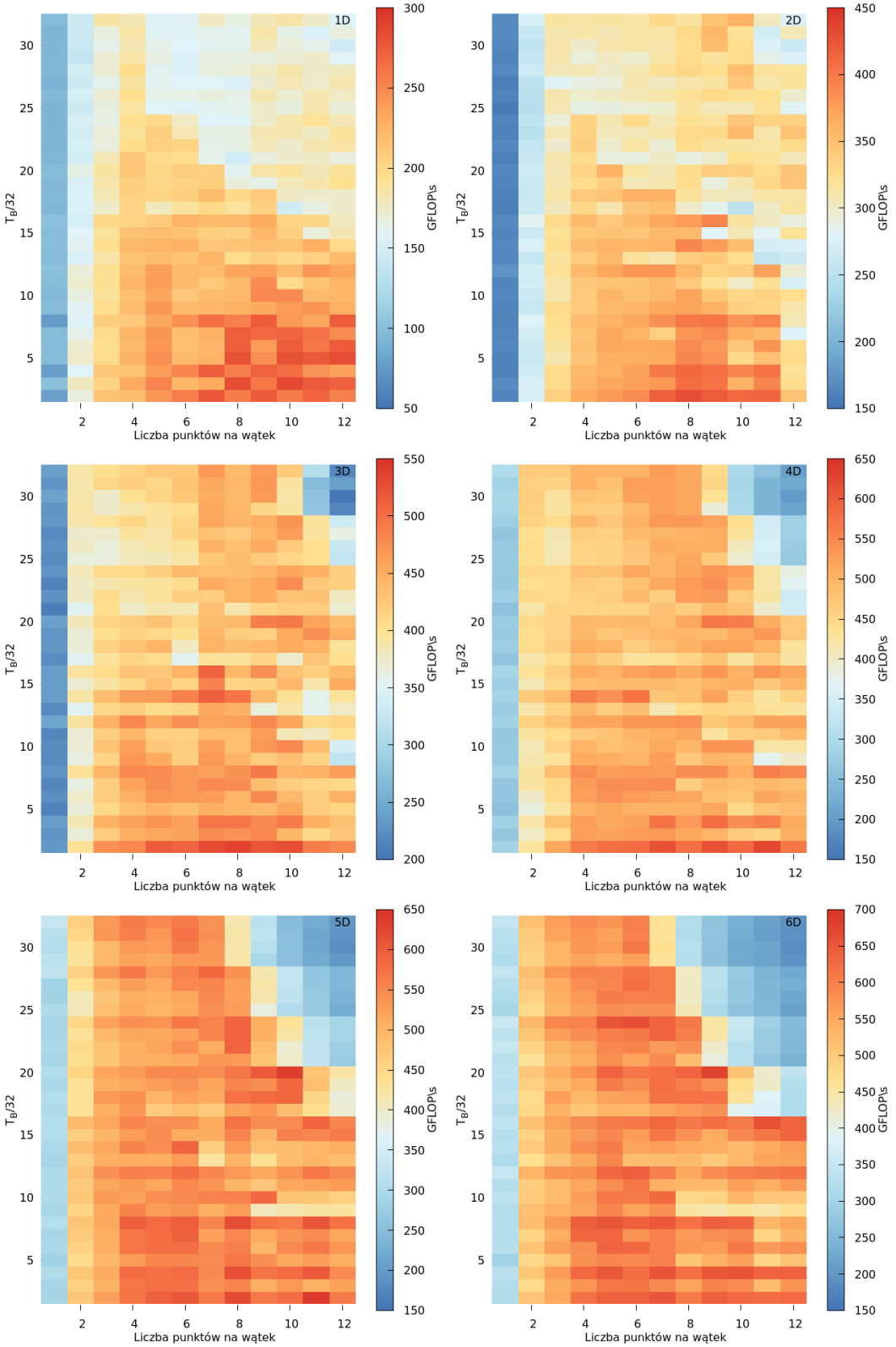
$$\simeq \frac{3 * |S|}{|S| \div IPT \div BS} = 3 * IPT * BS \tag{3.6}$$

Symbol IPT oznacza liczbę punktów przypadających na jeden wątek, a BS rozmiar bloku wątków. Jak widać w podstawowej wersji funkcja jądra jest zdecydowanie ograniczona przez przepustowość pamięci globalnej. Zgodnie z oczekiwaniami, optymalizacja odczytów z pamięci globalnej znacząco poprawia sytuację.

Na wykresach 3.5 i 3.6 zostały zaprezentowane mapy wydajności w zależności od parametrów konfiguracji funkcji jądra i wymiaru danych. Wydawać by się mogło, że powinien być wyraźnie widoczny malejący trend w liczbie przetwarzanych punktów na wątek wraz ze wzrostem wymiarowości informacji. Powinien być on powodowany rosnącym zużyciem rejestrów potrzebnych do zbuforowania aktualnie analizowanych punktów chmury przez pojedynczy wątek. Intensywne wykorzystanie pamięci prywatnej, przekłada się na malejącą liczbę jednocześnie aktywnych bloków wątków na wieloprocesorze. Wiadomo jednak, że nie zawsze musi to prowadzić do gorszej efektywności funkcji jądra. Można przypuszczać, że gdzieś musi się znajdować granica opłacalności wysokiego zużycia rejestrów. Niestety jak widać, testowana liczba wymiarów jest zbyt niska by to zjawisko unaocznić. Analizując wykresy możemy spostrzec, że z niemal całkowitą pewnością rozmiar 64 wątków w bloku jest najlepszym wyborem. Problemem pozostaje jedynie dobór właściwej ilości przypisanych punktów na jeden wątek. Oczywiście idealnym rozwiązaniem byłoby, znając wcześniej wymiar badanego problemu, dokonać odpowiednich testów. Jednakże trudno to sobie wyobrazić, by mieć specjalizowaną kompilację dla każdego wymiaru. W projekcie załączonym do niniejszej pracy został ustalony nominalny przydział 30 rejestrów na prywatny bufor punktów chmury. Ta wartość jest odpowiednio skalowana w zależności od wymiaru i typu danych (`float`, `double`).



Rysunek 3.5: Wydajność funkcji jądra, skompilowanej dla architektury 3.5, przetestowanej na karcie GTX TITAN, w zależności liczby wątków w bloku, liczby punktów przetwarzanych przez pojedynczy wątek i wymiaru danych. Zbiór wejściowy stanowi chmura 250 tys. punktów, tworząca odcinek o zadanej długości. Symbol T_B oznacza liczbę wątków w bloku.



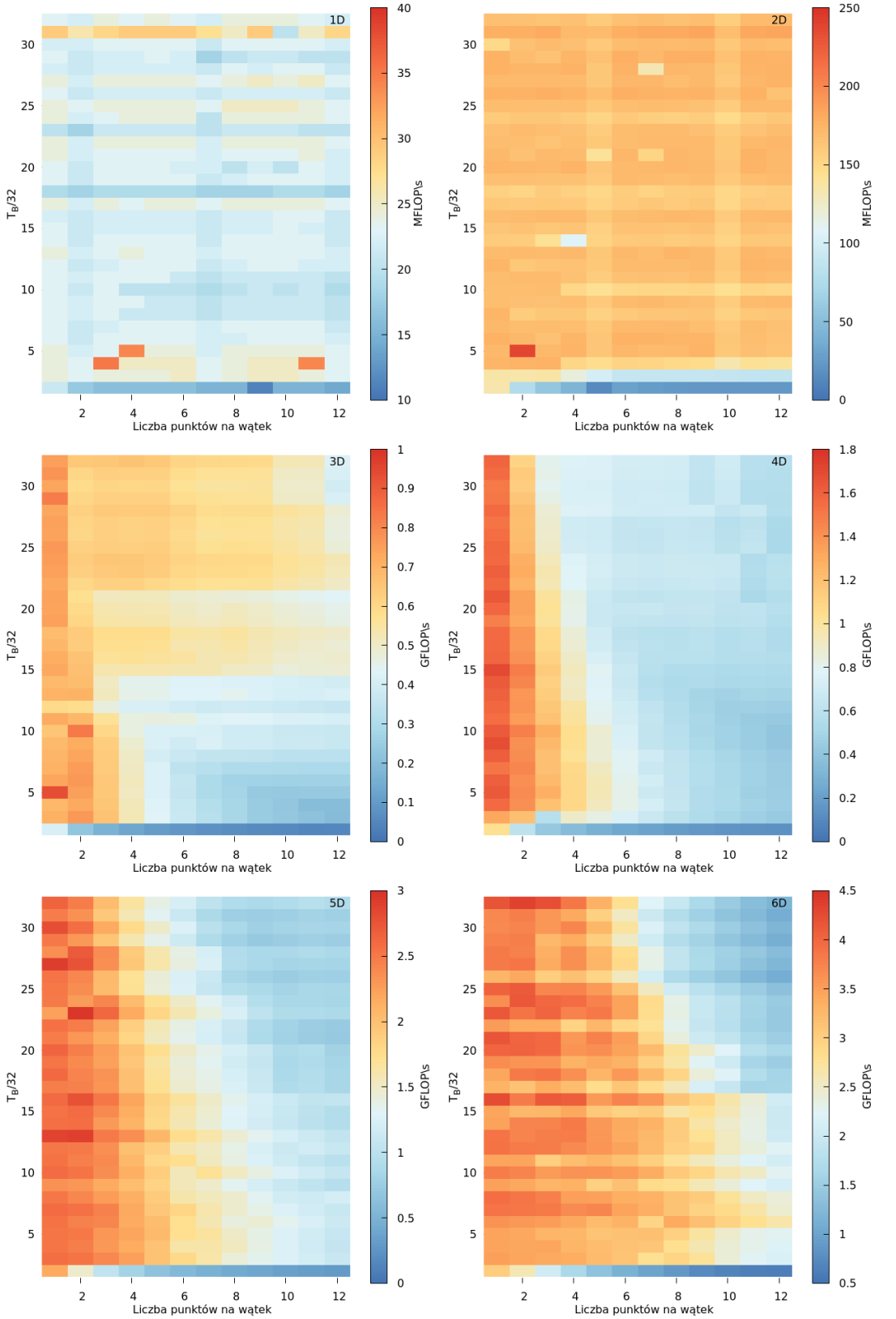
Rysunek 3.6: Wydajność funkcji jądra, skompilowanej dla architektury 5.0, przetestowanej na karcie GTX 750 Ti, w zależności liczby wątków w bloku, liczby punktów przetwarzanych przez pojedynczy wątek i wymiaru danych. Zbiór wejściowy stanowi chmura 250 tys. punktów, tworząca odcinek o zadanej długości. Symbol T_B oznacza liczbę wątków w bloku.

W algorytmie ewolucji pozostała nam jeszcze do opisania druga funkcja jądra, której zadaniem jest przesunięcie *wybrańców* do środka ciężkości przecięcia kuli i komórki Woronoja odpowiadających danemu *wybrańcowi*. Algorytm 10 przedstawia schemat pracy pojedynczego wątku. Jest to bardzo prosta funkcja jądra, mocno ograniczona przez dostępy do pamięci globalnej (współczynnik CGMA < 1) i wykonująca bardzo mało obliczeń. Toteż z pewnością nie przyczynia się ona do ogólnego wzrostu wydajności programu. Niemniej jednak rozwiązanie polegające na oddzielnym dedykowanym tej czynności kernelu jest zdecydowanie lepsze od (bardzo wolnego!) przesyłania danych w tę i z powrotem do CPU, by tam wykonać obliczenia. Warto tutaj jedynie zwrócić uwagę na warunek decydujący o przesunięciu punktu. Następuje ono tylko w sytuacji, gdy przesunięcie z aktualnej pozycji *wybrańca* do środkiem ciężkości jest „numerycznie zauważalna". Ponadto, w miarę możliwości, korzystamy z technik opisanych powyżej.

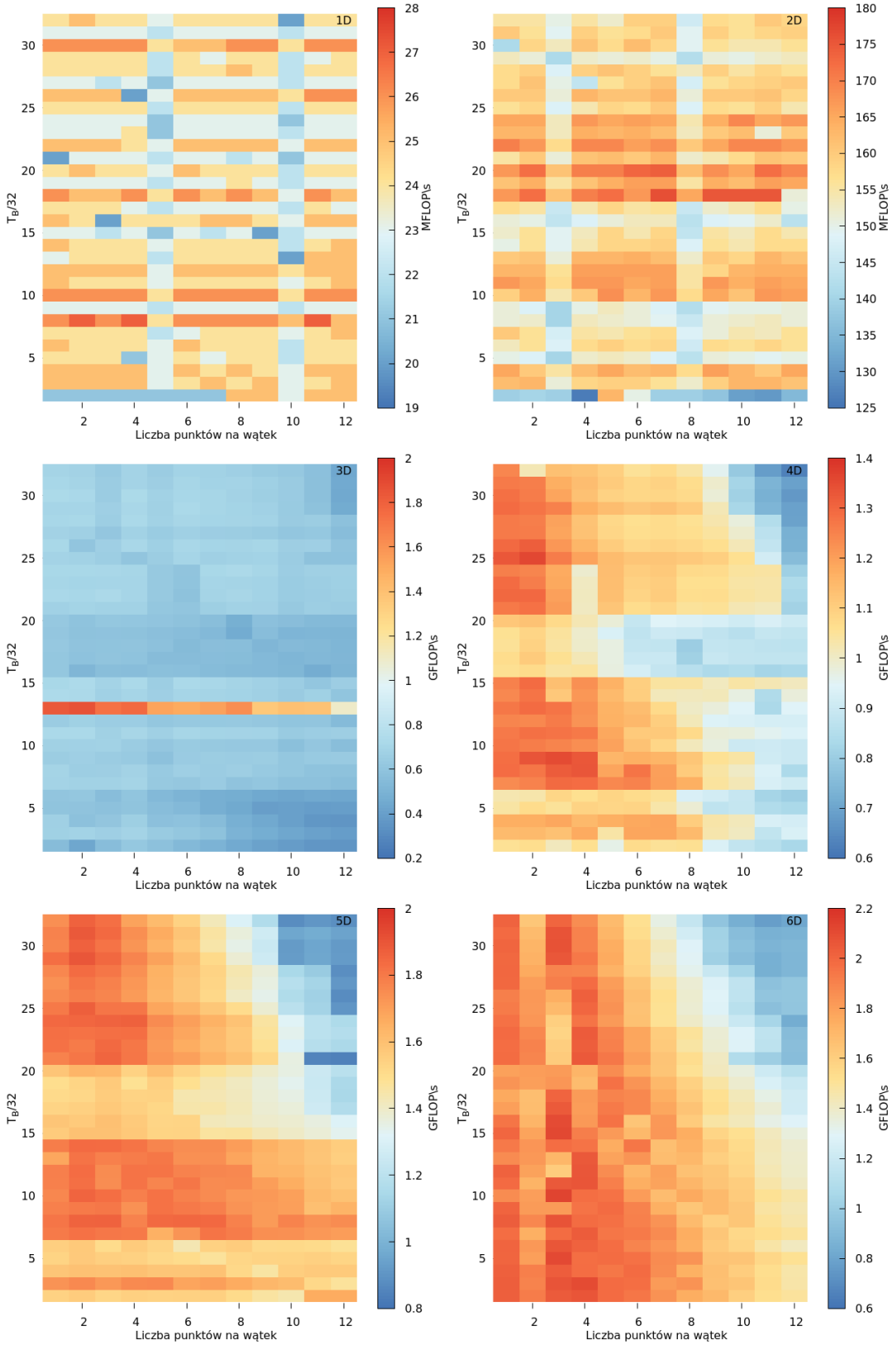
for Dla wszystkich pełnych kafelków w S **do**
  for $i = 0$ **to** $ITEMS\_PER\_THREAD$ **do**
    $spc_i \leftarrow SpherePointCount_{idx}$
    $cs_i \leftarrow CordSums_{idx}$
  for $i = 0$ **to** $ITEMS\_PER\_THREAD$ **do**
    $mc_{s_i} \leftarrow$ środek ciężkości przecięcia kuli z komórką Woronoja
    if $\| mc_{s_i} - s_i \| > 2 * | mc_{s_i} * spc_i * \varepsilon |$ **then**
      $s_i \leftarrow mc_{s_i}$
if Ostatni (nie)pełny kafelek **then**
  Wykonaj dokładnie to samo co wyżej, ale ze sprawdzaniem zakresu

Algorytm 10: Przesunięcie *wybrańców* do środków ciężkości.

Na rysunkach 3.7 i 3.8, widnieją wykresy ilustrujące uzyskaną w testach wydajność dla funkcji jądra przesuwającej wybrańców. W przypadku wymiarów 1-2 praktycznie nie zauważymy bardzo istotnej różnicy w zależności od wybranej konfiguracji, o czym świadczą zakresy wartości GFLOP/s. Jednak z dalszym wzrostem wymiarowości przestrzeni, rośnie również różnica pomiędzy najlepszym, a najgorszym wynikiem. Widzimy, że z mniejszą liczbą wątków w bloku coraz bardziej opłaca się zwiększać ILP, tutaj poprzez liczbę punktów przypadających na jeden wątek. Tymczasem duża liczba wątków w bloku również pozwala otrzymać dobrą efektywność. Nie mniej jednak, dla obu kart, najlepsze rezultaty od 4 wymiaru, prawie niezależnie od rozmiaru bloku, mamy dla



Rysunek 3.7: Przeskalowana wydajność funkcji jądra, skompilowanej dla architektury 3.5, przetestowanej na karcie GTX TITAN, w zależności od liczby wątków w bloku, liczby punktów przetwarzanych przez pojedynczy wątek i wymiaru danych. Zbiór wejściowy stanowi chmura 250 tys. punktów, tworząca odcinek o zadanej długości.



Rysunek 3.8: Przeskalowana wydajność funkcji jądra, skompilowanej dla architektury 5.0, przetestowanej na karcie GTX 750 Ti, w zależności od liczby wątków w bloku, liczby punktów przetwarzanych przez pojedynczy wątek i wymiaru danych. Zbiór wejściowy stanowi chmura 150 tys. punktów, tworząca odcinek o zadanej długości.

przedziału 1-4 punktów na wątek. Sytuacja jest podobnie skomplikowana jak dla poprzednio opisywanej funkcji jądra. Być może rozszerzenie testu o większą liczbę wymiarów, a także zbadanie czy najwyższe wyniki charakteryzują się podobną liczbą jednocześnie aktywnych splotów wątków i zużyciem rejestrów, pozwoliłyby wybrać właściwą konfigurację. Zwróćmy jeszcze uwagę na wykres dla 3 wymiarów na rysunku 3.8, gdzie obserwujemy wyraźną dominację jednego rozmiaru bloku wątków. Co więcej, to odstępstwo od normy powtarza się dla każdej testowanej liczby punktów przetwarzanych przez pojedynczy wątek, toteż ciężko powiedzieć o jakimś błędzie pomiarowym. Możemy jedynie przypuszczać, że mamy do czynienia z „perfekcyjnym"dopasowaniem się konfiguracji do architektury karty.

3.4. Decymacja

Ostatnim zrównoleglanym etapem w ramach algorytmu detekcji grani jest decymacja, mająca za zadanie usunięcie nadmiarowych punktów ze zbioru S. Zgodnie z regułą przedstawioną w rozdziale 2, badamy punkty należące do aktualnego zbioru *wybrańców* i sprawdzamy, czy w ich sąsiedztwie nie znajduje się za dużo, bądź za mało sąsiadów. W sytuacji spełnienia któregokolwiek z warunków dany punkt zostaje usunięty ze zbioru. Podobnie jak w fazie inicjalizacji zbioru *wybrańców*, kolejność z jaką następuje sprawdzanie punktów ma istotny wpływ na ostateczny rezultat, gdyż usunięcie jednego punktu warunkuje pozostanie lub nie, innych punktów. Stąd też mamy do wyboru dwa, opisane wcześniej w rozdziale 3.2, warianty możliwej implementacji tej części. Podejście *zachłanne*, pomimo iż znowu wielokrotnie szybsze, niestety zbyt intensywnie redukuje zbiór *wybrańców*, powodując liczne przerwania krzywej. Tym samym drastycznie obniża dokładność ostatecznego wyniku algorytmu, toteż nie jest ono wykorzystywane w niniejszej pracy.

Pozostaje zatem strategia *konserwatywna*, wykorzystująca jedynie jeden blok wątków. Schemat jej implementacji został zaprezentowany w algorytmie 11. Obliczenia są wielokrotnie powtarzane w pętli do momentu aż w ostatniej iteracji nie nastąpi żadna zmiana, bądź osiągniemy minimalną liczebność zbioru. Natomiast wewnętrzna pętla przebiega po całym zbiorze wybrańców,

badając po kolei, każdego *wybrańca*. Jak widzimy, funkcja jądra zawiera znaczną ilość skomplikowanej logiki wykonywanej przez każdy wątek. Tak naprawdę równoległa praca wątków następuje jedynie w liniach 5, 8, 13 i 16. Wyznaczenie liczby sąsiadów odbywa się dokładnie tak samo jak w etapie inicjalizacji zbioru *wybrańców*, przy użyciu algorytmu 7. Z kolei usuwanie punktów spełniających ograniczenia realizowane jest poprzez przesunięcie pozostałej, dalszej, części tablicy o jedną pozycję w lewo. Operacja ta jest wykonywana kolektywnie przez wszystkie wątki w bloku. Każdy z nich zapamiętuje w rejestrze jeden element tablicy, po czym zachodzi synchronizacja wątków i zapis danych w docelowej pozycji. Koordynacja pracy jest konieczna w celu uniknięcia sytuacji, gdzie jeden wątek czyta z miejsca, które zostało już nadpisane przez inny wątek.

```
1  lastCount ← 0
2  while lastCount ≠ |S| and |S| > 3 do
3      lastCount ← |S|
4      for i = 0 to |S| do
5          neighbours ← liczba sąsiadów w promieniu R_2 punktu s_i
6          if neighbours ≥ 4 then
7              |S| ← |S| − 1
8              Przesuń punkty {s_{i+1}, ..., s_{|S|−1}} o jedną pozycję w lewo
9              if |S| < 3 then
10                 break
11             else
12                 continue
13         neighbours ← liczba sąsiadów w promieniu 2R_2 punktu s_i
14         if neighbours ≤ 2 then
15             |S| ← |S| −1
16             Przesuń punkty {s_{i+1}, ..., s_{|S|−1}} o jedną pozycję w lewo
17             if |S| < 3 then
18                 break
19         else
20             i ← i + 1
```

Algorytm 11: Usuwanie nadmiarowych *wybrańców*.

Sprawdzanie warunków zostało rozbite na dwa rozłączne, następujące po sobie etapy, gdyż taki podział umożliwia zmniejszenie liczby wykonywanych operacji. W pierwszej kolejności analizujemy, czy dany *wybraniec* ma co najmniej trzech sąsiadów w otoczeniu R_2. W początkowym etapie detekcji grani to ten warunek odsieje większość punktów. Gdy zbiór *wybrańców* jest jeszcze

znacznych rozmiarów, punkty są ciasno upakowane, toteż większe jest prawdopodobieństwo posiadania zbyt dużej liczby sąsiadów. Ponadto ewaluacja tego kryterium nie musi pociągać za sobą konieczności przejrzenia całego zbioru *wybrańców*, w przeciwieństwie do drugiej części reguły. Natomiast funkcja zliczająca sąsiadów kończy swoje działanie jak tylko osiągnie zadany próg.

Analizując opisaną implementację, z pewnością razi nas wielokrotne kopiowanie tych samych danych wykonywane przy usuwaniu kolejnych punktów. Problem ten można rozwiązać dokonując kilku modyfikacji powyższego algorytmu. Zamiast przesuwania tablicy, przy spełnieniu poszczególnych warunków, będziemy oznaczać dany punkt flagą `NaN`. Pociąga to za sobą konieczność dodania kilku dodatkowych instrukcji `if` pomijających punkty ostemplowane jako do usunięcia. Po zakończeniu etapu etykietowania *wybrańców* przechodzimy do fizycznej redukcji zbioru, gdzie każdy wątek ma za zadanie przesunięcie na odpowiednią pozycję przypisanych mu punktów. W tym celu całym blokiem wątków poruszamy się w pętli po tablicy S, buforując kolejne punkty w rejestrach. Następnie wątki, które wczytały punkty nieoznakowane flagą `NaN`, atomowo zwiększają przechowywany w pamięci współdzielonej licznik, wyznaczający właściwy wyjściowy indeks. Pomimo iż jest to najprostsze i najkrótsze rozwiązanie, nie oznacza, że także najlepsze. Powyższy sposób generuje bardzo dużą ilość operacji atomowych, które mocno spowalniają pracę. Z pomocą przychodzi wspominana już w rozdziale 3.2 technika agregowanych na poziomie splotu wątków, operacji atomowych, opisana przez Andrew Adintez'a [2]. Zanim ją wykorzystamy, dokonujemy jeszcze tylko kilka następujących usprawnień. W artykule lider splotu wątków determinowany jest wywołaniem funkcji:

```
#define WARP_SZ 32
__device__
inline int lane_id(void) { return threadIdx.x % WARP_SZ; }
```

Natomiast my skorzystamy z zestawu rejestrów specjalnego przeznaczenia [38] i wplatanych w kod instrukcji ptx [37]:

```
__device__ __forceinline__ unsigned int cubLaneId()
{
    unsigned int ret;
    asm volatile("mov.u32 %0, %laneid;" : "=r"(ret) );
```

```
    return ret;
}
```

Dodatkowo obliczanie indeksu w ramach splotu wątków zamieniamy z instrukcji:

```
__popc(mask & ((1 << lane_id()) - 1));
```

na wywołanie funkcji:

```
__popc(mask & LaneMaskLt());
```

Gdzie funkcja `LaneMaskLt()` jest zdefiniowana następująco:

```
__device__ __forceinline__ unsigned int LaneMaskLt()
{
    unsigned int ret;
    asm volatile("mov.u32 %0, %lanemask_lt;" : "=r"(ret) );
    return ret;
}
```

Wygenerowany kod ptx, dla obu powyższych wersji widoczny jest na rysunku 3.9. Różnica jest wyraźna. Wyznaczanie „na piechotę" odpowiedniej maski

```
and.b32   %r8, %r1, 31;
mov.u32   %r9, 1;
shl.b32   %r10, %r9, %r8;
add.s32   %r11, %r10, -1;
and.b32   %r12, %r11, %r3;
popc.b32  %r13, %r12;
```

(a)

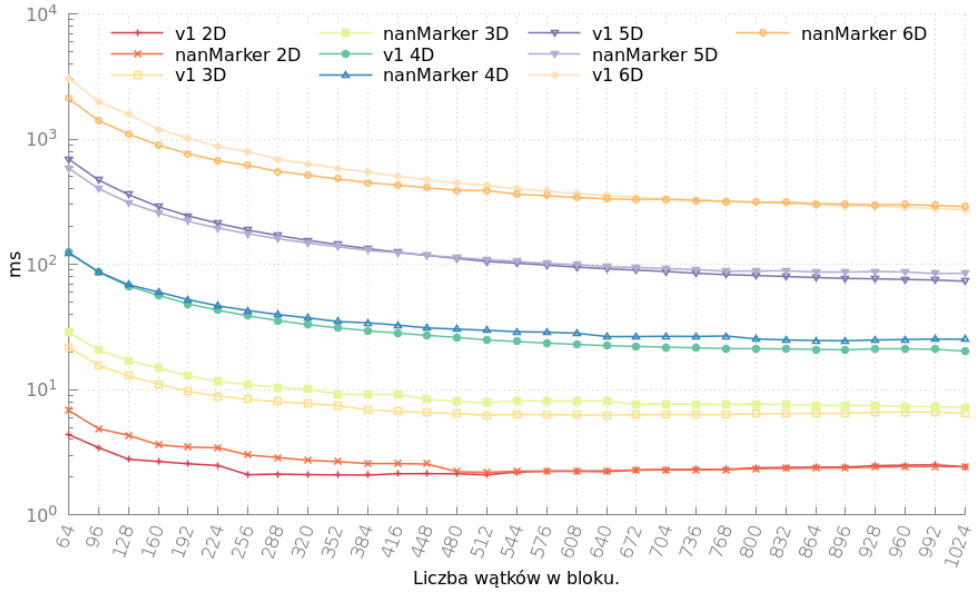
```
mov.u32 %r8, %lanemask_lt;
and.b32   %r9, %r8, %r2;
popc.b32  %r10, %r9;
```

(b)

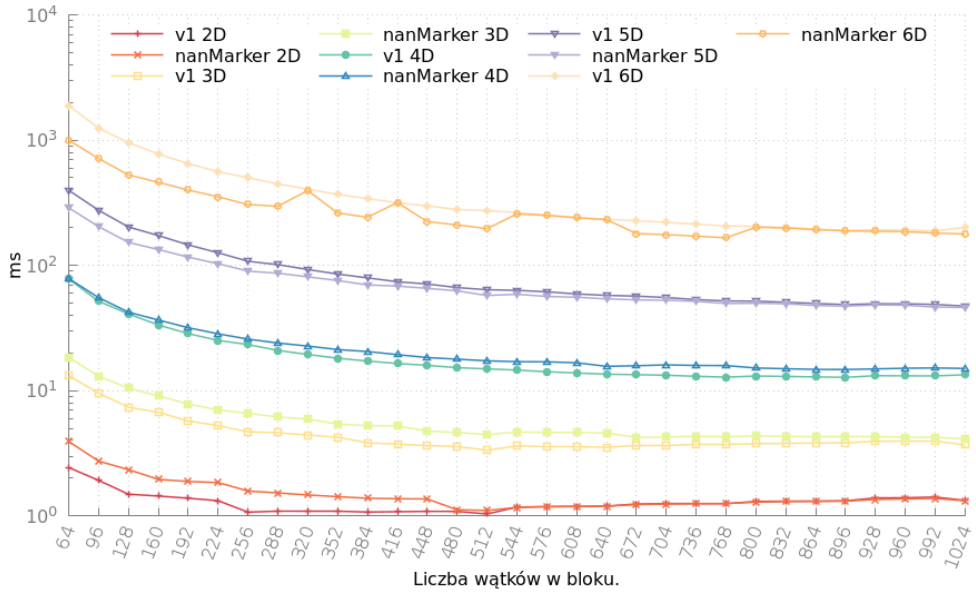
Rysunek 3.9: Instrukcje ptx wygenerowane dla (a) kodu Adintez'a i (b) zmodyfikowanej jego wersji.

kosztuje nas 4 instrukcje, które mogą być z łatwością zastąpione zaledwie jedną, szybką instrukcją czytającą wartość rejestru specjalnego przeznaczenia. Otrzymana wartość służy do ustalenia offsetu, od którego dany splot wątków może bezpiecznie rozpocząć zapisywanie w tablicy S. Dzięki wykorzystaniu synchronicznej pracy wątków w ramach splotu i operacji atomowych, policzenie indeksu konkretnego wątku, a zarazem cała redukcja, może odbywać się bez synchronizacji wątków w ramach bloku.

Wykres 3.10 porównuje wydajność obydwu dotychczasowych wersji. Jak się okazuje, nowa realizacja jest nieznacznie wolniejsza. Dokładna analiza



(a)



(b)

Rysunek 3.10: Czas wykonania decymacji w zależności od wersji, liczby wątków w bloku i wymiaru danych. Wykres (a) przedstawia wyniki uzyskane na karcie GTX TITAN, zaś (b) na karcie GTX 750Ti. Zbiór wejściowy stanowi chmura 150 tys. 2-6 wymiarowych punktów, tworząca odcinek o zadanej długości. Dane w pamięci uporządkowane wierszowo.

i profilowanie ukazują przyczyny tego stanu. Przede wszystkim za każdym obiegiem głównej pętli decymacji musimy czytać cały zbiór *wybrańców* razem

z punktami oznaczonymi już jako do usunięcia. Tymczasem w pierwszej implementacji, nasz zbiór S zmniejsza się prawie z każdą iteracją wewnętrznej pętli, podobnie jak ilość przesuwanych danych. Ponadto ciągłe sprawdzanie, czy dany punkt nie został już przeznaczony do usunięcia, zwiększyło dywergencję wątków.

Niestety, decymacja jest powtarzana w głównej pętli detekcji grani. Z tego względu jej efektywność, pomimo niewielkiego nakładu pracy, ma istotny wpływ na ogólną wydajność programu. Dlatego też podjęta została kolejna próba optymalizacji. Jak to zwykle w takich sytuacjach bywa, najlepszą poprawę otrzymuje się poprzez istotną reorganizację pracy lub zmianę algorytmu. Z takiego przekonania zrodził się pomysł, by oprzeć obliczenia o macierz odległości pomiędzy każdą parą *wybrańców*. Uzasadniony on został faktem, że kalkulacja tejże macierzy nie pociąga za sobą żadnego dodatkowego nakładu pracy. W rzeczywistości nawet go zmniejsza, ponieważ w dotychczasowej wersji, szukając sąsiadów danego *wybrańca*, wielokrotnie liczyliśmy dystans pomiędzy tymi samymi parami punktów.

O wyznaczaniu macierzy odległości wspominaliśmy już na wstępie tego rozdziału, przy okazji omówienia podobieństwa naszego algorytmu do znajdowania k najbliższych sąsiadów. Spośród wymienionych artykułów, praca Vincent'ego Garcia [52] okazuje się dla nas w obecnej chwili najbardziej interesująca, ze względu na macierzowe podejście do problemu. Autorzy publikacji zaproponowali następujący wzór na wyznaczenie kwadratu odległości euklidesowej:

$$\rho^2(x, y) = (x - y)^\top (x - y) = \| x \|^2 + \| y \|^2 - 2x^\top y \tag{3.7}$$

gdzie $\| . \|$ oznacza normę euklidesową, a x i y są wielowymiarowymi wektorami. Chcąc wyznaczyć macierz odległości pomiędzy punktami ze zbioru odniesienia i zbioru zapytań, powyższy wzór przyjmuje postać:

$$\rho^2(R, Q) = N_R + N_Q - 2R^\top Q \tag{3.8}$$

Macierze N_R i N_Q mają wszystkie elementy, w odpowiednio wierszach i kolumnach, równe $\|r_i\|^2$ i $\|q_j\|^2$. Z powyższej formuły, bibliotekę CUBLAS można wykorzystać do obliczenia jedynie $2R^\top Q$. Pozostałe operacje wymagają oddzielnych

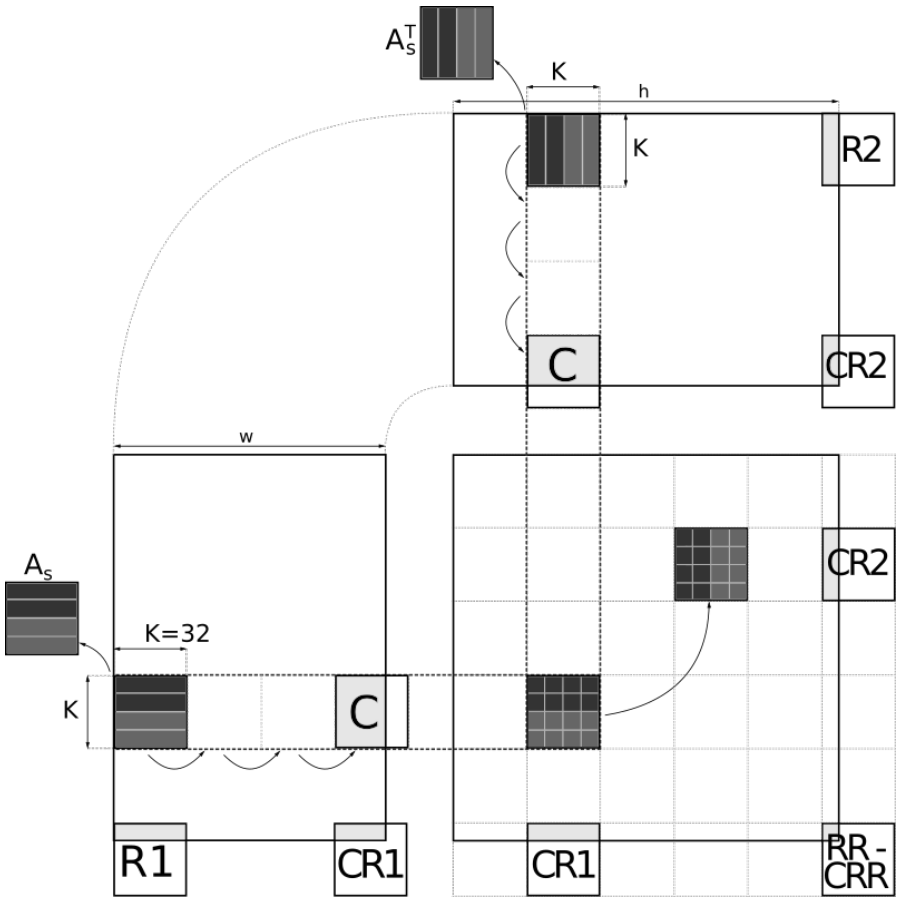
funkcji jądra. Adaptując rozwiązanie do naszych zapotrzebowań otrzymujemy wyrażenie:

$$\rho^2(S,S) = N_S + N_S^\top - 2S^\top S, \tag{3.9}$$

którego rezultatem jest symetryczna macierz. Niestety jedynie mnożenie macierzy daje tutaj możliwość wydajnej implementacji. Zwróćmy jednak uwagę, że wyznaczenie macierzy odległości można wykonać w jednej operacji, schematycznie podobnej do mnożenia macierzy. Dlatego też, napisana została własna implementacja tej czynności. Wykorzystano w niej efektywny algorytm zaproponowany przez Volkova [26, 54], umożliwiający zachowanie wysokiej lokalności danych. Polega on na sumowaniu odpowiednich macierzy, powstałych jako wynik iloczynu macierzowego pierwszego rzędu k-tej kolumny R przez k-ty wiersz Q. W naszej sytuacji będziemy mnożyć k-tą kolumnę S przez tę samą, ale transponowaną kolumnę. Co więcej, ponieważ wynikowa macierz jest symetryczna, wystarczy, że obliczymy jej macierz trójkątną dolną lub górną i przepiszemy wartości w odpowiednie pozycje.

Nasza nowa funkcja jądra dzieli wynikową macierz $SS^\top$, dla wierszowego układu S, a $S^\top S$ dla kolumnowego, na kwadratowe „kafelki" obliczane przez pojedynczy blok wątków. Każdy z nich ma wymiar 32x32, ponieważ wynik jest kwadratową, symetryczną macierzą, a także w celu zapewnienia wydajnej pracy wątków w ramach splotu. Taki sam rozmiar kafelka jest również wykorzystany przy poruszaniu się po odpowiednich „kafelkowych" wierszach, czy kolumnach wejściowych macierzy. Umożliwia on zaimplementowanie wydajnych, zgrupowanych dostępów do pamięci globalnej. Pojedynczy blok wątków jest dwuwymiarowy, jednak musi on spełniać pewne ograniczenia, sprawdzane na etapie kompilacji. Wymiar x powinien być równy szerokości kafelka. Natomiast wymiar y musi być całkowicie dzielić wysokość kafelka. Uruchamiana dwuwymiarowa sieć wątków ma tak dobierane rozmiary by zapełnić wszystkie wieloprocesory karty graficznej. Ponadto, jej niezależność od wielkości danych wejściowych pozwala na przetwarzanie danych dowolnego formatu, poprzez poruszanie się po wynikowej macierzy.

Rysunek 3.11 prezentuje schemat pracy pojedynczego bloku wątków. Każdy z uruchomionych wątków w bloku ma za zadanie wyznaczenie T_h/B_h



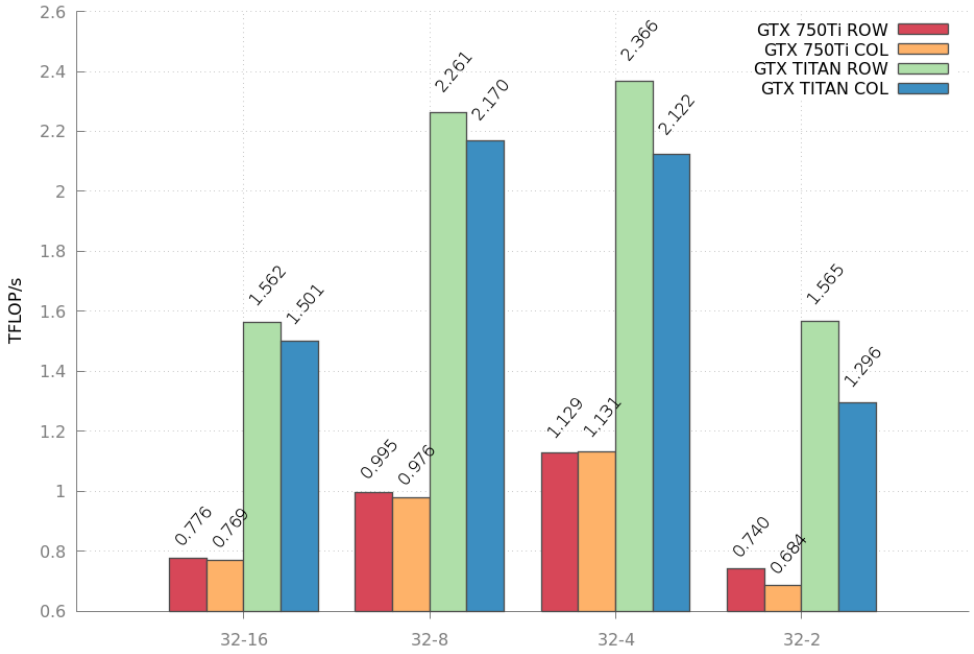
Rysunek 3.11: Schemat pracy pojedynczego bloku wątków, w funkcji jądra wyznaczającej symetryczną macierz odległości. Dane wejściowej macierzy uporządkowane wierszowo.

elementów w odpowiedniej kolumnie wynikowego kafelka, gdzie T_h jest wysokością kafelka, a B_h wymiarem y bloku wątków. Przed rozpoczęciem obliczeń, buforujemy w pamięci współdzielonej potrzebne nam dane. Współrzędne punktów z kafelka macierzy S są wykorzystywane przez wszystkie wątki, toteż w ten sposób wielokrotnie zmniejszamy liczbę dostępów do pamięci globalnej. Natomiast dane z kafelka macierzy $S^\top$ są używane tylko jednokrotnie. Jednakże ten zabieg ma na celu zagwarantowanie zgrupowanych odczytów z pamięci globalnej. Zwróćmy jeszcze uwagę, że w jednej transakcji splot wątków czyta całą kolumnę wejściowego kafelka danych, i w takim samym układzie zapisuje ją w pamięci współdzielonej. Taki wzorzec powodowałby 32-krotny konflikt banków, którego unikamy poprzez sztuczne rozszerzenie liczby kolumn kafelka w pamięci współdzielonej [26].

Oczywiście mało prawdopodobne jest by macierz S miała rozmiary będące wielokrotnością rozmiaru naszego kafelka. Ponieważ ciągłe sprawdzanie zakresów jest mało wydajnym rozwiązaniem, zastosowano nieco inne podejście. Mianowicie zawsze najpierw wykonujemy obliczenia na możliwie maksymalnej liczbie pełnych kafelków, a następnie stosujemy odpowiedni specjalizowany wariant dla danego przypadku brzegowego. Na rysunku 3.11, w przykładowych miejscach, zostały oznaczone różnymi symbolami poszczególne możliwe warianty. Spośród nich możemy pominąć CR2, ze względu na symetryczną wynikową macierz.

Obliczenia wykonują tylko bloki przypadające na kafelki macierzy trójkątnej dolnej. Zatem muszą one zapisać obliczone odległości do odpowiedniego kafelka macierzy trójkątnej górnej. Wyznaczony przez pojedynczy splot wątków wiersz staje się kolumną. Jednakże zapis danych w takim układzie byłby wysoce niewydajny, gdyż każdy wątek zapisywałby dane do bardzo odległych od siebie komórek pamięci, z których każda odpowiada innemu wierszowi wynikowej macierzy odległości. W związku z tym przed zapisem odpowiednio transponujemy nasz wynikowy kafelek do pamięci współdzielonej i stamtąd realizujemy efektywne dostępy do pamięci globalnej. Oczywiście kafelki leżące na diagonali nie wykonują żadnych dodatkowych zapisów. Na wykresie 3.12 przedstawiamy osiągniętą w testach wydajność.

Nowe podejście do decymacji, składa się z trzech etapów zaprezentowanych w algorytmie 12 i wymaga dwóch dodatkowych tablic: `distMtx` jest dwuwymiarową tablicą reprezentującą macierz odległości pomiędzy każdą parą punktów ze zbioru S, a `mask` jest jednowymiarową tablicą zawierającą tyle elementów ile jest *wybrańców*. Jej wartości stanowią maskę, decydującą o usunięciu, czy też pozostawieniu odpowiedniego punktu. Podobnie jak w pierwszej wersji uruchamiamy tylko jeden blok wątków. Na początku wykorzystujemy dynamiczną równoległość, by aktywować nową, odpowiednich rozmiarów, sieć wątków, do szybkiego obliczenia macierzy odległości. Kolejny etap, jak widzimy, ideowo jest bardzo podobny do poprzednich wersji algorytmu, z tym że pracujemy na macierzy odległości, a nie zbiorze *wybrańców*. Ta różnica pozwala usprawnić kod funkcji zliczającej sąsiadów. Ponieważ tym razem nie wczytujemy współrzędnych każdego punktu, tylko pojedyncze wartości, nie musimy już korzystać



Rysunek 3.12: Wydajność funkcji jądra wyznaczającej symetryczną macierz odległości w zależności od konfiguracji bloku wątków. Rozmiar testowanej macierzy 800x800.

z pamięci współdzielonej, by zapewnić odpowiedni wzorzec dostępu do pamięci globalnej, a w konsekwencji z synchronizacji wątków w ramach bloku. Poza tym, tak samo jak wcześniej, korzystamy z kafelkowania i potokowego przetwarzania danych. Na koniec wykonywane jest faktyczne usuwanie punktów. Ma ono identyczny przebieg jak w drugiej wersji z etykietowaniem flagą `NaN`. Wykres 3.13 przedstawia porównanie wydajności wszystkich, dotychczasowych trzech wersji decymacji.

Tym razem wysiłek nie poszedł na marne i odnotowujemy przyśpieszenie w porównaniu do pierwszej wersji od danych 4-o wymiarowych. Przewaga rośnie wraz z liczbą kardynalną zbioru *wybrańców*, jak i z wymiarem danych.

3.5. Całość

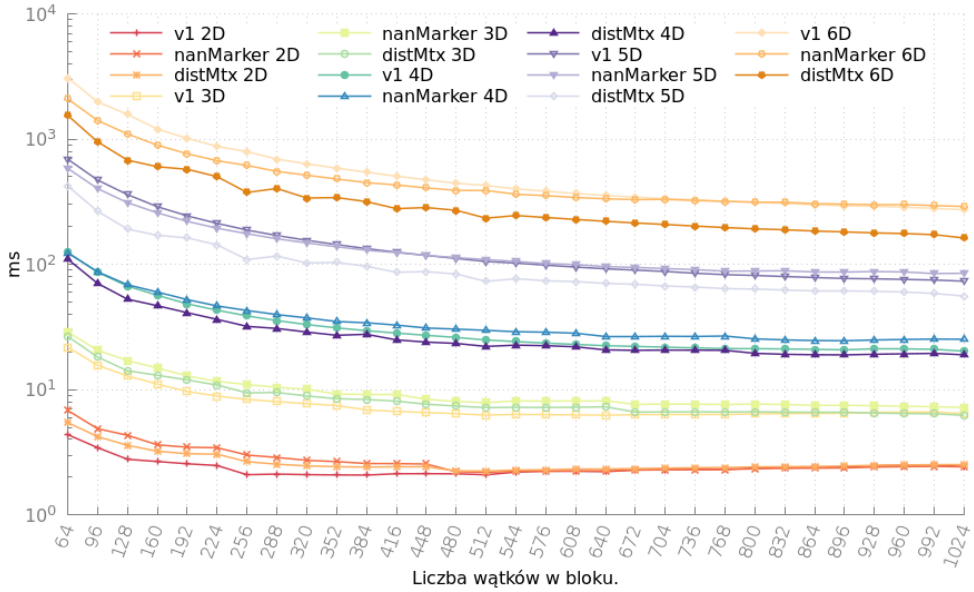
Omówiliśmy już poszczególne etapy w wersji siłowej, zatem pozostało tylko połączenie ich w całość. Na wstępie tego rozdziału na algorytmie 1 przedstawiliśmy kolejno wykonywane kroki. Spośród nich, obliczenia w ramach inicjalizacji zbioru wybrańców, ewolucji i decymacji zostały przeniesione na kartę

```
1  lastCount ← 0
2  currCount ← |S|
3  if threadIdx.x = 0 then
4      distMtx ← Macierz odległości                                  ▷ Etap 1
5  Synchronizuj wątki
6  while lastCount ≠ currCount and currCount > 3 do                  ▷ Etap 2
7      lastCount ← currCount
8      for i = 0 to |S| do
9          Synchronizuj wątki
10         if mask_i == 1 then
11             neighbours ← liczba sąsiadów w promieniu R_2 punktu s_i
12             if neighbours ≥ 4 then
13                 currCount ← currCount − 1
14                 if threadIdx.x = 0 then
15                     mask_i ← 0
16                 if currCount < 3 then
17                     break
18                 else
19                     continue
20             neighbours ← liczba sąsiadów w promieniu 2R_2 punktu s_i
21             if neighbours ≤ 2 then
22                 currCount ← currCount − 1
23                 if threadIdx.x = 0 then
24                     mask_i ← 0
25                 if currCount < 3 then
26                     break
27 Redukuj punkty oznaczone 0 w tablicy mask                          ▷ Etap 3
```

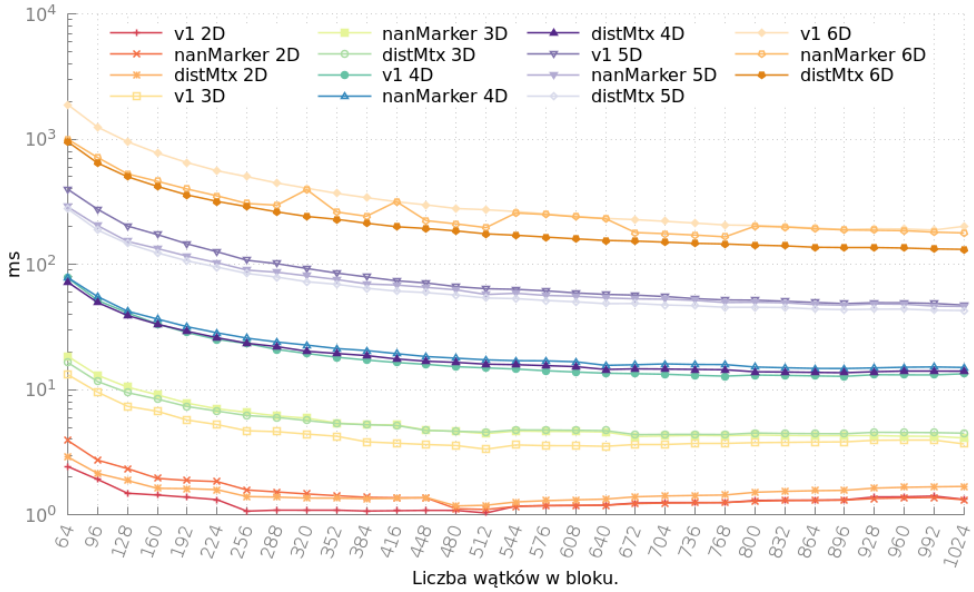
Algorytm 12: Usuwanie nadmiarowych *wybrańców* oparte o macierz odległości.

graficzną. Nadzorowanie pracy może odbywać się z poziomu gospodarza, bądź urządzenia. Jak to już zostało wspomniane, dynamiczna równoległość wymaga kart o współczynniku zdolności obliczeniowych ≥ 3.5, a nie wszystkie należące do rodziny Kepler takowy posiadają. Z tego względu na podstawie odpowiednich makr udostępnianych przez kompilator `nvcc` zostaje podjęta stosowna decyzja, którą implementację wybrać.

Na wykresie 3.14 prezentujemy uzyskane w testach przykładowe rezultaty. W etykietach, po nazwie urządzenia widnieje informacja o układzie danych w pamięci, mająca formę „X-Y”, gdzie X oznacza dane wejściowe (punkty chmury i niektóre tymczasowe tablice), a Y określa dane wyjściowe (punkty należące do zbioru S). W pojedynczej iteracji, z poziomu CPU, notowany był czas trwania detekcji grani, oprócz ostatniego nie przeprowadzanego etapu



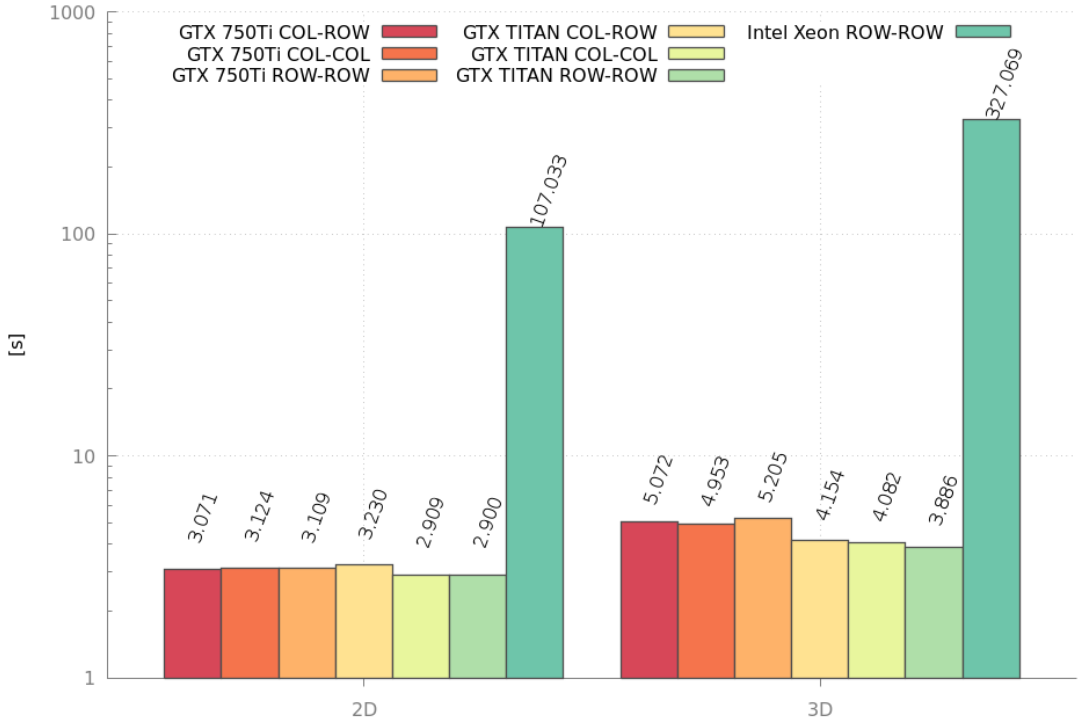
(a)



(b)

Rysunek 3.13: Czas wykonania decymacji w zależności od wersji, liczby wątków w bloku i wymiaru danych. Wykres (a) przedstawia wyniki uzyskane na karcie GTX TITAN, zaś (b) na karcie GTX 750Ti. Zbiór wejściowy stanowi chmura 150 tys. 2-6 wymiarowych punktów, tworząca odcinek o zadanej długości. Dane w pamięci uporządkowane wierszowo.

porządkowania wybrańców. Liczone także były takie miary jakości jak minimum, maksimum, mediana, wartość średnia odległości od idealnej krzywej, a także metryka Hausdorffa [21]. Ponadto, w przypadku wykonania na GPU,



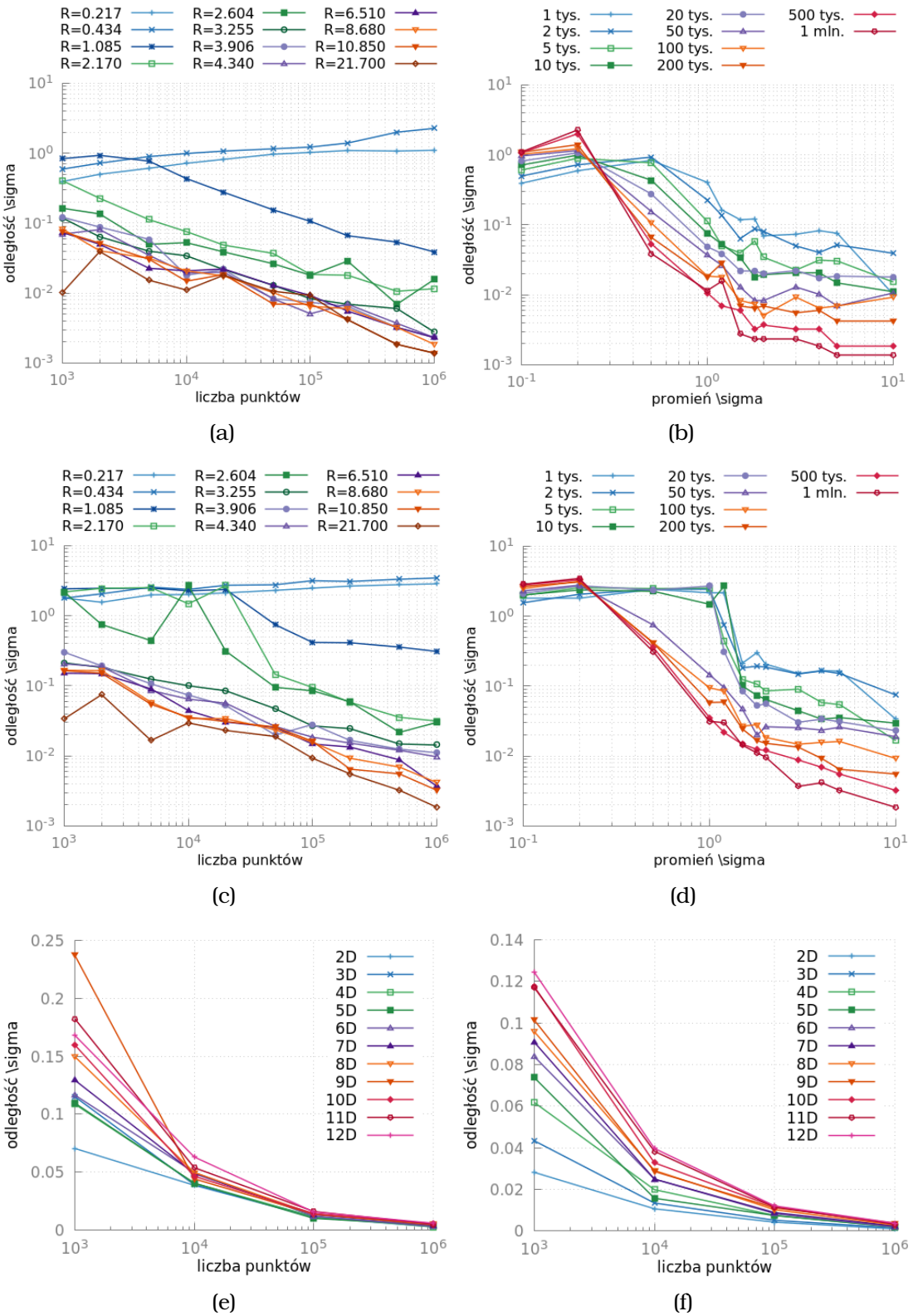
Rysunek 3.14: Średni czas pracy algorytmu detekcji grani z parametrami $R1 = 8,340$ i $R2 = 16,680$, przeprowadzonej na chmurze 100 tys. punktów, będącej dwu-, lub trójwymiarową spiralą, zaburzoną szumem o rozkładzie normalnym z wartością średnią 0 i odchyleniem standardowym 4.17.

odpowiednie dane znajdowały się już w pamięci urządzenia. Toteż w praktycznym wykorzystaniu należy mieć na uwadze dodatkowe opóźnienie związane z przesłaniem danych do i z karty graficznej. Szczegółowe porównanie czasów wykonania, wszystkich wersji detekcji grani, znajduje się na końcu rozdziału 4.

Zanim przystąpimy do omówienia dokładności rekonstrukcji krzywej, zwróćmy uwagę jeszcze na jeden fakt. Mianowicie, równoległa implementacja algorytmu detekcji grani, powoduje, że generowane wyniki są niedeterministyczne. Wynika to z wykorzystania operacji atomowych do akumulacji pomocniczych danych służących do wyznaczenia odpowiedniego środka ciężkości, gdyż nie możemy zapewnić kolejności ich realizacji.

Jak widzimy na wykresach przedstawionych na rysunku 3.15, jakość aproksymacji rośnie wraz z liczbą punktów chmury, co wydaje się naturalne, gdyż zwiększa się tym samym ilość informacji o szukanej krzywej. Ponadto,

wraz z wzrostem wielkości promienia w stosunku do wartości odchylenia standardowego dodanego szumu, również otrzymujemy lepszą aproksymację. Aczkolwiek w rzeczywistych zastosowaniach, przybliżając znacznie bardziej skomplikowane, niż testowy odcinek prosty, krzywe, wielkość promienia nie powinna być zbyt duża, gdyż będzie to prowadziło do coraz mniej dokładnego oszacowania.



Rysunek 3.15: Jakość aproksymacji grani dwuwymiarowej chmury punktów tworzącej odcinek prosty o długości 100, zaburzonej szumem o rozkładzie normalnym, z odchyleniem standardowym równym sigma. Wykresy (a) i (b) przedstawiają medianę odległości wynikowej krzywej od odcinka, zaś (c) i (d) odległość Hausdorffa. Wykresy (e) i (f) prezentują odpowiednio odległość Hausdorffa i medianę odległości, w zależności od wymiaru danych i ilości punktów, dla ustalonego parametru $R1$.

4. Wersja „kafelkowa”

Dotychczasowa implementacja detekcji grani świetnie wpasowuje się w model wykonania technologii CUDA, dzięki ogromnej ilości niezależnej od siebie pracy. Pomimo kilku etapów charakteryzujących się małą intensywnością obliczeń, odpowiednia dystrybucja pracy i optymalizacja dostępów do pamięci globalnej, pozwoliły uzyskać, w całości, znaczące przyśpieszenie w stosunku do wersji wielowątkowej na CPU. Jednakże podejście „siłowe” jest dalekie od ideału. Wiedząc, że interesują nas jedynie punkty leżące w zadanym promieniu, wyznaczanie odległości od danego *wybrańca* s_i do każdego punktu chmury p_i, jest absolutnie nadmiarowe. Dlatego też w tym rozdziale przeanalizowana zostanie nowa implementacja, częściowo rozwiązująca ten problem.

W celu zmniejszenia liczby wykonywanych kalkulacji, dokonujemy podziału przestrzeni zamykającej analizowaną chmurę punktów na wielowymiarowe „kafelki”. Jest to powszechnie stosowane podejście w grafice komputerowej do detekcji kolizji [27]. Również w sytuacjach, gdy badane przez nas zjawisko możemy przybliżyć z dużą dokładnością, mając do dyspozycji jedynie informacje o elementach znajdujących się w bliskim otoczeniu – np. symulacje N-ciał [27, 56], czy też analiza molekularna [16, 20, 46].

Tytułowy „kafelek”, w ogólności może być dowolną bryłą ograniczającą, jednakże ze względów praktycznych przyjmujemy pewne ograniczenia, które pozwolą uprościć implementację. Załóżmy zatem, że „kafelek” jest n-wymiarowym prostopadłościanem. Po segmentacji przestrzeni na tak określone części, możemy w każdej z nich indywidualnie przeprowadzić detekcję grani. Przejdźmy zatem do omówienia szczegółów tej metody.

4.1. Implementacja na CPU

Nasze rozważania rozpoczniemy od opracowania rozwiązania realizowanego na CPU, jako wstępne zbadanie tematu, przed przeniesieniem implementacji na kartę graficzną.

4.1.1. Budowa hierarchicznej struktury drzewiastej

Zanim przystąpimy do partycjonowania przestrzeni, przedstawimy kilka możliwych wariantów kafelków i związane z nimi struktury danych.

Kafelki *lokalne* są najprostszą i podstawową formą, z której wywodzą się pozostałe. Ich główną cechą jest „lokalność”, tzn. brak jakichkolwiek informacji o swoim otoczeniu. Reprezentowane są one przez następującą strukturę danych:

```
template <typename T>
struct LocalSamplesTile {
  int id;
  size_t pointsCnt;
  size_t chosenPointsCnt;
  size_t chosenPointsCapacity;
  size_t dim;
  std::list<T*> cpList;
  std::vector<T> samples;
  std::vector<T> chosenSamples;
  BoundingBox<T> bounds;
};
```

Posiada ona m.in. takie parametry jak liczba punktów zawierających się w kafelku, wymiar danych, czy liczba *wybrańców*; Jak widzimy, współrzędne punktów chmury i *wybrańców* przechowywane są w postaci wektorów. Składowa `cpList` jest listą wskaźników do elementów tablicy `chosenSamples`, której zastosowanie ma miejsce w trakcie decymacji. Znajduje się tutaj jeszcze obiekt typu `BoundingBox` opisujący obejmowany obszar przestrzeni, a także maksymalna liczba *wybrańców* mogąca się wewnątrz kafelka zmieścić.

Kafelki *grupowe* dziedziczą po kafelkach lokalnych, wnosząc dodatkowo pojęcie *sąsiadów*, dzięki któremu posiadają informacje o ich otoczeniu. Znaczenie tego terminu zależy od specjalizacji klasy. Mianowicie dzielimy kafelki

grupowe na zwykłe i rozszerzone[1]. W pierwszym przypadku sąsiedzi są wektorem wskaźników do bezpośrednio graniczących z nimi kafelków. Natomiast w drugim przypadku sąsiedzi są wektorem zawierającym współrzędne punktów chmury znajdujących się w bliskim otoczeniu kafelka.

1 $gBBox \leftarrow$ granice przestrzeni zawierającej punkty chmury
2 $hist \leftarrow$ przestrzenny histogram punktów
3 $tiles \leftarrow$ CREATETILES($P, hist, gBBox, initTileCnt$)
4 **for all** $t_i \in tiles$ **do**
5 ADDNODE($root, t_i, 1$)
6 COLLECTLEAFS( )

Algorytm 13: Budowa drzewa kafelków lokalnych

Algorytm 13 przedstawia kolejne kroki wykonywane w trakcie budowy drzewa kafelków lokalnych. Podział przestrzeni rozpoczynamy od znalezienia jej granic. W tym celu iterujemy po wszystkich punktach chmury szukając minimum i maksimum dla każdego wymiaru. Następnie wyznaczamy przestrzenny histogram, dający nam informację ile punktów chmury wpada do danego kafelka, którego obliczenie wymaga znajomości wymiarów przestrzeni ($gBBox$) i liczby kafelków na które dzielimy dany wymiar ($initTileCnt$). Jest to ważny parametr, mający istotny wpływ na czas budowania drzewa, jak i ogólnie na algorytm detekcji grani. Niestety nie ma sposobu na automatyczne, optymalne dobranie jego wartości, gdyż jest on mocno uwarunkowany danymi. Gruboziarnisty początkowy podział może skutkować w wielokrotnym, hierarchicznym podziale przestrzeni, a co za tym idzie wielokrotnym przenoszeniem danych. Z drugiej strony podział zbyt drobnoziarnisty może źle wpływać na jakość detekcji grani. W konsekwencji ciężar wyboru ziarnistości został przeniesiony na użytkownika programu. Obliczanie histogramu opiera się o mapowanie współrzędnych punktu na d-wymiarowe współrzędne kafelka zgodnie ze wzorem 4.1, a następnie ich linearyzacja przy użyciu formuły 4.2, w celu otrzymania odpowiedniego indeksu kubełka histogramu.

$$tileIdx_i = \left\lfloor \frac{\mid p_i - bbox_{min_i} \mid \cdot tileCnt_i}{bbox_{w_i}} \right\rfloor \tag{4.1}$$

[1] W dalszej części pracy przyjmujemy, że kafelki „grupowe" odpowiadają kafelkom „grupowym zwykłym", zaś kafelki „rozszerzone" – „grupowym rozszerzonym".

$$linIdx = k_0 + \sum_{i=1}^{dim-1} k_i \prod_{j=i-1}^{0} tileCnt_j \tag{4.2}$$

gdzie,

k_i – i-ta współrzędna kafelka

p_i – i-ta współrzędna punktu

$bbox_{min_i}$ – dolna granica i-tego wymiaru kafelka

$bbox_{w_i}$ – szerokość kafelka w i-tym wymiarze

$tileCnt_i$ – liczba kafelków w i-tym wymiarze

```
1  function CreateTiles(P, hist, bbox, tileCnt)
2      for i = 0 to |tileCnt| do
3          tiles[i] ← Initialize(hist)
4      for all p_i ∈ P do
5          idx ← GetTileIdx(p_i, tileCnt, bbox)
6          tiles[idx] ← tiles[idx] ∪ p_i
7      for all tile_i ∈ tiles do
8          Usuń kafelki puste i zawierające tylko 1 punkt
9          tile_i ← Inicjuj granice kafelka
10         Alokuj pamięć na wybrańców
11     return tiles
```

Algorytm 14: Inicjalizacja kafelków lokalnych

Kolejnym etapem budowy drzewa jest inicjalizacja pierwszego poziomu kafelków, realizowana przez wywołanie funkcji CreateTiles (por. algorytm 14). Przyjmuje ona jako parametry zbiór punktów P, a także listę $tileCnt$, określającą na ile kafelków ma zostać podzielony dany wymiar. Wartości histogramu służą do zaalokowania odpowiedniej ilości pamięci potrzebnej do przechowania punktów zawierających się w ramach kafelka. Aktualna implementacja detekcji grani wymaga, by dane wejściowe znajdowały się w spójnym obszarze pamięci. Z tego względu, w tym momencie kopiujemy punkty do odpowiednich tablic. Używamy przy tym tej samej metody mapowania współrzędnych punktu na liniowy indeks kafelka, co przy histogramie. Następnie ustalamy granice danego kafelka, trawersując przez jego punkty i alokujemy pamięć na *wybrańców*. Dotychczas, na zbiór *wybrańców* rezerwowaliśmy tyle samo pamięci co na zbiór punktów chmury, gdyż nie mogliśmy w żaden sposób z góry przewidzieć początkowej ilości *wybrańców*. Tym razem, znając ramy przestrzenne kafelka jesteśmy w stanie oszacować ich maksymalną liczbę stosując wzór 4.3, gdzie

$R1$ jest parametrem algorytmu detekcji grani, regulującym wybór początkowego zbioru wybrańców.

$$chosenPointsCapacity = \prod_{i=0}^{dim-1} \left\lceil \frac{bbox_{w_i}}{R1} \right\rceil \tag{4.3}$$

```
1  function AddNode(parent, tile, treeLevel)
2      if pointsCnt_tile < maxTileCapacity then
3          node ← Nowy węzeł zawierający tile
4          Podepnij node do parent
5      else
6          node ← Nowy węzeł
7          Podepnij node do parent
8          subTiles ← Subdivide(tile, treeLevel)
9          Usuń tile
10         for all t_i ∈ subTiles do
11             AddNode(node, t_i, treeLevel + 1)
```

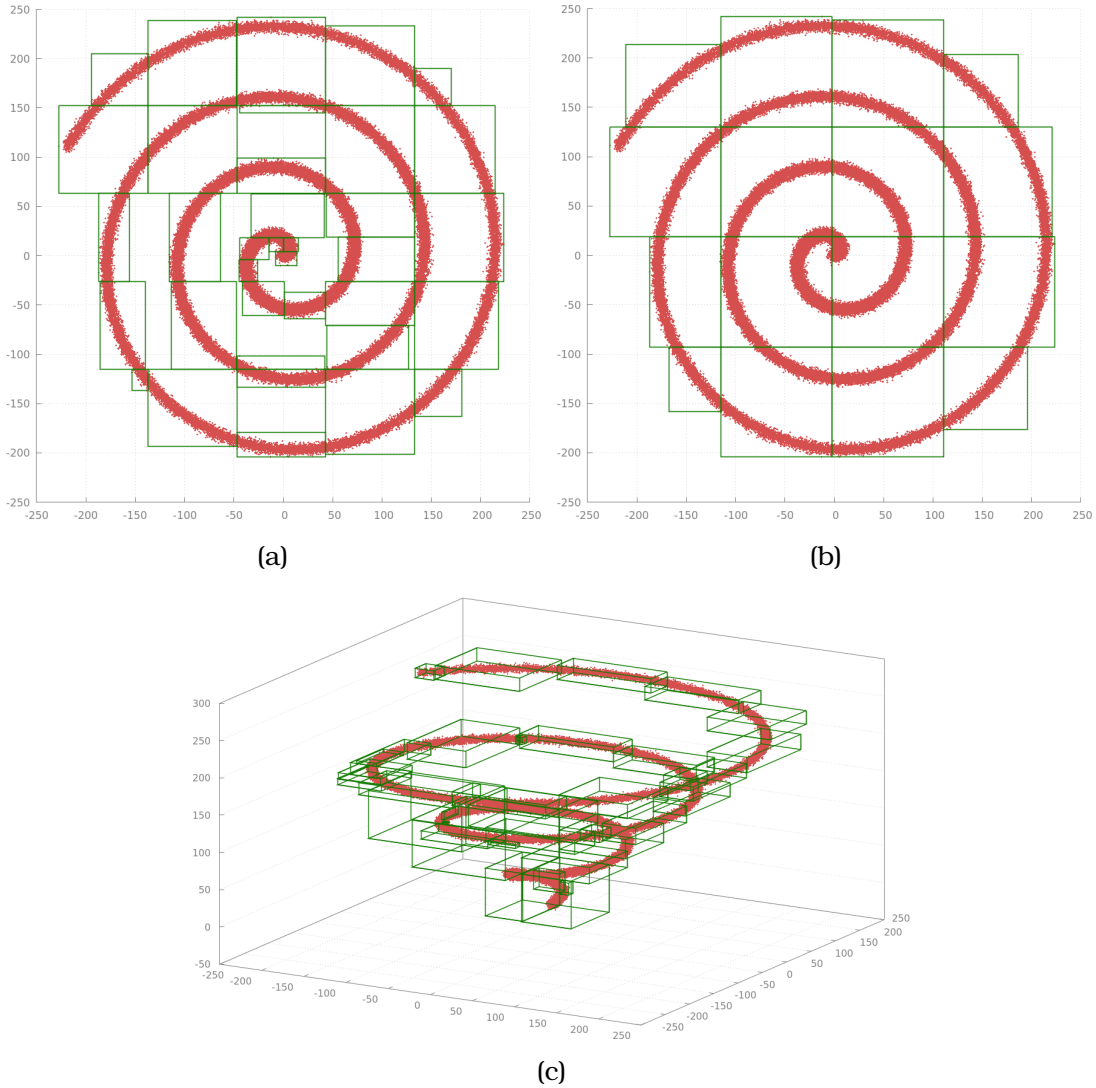
Algorytm 15: Dodawanie do drzewa nowych węzłów

W dalszej kolejności, utworzone przed chwilą kafelki, zostają jeden po drugim dodawane do drzewa przy użyciu rekurencyjnej funkcji AddNode zaprezentowanej jako algorytm 15. Operacja „Podepnij" polega na zapisaniu adresu węzła–dziecka w odpowiedniej składowej węzła–rodzica. Zanim jednak kafelek zostanie dodany, sprawdzamy czy przechowywana przezeń liczba punktów, nie przekracza zadanego maksymalnego limitu. W przypadku, gdy punktów jest zbyt dużo, dokonujemy przestrzennego podziału kafelka na dwie połowy (funkcja Subdivide algorytmu 16. Rozdzielenie następuje wzdłuż zmieniającego się wraz z poziomem drzewa, wymiaru. Dekompozycja przeprowadzana jest w sposób analogiczny do opisanej wcześniej początkowej segmentacji przestrzeni i korzysta z tych samych funkcji (CreateTiles). Podzielony kafelek zostaje usunięty, żeby zwolnić niewykorzystywane zasoby.

```
1  function Subdivide(tile, treeLevel)
2      tileCnt ← [1, 1]
3      tileCnt[treeLevel % tile_dim] ← 2
4      hist ← przestrzenny histogram punktów
5      subTiles ← CreateTiles(P_tile, hist, bbox_tile, tileCnt)
6      return subTiles
```

Algorytm 16: Podział kafelka na dwie części

Po zakończeniu budowy drzewa, tworzymy jeszcze listę zawierającą wskaźniki do węzłów liści. W ten sposób, chcąc wykonać obliczenia, nie będziemy musieli później za każdym razem trawersować całego drzewa. Ponadto taka forma dostępu do liści jest lepiej dostosowana do przetwarzania równoległego z wykorzystaniem biblioteki OpenMP.



(a) (b)

(c)

Rysunek 4.1: Przykładowy wynikowy podział przestrzeni danych na kafelki lokalne dla różnych parametrów budowy drzewa. Na rysunkach (a) i (b) przedstawiony został podział odpowiednio drobnoziarnisty i gruboziarnisty dwuwymiarowej chmury punktów, zaś na rysunku (c) podział drobnoziarnisty trójwymiarowej chmury punktów.

Na rysunku 4.1 prezentujemy przykładowe rezultaty budowy drzewa, składającego się z kafelków lokalnych. Jak widać, podział przestrzeni może być

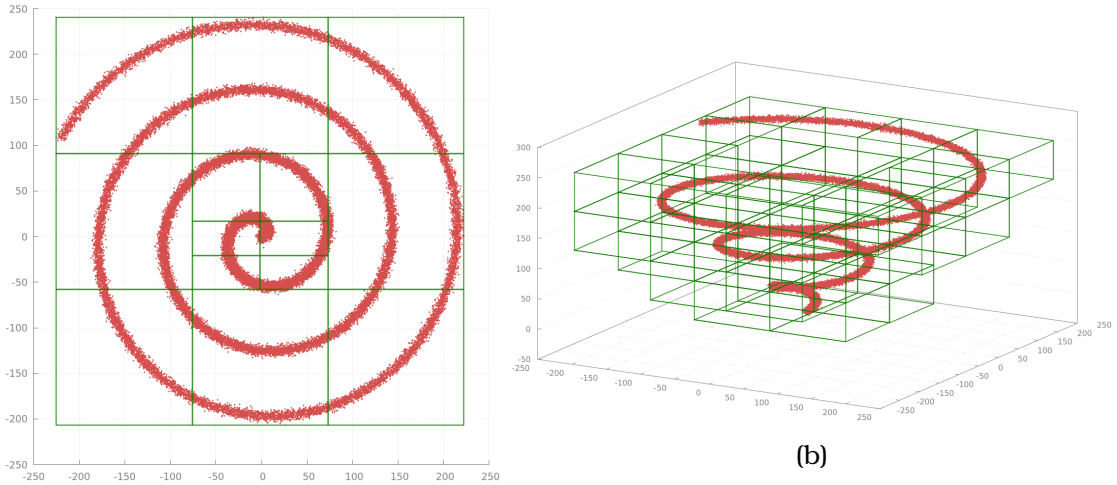
bardzo różny. Zwróćmy uwagę, że na rysunkach 4.1(a) i 4.1(c) nie zawsze stykają się ze sobą sąsiadujące kafelki. Jest to wynikiem użycia histogramu do znalezienia granic danego kafelka, które w rezultacie są maksymalnie dopasowane do danych. Takie podejście nie sprawdzi się w przypadku kafelków grupowych, gdyż uniemożliwiłoby ono znalezienie sąsiadujących kafelków.

Budowa drzewa kafelków grupowych, odróżnia się niewykorzystywaniem histogramu, a także obliczaniem granic kafelków zamiast ich wyszukiwaniem. W ten sposób wykonujemy mniej przebiegów po całym zbiorze danych - tylko przy wyznaczaniu granic całej chmury punktów i przy kopiowaniu punktów do kafelków. Brak wiedzy o liczbie punktów w ramach poszczególnych kafelków, pociąga konieczność korzystania z dynamicznych struktur danych, takich jak wektory ze standardowej biblioteki C++. Oczywiście nie odbywa się to za darmo, gdyż za każdym razem po przekroczeniu aktualnej pojemności wektora, wykonywane jest nowa alokacja pamięci i przeniesienie danych. Ponieważ z góry znamy granice danego kafelka, podczas partycjonowania danych, nie musimy wykonywać mapowania współrzędnych, lecz wystarczy, że sprawdzimy zawieranie się punktu w obrębie danego kafelka. Ponadto w przypadku kafelków rozszerzonych zawsze dodatkowo badamy, czy dany punkt jest w ich bliskim sąsiedztwie. Precyzyjniej rzecz ujmując, czy znajduje się on w odległości równej parametrowi detekcji grani $R1$, od granic kafelka. Takie rozszerzenie jest wystarczające, ponieważ *wybrańcy* są selekcjonowani jedynie z podstawowego obszaru kafelka. Zatem nawet jak *wybraniec* będzie położony dokładnie na granicy kafelka, to do wyznaczenia środka ciężkości mogą być wzięte punkty maksymalnie odległe właśnie o wartość $R1$. Dla zwykłych kafelków grupowych, po zakończeniu procesu budowy drzewa, wykonujemy jeszcze poszukiwanie ich sąsiadów, poprzez sprawdzanie pokrywania się granic.

Na rysunku 4.2 pokazane zostały testowe wyniki. Zauważmy, że tym razem sąsiednie kafelki współdzielą ze sobą granice.

4.1.2. Realizacja detekcji grani w kafelkowej strukturze.

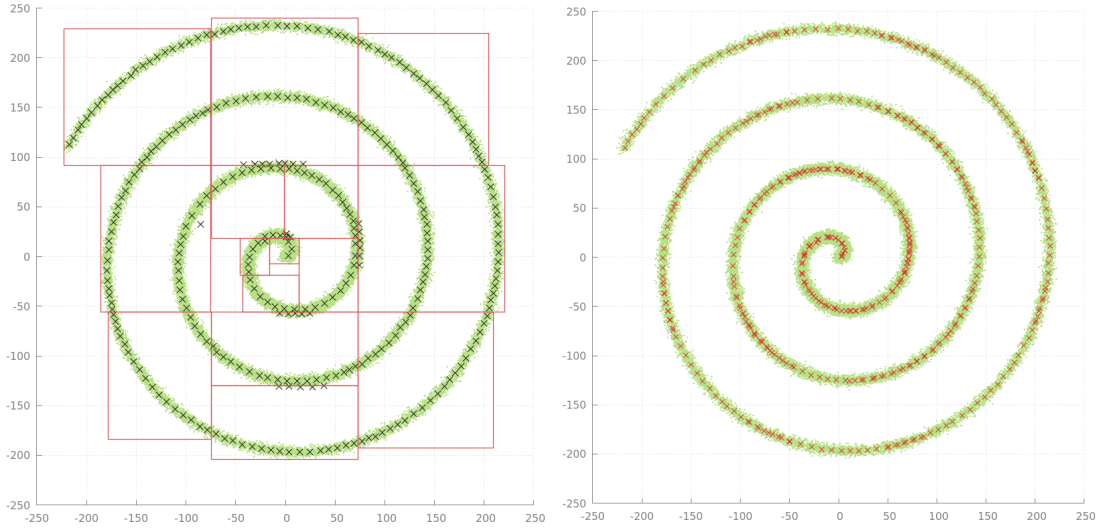
Mając gotowe, zbudowane drzewo kafelków, możemy przystąpić do detekcji grani. Ponieważ każdy z kafelków jest niezależny od innych, uruchamiamy



(a) (b)

Rysunek 4.2: Przykładowy wynikowy podział (a) – dwuwymiarowej i (b) – trójwymiarowej przestrzeni danych na kafelki grupowe.

obliczenia na wielu wątkach, przy użyciu dyrektyw OpenMP, z których każdy przetwarza jeden lub więcej kafelków.

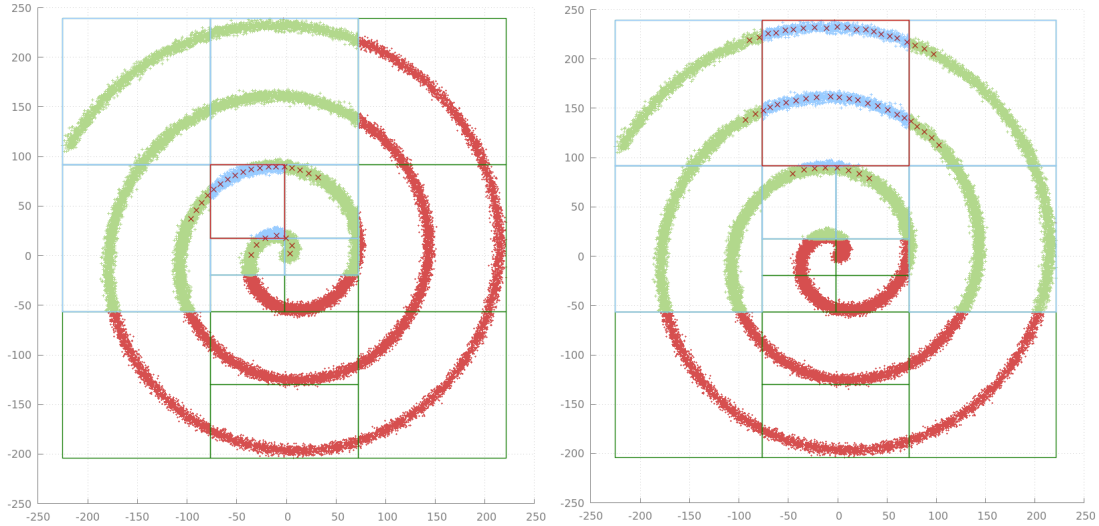


(a) (b)

Rysunek 4.3: Rezultaty lokalnej detekcji grani na kafelkach (a) lokalnych i (b) grupowych. Kolorem zielonym oznaczono punkty chmury, zaś czarnymi lub czerwonymi „x” ostateczny zbiór wybrańców.

Przyglądając się rysunkowi 4.3 spostrzegamy, że wynikowy zbiór zawiera duże ilości nadmiarowych *wybrańców*. Biorąc pod uwagę kafelki lokalne, ten fakt nie może dziwić, gdyż nie mają one żadnej informacji na temat swojego

otoczenia. Natomiast w odniesieniu do kafelków grupowych, to zjawisko staje się również oczywiste, po przeanalizowaniu wykresów na rysunku 4.4. Widzimy na nich, że w wyniku ewolucji *wybrańcy* danego kafelka rozsuwają się daleko na swoich sąsiadów.

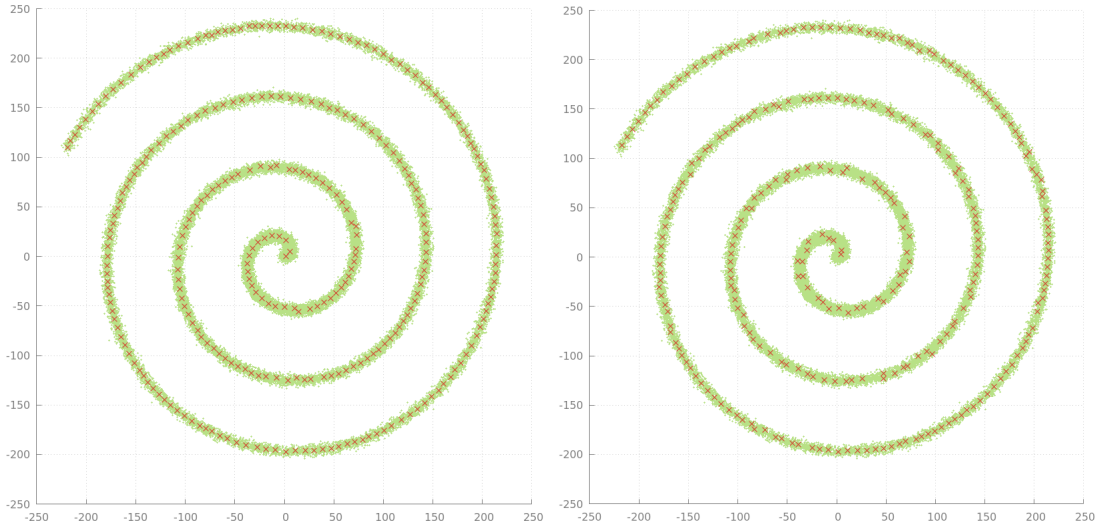


Rysunek 4.4: Pojedyncze grupowe kafelki z zaznaczonymi sąsiadami (niebieskie ramki, zielone punkty) wybranego kafelka (czerwona ramka) i *wybrańcami*(czerwone „x") w wyniku lokalnej detekcji grani.

Remedium na powyższe problemy może być dodanie końcowego etapu „wygładzania", polegającego na dodatkowej, globalnej iteracji ewolucji i decymacji. Wykresy na rysunku 4.5 prezentują uzyskane rezultaty. Niestety nie udało się uzyskać oczekiwanego efektu.

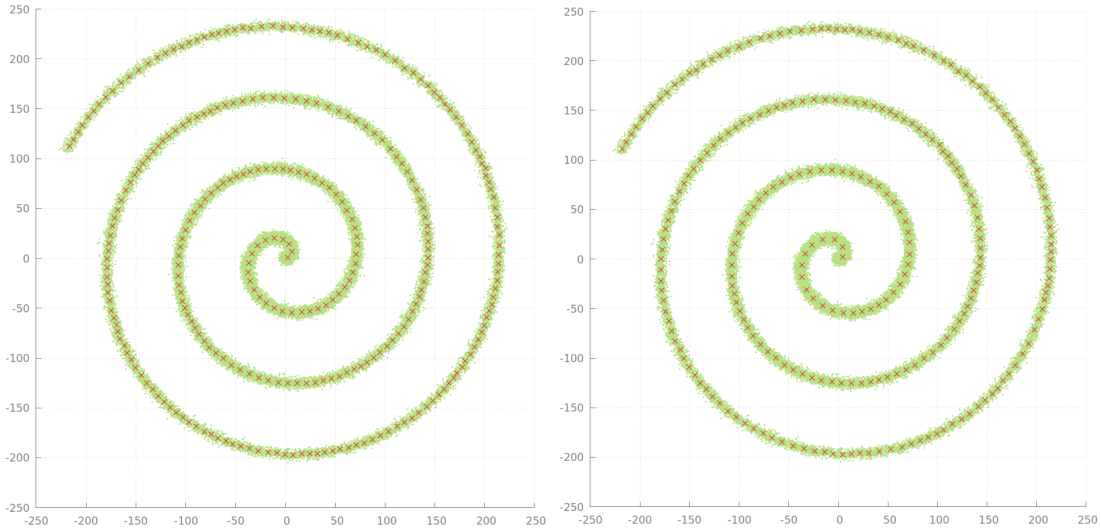
Spróbujmy jednak wykonać najpierw decymację, w celu usunięcia redundantnych *wybrańców*, a dopiero w drugiej kolejności ewolucję, która powinna rozprowadzić pozostałe punkty po aproksymowanej krzywej. Zaletą tej zmiany powinien być również szybszy czas wykonania, gdyż ewolucja będzie pracować na mniejszym zbiorze *wybrańców*. Przykładowe wyniki pokazane zostały na rysunku 4.6.

Tym razem osiągnięte efekty są zgodne z przypuszczeniami. Jednakże ten dodatkowy zabieg znacząco wydłuża czas całego algorytmu, zajmując ok. 70% czasu w przypadku kafelków lokalnych i ok. 25% w przypadku kafelków grupowych. W związku z tym powstał pomysł na hybrydową detekcję grani. Jej przebieg jest analogiczny do aktualnej wersji, z różnicą co do sposobu



(a) (b)

Rysunek 4.5: Rezultaty zastosowania dodatkowego etapu „wygładzania” (globalna ewolucja i decymacja), (a) dla kafelków lokalnych i (b) grupowych. Czerwonym kolorem oznaczono *wybrańców*.



(a) (b)

Rysunek 4.6: Rezultaty zastosowania dodatkowego etapu „wygładzania” (globalna decymacja i ewolucja), (a) dla kafelków lokalnych i (b) grupowych. Czerwonym kolorem oznaczono *wybrańców*.

wykonania decymacji. Mianowicie po zakończeniu lokalnej (wewnątrz każdego kafelka z osobna) ewolucji, decymacja realizowana będzie globalnie, biorąc pod uwagę wszystkie kafelki. Taki manewr powinien zapobiec efektom zachodzenia

na siebie *wybrańców* z sąsiednich kafelków, widocznego na dotychczasowych rysunkach.

Modyfikujemy zatem bieżącą wersję decymacji, tak by mogła ona pracować na liście list *wybrańców* z każdego kafelka, co umożliwi globalny zakres działań. Ponadto nie wykonujemy w tej chwili jeszcze żadnego usuwania danych, a jedynie operujemy na wskaźnikach do punktów, znajdujących się na listach. Jest to koniecznie do zachowania informacji z którego kafelka, które punkty zostały usunięte. Po zakończonym etapie globalnym, z każdego kafelka faktycznie usuwamy odpowiednie punkty z prywatnego wektora *wybrańców*.

Na początku tego podpunktu wspominaliśmy o jeszcze jednym rodzaju kafelków grupowych – tzw. kafelkach rozszerzonych. Ich idea jest kompromisem, pomiędzy zupełnym ignorowaniem otoczenia przez kafelki lokalne, a eksplorowaniem szerokiego sąsiedztwa przez zwykłe kafelki grupowe. Po wykorzystaniu kafelków rozszerzonych w połączeniu z hybrydową wersją detekcji grani, możemy się spodziewać najlepszych wyników. Rysunek 4.7 przedstawia porównanie rezultatów z i bez włączonego końcowego etapu „wygładzania".

Zastosowanie dodatkowego etapu wygładzania powoduje, że otrzymujemy porównywalne wyniki dla każdego rodzaju kafelka. Jednakże kafelki lokalne wciąż wymagają użycia końcowej fazy wygładzania. Natomiast kafelki grupowe i rozszerzone, zgodnie z oczekiwaniami, nawet bez niego dają bardzo dobre, zbliżone do siebie rezultaty. Przed ostatecznym wyborem najlepszej wersji spójrzmy jeszcze na zestawienie czasu wykonania poszczególnych wariantów zilustrowane na rysunku 4.8.

Oznaczenia legendy mają następujące znaczenie:

LT – kafelki lokalne

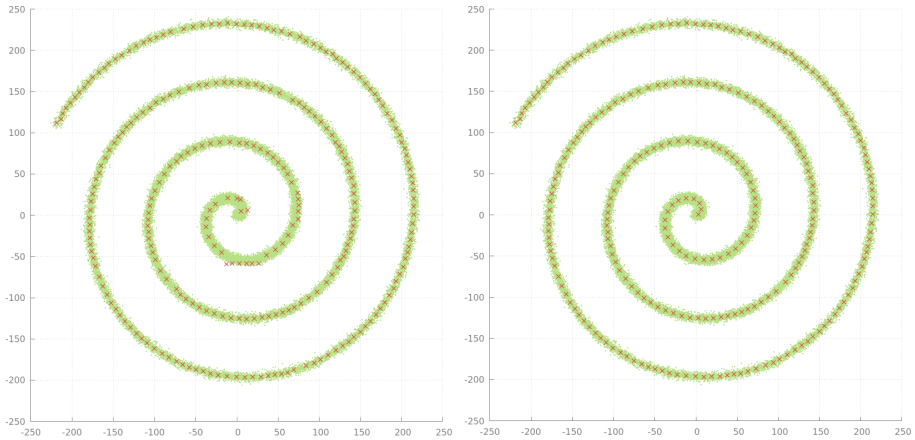
GT – kafelki grupowe

ET – kafelki rozszerzone

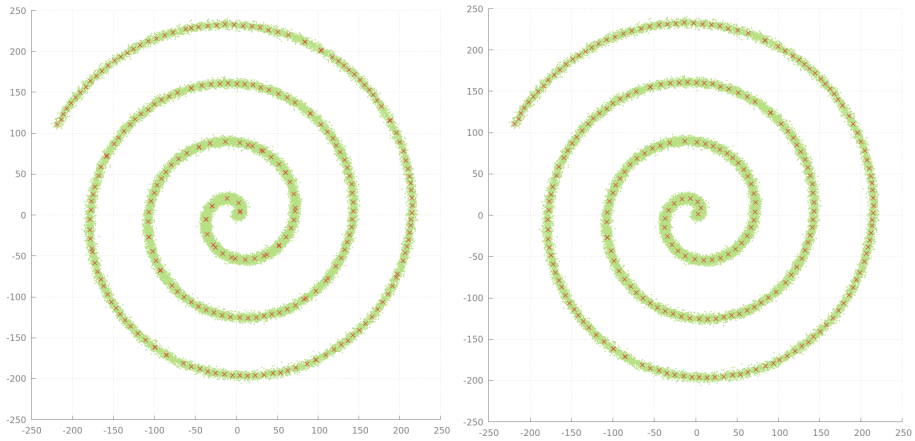
LRD – lokalna detekcja grani

MRD – hybrydowa detekcja grani

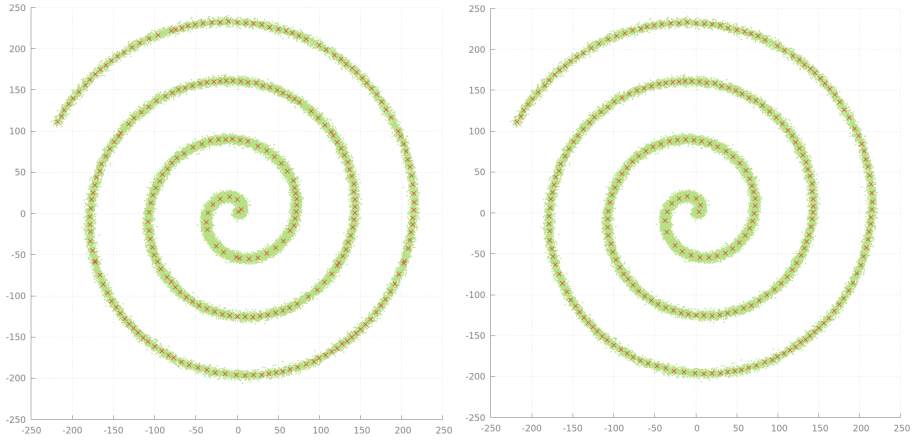
Wartości nad słupkami określają całkowity czas pracy detekcji, natomiast wartości wewnątrz słupków, czas danego etapu algorytmu. Każdy słupek składa się z trzech części. Zaczynając od dołu jest to budowa drzewa kafelkowego (najciemniejszy odcień), detekcja grani (odcień pośredni) i ewentualne wygładzanie



(a) (b)

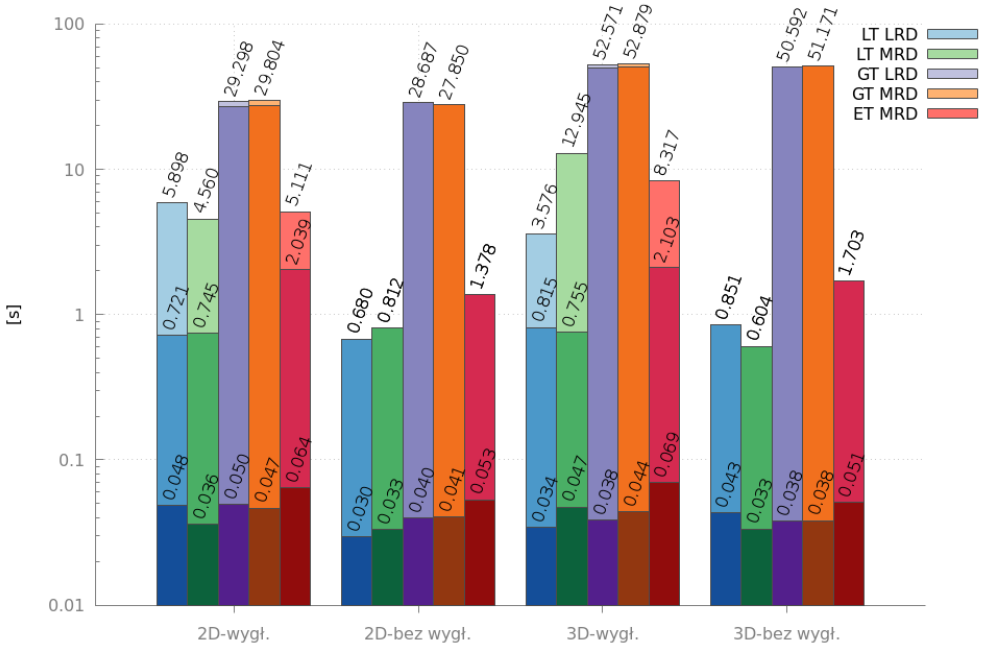


(c) (d)



(e) (f)

Rysunek 4.7: Hybrydowa detekcja grani z (rys. (b), (d), (f)) i bez (rys. (a), (c), (e)) włączonego etapu „wygładzania”. Od góry odpowiednio kafelki lokalne (rys. (b), (a)), grupowe (rys. (c) , (d)) i rozszerzone (rys (e), (f)). Czerwonym kolorem oznaczono finalny zbiór wybrańców.



Rysunek 4.8: Średni czas wykonania detekcji grani w zależności od rodzaju kafelków i algorytmu detekcji. Zbiór danych składa się z chmury 100-tys. dwu- lub trójwymiarowych punktów.

(najjaśniejszy odcień). Kafelki grupowe wypadają zdecydowanie najsłabiej, pod względem czasu pracy, co jest powiązane z ich bardzo dużą liczbą sąsiadów. Ponadto zauważamy, że etap wygładzania, często zajmuje wielokrotnie więcej niż sama detekcja grani w przypadku kafelków lokalnych i rozszerzonych. Ostatecznie biorąc pod uwagę, czas wykonania i jakość wyników, kafelki rozszerzone w połączeniu z hybrydową detekcją grani zdecydowanie tworzą najlepszy wariant. W związku z tym, tylko ta wersja zostaje przeniesiona na kartę graficzną.

4.2. Wersja „kafelkowa” na GPU

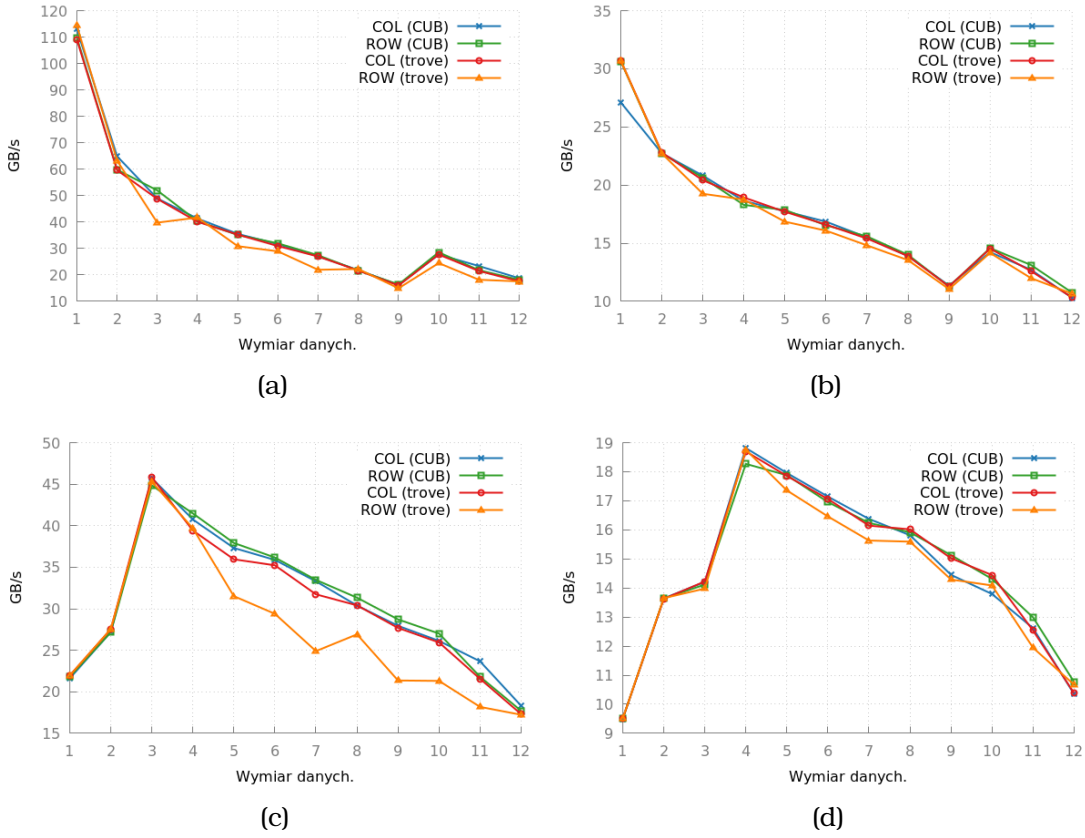
Proces budowy drzewa kafelkowego na karcie graficznej, koncepcyjnie jest identyczny jak jego odpowiednik na CPU. Wymaga jednak odpowiedniego dostosowania do architektury GPU. Przede wszystkim, środowisko CUDA nie dostarcza żadnych dynamicznych kontenerów na urządzeniu. W odpowiedzi na te braki, powstała autorska implementacja dynamicznego wektora. Udostępnionych jest kilka mechanizmów rozszerzania pojemności. Podstawowy polega

na skalowaniu aktualnej pojemności przez zadany współczynnik, za każdym razem po jej przekroczeniu. Pozostałe metody zakładają podanie przez użytkownika sekwencji rosnących wartości. Mogą być one całkowitoliczbowe, lub wyrażone w procentach. Klasa wektora przeznaczona jest do pracy tylko z jednym blokiem wątków. Jest to konsekwencją wykorzystania pamięci współdzielonej do udostępniania informacji na temat aktualnie przechowywanej liczby elementów, a także nowego, początkowego adresu pamięci, po rozszerzeniu wektora. Dane są zawsze przechowywane w pojedynczym, spójnym obszarze pamięci. Podczas rozszerzania następuje skopiowanie aktualnej zawartości do nowej, większej tablicy. Ponieważ alokacja następuje z poziomu urządzenia, fizycznie dane znajdują się w zarezerwowanym wcześniej obszarze sterty. Niestety, tutaj właśnie pojawia się problem, gdyż musimy z góry oszacować ilość potrzebnej pamięci. Ponadto, limit wielkości sterty urządzenia należy ustawić przed uruchomieniem jakiejkolwiek funkcji jądra z poziomu CUDA Runtime API i do momentu zresetowania urządzenia nie można go zmienić. Burzy to oczywiście cały *dynamizm* wektora. Z tego względu, żeby uniknąć rozwiązania zużywającego bardzo duże ilości pamięci (a w praktyce wykorzystującego tylko jej małą część) musimy liczyć histogramy. W ten sposób każdy kafelek będzie alokował tylko tyle pamięci, ile faktycznie mu potrzeba do przechowania zawartych w nim punktów.

Wyznaczanie histogramu w realizacji jednowątkowej jest bardzo prostym zadaniem. Jednakże mając na uwadze architektury wieloprocesorowe, takie jak GPU, nie jest to już takie oczywiste, jeśli chcemy wykonać to zadanie wydajnie. Do dyspozycji mamy dwa możliwe podejścia. Koncentrując się na danych wejściowych wystarczy tylko jeden raz przeczytać cały zbiór danych i aktualizować w trakcie odpowiednie wartości histogramu. Działając na wielu wątkach jednocześnie, musimy wykorzystać operacje atomowe, żeby zapewnić poprawność wyniku. Niestety atomowe inkrementowanie liczników mocno spowalnia pracę programu. Z drugiej strony, organizując obliczenia wokół danych wyjściowych zwalniamy się z konieczności użycia instrukcji atomowych. Niemniej jednak, te podejście wymusza wykonanie wielu iteracji po całym zbiorze danych, przez co jest ograniczone przepustowością pamięci. W związku z tym lepsza wydaje się być strategia zorientowana na dane wejściowe.

Większość dostępnych implementacji histogramu optymalizuje swoje działanie poprzez redukcję liczby wykonywanych operacji atomowych. Wilt [57] przedstawia rozwiązanie polegające na przypisaniu każdemu wątkowi prywatnego histogramu, trzymanego w całości w pamięci współdzielonej. Jednakże, jest ono dedykowane głównie do przetwarzania obrazów, gdzie mamy jedynie 256 wartości histogramu. Podobne podejście proponuje Ross [47], pakując wiele wartości histogramu do pojedynczego słowa w pamięci współdzielonej. Tym razem jednak każdy cały blok wątków ma swój prywatny histogram. W artykule [2] Adintez pokazuje ciekawą technikę agregowania kluczy (wartości histogramu) na poziomie splotu wątków. Niestety w naszym przypadku, gdzie mamy nieuporządkowane w żaden sposób dane i w ogólności dużą liczbę kluczy, takie postępowanie się nie sprawdzi, jak zresztą sam autor podkreśla. W tym projekcie zastosowano metodę opisaną przez Sakharnykh'a w artykule [28] i wdrożoną w bibliotece CUB [40]. Opiera się ona o fakt znaczącej poprawy wydajności operacji atomowych na pamięci globalnej w kartach z rodziny Kepler. Każdemu blokowi wątków przypisujemy obszar pamięci globalnej, gdzie budowany jest prywatny histogram. Po zakończeniu czytania danych następuje redukcja częściowych wyników do wyjściowego histogramu. Należy jeszcze tylko wziąć pod uwagę sytuację, gdy mamy rzeczywiście dużą liczbę wartości histogramu. Przykładowo, dzieląc każdy wymiar na 4 kafelki, dla zaledwie 10 wymiarów mamy już ponad 1 milion kluczy. Uruchamiając jedynie 256 bloków wątków, potrzebowalibyśmy zatem aż $4^{10} * 4 * 256 = 1GB$ pomocniczej pamięci na prywatne histogramy. Dlatego też w aktualnej implementacji, w momencie przekroczenia 2^{20} kluczy, wyłączamy użycie prywatnych histogramów. Na koniec dodajmy jeszcze, że w celu maksymalnie wydajnego czytania danych wejściowych stosujemy wektoryzację odczytów i przetwarzanie kafelkowe, opisane w rozdziale 6.

Jak się okazało, co widać na wykresach 4.9(a) i 4.9(b), po przełączeniu na implementację niewykorzystującą prywatnych histogramów, uzyskaliśmy wzrost wydajności. Taki rezultat, spowodował uruchomienie kolejnego testu, w którym cały czas wykonujemy operacje atomowe bezpośrednio do wyjściowego histogramu w pamięci globalnej. Wyniki są widoczne na wykresach 4.9(c) i 4.9(d). Na karcie TITAN, już od 5-go wymiaru (1024 kluczy), druga wersja

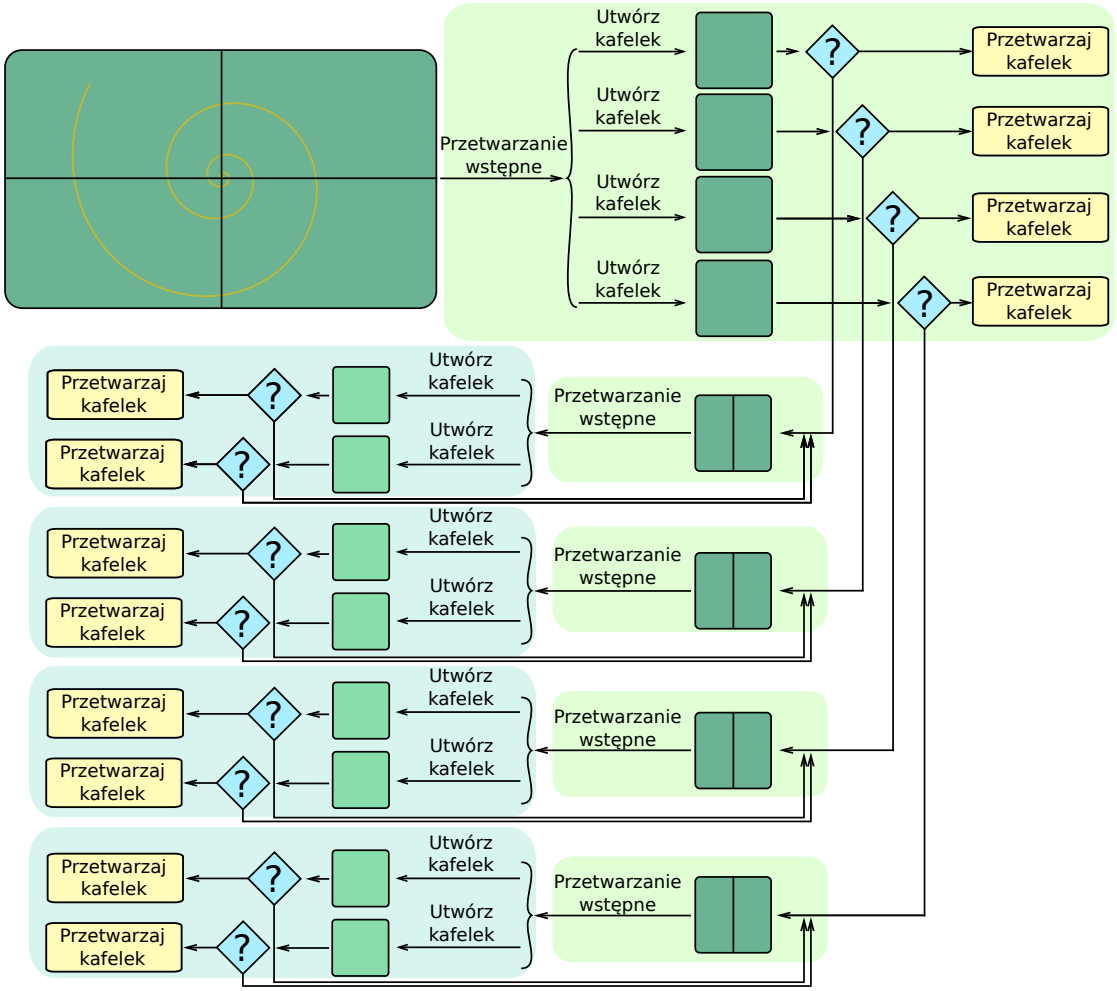


Rysunek 4.9: Maksymalna przepustowość podczas budowania histogramu w zależności od liczby wymiarów, układu danych w pamięci, a także wykorzystanej biblioteki do realizacji wektorowych zapisów, odczytów pamięci globalnej. Zbiór wejściowy składał się z 1 mln. punktów. Wykresy (a) i (c) przedstawiają wyniki uzyskane na karcie GTX TITAN, a (b) i (d) na karcie GTX 750Ti. Wykresy (a) i (b) prezentują wyniki wersji wykorzystującej prywatne histogramy, zaś (c) i (d) wersji wykonującej operacje atomowe bezpośrednio do wynikowego histogramu. Liczba wartości danego histogramu jest równa 4^d, gdzie d odpowiada wymiarowi danych.

okazuje się wydajniejsza. Natomiast w przypadku karty GTX 750Ti, nawet wcześniej, bo od 4-go wymiaru (512 kluczy). To spowodowało, że przesunęliśmy odpowiednio moment przełączenia na wersję bez prywatnych histogramów.

Wracając do budowy drzewa kafelkowego, zwróćmy uwagę, że dodawanie każdego kafelka jest niezależne od pozostałych. W związku z tym korzystamy z tej sposobności i wykonujemy te operacje równolegle, przydzielając jeden blok wątków do budowy jednego kafelka. Co więcej, możliwa jest sytuacja, gdzie w tym samym czasie jeden kafelek jest w trakcie konstrukcji struktury danych, a inny już zakończył pracę i może rozpocząć przetwarzanie swoich

danych w nowej funkcji jądra. Implementacja tak wielotorowej pracy, jest możliwa tylko przy użyciu dynamicznej równoległości. Wykorzystywaliśmy już tę technikę przy okazji ewolucji, decymacji i całościowego nadzorowania pracy algorytmu detekcji grani. Jednakże dotychczas stosowaliśmy dynamiczną równoległość jedynie w celu minimalizacji komunikacji urządzenia z gospodarzem. Tym razem, używamy tego narzędzia zgodnie z jego właściwym przeznaczeniem, a dokładniej do adaptacyjnej dystrybucji pracy [1, 36]. Dynamiczna równoległość jest techniką dającą programistom potężne i do tej pory niedostępne, możliwości w budowaniu bardziej optymalnych rozwiązań. Nie mniej jednak, efektywne jego wykorzystanie wymaga skrupulatnej analizy algorytmu, gdyż łatwo doprowadzić do sytuacji w której możemy znacznie pogorszyć wydajność programu [58].



Rysunek 4.10: Schemat budowy drzewa kafelkowego na GPU.

Rysunek 4.10 przedstawia ogólny schemat pracy zastosowany przy budowie kafelkowego drzewa na GPU. Kolorem tła wyróżniono pracę wykonywaną w tym samym strumieniu. Nawias klamrowy oznacza uruchomienie nowej funkcji jądra, a odchodzące od niego strzałki symbolizują bloki wątków. Zanim rozpoczniemy dodawanie kolejnych kafelków do drzewa, potrzebujemy kilku informacji wyznaczanych w ramach globalnego *przetwarzania wstępnego* i nadzorowanych jeszcze z poziomu gospodarza. W pierwszym kroku szukamy granic przestrzeni ograniczającej wejściową chmurę punktów. Sprowadza się to do powszechnie znanego problemu redukcji [14, 56]. W naszym przypadku jest to poszukiwanie minimum i maksimum, do którego istnieją już bardzo dobre gotowe rozwiązania m.in. w bibliotece Thrust [17, 29], czy też CUB [40]. Jednakże, o ile w bibliotece Thrust jest dostępna funkcja znajdująca jednocześnie elementy minimum i maksimum ze zbioru wejściowego, to CUB takiej możliwości nie udostępnia. Ponadto tak naprawdę potrzebujemy nie jedną parę minimum i maksimum, a tyle ile wymiarów mają punkty. W związku z tym, opierając się na implementacji z biblioteki CUB, powstała autorska funkcja jądra, realizująca te zadanie. Składa się ona z dwóch następujących po sobie faz, realizowanych przez oddzielne funkcje jądra. W pierwszej z nich każdy wątek wyznacza minimum i maksimum dla przypisanych mu punktów, zapisując wynik w rejestrach. Następnie, każdy blok wątków redukuje wyniki częściowe do przypisanego mu, prywatnego obszaru pamięci globalnej. W drugim kroku uruchamiamy tyle bloków wątków ile wymiarów mają punkty razy dwa. Bloki o parzystych indeksach znajdują maksimum odpowiadającego im wymiaru, podczas gdy pozostałe szukają minimum odpowiadających im wymiarów.

Uzyskane w powyższy sposób granice, pozwalają oszacować maksymalną, ostateczną liczbę *wybrańców* w celu zaalokowania dla nich minimalnego obszaru pamięci. Mając te informacje do dyspozycji przystępujemy do budowy drzewa uruchamiając nową funkcję jądra z pojedynczym wątkiem. Z jej poziomu alokujemy pamięć na kafelki znajdujące się na pierwszym poziomie hierarchii. Zlecamy funkcję jądra, której zadaniem jest obliczenie granic nowych kafelków, a także jeszcze kolejną funkcję jądra liczącą przestrzenny histogram. Po odczekaniu aż obliczenia się zakończą, zaczynamy dodawanie pierwszych kafelków w nowej funkcji jądra. Na rysunku 4.10 są to pierwsze, duże, nawiasy

klamrowe. Uruchamiamy tyle bloków wątków, ile jest kafelków i każdemu z nich zadajemy do wykonania pracę przedstawioną na algorytmie 17.

```
1 if pointsCnt < 2 then
2     return
3 Alokuj pamięć na punkty chmury i punkty sąsiadujące
4 Znajdź i skopiuj odpowiednie punkty
5 if pointsCnt <= maxTileCapacity then
6     Alokuj pamięć na wybrańców
7     Przetwarzaj kafelek
8 else
9     SUBDIVIDE( )
```

Algorytm 17: Dodawanie kafelków do drzewa.

Po alokacji kontenerów na punkty, każdy blok rozpoczyna kopiowanie wpadających w jego granice punktów. Aktualnie uruchomiona sieć wątków jest dopasowana właśnie do tego zadania, żeby zmniejszyć liczbę zlecanych funkcji jądra.

Zadanie wykonywane w linii 5 na algorytmie 17, możemy scharakteryzować jako powszechnie znany problem selekcji elementów spełniających dany predykat. W bibliotece CUB istnieje implementacja wykonująca taką operację, aczkolwiek całą siecią wątków i przy użyciu tylko jednego predykatu. Natomiast w naszym przypadku każdy blok wątków kopiuje punkty znajdujące się w granicach przypisanego mu kafelka. Dlatego też w niniejszej pracy wykorzystujemy własną implementację, wzorującą się na tej z biblioteki CUB. Różnice są następujące: przede wszystkim nie zachowujemy porządku punktów, gdyż nie ma takiej potrzeby. Dalej, każdy blok wątków musi przeczytać cały wejściowy zbiór punktów. Ponieważ zawsze iterujemy po danych w tej samej kolejności, mocno polegamy tutaj na pamięci podręcznej L2, buforującej odczyty dla następnych bloków. Ponadto wybór punktów następuje do prywatnych rejestrów wątków, po czym wątki w ramach splotu wspólnie określają docelowy obszar w pamięci globalnej, gdzie zapisują punkty spełniające dany warunek. Orientacja pracy na poziomie splotu wątków przynosi szereg zalet. Po pierwsze, nie wykonujemy ani jednej synchronizacji wątków. Po drugie, nie musimy posługiwać się pamięcią współdzieloną do pośredniczenia przy zapisie wybranych punktów do pamięci globalnej, tak jak to jest w bibliotece CUB.

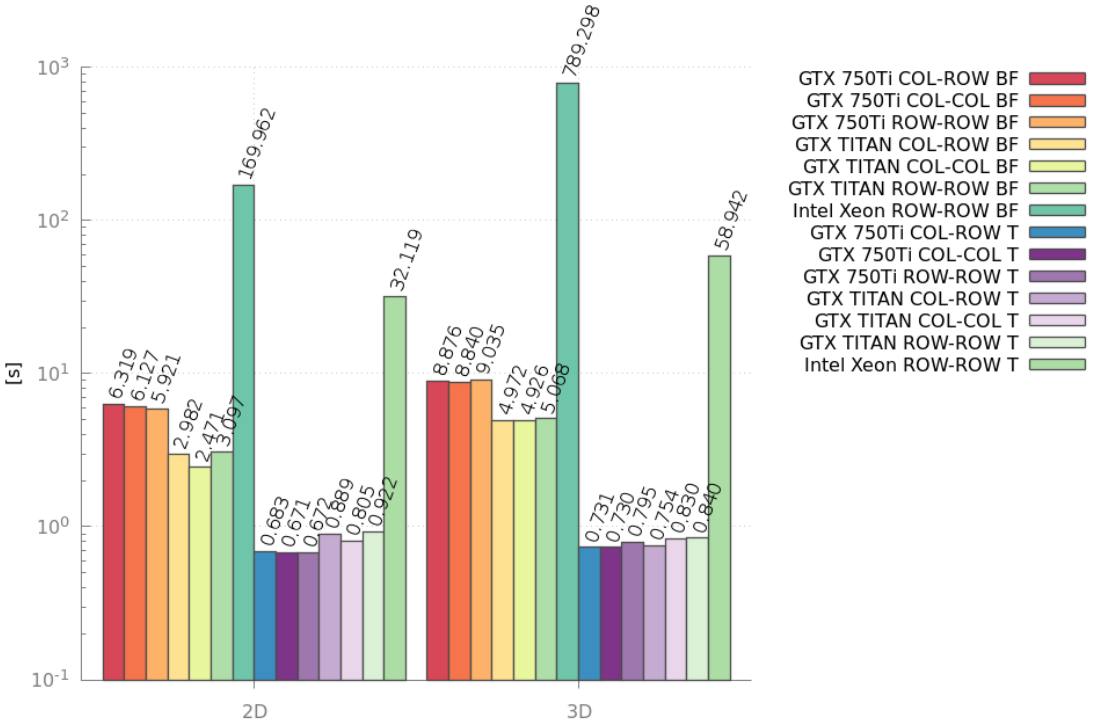
Mając zebrane punkty należące do danego kafelka, sprawdzamy czy nie przekraczamy jego maksymalnej pojemności. Jest to bardzo ważny parametr budowy drzewa kafelkowego na GPU, mający istotny wpływ na wydajność tego procesu, decydując o liczbie podziałów, a tym samym o liczbie uruchamianych funkcji jądra. W sytuacji, gdy mieścimy się w zadanym limicie, dany blok wątków może rozpocząć przetwarzanie swojego kafelka. W przeciwnym wypadku wykonujemy czynności wymienione na algorytmie 18, analogiczne do tych przeprowadzonych na początku procedury budowy drzewa. Dlatego też, na schemacie zostały również ujęte jako *przetwarzanie wstępne*. Na koniec zlecamy nową funkcję jądra z jedynie dwoma blokami wątków, dla lewego i prawego pod-kafelka. Jej działanie jest identyczne jak na poprzednio zaprezentowanym algorytmie 17 z jednym małym dodatkiem. Mianowicie ostatni (z dwóch) bloków wątków, który zakończył selekcję punktów, może zwolnić zasoby zajmowane przez kafelka-rodzica.

```
1 function SUBDIVIDE( )
2     Alokuj kafelki-dzieci
3     Oblicz granice kafelków-dzieci
4     Oblicz histogram przestrzenny
5     Dodaj kafelki do drzewa w nowej funkcji jądra
```

Algorytm 18: Podział kafelka na dwie równe części.

Pozostało już tylko uruchomić detekcję grani w każdym kafelku z osobna. W przypadku detekcji lokalnej, nie musimy wprowadzać żadnych dodatkowych zmian. Natomiast detekcja hybrydowa, niestety wymaga powrotu do podstawowej wersji decymacji ze względu na rozproszone po kafelkach lokalne zbiory *wybrańców*, a także dostosowania jej do pracy z wieloma kafelkami. Przeanalizujmy zatem otrzymane w testach wyniki i porównajmy wydajność z poprzednimi wersjami.

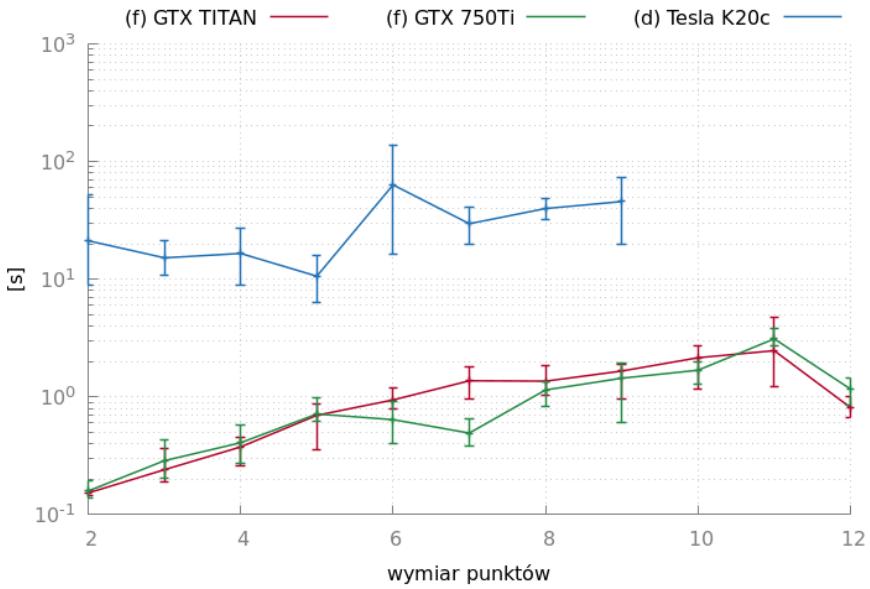
Jak widać na wykresie 4.11, udało się otrzymać ok. 47x przyśpieszenie w wersji „siłowej” i ok. 80x w wersji kafelkowej, w stosunku do odpowiednika wykonywanego w wielu wątkach na CPU. Zwróćmy uwagę, że przy dwóch–trzech wymiarach sposób układu danych w pamięci urządzenia nie ma większego znaczenia. Ponadto interesujący jest wynik karty GTX 750Ti w wersji kafelkowej, gdzie jest odrobinę szybsza niż, znacznie większa karta



Rysunek 4.11: Średni czas (zaprezentowany na logarytmicznej skali) wykonania detekcji grani dwu-, lub trójwymiarowej spirali składającej się z 1 mln. punktów, zaburzonych szumem o rozkładzie normalnym. „BF” oznacza wersję „siłową”, natomiast „T”, wersję kafelkową z kafelkami rozszerzonymi i hybrydową detekcją grani. W wersji kafelkowej następował podział każdego wymiaru przestrzeni na 3 kafelki, z których każdy miał maksymalną pojemność równą 200 tys. pkt.

GTX TITAN. Generacja kart graficznych Maxwell, do której należy GTX 750Ti powiększyła pojemność pamięci podręcznej L2 z 1.5MB (GTX TITAN) do 2MB. Dodatkowo, wyodrębniona została pamięć podręczna L1, co pozwoliło także zwiększyć ilość pamięci współdzielonej dostępnej dla jednego wieloprocesora z 48KB (GTX TITAN) do 64KB. Te, i inne zmiany w architekturze rodziny Maxwell pozwalają zwiększyć ilość aktywnych bloków wątków na pojedynczym wieloprocesorze. A to z kolei prowadzi do większej wydajności.

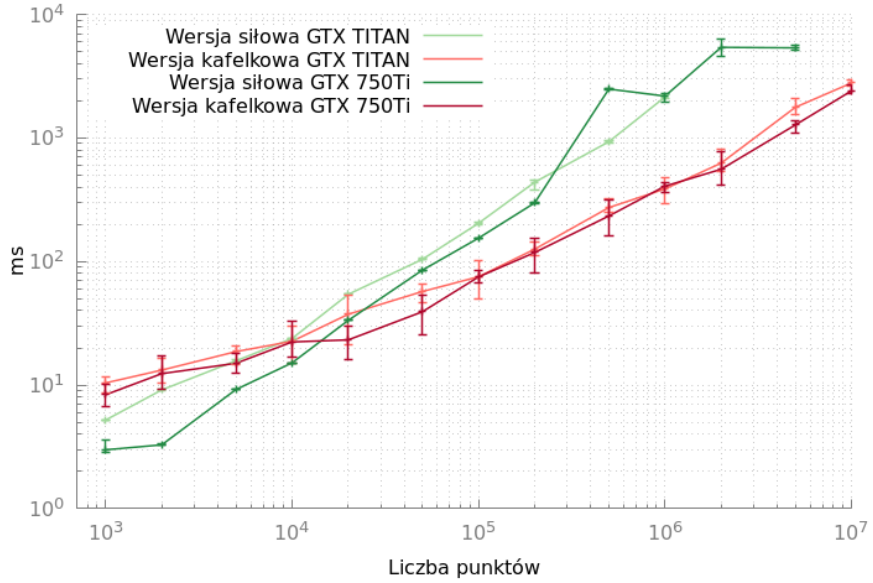
Spójrzmy, jeszcze na wykresy 4.12 i 4.13, prezentujące skalowalność kafelkowej detekcji grani w zależności od liczby i wymiaru punktów chmury. Pionowe słupki oznaczają minimalny i maksymalny uzyskany w testach wynik. Jak widać, w obydwu przypadkach otrzymujemy liniowy wzrost czasu.



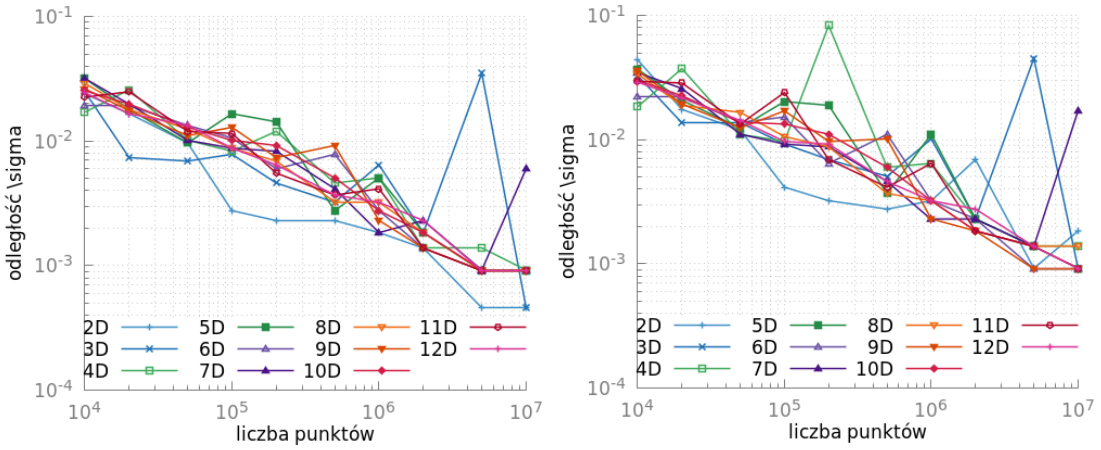
Rysunek 4.12: Średni czas kafelkowej detekcji grani w zależności od wymiaru danych, przeprowadzonej na chmurze składającej się z 1 mln. punktów tworzącej odcinek prosty o długości 100, zaburzony szumem o rozkładzie normalnym. Początkowy podział przestrzeni: 16 kafelków w pierwszym wymiarze i 1 kafelek w pozostałych. Literka „f” oznacza obliczenia zmiennoprzecinkowe w pojedynczej precyzji, zaś „d” w podwójnej precyzji.

Ponadto, zgodnie z oczekiwaniami, wersja kafelkowa potrzebuje odpowiednio dużej liczby punktów, by zamortyzować czas poświęcony na budowę drzewa.

Ocena jakości aproksymacji krzywej została przedstawiona na wykresach 4.14 i 4.15. Podobnie jak dla wersji „siłowej”, wraz ze wzrostem liczby punktów i wartości parametru $R1$ w stosunku do odchylenia standardowego dodanego szumu, obserwujemy coraz lepsze odwzorowanie poszukiwanej krzywej. Jednakże, algorytm kafelkowy wymaga zastosowania większej wartości promienia $R1$ Wynika to bezpośrednio z wybranego modelu kafelka. Mianowicie, początkowy zbiór *wybrańców* danego kafelka w wyniku ewolucji może poruszać się jedynie w obrębie danego kafelka i jego sąsiadów. Ponieważ kafelki rozszerzone jako sąsiadujące punkty traktują te leżące w promieniu $R1$ od ich granic, to im większy promień, tym większy obszar przestrzeni „widzi” dany kafelek. W konsekwencji początkowy podział przestrzeni na kafelki, jak i ich maksymalna pojemność (która ma wpływ na ich ewentualny dalszy podział), mają niezwykle istotny wpływ na jakość aproksymacji. Niestety, wydaje się, że nie jest możliwe automatyczne dobranie tych parametrów, gdyż zależą one od

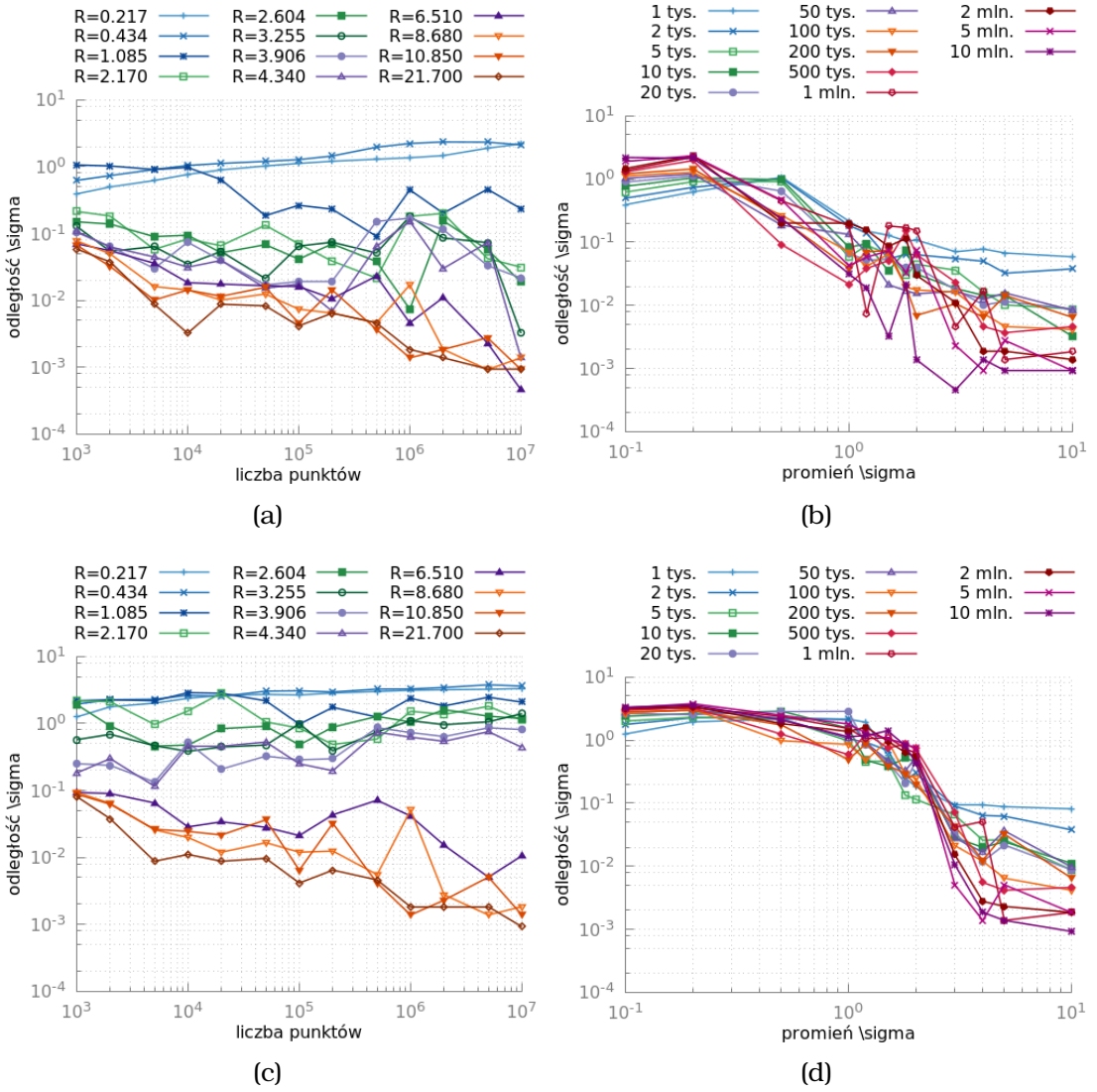


Rysunek 4.13: Średni czas kafelkowej detekcji grani, dla parametru $R1 = 4.34$, w zależności od liczby punktów chmury tworzącej odcinek prosty o długości 100, zaburzony szumem o rozkładzie normalnym. Parametry budowy drzewa w wersji kafelkowej: początkowy podział przestrzeni: 4 kafelki w pierwszym wymiarze i 1 kafelek w pozostałych, maksymalna pojemność kafelka $maxp = 0.2 * N$, gdzie N to liczba punktów chmury. Obliczenia w pojedynczej precyzji.



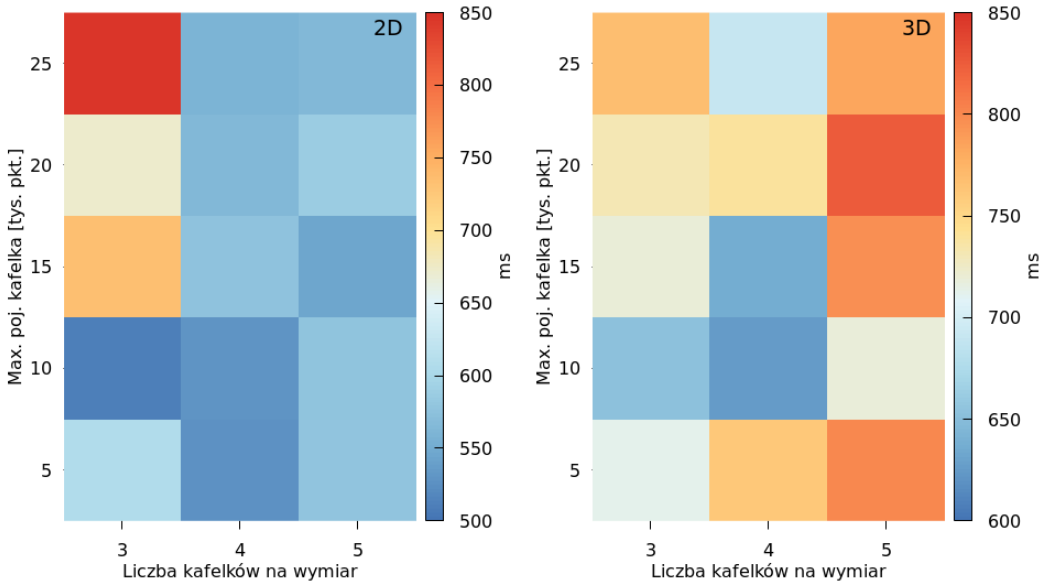
Rysunek 4.14: Jakość kafelkowej aproksymacji grani, dla ustalonego parametru $R1 = 3.689$, w zależności od wymiaru danych i ilości punktów. Wykres (b) prezentuje odległość Hausdorffa, zaś (a) medianę odległości. Dane wejściowe stanowiła chmura punktów tworząca odcinek prosty o długości 100, zaburzona szumem o rozkładzie normalnym z odchyleniem standardowym równym 2.17.

układu punktów chmury rekonstruowanej krzywej. Oczywiście można powiększyć region „widoczny” dla danego kafelka, ale trzeba pamiętać, że powoduje

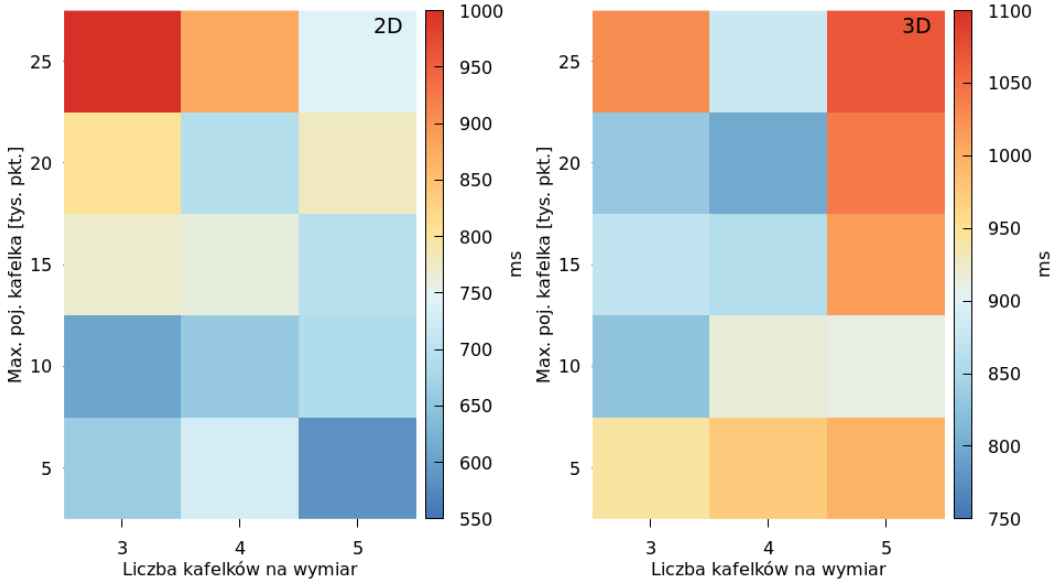


Rysunek 4.15: Jakość kafelkowej aproksymacji grani dwu wymiarowej chmury punktów tworzącej odcinek prosty o długości 100, zaburzonej szumem o rozkładzie normalnym, z odchyleniem standardowym równym sigma. Parametry budowy drzewa w wersji kafelkowej: początkowy podział przestrzeni: 4 kafelki w pierwszym wymiarze i 1 kafelek w pozostałych, maksymalna pojemność kafelka $maxp = 0.2 * N$, gdzie N to liczba punktów chmury. Wykresy (a) i (b) przedstawiają medianę odległości wynikowej krzywej od odcinka, zaś (c) i (d) odległość Hausdorffa.

to wydłużenie czasu pracy. Na wykresie 4.16 możemy zobaczyć przykładową zależność średniego czasu wykonania od początkowego podziału przestrzeni i pojemności kafelków.



(a)



(b)

Rysunek 4.16: Średni czas kafelkowej detekcji grani, w zależności od maksymalnej pojemności kafelka i początkowego podziału przestrzeni na kafelki, przeprowadzonej na chmurze składającej się z 1 mln. punktów, tworzącej dwu- (lewa strona) lub trójwymiarową (prawa strona) spiralę, zaburzoną szumem o rozkładzie normalnym. Wykres (a) prezentuje czasy otrzymane na karcie GTX 750Ti, natomiast (b) na karcie GTX TITAN.

5. Struktury drzewiaste w poszukiwaniu najbliższego sąsiada

Na początku rozdziału 4 powiedzieliśmy, że kafelkowanie stanowi jedynie częściowe złagodzenie problemu dużej liczby porównań w algorytmie poszukiwania najbliższych sąsiadów. W końcu przecież, wewnątrz pojedynczego kafelka, w dalszym ciągu używamy implementacji *siłowej*. Jak się dowiedzieliśmy z dalszej części tego rozdziału, lepszym rozwiązaniem jest zastosowanie hierarchicznych struktur drzewiastych. Najczęściej wykorzystywane w tym celu są drzewa k-wymiarowe. Ponadto, pochodzące z dziedziny grafiki komputerowej, drzewa brył ograniczających, również wydają się dobrze odpowiadać zagadnieniu. W przypadku detekcji grani, taką strukturę moglibyśmy budować dla zbioru wybrańców, by punkty w ramach jednego kafelka mogły następnie szybko znaleźć najbliższego sobie *wybrańca*. Niemniej jednak, w praktyce niekoniecznie takie podejście może się okazać skuteczne. Jego słabością jest konieczność budowania od nowa drzewa po każdej ewolucji. Ponadto z każdą iteracją decymacji maleje zbiór *wybrańców*, zmniejszając tym samym uzasadnienie wykorzystania drzewa. Ostatnim narastającym dylematem jest malejąca wydajność struktur drzewiastych wraz z wzrostem liczby wymiarów. Niestety, ze względu na ograniczone ramy czasowe, nie udało się w niniejszej pracy przeprowadzić eksperymentów z użyciem ani drzew k-wymiarowych, ani drzew brył ograniczających.

6. Zastosowane techniki optymalizacji wydajności funkcji jądra.

Zaprojektowanie wydajnej funkcji jądra jest niezwykle trudnym zadaniem. Wymaga dogłębnej znajomości architektury karty graficznej, badanej dziedziny, a ponadto zdolności „myślenia równoległego". Z pomocą przychodzą rozmaite wzorce projektowe, porady, schematy, techniki czy też sztuczki poprawiające efektywność naszego programu, dostępne w szerokim wachlarzu literatury [16, 17, 19, 20, 26, 56], wymieniając zaledwie kilka przykładowych źródeł. W tym rozdziale przedstawimy i opiszemy najciekawsze i najważniejsze z wykorzystywanych w tej pracy.

6.1. Wyznaczanie optymalnej konfiguracji funkcji jądra

Każda funkcja jądra wymaga podania, podczas zlecania do wykonania, zestawu parametrów konfiguracji uruchomienia. Pomijając przypadki, gdzie implementacja funkcji z góry narzuca konkretne wartości, ich wybór może mieć bardzo duży wpływ na wydajność. W celu ułatwienia tego zadania Nvidia udostępnia arkusz kalkulacyjny służący do wyznaczania konfiguracji, maksymalizującej wykorzystanie zasobów urządzenia. Dodatkowo od wersji 6.5 CUDA SDK dostępny jest interfejs umożliwiający realizację tego zadania w trakcie działania programu [15]. Znacznie ułatwia on optymalizację funkcji jądra, jednakże skupia się na metryce, która nie zawsze zapewnia najlepszą wydajność. Ponadto w przypadku, tak jak to ma miejsce w tej pracy, gdzie funkcje jądra często są szablonami zależącymi od wielu parametrów, z których niektóre mają istotny wpływ na zużycie zasobów urządzenia, jego zastosowanie jest utrudnione, bądź nawet bezzasadne. W związku z tym do projektu załączonego do niniejszej pracy dołączony jest zestaw testów badających wydajność danej funkcji jądra w zależności od parametrów szablonu (jeżeli takowe są) i rozmiaru bloku wątków. Na podstawie zebranych wyników,

z przeprowadzonych pomiarów wybierane są optymalne, w sensie czasu wykonania i wydajności, dla docelowej architektury (Kepler, Maxwell), rozmiary bloku wątków i jeżeli jest używany, parametr szablonu decydujący o ilości pracy wykonywanej przez pojedynczy wątek. W celu automatyzacji wyboru odpowiedniej konfiguracji, każda tak przetestowana funkcja jądra udostępnia interfejs pośredniczący w jej uruchomieniu. Ma on za zadanie wybranie konfiguracji przeznaczonej dla wirtualnej architektury, dla której program został skompilowany. Następnie bazując na niej, jeżeli zachodzi taka konieczność, dostosowujemy ją do aktualnych parametrów szablonu wołanej funkcji jądra. Na koniec, funkcja z interfejsu CUDA'ownego kalkulatora wykorzystania zasobów, `cudaOccupancyMaxActiveBlocksPerMultiprocessor`, okazuje się wysoce przydatna do wyznaczenia rozmiarów uruchamianej sieci bloków wątków. Dokładniej rzecz ujmując, za jej pomocą uzyskujemy maksymalną liczbę bloków wątków mogących jednocześnie rezydować na pojedynczym wieloprocesorze. Dalej, mnożymy ją przez liczbę wieloprocesorów wybranej karty graficznej i w większości przypadków, przez heurystycznie dobraną stałą zależną od aktualnej wirtualnej architektury. Ostatnia wartość jest makrem zdefiniowanym w bibliotece CUB [40].

6.2. Ziarnistość zadań

W najprostszej formie, pojedyńczy wątek w ramach funkcji jądra przetwarza jeden element danych wejściowych. Tak drobnoziarnisty podział pracy wymaga bardzo dużych rozmiarów sieci wątków, do ukrycia opóźnień dostępów do pamięci globalnej, czy opóźnień operacji arytmetycznych. Jednakże, nawet korzystając z CUDA'ownego interfejsu do wyznaczenia konfiguracji zapewniającej maksymalną liczbę aktywnych splotów wątków, wynikowa wydajność będzie daleka od maksymalnej, możliwej do osiągnięcia przez kartę graficzną. Żeby sytuację poprawić musimy korzystać z rejestrów, gdyż to one zapewniają największą przepustowość, jak i wydajność. Tymczasem większe zużycie zasobów pociąga za sobą spadek liczby aktywnych splotów wątków. Niemniej jednak okazuje się, że wykorzystując mniej wątków, z których każdy wykonuje więcej pracy, możemy również dobrze ukryć opóźnienia, a na nowszych kartach

```
template <typename ITEMS_PER_THREAD>
__global__ void kernel()
{
  //...
  a[ITEMS_PER_THREAD];
  b[ITEMS_PER_THREAD];
  c[ITEMS_PER_THREAD];
  //...
  #pragma unroll
  for (int i = 0; i < ITEMS_PER_THREAD; ++i)
  {
    a[i] = A[idx];
    b[i] = B[idx];
    c[i] = C[idx];
  }
  //...
  #pragma unroll
  for (int i = 0; i < ITEMS_PER_THREAD; ++i)
  {
    c[i] = c[i] + a[i] * b[i];
  }
  //...
```

Wydruk 6.1: Wykorzystanie niezależności instrukcji do ukrycia opóźnień w dostępie do pamięci i operacji arytmetycznych.

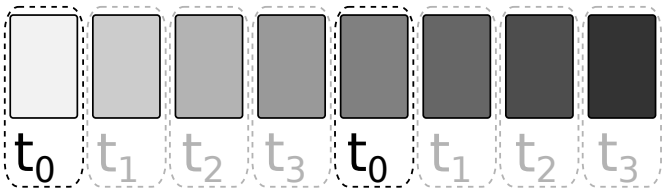
jest to jedynym sposobem na uzyskanie maksymalnej wydajności, co wykazał Volkov [55]. Zatem, korzystając z tej wiedzy, większość zaimplementowanych w tej pracy funkcji jądra ma schemat analogiczny do przedstawionego na wydruku 6.1.

Dzięki parametryzacji szablonu funkcji jądra liczbą elementów przetwarzanych przez jeden wątek, umożliwiamy statyczne adresowanie tablic, które w konsekwencji mogą zostać umieszczone w rejestrach. Ponadto wszelkie pętle wykonywane po elementach przypisanych wątkowi, mają z góry znaną liczbę iteracji. Pozwala to na ich całkowite rozwinięcie przy użyciu dyrektywy `#pragma unroll`. Redukujemy tym samym dodatkowo obciążające procesor instrukcje obsługujące iteracje pętli. Co więcej, zwiększamy niezależność instrukcji, przez co jednostka zarządzająca pracą splotu, ma do dyspozycji nie jedną instrukcję, a aż `ITEMS_PER_THREAD`.

6.3. Wektoryzacja dostępów do pamięci

Jest to technika szczególnie przydatna w przypadku funkcji jądra ograniczonych przepustowością magistrali pamięci globalnej. W odniesieniu do kafelkowej detekcji grani są to: wyznaczanie histogramu, szukanie granic przestrzeni zawierającej chmurę punktów, a także selekcja punktów spełniających predykat. Obecnie dostępne karty graficzne firmy Nvidia umożliwiają wykonywanie dostępów do pamięci globalnej w transakcjach 32-, 64- lub 128-bajtowych [36]. Ponadto, dostęp do pamięci globalnej jest realizowany w jednej instrukcji, wyłącznie gdy rozmiar typu danych jest równy 1, 2, 4, 8, 16 bajtów i czytany/zapisywany adres jest naturalnie wyrównany. Zatem, w pojedynczej instrukcji, możliwe jest pobranie aż 512 bajtów - co zostanie zrealizowane w minimum 4 transakcjach, przy właściwym wyrównaniu początkowego adresu. Pracując na 32-bitowych danych (`int, float`), każdy wątek może więc pobierać aż 4 kolejne wartości, zmniejszając tym samym liczbę instrukcji programu. W celu ułatwienia wektoryzacji napisane zostały dwie klasy zapewniające tą funkcjonalność dla odczytów i zapisów. Ich interfejs został zaprojektowany opierając się na kilku ideach i założeniach:

— dane są przetwarzane kafelkami[1]
— dostęp do danych następuje w przeplatanym układzie (patrz rysunek 6.1
— jeden wątek czyta N punktów D-wymiarowych
— dane wejściowe i wyjściowe mogą być w układzie kolumnowym lub wierszowym

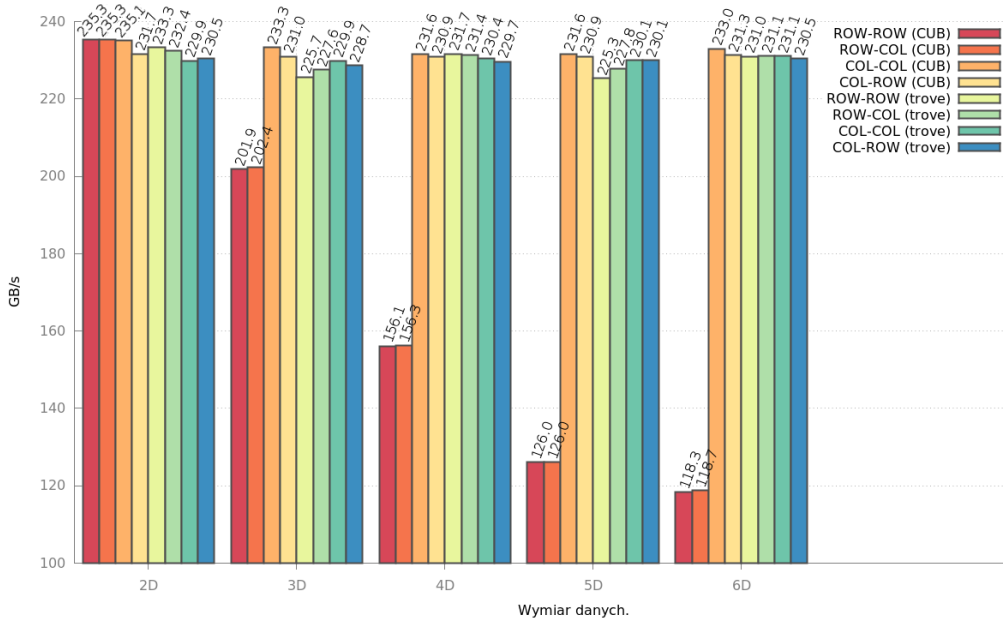


Rysunek 6.1: Przeplatany układ dostępu do danych

Korzystając z tych klas, mamy do wyboru dwa interfejsy realizujące faktyczne dostępy do danych: CUB, i Trove [7]. Biblioteka CUB dodatkowo, udostępnia zastosowanie różnych modyfikatorów dostępu do pamięci, gdyż

[1] Patrz rozdział 6.4

realizowane są one poprzez wplatane w kod instrukcje ptx. Zadaniem klas jest taki podział wczytywanych danych pomiędzy wątki, by liczba instrukcji i transakcji konieczna do ich odczytu/zapisu była jak najmniejsza. Na wykresie 6.2 przedstawione zostały przykładowe średnie wyniki ze 100 iteracji, podczas testu polegającego na kopiowaniu danych w ramach pamięci urządzenia. Współrzędne punktów przechowywane były jako liczby zmiennoprzecinkowe pojedynczej precyzji. Opis etykiet w legendzie ma postać „X - Y (Z)", gdzie X jest układem danych wejściowych i wyjściowych w pamięci urządzenia, Y jest układem danych w rejestrach, a Z oznacza wykorzystywaną bibliotekę. Jak widać, niezależnie od wymiaru danych, możliwe jest czytanie na poziomie ok. 80% maksymalnej przepustowości pamięci w przypadku karty GTX TITAN.



Rysunek 6.2: Maksymalna uzyskana przepustowość na karcie GTX TITAN w zależności od porządku i wymiaru danych. Zbiór wejściowy składał się z 1 mln. punktów.

6.4. Kafelkowe przetwarzanie danych

Kafelkowanie danych polega na podzieleniu wejściowego zbioru danych na *kafelki*, tj. podzbiory przetwarzane przez pojedynczy blok wątków. Taki podział niesie ze sobą szereg zalet. Znając rozmiar kafelka na etapie kompilacji programu, możemy statycznie zaalokować potrzebną ilość pamięci współdzielonej,

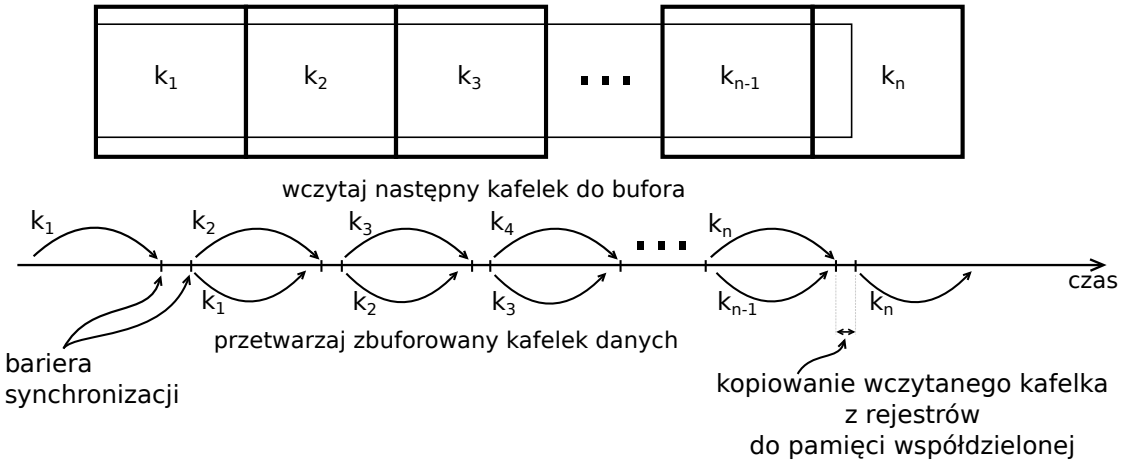
a także ułatwiamy kompilatorowi zastosowanie różnych optymalizacji, w tym techniki rozwijania pętli. Ponadto nieprzypadkowo wybór rozmiaru kafelka często jest wielokrotnością rozmiaru bloku wątków. Upraszczamy w ten sposób kod funkcji jądra, a w szczególności zmniejszamy rozbieżność wątków w ramach splotu. Nie musimy wykonywać sprawdzania przekroczenia rozmiaru kafelka, bo np. każdy wątek czyta do pamięci współdzielonej tylko jeden element. Co więcej, podział danych na kafelki, umożliwia prawie całkowite pozbycie się instrukcji warunkowych badających wyjście poza zakres danych [56]. Wystarczy, że dokonamy podziału pracy na kafelki pełne i częściowe. Ostatnią, ale nie mniej ważną zaletą jest regularyzacja schematu funkcji jądra, który przyjmuje postać analogiczną do przedstawionego jako algorytm 19. Zwiększamy tym samym czytelność kodu, zmniejszamy możliwość popełnienia błędów i ułatwiamy jego modyfikację.

```
1 Nk ← liczba pełnych kafelków pokrywająca dane wejściowe
2 for k = 1 to k = Nk − 1 do
3     Wczytaj pełny kafelek
4     Wykonaj obliczenia na pełnym kafelku
5     Zapisz wynik
6 Wczytaj ostatni, (nie)pełny kafelek
7 Wykonaj obliczenia na (nie)pełnym kafelku
8 Zapisz wynik
```

Algorytm 19: Schemat pracy funkcji jądra z wykorzystaniem idei kafelkowania.

6.5. Przetwarzanie potokowe

Przetwarzanie potokowe pozwala na ukrycie opóźnień w dostępie do pamięci globalnej. W tym celu równolegle wykonujemy wczytywanie danych z obliczeniami na już pobranych danych [19, 26]. Przykładowy schemat działania funkcji jądra adaptującej tę technikę został zilustrowany na rysunku 6.3. Oczywiście, aby w pełni ukryć opóźnienie związane z wczytywaniem kolejnego kafelka danych, konieczna jest odpowiednia ilość obliczeń. Minusem tej metody może być dodatkowe duże zapotrzebowanie na zasoby wieloprocesora takie jak rejestry i pamięć współdzielona, także konieczność częstej synchronizacji. Toteż



Rysunek 6.3: Potokowe przetwarzanie kafelków.

raczej nie sprawdzi się ona w sytuacji, gdy mamy dostateczną ilość aktywnych splotów wątków do ukrycia opóźnień. Natomiast może przynieść korzyści w przypadku funkcji jądra wykorzystujących bloki wątków o niewielkich rozmiarach lub gdy z różnych powodów mamy niski poziom jednocześnie aktywnych splotów wątków. Przetwarzanie potokowe może w ogólności zachodzić na różnych poziomach. Przykładowo, bardzo dobrze współgra z opisywanym wcześniej w rozdziale 6.2 zwiększeniem liczby niezależnie wykonywanych od siebie instrukcji.

7. Wykorzystane narzędzia

7.1. CUB

CUB [40], jest biblioteką dostarczającą gotowych rozwiązań popularnych problemów takich jak, sortowanie, segmentacja, wyznaczanie histogramu, redukcja, skanowanie, selekcja, czy też mnożenie rzadkiej macierzy przez wektor, wykonywanych na karcie graficznej za pośrednictwem technologii CUDA. Jej głęboko przemyślana struktura była inspiracją podczas budowy projektu detekcji grani. Udostępniane algorytmy zostały rozdzielone na abstrakcyjne warstwy określające poziom pracy, jak również znacznie ułatwiające wysoce wydajną implementację, modyfikację i pozwalające na swobodne wykorzystywanie elementów należących do każdej z nich z osobna. Możemy wyróżnić następujące poziomy: urządzenia, bloku wątków, splotu wątków i pojedynczego wątku. Realizacja poszczególnych zadań została zamknięta w szablony klas. Zawierają one szereg parametrów, przy użyciu których możemy swobodnie, a zarazem dokładnie dopasować je do naszych potrzeb zapewniając optymalne działanie. Z tego względu biblioteka jest opublikowana w postaci plików nagłówkowych z otwartym kodem. Ma ona tym samym ogromną wartość edukacyjną dla użytkownika pragnącego zapoznać się z implementacją najwyższej klasy. W niniejszej pracy korzystamy z tej biblioteki w zakresie redukcji elementów w ramach bloku wątków (podczas poszukiwania granic przestrzeni zamykającej chmurę punktów), funkcji ułatwiających wektorowe wczytywanie danych, a także wielu innych drobnych, ale jakże użytecznych funkcji, czy struktur.

7.2. Trove

Trove [7], jest znacznie mniejszą biblioteką zaprojektowaną z myślą o optymalnym wektorowym wczytywaniu i zapisywaniu danych w układzie SoA (Structure of Arrays). Realizacja odbywa się w oparciu o transpozycję, która wykorzystuje wbudowaną operację `__shuffle` do odpowiedniej dystrybucji danych pomiędzy wątki w ramach splotu. Dzięki takiej metodzie komunikacji wątków nie jest potrzebna pamięć współdzielona. Wewnętrzna implementacja transpozycji opiera się o statyczną, rekursywną strukturę tablicy, co pozwala jeszcze w czasie kompilacji wyznaczyć indeksy i wczytać dane do rejestrów. Niestety skutkuje to dużym zużyciem rejestrów. Natomiast chcąc wykorzystać wewnętrzną strukturę tablicy, żeby złagodzić ten problem, trzeba pamiętać o jej statycznym adresowaniu, co stwarza dodatkowe trudności. Niemniej jednak, w ramach projektu, skorzystano z tej biblioteki w ramach jednego z wariantów wewnętrznej realizacji czytania kafelków.

8. Zakończenie

W niniejszej pracy zaprojektowaliśmy i zaimplementowaliśmy kilka równoległych wersji algorytmu detekcji grani funkcji gęstości wielowymiarowej zmiennej losowej w technologii CUDA. Pierwszy wariant, tzw. podejście „siłowe", świetnie wpasowuje się w model wykonania programu przez karty graficzne, ze względu na niezwykle dużą niezależność obliczeń. Zastosowanie szeregu technik optymalizacji kodu wykonywanego na GPU pozwoliło uzyskać ok 56× i 157× krotne przyśpieszenie w stosunku do wielowątkowej realizacji na CPU, dla wejściowej chmury składającej się z 1 mln., odpowiednio dwu- lub trójwymiarowych punktów.

Pomimo tak znacznego przyśpieszenia mieliśmy świadomość, że istotna część obliczeń wykonywanych przez aktualny algorytm jest redundantna. Wiedzieliśmy, że można temu chociaż w pewnym stopniu zapobiec. Mając powyższe na uwadze, stworzono kolejną implementację wykorzystującą popularną strategię zawężania obszaru zainteresowania poprzez podział przestrzeni na kafelki. W tym celu zbudowaliśmy hierarchiczną strukturę drzewa brył ograniczających. Przeprowadziliśmy następnie, na CPU, badania różnych rodzajów kafelków (lokalne, grupowe, rozszerzone) i sposobu realizacji detekcji grani (lokalna, hybrydowa) wewnątrz każdego z nich, by wybrać optymalną pod względem czasu wykonania i jakości konfigurację. Przeniesienie jej na kartę graficzną stawiało wiele nowych problemów i wyzwań. Ich rozwiązanie ostatecznie zaowocowało ponownie sporym, bo ok. 47× i 80× dla zagadnienia odpowiednio dwu- i trójwymiarowego, wzrostem wydajności w stosunku do wielowątkowej wersji na CPU.

Na mniejsze przyśpieszenie równoległej wersji kafelkowej wykonywanej na GPU ma wpływ kilka czynników. Pierwszym z nich jest globalna decymacja, będąca częścią algorytmu hybrydowej detekcji grani. Pracuje ona na wielu (coraz mniejszych) zbiorach *wybrańców* rozproszonych po wszystkich kafelkach,

a co za tym idzie nie znajdujących się w spójnym obszarze pamięci. Taki układ danych wnosi duży narzut logiki zarządzającej obliczeniami, jak również jest gorzej dopasowany do architektury GPU. Warto tutaj przypomnieć, że karta GTX 750Ti z rodziny Maxwell, nieznacznie lepiej sobie radziła w tej sytuacji, w porównaniu do drugiej (starszej generacji) karty GTX TITAN, właśnie dzięki m.in. większym rozmiarom pamięci podręcznej. Drugim elementem mającym wpływ na przyśpieszenie jest wielkość kafelka – im jest ich mniej i ich pojemność jest większa, tym coraz bardziej wracamy do podejścia „siłowego". Dlatego też prawdopodobnie jest możliwe uzyskanie lepszego przyśpieszenia dla innej konfiguracji rozmiaru i liczby kafelków. Porównując natomiast osiągnięte czasy wykonania algorytmu równoległej detekcji grani do oryginalnej, sekwencyjnej implementacji Marka Rupniewskiego, otrzymujemy wzrost wydajności o trzy–cztery rzędy wielkości.

Na koniec, przeprowadziliśmy jeszcze zestaw testów obrazujących skalowalność finalnego rozwiązania względem liczby wymiarów, rozmiaru chmury punktów, czy też weryfikujących jakość aproksymacji w zależności od parametrów wejściowych algorytmu detekcji grani. W ich wyniku otrzymaliśmy liniowy wzrost czasu pracy programu, a także stosowne wytyczne co do wyboru wartości parametru R_1. Należy zwrócić uwagę, że wciąż jednak wykorzystujemy podejście „siłowe" w obrębie pojedynczego kafelka. Toteż w jednym z rozdziałów zarysowaliśmy krótko możliwe zastosowanie struktur drzewiastych, zwracając uwagę na ich ograniczenia. Z doświadczenia wiemy, że aby uzyskać optymalną wydajność, wiele algorytmów wymaga specjalizowanych wersji do działania na „dużych" i „małych" zbiorach danych. Stąd też obecna implementacja zaprojektowana została z myślą o pracy na niewielkiej liczbie wymiarów. Problem dużej liczby wymiarów, jest tak naprawdę prostszy i wymaga zupełnie innego podejścia, zakładającego obliczenia na macierzach dużych rozmiarów. Niestety, ze względu na ograniczony czas, nie udało się podjąć próby realizacji takiej specjalizacji wersji programu.

Bibliografia

[1] Adintez Andrew. *Adaptive Parallel Computation with CUDA Dynamic Parallelism*. https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/, May 2014.

[2] Adintez Andrew. *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics*. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/, October 2014. Nvidia Parallel For All Blog.

[3] S. Arya and D. Mount. Biblioteka ANN. http://www.cs.umd.edu/~mount/ANN/.

[4] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 271–280, 1993.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, and R. Silverman et A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[6] Arefin A.S., Riveros C., Berretta R., and Moscato P. Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus. *PLoS ONE*, 7(8):e44000, 2012.

[7] Catanzaro Bryan. Trove library. https://github.com/bryancatanzaro/trove, 2013.

[8] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and et al. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[9] Knuth D.E. *The art of computer programming*. Addison-Wesley, 2nd edition, 1998.

[10] Lian Fang and David C. Gossard. Multidimensional curve fitting to unorganized data points by nonlinear minimization. *Computer-Aided Design*, 27(1):48–58, 1995.

[11] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *CVPR Workshop on Computer Vision on GPU*, Anchorage (AK), USA, 2008.

[12] B. Gaster, L. Howes, and D.R. Kaeli. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.

[13] A. Ardeshir Goshtasby. Grouping and parameterizing irregularly spaced points for curve fitting. *ACM Trans. Graph.*, 19(3):185–203, July 2000.

[14] Mark Harris. Optimizing parallel reduction in cuda. *http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf*, 2007.

[15] Mark Harris. *CUDA Pro Tip: Occupancy API Simplifies Launch Configuration*. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/, July 2014.

[16] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[17] Wen-mei W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[18] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39, pages 877–892. CRC Press, 2004.

[19] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 1st edition, 2010.
[20] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann. Morgan Kaufmann, 2nd edition, 2013.
[21] Jacek Kudrewicz. *Fraktale i chaos*. Wydawnictwo WNT, Warszawa, 2015.
[22] In-Kwon Lee. Curve reconstruction from unorganized points. *Computer Aided Geometric Design*, 17(2):161 – 177, 2000.
[23] David Levin. The approximation power of moving least-squares. *Math. Comput.*, 67(224):1517–1531, October 1998.
[24] D.M. Mount, Siu-Wing Cheng, Stefan Funke, Mordecai Golin, Piyush Kumar, Sheung-Hung Poon, and Edgar Ramos. Curve reconstruction from noisy samples. *Computational Geometry*, 31(1):63 – 100, 2005.
[25] A. Munshi, B.R. Gaster, and T.G. Mattson. *OpenCL Programming Guide*. Graphics programming. ADDISON WESLEY, 2011.
[26] Marek Nałęcz. *Równoległe Implementacje Metod Numerycznych*, 2013. Slajdy do wykładu z przedmiotu RIM, Politechnika Warszawska WEiTI.
[27] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.
[28] Sakharnykh Nikolay. *GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell*. `https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/`, March 2015.
[29] Nvidia. Thrust library. `https://developer.nvidia.com/thrust`.
[30] Nvidia. *NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built*, 2012.
[31] Nvidia. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.
[32] Nvidia. *Nvidia GeForce GTX 980*, 2014.
[33] Nvidia. *CUBLAS Library User Guide v7.5*, September 2015.
[34] Nvidia. *CUDA Binary Utilities v7.5*, 2015.
[35] Nvidia. *CUDA C Best Practices Guide v7.5*, 2015.
[36] Nvidia. *CUDA C Programming Guide v7.5*, 2015.
[37] Nvidia. *Inline PTX Assembly in CUDA. Application Note*, September 2015.
[38] Nvidia. *Parallel Thread Execution ISA. Application Guide*, September 2015.
[39] Nvidia. *Tuning CUDA aplications for Kepler*, 2015.
[40] Nvidia. Cuda unbound library. `https://nvlabs.github.io/cub/`, 2016.
[41] Strona Nvidia CUDA developer zone. `https://developer.nvidia.com/category/zone/cuda-zone`.
[42] Strona domowa Nvidia CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.
[43] R. Panigrahy. *Hashing, Searching, Sketching*. PhD thesis, Stanford University, 2006.
[44] H. Pottmann and T. Randrup. Rotational and helical surface approximation for reverse engineering. *Computing*, 60(4):307–322, June 1998.
[45] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2010.
[46] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten, and Wen-Mei W. Hwu. Gpu acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 273–282, New York, NY, USA, 2008. ACM.
[47] G. J. Ross. Hight performance histogramming on massively parallel processors, 2014.
[48] Marek W. Rupniewski. Curve reconstruction from noisy and unordered samples. In *Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods*, pages 183–188, 2014.

[49] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
[50] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-neighbor Methods in Learning and Vision: Theory and Practice*. MIT Press, 2005.
[51] S.S. Skiena. *The Algorithm Design Manual*. Springer London, 2009.
[52] F. Nielsen V. Garcia, E. Debreuve and M. Barlaud. k-nearest neighbor search: fast gpu-based implementations and application to high-dimensional feature matching. In *In Proceedings of the IEEE International Conference on Image Processing (ICIP)*, Hong Kong, China, September 2010.
[53] Glenn Volkema and Gaurav Khanna. Scientific computing using consumer video-gaming hardware devices. preprint, July 2016.
[54] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Conf. on Supercomputing*, Piscataway, NJ, USA, 2008. ACM/IEEE.
[55] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 08 2010.
[56] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.
[57] N. Wilt. Histograms privatized for fast, level performance. In *GPU Technology Conference*, 2014.
[58] Hancheng Wu, Da Li, and Michela Becchi. Compiler-assisted workload consolidation for efficient dynamic parallelism on gpu. jun 2016.