STUDIA INFORMATICA 2011
Volume 32 Number 2A (96)

Robert PAWŁOWSKI, Dariusz MROZEK
Politechnika Śląska, Instytut Informatyki

PRZYSPIESZENIE ALGORYTMU SMITHA-WATERMANA Z UŻYCIEM PROCESORA GRAFICZNEGO

Streszczenie. Wraz z wprowadzeniem przez firmę NVIDIA technologii CUDA wykorzystanie potencjału kart graficznych stało się łatwiejsze. W artykule przedstawiono metodę istotnego przyśpieszenia wykonania algorytmu Smitha-Watermana, znajdującego optymalne, lokalne dopasowanie dwóch sekwencji, takich jak sekwencje aminokwasów lub nukleotydów. Uzyskane wyniki sugerują, że możliwe jest dokładne przeszukiwanie bioinformatycznych baz danych w rozsądnym czasie.

Słowa kluczowe: bioinformatyka, dopasowanie sekwencji, GPU, CUDA

ACCELERATING SMITH-WATERMAN ALGORITHM WITH THE USE OF GRAPHICS PROCESSING UNIT

Summary. CUDA is a technology introduced by NVIDIA Corporation, which allows software developers to take advantage of GPU resources relatively easily. This paper presents an approach leading to significant acceleration of the execution of the Smith-Waterman algorithm. The algorithm finds the best local alignment of two sequences, such as amino acid or nucleotide sequences. The results show that it is possible to search bioinformatics databases accurately within a reasonable time.

Keywords: bioinformatics, sequence alignment, GPU, CUDA

1. Wprowadzenie

Bioinformatyczne bazy danych stanowią nieocenione narzędzie przy realizacji badań z zakresu biologii molekularnej oraz dziedzin pokrewnych. Porównując do znanych sekwencji, które zgromadzono w bazie danych, niezidentyfikowaną sekwencję nukleotydów lub aminokwasów, którą uzyskano w trakcie badań, można wnioskować o pokrewieństwie ewo-

lucyjnym niezidentyfikowanej sekwencji, o kodowanej przez nią strukturze, a nawet o roli biologicznej kodowanej struktury.

Siła narzędzia, jakim są bioinformatyczne bazy danych stale wzrasta. Przez kilka ostatnich lat liczba dodawanych do baz danych sekwencji aminokwasów i nukleotydów w ciągu roku niemal się podwaja [4], [5]. Nieustannie poszukiwane są wydajniejsze metody przeszukiwania zebranej informacji. W związku z zahamowaniem tempa przyrostu taktowania jednostek centralnych oraz rosnącą dostępnością układów wieloprocesorowych i wielordzeniowych, warto podejmować próby opracowania algorytmów przeszukujących bazy danych w sposób współbieżny.

W artykule przedstawiono możliwość wykorzystania potencjału zawartego w układach kart graficznych wspierających technologię CUDA (ang. *Compute Unified Device Architecture*) [11] do przyspieszenia operacji przeszukiwania baz danych sekwencji nukleotydów lub aminokwasów. Za wyborem technologii CUDA przemawia kilka ważnych czynników: dostępność i duża liczba urządzeń wspierających tę technologię, przystępny i dobrze udokumentowany interfejs programowania aplikacji, przenośność oraz duża skalowalność opracowanych rozwiązań.

Spośród znanych algorytmów porównywania sekwencji, skoncentrowano się na algorytmie Smitha-Watermana, który ma duże znaczenie praktyczne, ale którego złożoność obliczeniowa nie pozwala na jego powszechne stosowanie.

2. Algorytm Smitha-Watermana

Algorytm T. F. Smitha oraz M. S. Watermana [13] jest algorytmem lokalnego dopasowania dwóch sekwencji nukleotydów lub aminokwasów. Wskazuje on relatywnie krótkie i wykazujące duże podobieństwo (ang. *similarity*) odcinki pary porównywanych sekwencji. Pozwala to wychwycić konserwatywne motywy sekwencji oraz mutacje strukturalne powstałe w procesie ewolucji, takie jak translokacje czy duplikacje fragmentów sekwencji. W odróżnieniu od algorytmów korzystających z heurystyki, których najpopularniejszym przedstawicielem jest BLAST (ang. *Basic Local Alignment Search Tool*) [1], algorytm Smitha-Watermana charakteryzuje się większą czułością. Przekłada się to na zdolność do wykrywania bardziej odległego podobieństwa porównywanych sekwencji.

W algorytmie Smitha-Watermana wykorzystano ideę programowania dynamicznego. Programowanie dynamiczne jest strategią tworzenia algorytmów, zasadniczo algorytmów optymalizacji, w której znaczne zmniejszenie złożoności obliczeniowej uzyskuje się przez wykorzystanie pomocniczej struktury danych. W tym algorytmie taką strukturą danych jest tablica dwuwymiarowa, która dalej będzie oznaczana przez M i będzie nazywana macierzą

dopasowania. Macierz M ma wymiar $(p+1)\times(q+1)$, gdzie p i q są długościami dopasowywanych sekwencji. Niech $P(i)$ oraz $Q(j)$, $i=2,...,p+1$, $j=2,...,q+1$ będą symbolami w porównywanych sekwencjach P i Q na pozycji odpowiednio $i-1$ oraz $j-1$, wówczas, rozpoczynając od $i=2$ i $j=2$, elementy macierzy M wyznacza się z zależności:

$$M_{ij}=\max\left\{\begin{array}{c} S_{ij}+M_{(i-1)(j-1)} \\ \max\limits_{1\le k\le i-1,k\in N}\left\{M_{kj}-\left|G_P\right|\right\} \\ \max\limits_{1\le l\le j-1,l\in N}\left\{M_{il}-\left|G_P\right|\right\} \\ 0 \end{array}\right\}, \tag{1}$$

$$S_{ij}=\begin{cases} M_A, gdy\, P(i)=Q(j) \\ M_P, gdy\, P(i)\neq Q(j) \end{cases}. \tag{2}$$

Dla pierwszego wiersza i pierwszej kolumny macierzy:

$$M_{i1}=M_{1j}=0. \tag{3}$$

Wartość M_A jest premią za zgodność, a M_P jest karą za niezgodność symboli *P(i)* i *Q(j)*. Niezgodność elementów sekwencji jest interpretowana jako efekt mutacji – substytucji – w procesie ewolucji. Wartości M_A oraz M_P mogą być obrane arbitralnie, zgodnie z zasadami przedstawionymi w [13] lub według jednej ze znanych macierzy substytucji (PAM, BLOSUM lub innej) [9]. G_P oznacza karę za przerwę w dopasowywanych sekwencjach, która powstała w wyniku hipotetycznych delecji lub insercji. Zależnie od wartości parametrów, G_P może mieć postać funkcji stałej lub liniowej:

$$G_P=G_O+G_E n, \tag{4}$$

gdzie: G_O jest karą za otwarcie przerwy, a $G_E n$ karą za jej wydłużanie, proporcjonalną do długości przerwy n; G_E jest współczynnikiem proporcjonalności. Przez $|G_P|$ rozumie się wartość bezwzględną wartości funkcji G_P.

Oceną dopasowania pary sekwencji jest największa wartość w wypełnionej macierzy M. Im większa wartość oceny dopasowania, tym bardziej sekwencje są do siebie strukturalnie podobne.

Algorytm Smitha-Watermana (w przedstawionej postaci) charakteryzuje się złożonością czasową, rzędu *O(pq)*. Mimo znacznej mocy obliczeniowej współczesnych komputerów, nadal jest to złożoność zbyt wysoka, aby stosować algorytm Smitha-Watermana jako rutynową metodę przeszukiwania dużych zbiorów sekwencji.

3. Model programowania z CUDA

W układach kart graficznych z technologią CUDA wysoką skalowalność osiągnięto poprzez hierarchiczne zorganizowanie podstawowych jednostek wykonawczych, którymi są wątki (ang. *thread*). Każdy wątek posiada swój indeks, wektor współrzędnych, odpowiadający jego lokalizacji w jedno-, dwu- lub trójwymiarowej strukturze organizacyjnej, zwanej blokiem (ang. *block*). Indeks wątku jest unikatowy w ramach bloku. Maksymalna liczba aktywnych wątków składających się na blok jest ograniczona, zależy od sprzętu, lecz nigdy nie jest mniejsza niż 768.

Bloki wątków tworzą jedno- lub dwuwymiarową strukturę wyższego poziomu – siatkę (ang. *grid*). Podobnie jak wątki, również bloki mają swój unikatowy identyfikator. Co bardzo istotne, w przeciwieństwie do liczby wątków w bloku, liczba bloków w siatce jest teoretycznie nieograniczona [10]. Każdy blok wątków obsługiwany jest przez jeden multiprocesor strumieniowy (SM, ang. *Streaming Multiprocessor*). Liczba dostępnych multiprocesorów zależy od posiadanej karty graficznej. Niemniej w układach z CUDA 1.1 każdy z nich złożony jest z ośmiu rdzeni procesora skalarnego (SP, ang. *Scalar Processor*), dwóch specjalnych jednostek funkcyjnych (SFU, ang. *Special Function Unit*), jednostki sterującej i szybkiej pamięci współdzielonej [10].

Na potrzeby technologii CUDA opracowano nową architekturę, nazwaną SIMT (ang. *Single Instruction, Multiple Thread*). W architekturze tej multiprocesor mapuje każdy z wątków na jeden rdzeń procesora skalarnego, gdzie każdy z wątków wykonuje się niezależnie z własnym licznikiem rozkazów i stanem rejestrów. Jednostka sterująca multiprocesora tworzy, zarządza, planuje i wykonuje wątki w grupach nazywanych osnową (ang. *warp*). Na osnowę składają się 32 wątki. Przez półosnowę rozumie się pierwsze bądź drugie 16 kolejnych wątków osnowy. Wątki w osnowie wykonują tę samą instrukcję, ale operują na różnych danych tak, jak ma to miejsce w architekturze SIMD. Pełna wydajność w architekturze SIMT jest osiągana, kiedy wszystkie 32 wątki mają tę samą ścieżkę wykonywania.

W układach kart graficznych z CUDA wyróżnia się pięć typów pamięci. Układ NVIDIA GeForce 88000 GT wykorzystywany na potrzeby testów charakteryzuje się następującymi zasobami każdego z rodzajów pamięci:

- 8192 32-bitowe rejestry, przypadające na jeden multiprocesor i dzielone pomiędzy wszystkie wątki w bloku,
- 16 kB pamięci współdzielonej, fizycznie zlokalizowanej w 16 bankach o dostępnie równoległym,
- 8 kB buforu pamięci stałej, przypadającej na jeden multiprocesor,
- 6÷8 kB buforu pamięci tekstur, przypadającej na jeden multiprocesor,

- 512 MB pamięci globalnej, w której zlokalizowane są obszary pamięci stałej (64 kB), pamięci tekstur i pamięci lokalnej.

Pierwsze cztery typy pamięci są fizycznie zlokalizowane w multiprocesorze i zapewniają bardzo szybki dostęp do przechowywanych danych. Ich wadą jest niewielki rozmiar. Natomiast pamięć globalna i pamięć lokalna, chociaż mają znacznie większe rozmiary, to dostęp do nich – z powodu braku buforowania – jest dwa rzędy wielkości wolniejszy [10]. Dlatego dostęp do pamięci globalnej należy zawsze optymalizować, zapewniając tzw. dostęp (odczyt lub zapis) spójny, w którym kolejne wątki półosnowy odwołują się do odpowiednio wyrównanych, kolejnych 32-, 64- lub 128-bitowych słów.

Pamięć stała i pamięć tekstur są pamięciami buforowanymi. Rozmiar pamięci stałej jest ściśle określony (tutaj 64 kB), natomiast pamięć tekstur to dynamicznie wydzielony obszar pamięci globalnej. Program wykonywany przez układ karty graficznej może tylko odczytywać zawartość obu tych pamięci. Gdy wartość z odczytywanej komórki pamięci stałej lub pamięci tekstur nie znajduje się w buforze, czas dostępu jest bliski czasowi dostępu do pamięci globalnej. Z kolei, gdy pożądana komórka pamięci została wcześniej zbuforowana, czas dostępu jest porównywalny z czasem dostępu do rejestru. Pamięć tekstur dodatkowo ma kilka użytecznych własności. Dla problemu rozważanego w artykule najważniejszą własnością jest lokalne buforowanie odczytywanych danych.

W przypadku urządzeń ze zdolnością obliczeniową w wersji 1.x, pamięć współdzielona jest fizycznie zaimplementowana jako szesnaście banków o równej pojemności. Dostęp do poszczególnych banków może odbywać się równolegle. To sprawia, że wątki półosnowy zamiast wykonywać kilka sekwencyjnych odczytów lub zapisów danych do jednego banku pamięci, mogą równolegle dokonać operacji na danych rozmieszczonych w różnych bankach. Czas wykonywania takiej równoległej operacji będzie równy czasowi realizacji pojedynczej operacji odczytu lub zapisu.

Konflikt banków (ang. *bank conflict*) jest sytuacją, w której dwa lub więcej wątków półosnowy prosi o dostęp do tego samego banku pamięci współdzielonej. Wystąpienie konfliktu banków implikuje konieczność sekwencyjnego dostępu do danych w banku. Całkowicie niezoptymalizowany dostęp do banków pamięci współdzielonej może sumarycznie trwać dłużej niż dostęp do pamięci globalnej [2].

4. Przegląd istniejących rozwiązań

Problem współbieżnej realizacji algorytmu Smitha-Watermana z zastosowaniem technologii CUDA jest zagadnieniem relatywnie nowym. Jedną z pierwszych prac, w której osiągnięto znaczne przyśpieszenie jest [8]. Zaprezentowany algorytm SW-CUDA był realizowany

przez układ NVIDIA GeForce 8800 GTX, osiągając wydajność do 1,8 GCUPS (ang. *Giga Cell Updates Per Second*). Wydaje sie, że obciążeniem dla tej metody jest narzut, związany z wyznaczaniem i późniejszym wykorzystywaniem tzw. profilu zapytania (ang. *query-profile*). Eliminuje on konieczność swobodnego dostępu do pamięci globalnej, w celu odczytania wartości elementów macierzy substytucji. Profil zapytania jest przechowywany w pamięci tekstur. Stanowi to istotny narzut w przypadku długich sekwencji wejściowych, ponieważ – potencjalnie – często może zachodzić nietrafienie bufora (ang. *cache miss*). Obserwacje te potwierdza się w [14]. W niniejszym artykule pokazano, że unikając narzutu wynikającego ze stosowania profilu zapytania i bezpośrednio umieszczając macierz substytucji w pamięci tekstur uzyskuje się większą wydajność.

Kolejną najwydajniejszą implementację dla układów z CUDA 1.1 przedstawiono w [6], gdzie algorytm CUDASW++ 1.0 osiągał do 10 GCUPS na jednej karcie graficznej z układem NVIDIA GeForce GTX 280. Słabym punktem algorytmu jest przechowywanie macierzy substytucji w pamięci współdzielonej, co wiąże się z występowaniem konfliktu banków. W CUDASW++ 1.0 pamięć tekstur wykorzystywana jest do buforowania odczytów symboli sekwencji z bazy danych. Podczas testów przeprowadzanych w ramach niniejszego artykułu okazywało się jednak, że przyrost wydajności, wynikający z lokalnego buforowania jest relatywnie niewielki. Lepszym rozwiązaniem jest zagwarantowanie spójnych odczytów z pamięci globalnej. Wtedy własność lokalnego buforowania pamięci tekstur można wykorzystać dla osiągnięcia innych celów. Co więcej algorytm CUDASW++ 1.0 przechowuje sekwencję zadaną w pamięci stałej, której rozmiar jest istotnie ograniczony. W rezultacie CUDASW++ 1.0 działa dla sekwencji wejściowej nie dłuższej niż 59 tysięcy symboli [6].

Obecnie najwydajniejszą, opublikowaną implementacją, dla układów wspierających technologię CUDA w wersji co najmniej 1.2, jest algorytm CUDASW++ 2.0 [7], osiągający około 17 GCUPS na układzie NVIDIA GeForce GTX 280. W [7] opisano zupełnie nowe podejście, zainspirowane pracą Farrara [3], jak również poprawiony algorytm, znany z [6]. W poprawionym algorytmie powraca się do idei profilu zapytania, stosowanego w algorytmie SW-CUDA, jednakże ze zmienioną organizacją struktury danych. Dodatkowy przyrost wydajności osiągnięto przez optymalizację kodu i przechowywanie 4 kolejnych symboli zarówno sekwencji zadanej, jak i każdej z sekwencji z bazy danych, w 32-bitowych słowach. Podobne rozwiązanie przedstawiono dalej w niniejszym artykule.

5. Implementacja algorytmu Smitha-Watermana z użyciem procesora graficznego i technologii CUDA

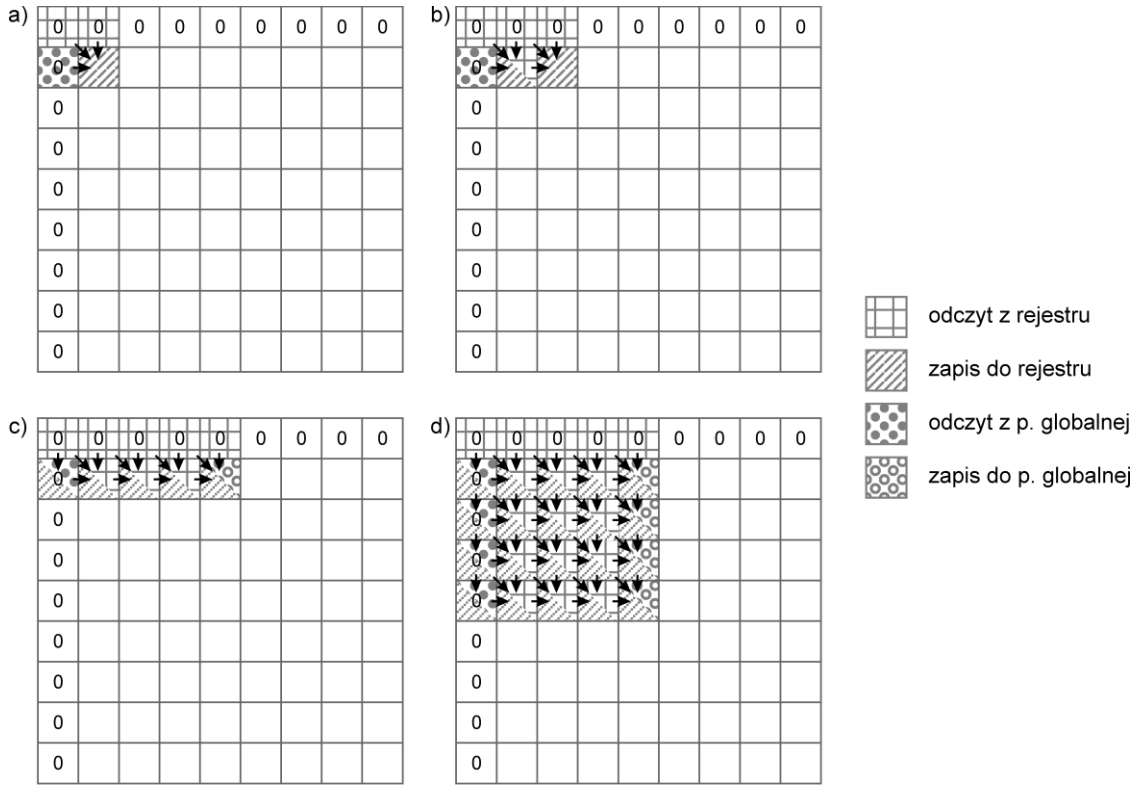
Zasadniczym założeniem proponowanej metody przyśpieszenia algorytmu Smitha-Watermana jest przeprowadzanie dopasowania pary sekwencji tylko przez jeden wątek. Stąd też zaimplementowany algorytm określono mianem SW-CUDA-STSA (STSA, ang. *Single-Thread-Single-Alignment*). Takie podejście pozwala na wyznaczanie elementów macierzy dopasowania kolumnami lub wierszami. Zyskuje się przy tym całkowitą niezależność przy wyznaczaniu wartości elementu M_{ij}. Różne dopasowania nie są względem siebie w żaden sposób zależne. W ogóle nie jest konieczna żadna wymiana informacji pomiędzy wątkami i ewentualna ich synchronizacja.

5.1. Sposób prowadzenia obliczeń w macierzy dopasowania

Ze swobodą w wyznaczaniu elementów macierzy dopasowania, można poszukiwać lepszej metody niż proste obliczanie kolejnych elementów wierszami lub kolumnami, jak ma to miejsce w przypadku bezpośredniej implementacji na standardowym procesorze. Wydaje się, że najkorzystniejszym sposobem jest przeprowadzanie obliczeń obszarami, kwadratami o boku 4×4 elementy. Do wyznaczenia wartości elementów takiego obszaru niezbędna jest znajomość wartości 9 elementów, położonych wzdłuż brzegów lewego i górnego obszaru, co pokazano na rys. 1d.

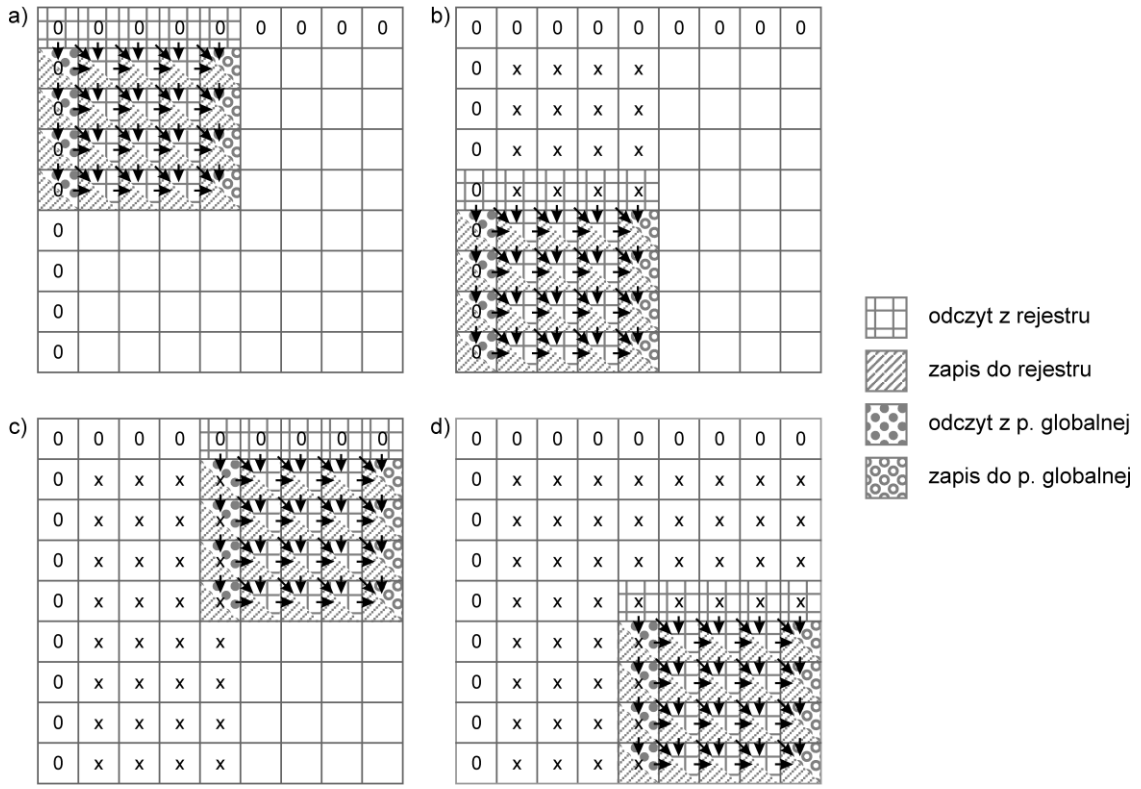
Przyjmijmy, że kolejne obszary w macierzy dopasowania obliczamy kolumnami (rys. 2), a elementy obszaru obliczamy wierszami (rys. 1). Na rys. 2 widać, że dla pierwszego obszaru w kolumnie wartości wzdłuż górnego brzegu są zawsze równe 0. Zatem przechowywanie wartości elementów górnego brzegu w pamięci globalnej byłoby niekorzystnym rozwiązaniem. Z pamięcią współdzieloną związany jest z kolei problem konfliktu banków. Dlatego wartości te najlepiej jest przechowywać w rejestrach. Również wartość elementu z lewego górnego rogu obszaru jest przenoszona pomiędzy iteracjami algorytmu za pomocą rejestru.

Jeżeli chodzi o wartości lewego brzegu obszaru, to dla każdej kolumny obszarów są one inne i równe wartościom w ostatnich komórkach wcześniejszej kolumny obszarów. Całkowita liczba tych wartości jest równa długości sekwencji odłożonej wzdłuż pionowego brzegu macierzy dopasowania. Dobrze jest, żeby dla każdego wątku liczba ta była taka sama, co pozwoliłoby łatwo zorganizować spójny dostęp do pamięci globalnej. Dla wyznaczania obszarów kolumnami, wzdłuż pionowego brzegu macierzy dopasowania zawsze musi być odłożona sekwencja taka sama dla każdego wątku, czyli sekwencja wejściowa.



Rys. 1. Kolejność wyznaczania elementów określonego obszaru macierzy dopasowania dla zaproponowanej metody

Fig. 1. Order of elements determination for a particular area in the alignment matrix for the proposed method



Rys. 2. Kolejność wyznaczania obszarów w macierzy dopasowania dla zaproponowanej metody

Fig. 2. Order of areas determination in the alignment matrix for the proposed method

5.2. Redukcja liczby transakcji

Wyznaczając wartości elementów macierzy dopasowania obszarami 4×4 elementy, ograniczona została liczba dostępów do pamięci globalnej do 4 odczytów i 4 zapisów, przypadających aż na 16 elementów. Do tych 8 dostępów trzeba jeszcze dodać 8 dostępów związanych z odczytywaniem i zapamiętywaniem danych koniecznych do wyznaczenia wartości funkcji G_P.

Liczbę dostępów do pamięci globalnej można dalej zmniejszyć. Wystarczy przechowywać 4 odczytywane lub zapisywane wartości z obszaru pamięci globalnej, jako wartości pól typu 4-bajtowego (*int* lub *float*) struktury języka C (128-bitowe słowo). Wówczas mechanizm transakcyjny technologii CUDA zapewnia, że takie 128-biotowe słowa będą odczytane lub zapisywane przez wątki osnowy tylko w dwóch transakcjach, a nie w czterech. Pod warunkiem jednak, że dostęp ten jest spójny – kolejne wątki odczytują lub zapisują kolejne 128-bitowe słowa. Reasumując, w rozważanym algorytmie liczbę wymaganych dostępów do pamięci globalnej należy podzielić jeszcze przez 2.

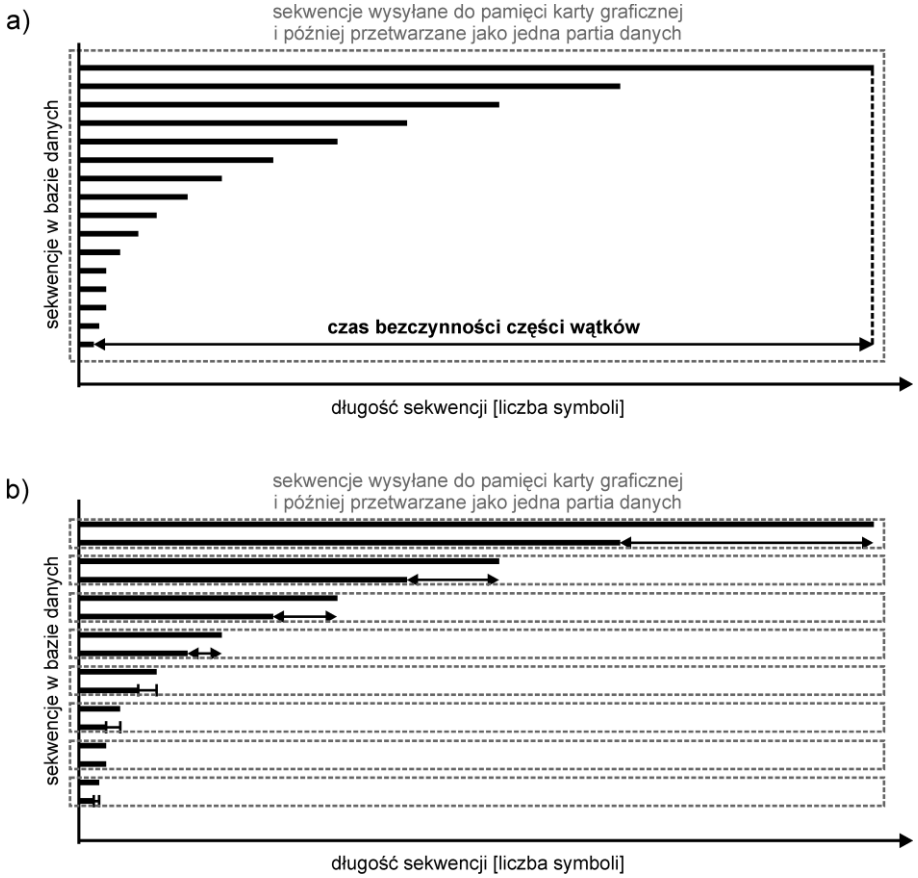
5.3. Ograniczenie czasu bezczynności wątków

Każdy wątek przeprowadza jedno dopasowanie. Dla współbieżnego działania możliwie dużej liczby wątków konieczne jest dostarczenie jednorazowo stosownie dużej części danych do pamięci układu karty graficznej. Ponieważ wątki uruchamiane są blokami, w analizowanym algorytmie liczba potrzebnych danych jest w dużym przybliżeniu proporcjonalna do liczby wątków w bloku. Po uruchomieniu programu dla karty graficznej, bloki będą pozostawały bezczynne przez pewien okres, zanim przystąpią do działania, co wynika z konieczności transferu odpowiednio licznego zbioru danych.

Czas bezczynności przy uruchamianiu kolejnych bloków można skutecznie ograniczyć, przeprowadzając transfer danych do urządzenia partiami z użyciem strumieni i odpowiednio dobierając liczbę wątków w bloku. Podczas testów najkorzystniejsze rezultaty uzyskiwano dla 64 wątków w bloku i rozmiaru partii danych około 20 MB. Podział danych na partie o stosownym rozmiarze jest całkowicie niezależny od długości czy postaci sekwencji zadanej. Zatem podziału można dokonać wcześniej, wczytywać odpowiednio podzielone dane z pliku i później przechowywać dane w pamięci RAM, wielokrotnie je wykorzystując do realizacji zapytań.

Z organizacją danych, które są przesyłane do urządzenia związany jest jeszcze jeden problem. Znaczna większość sekwencji zgromadzonych w typowej bazie danych to sekwencje relatywnie krótkie. Sekwencje bardzo długie zwykle stanowią tylko mały procent bazy danych. Taką hipotetyczną sytuację przedstawiono (w dużym uproszczeniu) na rys. 3.

Pamiętajmy, że wątki w bloku działają współbieżnie. Jeżeli wykonywany program dla części wątków bloku jest z jakiś względów krótszy, to wątki te nie stają się częścią innego bloku. Przedwcześnie zakończone wątki nadal zajmują zasoby, chociaż nie wykonują już żadnych instrukcji – przedziały bezczynności zostały oznaczone strzałką na rys. 3a. Dlatego sekwencje z bazy danych przetwarzane przez wątki bloku powinny mieć zbliżoną długość. Dzięki temu skraca się czas pomiędzy zakończeniem działania „najszybszego" i „najwolniejszego" wątku. Zbliżoną długość sekwencji, składających się na dane dla wątków bloku, można osiągnąć przez wstępne przetworzenie bazy danych, które następuje jeszcze przed podzieleniem danych na partie, a którym jest posortowanie sekwencji względem ich długości malejąco lub rosnąco.



Rys. 3. Wpływ długości sekwencji w zbiorze przetwarzanych danych na czas bezczynności części wątków bloku: a) cały zbiór danych przetwarzany na raz, b) cały zbiór danych podzielony na mniejsze podzbiory, przetwarzane niezależnie. Liczba sekwencji w podzbiorze jest proporcjonalna do liczby wątków w bloku

Fig. 3. Dependence of sequences length in a set of processed data for the idle time of some threads in a block: a) entire data set processed at once, b) entire data set divided into smaller subsets, processed independently. The number of sequences in the subset is proportional to the number of threads in the block

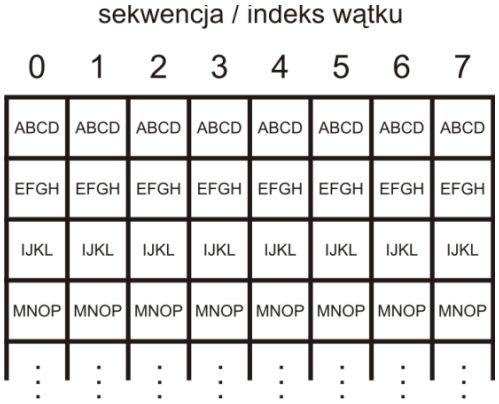
Dla uproszczenia, niech na rys. 3 partia danych oznacza podzbiór ze zbioru sekwencji z bazy danych, który przetwarzany jest tylko przez jeden blok wątków. Jak widać na rys. 3b,

po posortowaniu sekwencji i obraniu mniejszej liczby wątków w bloku (mniejsze partie danych), różnica długości pomiędzy sekwencjami najdłuższą i najkrótszą jest mniejsza (krótsze strzałki przedziałów bezczynności) i krótszy jest czas blokowania zasobów niż dla przypadku z rys. 3a.

5.4. Organizacja danych

Dane przetwarzane przez wątki bloku umieszczane są w pamięci globalnej, ze względu na ich liczbę. Dostęp do nich można usprawnić, obierając odpowiednią strukturę ich przechowywania jeszcze w pamięci operacyjnej komputera, zanim zostaną one wysłane do pamięci karty graficznej.

Wyobrazimy sobie pamięć operacyjną komputera jako dwuwymiarową tablicę z adresacją wierszami. Załóżmy ponadto, że każda z komórek takiej tablicy mieści 32-bitowe słowo, a długość wiersza tablicy – liczba 32-bitowych komórek w wierszu – jest równa obranej przez programistę liczbie wątków w bloku. Do każdej komórki możemy zapisać 4 symbole sekwencji, bo każdy symbol (typ *char* języka C) zajmuje 8 bitów. Przy takich założeniach, sekwencje najlepiej jest umieszczać w tablicy kolumnami, co zilustrowano na rys. 4. Każda kolumna stanowi zbiór danych (sekwencję z bazy) do przetwarzania dla jednego wątku bloku. Jeżeli sekwencja jest krótsza od najdłuższej sekwencji w tablicy, brakujące symbole uzupełnia się dowolnie obranym symbolem pustym, który należy uwzględnić w macierzy substytucji, przypisując odpowiadającym mu elementom wartość 0.



Rys. 4. Organizacja sekwencji z bazy danych w pamięci operacyjnej
Fig. 4. Arrangement of database sequences in RAM

Implementując opisany sposób organizacji przetwarzanych danych, zyskuje się spójne odczyty danych przez wszystkie wątki w bloku. Do sukcesywnych kolumn tablicy przypisane są sukcesywne wątki bloku. W razie potrzeby, liczbę kolumn, a tym samym liczbę sekwencji w tablicy danych można zwiększać o wielokrotność liczby wątków w bloku i przypisywać do każdego wątku kolumny „oddalone” od siebie o liczbę wątków w bloku. Co więcej, zamiast

odczytywać kolejne symbole sekwencji pojedynczo, odczytujemy aż 4 symbole podczas jednego dostępu do pamięci globalnej.

5.5. Przechowywanie sekwencji wejściowej i macierzy substytucji

W zaproponowanym algorytmie wykorzystanie pamięci buforowanej, konkretnie pamięci tekstur, ograniczono do przechowywania wartości macierzy substytucji i sekwencji wejściowej. Pamięć tekstur najlepiej nadaje się do przechowywania danych o charakterze przestrzennym, a za takie można uznać właśnie wartości macierzy substytucji. Sekwencję wejściową również umieszczono w pamięci tekstur, gdyż pamięć ta charakteryzuje się buforowaniem lokalnym. Przy ładowaniu elementu wektora lub macierzy do bufora, ładowane są również elementy sąsiednie. W konsekwencji, odczytując z pamięci tekstur pierwszy symbol sekwencji wejściowej, zaraz po jego załadowaniu do bufora, niejako „w tle", ładowany jest kolejny symbol.

Założono, że porównywane sekwencje składają się tylko z dużych liter alfabetu łacińskiego. Aby przyśpieszyć mapowanie wartości odpowiadających symbolom alfabetu na indeksy wierszy i kolumn macierzy substytucji, przyjęto rozmiar macierzy równy $27 \times 27 - 26$ liter alfabetu oraz tzw. symbol pusty. Za symbol pusty najlepiej jest przyjąć następnik symbolu 'Z' w kodzie ASCII, wtedy łatwo można wyliczać indeksy elementów macierzy substytucji, odejmując od wartości kodu ASCII symbolu z sekwencji wartość kodu symbolu 'A'. Rozmiar macierzy równy 27×27 elementów złożonej z 32-bitowych słów oznacza, że cała macierz zajmuje 2916 bajtów. Rozmiar buforu pamięci tekstur w technologii CUDA nie jest ściśle określony, ale wynosi co najmniej 6 kB. Po pierwszym odczycie danych z pamięci globalnej do bufora, wszystkie kolejne odczyty powinny odbywać się już tylko z bufora, czyli bardzo szybko. Ewentualne, raczej sporadyczne, nietrafienia bufora pamięci tekstur są spowodowane równoczesnym używaniem bufora do odczytu symboli sekwencji wejściowej.

6. Testy wydajnościowe

Algorytmy oceniano na podstawie wyników pomiarów dla 20 zadanych sekwencji, o długości od 127 do 2999 aminokwasów, z przeszukiwanej bazy danych Swiss-Prot 2010_08 z 13 lipca 2010 r., składającej się z 518415 rekordów (sekwencji). Do badań wykorzystano kartę graficzną firmy ZOTAC z układem NVIDIA GeForce 8800 GT z 14 multiprocesorami (96 procesorami skalarnymi) oraz z 512 MB pamięci globalnej. Karta graficzna wspiera technologię CUDA w wersji 1.1. Urządzenie zainstalowano w komputerze osobistym z systemem operacyjnym Windows 7 w wersji 64-bitowej i CUDA SDK (ang. *Software Development Kit*)

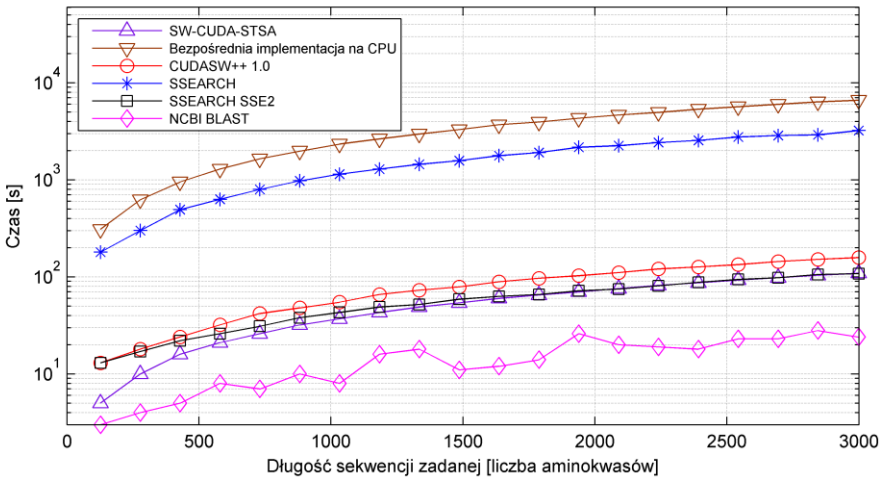
w wersji 2.3. Wykorzystywany komputer miał procesor Intel Core 2 Quad Q9300 2.5 GHz oraz 8 GB pamięci RAM.

Wyniki uzyskane dla opracowanego algorytmu SW-CUDA-STSA porównano z wynikami ogólnie dostępnych implementacji. Algorytm porównano z własną, sekwencyjną, bezpośrednią implementacją algorytmu Smitha-Watermana, z algorytmem CUDA-SW++ 1.0 [6] oraz z uruchamianymi lokalnie: algorytmem dla programu SSEARCH z pakietu FASTA w wersji 3.5.4.11 [12] i algorytmem NCBI BLAST 2.2.23.

CUDA-SW++ 2.0 [7] jest obecnie najpopularniejszą opublikowaną implementacją algorytmu Smitha-Watermana, dedykowaną na architekturę SIMT. Jednakże do badań wybrano CUDA-SW++ w wersji 1.0. Nowsza wersja CUDA-SW++ 2.0 wymaga urządzenia wspierającego technologię CUDA w wersji co najmniej 1.2.

Program SSEARCH występuje w dwóch odmianach. Podstawowa wersja to zoptymalizowany algorytm Smitha-Watermana, który jest około dwukrotnie szybszy od implementacji bezpośredniej [3], co potwierdzają zamieszczone dalej wyniki pomiarów (rys. 5). Druga wersja programu jest ukierunkowana na procesory firmy Intel, dysponujące technologią rozkazów wektorowych SSE2 (ang. *Streaming SIMD Extensions 2*).

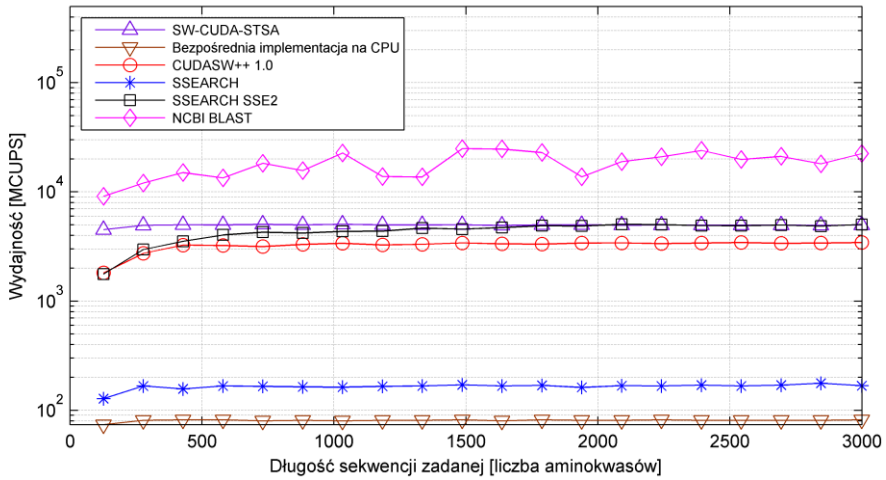
Pomiarów dla wszystkich implementacji dokonano dla tej samej macierzy substytucji, którą była macierz BLOSUM62, jak również dla tych samych wartości kary za otwarcie przerwy (-10) i za wydłużanie przerwy (-2). Wartości dobrano zgodnie z zaleceniami przedstawionymi w pracy [13]. Na rys. 5 zamieszczono rezultaty pomiarów czasów realizacji poszczególnych implementacji.



Rys. 5. Czas wykonania algorytmu SW-CUDA-STSA i algorytmów odniesienia dla różnych długości sekwencji zadanej

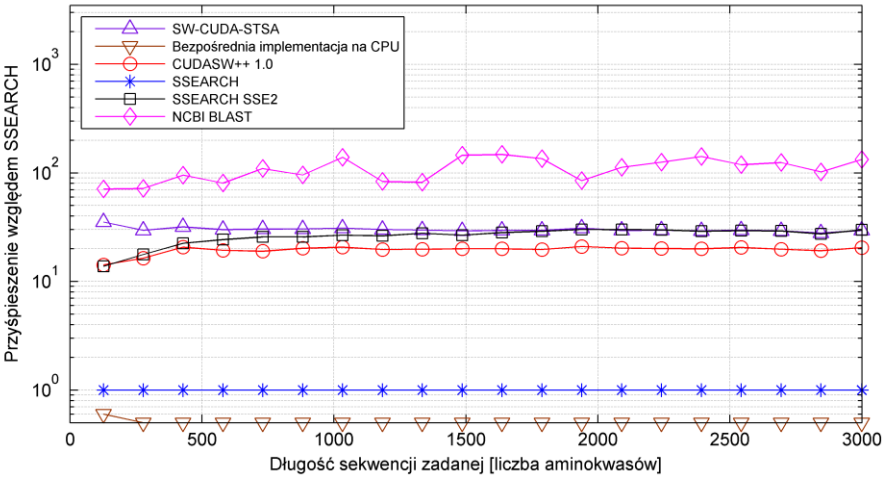
Fig. 5. Execution times for SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence

By usunąć czynnik różnego rozmiaru problemu, rozwiązywanego na potrzeby pomiarów, na rys. 6 wyniki pomiarów wyrażono w MCUPS (ang. *Million Cell Updates Per Second*). Implementacja algorytmu Smitha-Watermana z programu SSEARCH zdecydowanie stanowi uniwersalny punkt odniesienia do obliczenia przyśpieszenia pozostałych algorytmów. Algorytm ten jest efektem wieloletniej optymalizacji kodu programów z pakietu FASTA i z pewnością uwzględnia większość metod przyśpieszenia implementacji algorytmu Smitha-Watermana dla architektury SISD. Przyśpieszenie poszczególnych algorytmów, obliczone względem algorytmu z programu SSEARCH, zobrazowano na rys. 7.



Rys. 6. Wydajność algorytmu SW-CUDA-STSA i algorytmów odniesienia dla różnych długości sekwencji zadanej

Fig. 6. Efficiency of SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence



Rys. 7. Przyspieszenie algorytmu SW-CUDA-STSA i algorytmów odniesienia dla różnych długości sekwencji zadanej. Przyspieszenie liczone jest względem programu SSEARCH

Fig. 7. Acceleration of SW-CUDA-STSA algorithm and referential algorithms for various lengths of the query sequence. The acceleration is relative to SSEARCH program

Wyniki pomiarów zestawione na rys. 7 pokazują, że opisana w niniejszym artykule implementacja algorytmu Smitha-Watermana (SW-CUDA-STSA), dla wykorzystywanej konfiguracji sprzętowej, jest prawie trzydzieści razy szybsza od implementacji z podstawowej wersji programu SSEARCH i sześćdziesiąt razy szybsza względem zaproponowanej implementacji bezpośredniej. Tylko nieznacznie mniej wydajnym okazał się algorytm z programu SSEARCH w wersji korzystającej z technologii SSE2. W dodatku, dla krótszych sekwencji wejściowych zaproponowany algorytm jest znacznie wydajniejszy od szybszej wersji programu SSEARCH. Oba algorytmy – SW-CUDA-STSA i SSEARCH SSE2 – są o około 20% szybsze od pierwszej wersji najszybszego, opublikowanego do tej pory, algorytmu Smitha-Watermana na architekturę SIMT – CUDASW++.

Tak dobry rezultat opisanej metody wynika z niezależności pracujących wątków, z zupełnego wyeliminowania instrukcji synchronizacji oraz z obranej dla tego algorytmu struktury danych, która to z kolei jest konsekwencją próby pogodzenia specyfiki architektury kart graficznych z CUDA ze strategią programowania dynamicznego.

7. Podsumowanie

Algorytmy dopasowania sekwencji stanowią ważne narzędzie w rękach bioinformatyka. Z pewnością warto podejmować próby przyśpieszania wykonywania tych algorytmów, gdyż zakończone sukcesem przynoszą wymierne korzyści. Zarówno dla aplikacji komercyjnych, jak i darmowych, wolno stojących lub serwerów, szybsza realizacja algorytmu oznacza szybszą obsługę klienta-użytkownika. Wpływa to na końcową ocenę jakości produktu i sprzyja dalszemu korzystaniu z aplikacji przez użytkownika.

W artykule zaproponowano ulepszoną metodę implementacji algorytmu Smitha-Watermana na urządzeniach wspierających technologię CUDA. Warto zwrócić uwagę na fakt, że teoretyczna złożoność obliczeniowa tak zrealizowanego algorytmu Smitha-Watermana pozostaje taka sama, poprawia się natomiast jego implementacja, prowadząc do zmniejszenia praktycznej złożoności obliczeniowej. Opisana metoda, chociaż nie jest całkowicie nowatorska, ponieważ łączy już znane rozwiązania, to odznacza się kilkoma cechami, które przekładają się na zwiększoną uniwersalność implementacji i dodatkowe skrócenie czasu realizacji algorytmu Smitha-Watermana. Projektując algorytm starano się uwzględnić najlepsze cechy znanych rozwiązań, minimalizując koszty wstępnego przetwarzania danych i unikając operacji, które mogłyby stanowić dodatkowy narzut.

Opisany w [6] algorytm CUDASW++ jest właściwie połączeniem dwóch algorytmów. W CUDASW++ sekwencja zadana jest dopasowywana do sekwencji z bazy danych jedną z dwóch odmiennych implementacji algorytmu Smitha-Watermana, w zależności od długości

sekwencji zadanej. Przy czym obie implementacje zawarte w CUDASW++ wykazują znaczną różnicę wydajności. W konsekwencji wydajność realizacji algorytmu CUDASW++ silniej niż w zaproponowanej metodzie SW-CUDA-STSA zależy od długości sekwencji zadanej. Algorytm CUDASW++ przejawia wyraźny spadek wydajności, zwłaszcza dla krótkich sekwencji (rys. 6), podczas gdy proponowana metoda SW-CUDA-STSA zachowuje stałość wydajności w szerokim zakresie długości sekwencji wejściowej.

Opracowane rozwiązanie SW-CUDA-STSA teoretycznie działa dla sekwencji dowolnej długości. Ograniczenie stanowi jedynie ilość pamięci globalnej, zainstalowanej na urządzeniu. Natomiast dla konkurencyjnego algorytmu CUDASW++ 1.0 ograniczeniem jest rozmiar pamięci stałej, w której przechowywana jest sekwencja zadana i który wynosi zaledwie 64 kB w urządzeniach z CUDA 1.1.

Algorytm został zaprojektowany dla urządzeń wspierających CUDA od wersji 1.1. Korzyści wynikające z szybszej realizacji algorytmu Smitha-Watermana można czerpać nawet dysponując jednym z obszernej grupy starszych układów kart graficznych.

Za wadę algorytmu SW-CUDA-STSA można uznać niewykorzystywanie pamięci współdzielonej i duże zapotrzebowanie wątków na rejestry. Ostatnia cecha bezpośrednio przekłada się na liczbę współbieżnie działających bloków wątków. Zależnie od szczegółów implementacji i wersji wykorzystywanego CUDA SDK, jeden wątek wymaga aż ponad 40 rejestrów. Mimo że niezbędną liczbę rejestrów dodatkowo ograniczono przez zastosowanie techniki rozwijania pętli (ang. *loop unrolling*), duże zapotrzebowanie wynika głównie z potrzeby przechowywania w rejestrach wartości 4 elementów, położonych wzdłuż górnego brzegu obliczanego obszaru macierzy dopasowania. Elementy te można byłoby przechowywać w pamięci współdzielonej, lecz takie rozwiązanie okazuje się wolniejsze, nawet po zwalczeniu konfliktu banków.

W urządzeniach z technologią CUDA od wersji 1.2 całkowita liczba rejestrów została co najmniej podwojona. Fakt ten potencjalnie pozwala uzyskać jeszcze wyższą wydajność algorytmu, ponieważ rozmiar obliczanego obszaru można zwiększyć do 4×8 lub nawet do 4×12 elementów, przy niezmienionej liczbie dostępów do pamięci globalnej, liczonych na jeden obszar.

BIBLIOGRAFIA

1. Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J.: Basic Local Alignment Search Tool. Journal of Molecular Biology, Vol. 215, 1990, s. 403÷410.
2. Boyer M., Skadron K., Weimer W.: Automated Dynamic Analysis of CUDA Programs. University of Virginia, USA, 2008. http://www.nvidia.com/docs/IO/67190/stmcs08.pdf.

3. Farrar M.: Striped Smith–Waterman speeds database searches six times over other SIMD implementations. Bioinformatics, Vol. 23, No. 2, 2007, s. 156÷161.
4. GenomeNet. http://www.genome.jp/en/db_growth.html.
5. Gough E. S., Kane M. D.: Evaluating Parallel Computing Systems in Bioinformatics. Proceedings of the Third International Conference on Information Technology: New Generations, Las Vegas, NV 2006, s. 233÷238.
6. Liu Y., Maskell D., Schmidt B.: CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Research Notes, Vol. 2:73, 2009, s. 1÷10.
7. Liu Y., Maskell D., Schmidt B.: CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC Research Notes, Vol. 3:93, 2010, s. 1÷12.
8. Manavski S. A., Valle G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, Vol. 9, 2008, s. 1÷9.
9. Mount D. W.: Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, 2001.
10. NVIDIA CUDA programming guide 2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.
11. NVIDIA. http://www.nvidia.pl/object/cuda_home_new_pl.html.
12. Pearson W. R., Lipman D. J.: Improved tools for biological sequence analysis. Proceedings of the National Academy of Sciences, Vol. 85, 1988, s. 2444÷2448.
13. Smith T. F., Waterman M. S.: Identification of common molecular subsequences. Journal of Molecular Biology, Vol. 147, 1981, s. 195÷197.
14. Striemer G. M., Akoglu A.: Sequence Alignment with GPU: Performance and Design Challenges. IPDPS, IEEE International Symposium on Parallel & Distributed Processing, 2009, s. 1÷10.

Recenzenci: Prof. dr hab. inż. Zbigniew Huzar
Dr Ewa Romuk

Wpłynęło do Redakcji 31 stycznia 2011 r.

Abstract

The quadratic computational complexity is the essential reason why the Smith-Waterman algorithm is not commonly used for searching databases of biological sequences. However, the algorithm is characterized by very much desired feature, which is high sensitivity. It is able to point out more distant relations between two sequences than popular heuristic methods. Fortunately, new graphics processing units supporting a novel technology called CUDA are a hope for a wider use of the Smith-Waterman algorithm.

Presented approach to the Smith-Waterman algorithm implementation on a graphics card with CUDA was benchmarked on a single-GPU NVIDIA GeForce 8800 GT with CPU Intel Core 2 Quad Q9300 2.5 GHz. The results were compared with alternative publicly available solutions: CUDASW++ 1.0, SSEARCH (SSE2) included in FASTA 3.5.4.11 and NCBI BLAST 2.2.23. With hardware used for tests our approach provides about 20 percent acceleration over CUDASW++ 1.0 and a slightly better performance than SSEARCH SSE2. Still, the NCBI BLAST is more than two times faster, but not so accurate.

The described method is probably slower than CUDASW++ 2.0. Even though, it has a couple of distinctive qualities. In contrast to CUDASW++, the length of input sequence is limited only by the size of the global memory installed on the GPU. The performance of our approach is very stable within a wide range of the input sequence length and it depends mainly on the size of the available registers pool. Therefore, the performance should grow with newer devices, which have a greater number of registers, but a small modification of the implementation would be required.

Adresy

Robert PAWŁOWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, rob.paw@o2.pl.
Dariusz MROZEK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, dariusz.mrozek@polsl.pl.