STUDIA INFORMATICA 2013
Volume 34 Number 4 (114)

Jacek WIDUCH, Rafał KRAWCZYK
Politechnika Śląska, Instytut Informatyki

ROZWIĄZANIE PROBLEMU TRÓJSPEŁNIALNOŚCI FORMUŁ LOGICZNYCH Z UŻYCIEM ARCHITEKTURY CUDA

Streszczenie. Architektura CUDA firmy NVIDIA jest architekturą wielordzeniowych procesorów graficznych, w której jest stosowany model przetwarzania wielowątkowego. Procesor graficzny w architekturze CUDA może być traktowany jako procesor SIMD z pamięcią wspólną. W niniejszym artykule przedstawiono zastosowanie CUDA do rozwiązania problemu 3–SAT. Przedstawione zostały 3 wersje algorytmu oraz wyniki przeprowadzonych badań eksperymentalnych.

Słowa kluczowe: GPGPU, architektura CUDA, wielowątkowość, model SIMD, problem 3–SAT

SOLVING THE 3-SATISFIABILITY PROBLEM USING CUDA ARCHITECTURE

Summary. The NVIDIA's CUDA architecture is multi–core GPU architecture with the multithreaded processing model. The GPU of CUDA architecture can be treated as a SIMD processor with shared memory. This work presents solving the 3–SAT problem using CUDA architecture. Three versions of algorithm are proposed. Apart from that the results of experimental tests are presented.

Keywords: GPGPU, CUDA architecture, multithreading, SIMD model, 3–SAT problem

1. Wstęp

CUDA (ang. *Compute Unified Device Architecture*) jest uniwersalną architekturą wielordzeniowych procesorów graficznych opracowaną przez firmę NVIDIA [19]. Jest to technologia typu GPGPU (ang. *General-Purpose Computing on Graphics Processing Units*), umożliwiająca użycie mocy obliczeniowej GPU (ang. *Grapics Processing Unit*) do

wykonywania obliczeń numerycznych znacznie wydajniej niż z użyciem CPU (ang. *Central Processing Unit*). Wraz z kartami graficznymi dostępne jest API (ang. *Application Programming Interface*), dostarczające interfejs programistyczny programowania wielowątkowego; najnowszą wersją jest API 5.0 [22, 23, 24].

GPU jest złożony z wielu multiprocesorów strumieniowych (SM, ang. *Streaming Multiprocessor*), z których każdy zawiera wiele procesorów strumieniowych (SP, ang. *Streaming Processor*), zwanych potocznie rdzeniami, przetwarzających wątki programu. Wątki mogą wymieniać dane między sobą, korzystając z pamięci współdzielonej lub globalnej pamięci karty graficznej. Komunikacja z użyciem globalnej pamięci karty jest jednak mniej efektywna ze względu na opóźnienie w dostępie do pamięci, które wynosi 400–600 cykli zegara (w przypadku pamięci współdzielonej wynosi ono 8 cykli).

Pojedynczy SP wykonuje jeden wątek programu, a w każdym cyklu wszystkie SP multiprocesora wykonują jednocześnie identyczne instrukcje poszczególnych wątków, stąd SM, zgodnie z taksonomią Flynna, może być traktowany jako procesor SIMD z pamięcią współdzieloną [6]. Wątki, które są przetwarzane w tym samym SM, są grupowane w bloki o geometrii 1D, 2D lub 3D. W pojedynczym cyklu SM przetwarza 32 wątki, zwane wiązką (ang. *warp*). Bloki wątków są z kolei grupowane w siatkę (ang. *grid*) o geometrii 1D, 2D lub 3D.

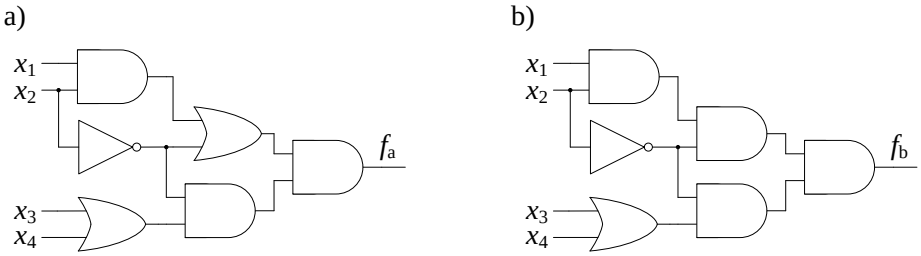
W niniejszym artykule przedstawiono zastosowanie GPU oraz technologii CUDA do rozwiązywania problemu trójspełnialności formuł logicznych. Problem ten jest poddawany badaniom, istnieją sekwencyjne algorytmy jego rozwiązywania [10, 15, 16] oparte na zmodyfikowanym algorytmie DPLL [7, 8] bazującym na algorytmie z powrotami. Problem ten jest także rozwiązywany z użyciem metod heurystycznych [1, 2, 4, 11, 21]. Podejmowane są również próby użycia technologii CUDA do rozwiązywania niniejszego problemu [9, 17].

Struktura niniejszej pracy przedstawia się następująco. W rozdziale 2 został omówiony problem 3–SAT. W rozdziale 3 przedstawiono trzy wersje algorytmów rozwiązywania problemu 3–SAT z użyciem architektury CUDA. Wyniki przeprowadzonych badań eksperymentalnych omówiono w rozdziale 4. Rozdział 5 jest podsumowaniem pracy.

2. Problem 3-SAT

Problem spełnialności funkcji (formuł) logicznych (SAT, ang. *Satisfiability Problem*) jest problemem teorii układów logicznych. Układ logiczny jest zbudowany z połączonych ze sobą bramek logicznych realizujących operacje algebry Boole'a: sumę logiczną, iloczyn logiczny i negację [3, 5, 14]. Zawiera on pewną liczbę wejść i wyjść, realizuje dokładnie określoną funkcję logiczną. Wartości na wejściu i wyjściu układu oraz na wejściu i wyjściu

poszczególnych bramek należą do zbioru {0, 1} i są one oznaczane odpowiednio jako logiczny fałsz i logiczna prawda. Przez wartościowanie układu logicznego rozumie się zbiór wejściowych wartości logicznych. Układ logiczny jest spełniany, jeżeli ma wartościowanie spełniające (wartościowanie, dla którego na wyjściu jest wartość 1). Układ logiczny na rys. 1a jest spełnialny np. dla wartościowania: $\{x_1 = 1; x_2 = 0; x_3 = 0; x_4 = 1\}$, a dla układu z rys. 1b nie istnieje wartościowanie, dla którego wyjście $f_b = 1$, dlatego układ ten nie jest spełnialny.



Rys. 1. Przykładowe układy logiczne
Fig. 1. A sample of logic circuits

Układ logiczny o n wejściach $x_1, \ldots, x_n$ jest opisany za pomocą funkcji (formuły) logicznej $f(x_1, \ldots, x_n)$ realizowanej przez ten układ. Wejścia układu odpowiadają zmiennym funkcji logicznej, w której zmienne mogą występować w postaci prostej lub zanegowanej. Funkcja logiczna zawiera także nawiasy określające kolejność wykonywanych operacji oraz operatory określające wykonywaną operację:

- $\cup$ – operator sumy logicznej (alternatywy),
- $\cap$ – operator iloczynu logicznego (koniunkcji),
- $\sim$ – operator negacji.

Powyższe operatory są jednocześnie funkcjami logicznymi unarnymi (negacja) lub binarnymi (alternatywa i koniunkcja) i jednej wartości wyjściowej, stanowią one zestaw operacji funkcjonalnie pełny. Wartościowaniem spełniającym funkcji logicznej jest zbiór wartości zmiennych, dla których funkcja przyjmuje wartość 1. Jeżeli istnieje wartościowanie spełniające, to funkcja jest nazywana funkcją spełnialną. Układy logiczne przedstawione na rys. 1 realizują odpowiednio funkcje logiczne przedstawione w postaci wzorów (1) i (2).

$$f_a = f_a(x_1,...,x_4) = ((x_1 \cap x_2) \cup \sim x_2) \cap ((x_3 \cup x_4) \cap \sim x_2) \quad (1)$$

$$f_b = f_b(x_1,...,x_4) = ((x_1 \cap x_2) \cap \sim x_2) \cap ((x_3 \cup x_4) \cap \sim x_2) \quad (2)$$

Funkcja logiczna jest w CNF (CNF, ang. *Conjunctive Normal Form*), jeżeli jest koniunkcją alternatyw zmiennych, gdzie alternatywy są nazywane makstermami. Funkcja jest w postaci 3–CNF, jeżeli każda maksterma jest alternatywą dokładnie trzech zmiennych. Funkcję $f(x_1, \ldots, x_n)$ będącą w 3–CNF można opisać zależnością (3), gdzie k jest liczbą maksterm, a c_{ij} ($i = 1, \ldots, k$; $j = 1, \ldots, 3$) jest j–tą zmienną i–tej makstermy i jest jedną ze

zmiennych x_1, …, x_n w postaci prostej lub zanegowanej. Problem trójspełnialności funkcji logicznych (3–SAT) polega na sprawdzeniu, czy funkcja będąca w 3–CNF jest spełnialna.

$$f(x_1,...,x_n)=(c_{11}\cup c_{12}\cup c_{13})\cap(c_{21}\cup c_{22}\cup c_{23})\cap...\cap(c_{k1}\cup c_{k2}\cup c_{k3}) \tag{3}$$

W celu obliczenia wartości funkcji $f(x_1, …, x_n)$ nie ma konieczności wyznaczania wartości wszystkich maksterm. Zgodnie z własnością koniunkcji wartościowanie funkcji może być przerwane w przypadku wykrycia makstermy mającej wartość logicznego fałszu. W takim przypadku, niezależnie od wartości pozostałych maksterm, funkcja przyjmuje wartość logicznego fałszu.

Wartościowanie funkcji $f(x_1, …, x_n)$ można przedstawić w postaci ciągu n bitów, gdzie wartość i–tego bitu jest równa wartości zmiennej x_i. Przykładowo dla funkcji $f(x_1, …, x_5)$ wartościowanie $\{x_1 = 1; x_2 = 1; x_3 = 0; x_4 = 0; x_5 = 1\}$ można zapisać jako ciąg bitów: 10011, stąd dla funkcji o n zmiennych liczba wszystkich wartościowań jest równa liczbie wszystkich n bitowych liczb, która jest równa $N_e = 2^n$. Rozwiązując problem spełnialności algorytmem siłowym, w pesymistycznym przypadku należy sprawdzić 2^n wartościowań. Jak wykazano w pracach [3, 5, 6, 18], problem spełnialności funkcji i układów logicznych należy do problemów NP–zupełnych.

3. Algorytm rozwiązywania problemu 3-SAT dla architektury CUDA

3.1. Wersja podstawowa algorytmu

Badana funkcja $f(x_1, …, x_n)$ (3) jest reprezentowana za pomocą dwuwymiarowej tablicy o wymiarach $k \times 3$, gdzie k jest równe liczbie maksterm. W tablicy i–ty wiersz zawiera indeksy zmiennych tworzących i–tą makstermę, przy czym indeks dodatni oznacza wystąpienie zmiennej w postaci prostej, a indeks ujemny jego wystąpienie w postaci zanegowanej. Indeksy są uporządkowane rosnąco według wartości bezwzględnej indeksu. Niech przykładowo będzie dana funkcja opisana zależnością (4). Jej reprezentację w postaci tablicowej przedstawia zależność (5).

$$f(x_1,...,x_8)=(x_1\cup\sim x_6\cup x_7)\cap(x_4\cup x_5\cup x_7)\cap(\sim x_2\cup x_3\cup\sim x_8) \tag{4}$$

$$\begin{bmatrix} 1 & -6 & 7 \\ 4 & 5 & 7 \\ -2 & 3 & -8 \end{bmatrix} \tag{5}$$

W zaproponowanej implementacji algorytmu zastosowano grupowanie wątków w bloki i siatkę o geometrii 1D. Jak wykazano w rozdziale 2, liczba wszystkich wartościowań dla

funkcji o n zmiennych jest równa $N_e = 2^n$. Wartościowanie jest reprezentowane w postaci n bitowej liczby całkowitej, w której i–ty bit odpowiada wartości zmiennej x_i. W algorytmie następuje sprawdzenie kolejnych wartościowań przez poszczególne wątki. Liczba sprawdzanych wartościowań przez każdy wątek jest równa N_{pt} i jest określona zależnością (6), gdzie *block.x* i *grid.x* są równe odpowiednio rozmiarowi bloku i rozmiarowi siatki. W sytuacji, kiedy N_e nie jest podzielne przez $grid.x \cdot block.x$, liczba wartościowań sprawdzanych przez ostatni wątek wynosi N_{pt}' (7).

$$N_{pt} = \left\lceil \frac{N_e}{grid.x \cdot block.x} \right\rceil \tag{6}$$

$$N_{pt}' = N_e - \left\lceil \frac{N_e}{grid.x \cdot block.x} \right\rceil \cdot (grid.x \cdot block.x - 1) \tag{7}$$

Sprawdzanie wartościowań może zostać zoptymalizowane przez pominięcie części wartościowań, dla których bez konieczności ich sprawdzania można stwierdzić, że funkcja nie jest spełnialna [12]. Na podstawie informacji o niespełnialności jednej z klauzul można określić zakres wartościowań, dla których funkcja nie jest spełniana; zakres ten może zostać pominięty bez konieczności sprawdzania go. Niech dana będzie funkcja $f(x_1, \ldots, x_8)$ (4) i aktualnie sprawdzane wartościowanie, które jest równe $161_{(10)}$, co odpowiada wartości $10100001_{(2)}$ i wartościom zmiennych: $x_1 = x_6 = x_8 = 1$, $x_2 = x_3 = x_4 = x_5 = x_7 = 0$. Sprawdzanie spełnialności funkcji zostanie przerwane po zwartościowaniu makstermy $(x_4 \vee x_5 \vee x_7)$, w której wszystkie zmienne mają wartość fałszu, a co za tym idzie – maksterma jest niespełnialna i funkcja dla wartościowania $161_{(10)}$ jest niespełnialna. Tak więc dla każdego wartościowania, w którym $x_4 = x_5 = x_7 = 0$, funkcja jest niespełniana i wartościowanie to może zostać pominięte bez konieczności sprawdzania go.

Pomijanie sprawdzania części wartościowań jest realizowane w sposób uproszczony. W pierwszym kroku w niespełnianej makstermie jest wyznaczana zmienna x_i będąca zmienną o minimalnym indeksie. Dla makstermy $(x_4 \cup x_5 \cup x_7)$ funkcji (4) jest nim zmienna x_4. Następnie wszystkie zmienne x_j, gdzie $1 \leq j < i$, otrzymują wartość logicznej prawdy, co odpowiada nadaniu wartości 1 bitom o numerach 1, …, i–1 w liczbie reprezentującej wartościowanie, a następnie do uzyskanego wyniku jest dodawana wartość 1. Przykładowo dla wartościowania $161_{(10)} = 10100001_{(2)}$ operacja ta jest opisana zależnością (8), co oznacza, że kolejnym sprawdzanym wartościowaniem będzie $168_{(10)}$. Pozostałe wartościowania z zakresu $162_{(10)}, \ldots, 167_{(10)}$ zostaną pominięte i nie będą sprawdzane.

$$10100001_{(2)} \,|\, 111_{(2)} + 1_{(2)} = 10100111_{(2)} + 1_{(2)} = 10101000_{(2)} = 168_{(10)} \tag{8}$$

```
Wejście: n, k – liczba zmiennych i maksterm badanej funkcji
         f – badana funkcja
         grid, blok – rozmiar siatki i bloku wątków
Wyjście: res – informacja o tym, czy funkcja jest spełnialna
         res_val – wartościowanie, dla którego funkcja jest spełnialna
 1: void SAT_CUDA_1(int n, int k, int** f, dim3 grid, dim3 block, bool* res,
                    unsigned long long* res_val)
 2: {
 3:   unsigned long long n_eval = (unsigned long long)pow((double)2, n);
 4:   unsigned long long n_per_thr = ceil((double)n_eval / (block.x * grid.x));
 5:   // przydzielenie pamięci w obszarze pamięci karty graficznej
      // i przekopiowanie do niej badanej funkcji:
 6:   int *gpu_f;  size_t pitch;
 7:   cudaMallocPitch((void**)&gpu_f, &pitch, sizeof(int*) * 3, (size_t)k);
 8:   for (int i = 0; i < k; i++)
 9:     cudaMemcpy(((int*)((char*)gpu_f + i * pitch)), f[i],
                   sizeof(int) * 3, cudaMemcpyHostToDevice);
10:   // przydzielenie w obszarze pamięci karty pamięci, do której zostanie
11:   // zapisany wynik obliczeń:
12:   bool* gpu_res;  unsigned long long* gpu_res_val;
13:   cudaMalloc((void**)&gpu_res_val, sizeof(unsigned long long));
14:   cudaMalloc((void**)&gpu_res, sizeof(bool));
15:   cudaMemcpy(gpu_res, false, sizeof(bool), cudaMemcpyHostToDevice);
16:   // wywołanie kodu jądra i wykonanie obliczeń z użyciem GPU:
17:   SAT3_1<<<grid, block>>>(gpu_f, pitch, n_per_thr, n_eval, k,
                              gpu_res, gpu_res_val);
18:   // przekopiowanie wyników z pamięci karty:
19:   cudaMemcpy(&res, gpu_res, sizeof(bool), cudaMemcpyDeviceToHost);
20:   if (*res)
21:     cudaMemcpy(res_val, gpu_res_val, sizeof(unsigned long long),
                  cudaMemcpyDeviceToHost);
22:   // zwolnienie przydzielonej pamięci w obszarze pamięci karty graficznej:
23:   cudaFree(gpu_res); cudaFree(gpu_res_val); cudaFree(gpu_f);
24: }
```

Implementacja algorytmu została przestawiona w postaci funkcji *SAT_CUDA_1*. W części inicjalizacyjnej jest wyznaczana liczba wszystkich wartościowań badanej funkcji *f* (wiersz 3) oraz liczba wartościowań sprawdzanych przez każdy wątek (wiersz 4). Następnie w obszarze pamięci karty graficznej jest przydzielana pamięć (wiersz 7), do której jest kopiowana funkcja *f* (wiersze 8–9), oraz pamięć, w której zostanie zapisana informacja o spełnialności funkcji *f* (wiersz 14) i wartościowanie, dla którego funkcja jest spełnialna (wiersz 13)[1]. Ponieważ funkcja jest reprezentowana w postaci tablicy dwuwymiarowej, więc do przydziału pamięci użyta została funkcja *cudaMallocPitch*, gwarantująca rozmieszczenie w pamięci tablicy w sposób minimalizujący czas dostępu do poszczególnych wierszy. W tym celu każdy wiersz może zostać dopełniony dodatkowymi bajtami, a rzeczywisty rozmiar wiersza jest zapisywany w zmiennej *pitch*.

Najważniejszym fragmentem funkcji *SAT_CUDA_1* jest wiersz 17, gdzie następuje wywołanie kodu jądra *SAT3_1* (ang. *kernel*) wykonywanego przez GPU [13, 20, 22, 23], w którym następuje sprawdzenie, czy funkcja *f* jest spełnialna. Następnie z pamięci karty graficznej jest kopiowana informacja o tym, czy funkcja *f* jest spełnialna (wiersz 19). Jeżeli

[1] Jeżeli funkcja nie jest spełniana, to w obszarze pamięci przydzielonym dla wartościowania nie jest zapisywana żadna wartość.

funkcja jest spełniana, to dodatkowo jest kopiowane wartościowanie, dla którego jest ona spełnialna (wiersz 21). W ostatnim kroku jest zwalniana przydzielona pamięć w obszarze pamięci karty graficznej (wiersz 23).

```
Wejście: f – badana funkcja
         pitch – liczba bajtów w pamięci zajmowanych przez makstermę
         n_per_thr – liczba wartościowań sprawdzanych przez pojedynczy wątek
         n_eval – liczba wszystkich wartościowań funkcji
         k – liczba maksterm z których składa się funkcja f
Wyjście: res – informacja o tym, czy funkcja jest spełnialna
         res_val – wartościowanie, dla którego funkcja jest spełnialna
 1: __global__ static void SAT3_1(int* f, size_t pitch,
                    unsigned long long n_per_thr, unsigned long long n_eval,
                    int k, bool* res, unsigned long long* res_val)
 2: {
 3:   // identyfikator wątku i maksymalna wartość sprawdzana przez wątek:
 4:   int id = blockDim.x * blockIdx.x + threadIdx.x;
 5:   unsigned long long max_val = (id + 1) * n_per_thr;
 6:   if (max_val > n_eval)
 7:     max_val = n_eval;
 8:   // licznik sterujący sprawdzaniem czy zostało znalezione rozwiązanie:
 9:   int check = 0;
10:   int max_check = n_per_thr / 100;
11:   // sprawdzanie wartościowań przez dany wątek:
12:   for (unsigned long long val = id * n_per_thr; val < max_val; val++)
13:   {
14:     // sprawdzenie, czy zostało już znalezione rozwiązanie:
15:     if (++check >= max_check)
16:     {
17:       if ((*res) == true)
18:         return;
19:       check = 0;
20:     }
21:     // sprawdzenie kolejnych maksterm funkcji dla bieżącego wartościowania:
22:     bool satisfied = true;
23:     for (int i = 0; i < k; i++)
24:     {
25:       int* clause = ((int*)((char*)f + i * pitch));
26:       // maksterm nie jest prawdziwa, czyli funkcja nie jest spełnialna:
27:       if (!GetLitVal_1(clause[0], val) && !GetLitVal_1(clause[1], val) &&
              !GetLitVal_1(clause[2], val))
28:       {
29:         satisfied = false;
30:         int lit_abs = abs(clause[0]);
31:         if (lit_abs > 1)
32:           val |= 2 ^ (lit_abs - 1) - 1;
33:         break;
34:       }
35:     }
36:     // znaleziono rozwiązanie (funkcja jest spełnialna):
37:     if (satisfied)
38:     {
39:       (*res) = true;
40:       (*res_val) = val;
41:       return;
42:     }
43:   }
44: }
```

Kod jądra jest reprezentowany przez funkcję *SAT3_1*. W części inicjalizacyjnej są wyznaczane: identyfikator wątku (wiersz 4), maksymalna wartość sprawdzana przez dany wątek (wiersze 5–7) oraz wartość *max_check* (wiersz 10). Rozwiązanie jest wyznaczane

równolegle przez wątki, dlatego każdy wątek sprawdza, czy inny wątek wyznaczył już rozwiązanie. Odbywa się to przez sprawdzenie wartości zmiennej *res* znajdującej się w pamięci globalnej karty. Aby zminimalizować liczbę odwołań do pamięci karty, zmienna *res* jest sprawdzana po każdorazowym sprawdzeniu *max_check* wartościowań.

Sprawdzanie wartościowań przez wątek odbywa się w iteracji *for* (wiersze 12–43). W pierwszym kroku iteracji jest sprawdzane, czy zostało już znalezione rozwiązanie (wiersze 15–20). Następnie są obliczane wartości kolejnych maksterm funkcji *f* dla wartościowania *val* (iteracja *for* w wierszach 23–35). W tym celu z pamięci globalnej karty jest pobierany opis obliczanej klauzuli (wiersz 25), a następnie z użyciem funkcji *GetLitVal_1* są obliczane wartości wszystkich jej zmiennych dla wartościowania *val* (wiersz 27). Jeżeli maksterma ma wartość logicznego fałszu, to następuje pominięcie części wartościowań (wiersze 30–32) oraz przerwanie analizy bieżącego wartościowania *val* (wiersz 33). W przypadku wykrycia spełnialności funkcji *f* dla wartościowania *val* informacja ta jest zapisywana w pamięci globalnej karty pod adresem *res*, a pod adresem *res_val* jest zapisywane wartościowanie i następuje zakończenie wykonywania wątku (wiersze 39–41).

Obliczanie wartości zmiennych dla wartościowania *val* odbywa się z użyciem funkcji *GetLitVal_1*, która zwraca wartość *true*, jeżeli zmienna ma wartość logicznej prawdy dla wartościowania *val*, i wartość *false* w przeciwnym przypadku.

```
Wejście: id_lit – indeks zmiennej
         val – sprawdzane wartościowanie funkcji
Wyjście: wartość logiczna zmiennej dla wartościowania val
1: __device__ static bool GetLitVal_1(int id_lit, unsigned long long val)
2: {
3:   // zmiennna w postaci prostej:
4:   if (id_lit > 0)
5:     return (short)((val >> id_lit) & 1) == 1;
6:   else    // zmienna zanegowana:
7:     return (short)((val >> -id_lit) & 1) == 0;
8: }
```

3.2. Algorytm z dekodowaniem wartości zmiennych

W algorytmie przedstawionym w podrozdziale 3.1 dla każdej sprawdzanej makstermy (wiersz 27 funkcji *SAT3_1*) w funkcji *GetLitVal_1* jest pobierana wartość zmiennej. Dana zmienna może występować w wielu makstermach funkcji *f*, stąd dla tej samej zmiennej operacja ta może być wykonywana wielokrotnie. W funkcji *SAT3_1* wprowadzono modyfikację, zgodnie z którą przed sprawdzaniem wartościowania *val* (wiersze 22–35) dekodowane są wartości zmiennych z wartościowania *val* i zapamiętywane są w dodatkowej tablicy *literals*. Dekodowanie jest realizowane za pomocą następującego ciągu instrukcji:

```
1: bool literals[MAX_N];   // MAX_N – maksymalna liczba zmiennych
2: for (int i = 0; i < k; i++)
```

```
3: {
4:   literals[i] = (short)(val & 1) == 1;
5:   val = val >> 1;
6: }
```

Podczas wartościowania makstermy (wiersz 27) nie ma więc konieczności pobierania wartości zmiennej z wartościowania *val*, tylko jest ona odczytywana z tablicy *literals*. W związku z tym modyfikacji ulega funkcja *GetLitVal_1*, a jej zmodyfikowana wersja jest przedstawiona w postaci funkcji *GetLitVal_2*.

```
Wejście: id_lit – indeks zmiennej
         literals – wartości zmiennych w sprawdzanym wartościowaniu funkcji
Wyjście: wartość logiczna zmiennej dla wartościowania opisanego przez literals
1: __device__ static bool GetLitVal_2(int id_lit, bool* literals)
2: {
3:   // zmienna w postaci prostej:
4:   if (id_lit > 0)
5:     return literals[id_lit - 1];
6:   else      // zmienna zanegowana:
7:     return !literals[-id_lit - 1];
8: }
```

Wymienione modyfikacje są jedynymi, które zostały wprowadzone w funkcji *SAT3_1*, stąd pełna treść zmodyfikowanej funkcji zostanie pominięta.

3.3. Algorytm uwzględniający optymalizacje sprzętowe

Kolejną wprowadzoną modyfikacją była optymalizacja na poziomie sprzętowym. Pierwszą modyfikacją jest przeniesienie badanej funkcji do pamięci współdzielonej. Wstępnym krokiem do wprowadzenia tej modyfikacji jest zamiana dwuwymiarowej tablicy reprezentującej badaną funkcję *f* na tablicę jednowymiarową. Każda maksterma jest złożona z trzech zmiennych będących wartościami typu *int*. Pojedyncza wartość *int* zajmuje 4B, co oznacza, że maksterma zajmuje 12B. Tak więc makstermy byłyby rozmieszczone co 12B w pamięci. Zgodnie z zaleceniami firmy NVIDIA większa przepustowość w dostępie do pamięci (ang. *bandwith*) jest uzyskiwania w przypadku, kiedy wartości są oddalone co wielokrotność 8B, zatem w tym przypadku co 16B [22][2]. Stąd zamiast przechowywania kolejnych trzech wartości *int* w pamięci są przechowywane cztery wartości *int* (czwarta wartość nie jest używana). Modyfikacja ta powoduje konieczność przydziału w obszarze pamięci karty graficznej większego obszaru pamięci dla funkcji *f*. W tym celu modyfikowane są wiersze 6–9 funkcji *SAT_CUDA_1*, które przyjmują postać:

```
1: int* gpu_f;
2: cudaMalloc((void**)&gpu_f, 4 * k * sizeof(int));
3: cudaMemcpy(gpu_f, f, 4 * k * sizeof(int), cudaMemcpyHostToDevice);
```

[2] Własność ta została potwierdzona przez wstępnie przeprowadzone badania eksperymentalne.

Kopiowanie funkcji do pamięci współdzielonej jest wykonywane przez poszczególne wątki w ramach danego bloku. W celu zapewnienia możliwie najszybszego transferu zastosowano łączony odczyt z pamięci globalnej, a warunkiem, jaki musi być spełniony w takim przypadku, jest kopiowanie również czwartej, nieużywanej wartości *int*. Kopiowanie badanej funkcji do pamięci współdzielonej odbywa się przed rozpoczęciem sprawdzania wartościowań przez wątek (przed wierszem 11 funkcji *SAT3_1*) i jest realizowane przez następujące instrukcje:

```
 1: __shared__ int shm_f[4 * MAX_K];   // MAX_K – maksymalna liczba maksterm
 2: short size_f = ceil((float)(4 * k) / (float)blockDim.x);
 3: int id_lit = threadIdx.x;
 4: for (int i = 0; i < size_f; i++)
 5: {
 6:   if (id_lit < 4 * MAX_K)
 7:     shm_f[id_lit]= f[id_lit];
 8:   id_lit += blockDim.x;
 9: }
10: __syncthreads();
```

W wierszu 1 w pamięci współdzielonej jest deklarowana tablica *shm_f*, do której zostanie przekopiowana funkcja *f*, po czym jest wyznaczana liczba kopiowanych danych przez każdy wątek (wiersz 2). Kopiowanie odbywa się współbieżnie przez poszczególne wątki (wiersze 4–9), stąd w ostatnim kroku jest dokonywana synchronizacja wątków (wiersz 10).

4. Wyniki badań eksperymentalnych

W badaniach eksperymentalnych użyto funkcji logicznych różniących się liczbą zmiennych oraz maksterm. Została także zaimplementowana sekwencyjna wersja algorytmu przedstawionego w podrozdziale 3.2, a jej celem było zbadanie przyspieszenia obliczeń z użyciem GPU i CPU. Badania przeprowadzono na komputerze o następujących parametrach:

- CPU: AMD FX(tm)–6200 Six–Core 3.80 GHz,
- 16 GB RAM,
- GPU: GeForce GTX 650 (384 rdzenie, 1 GB pamięci, zgodność z CC 3.0)[3],
- system operacyjny: Windows 7 Enterprise 64 bitowy.

W przedstawionych wynikach zostały użyte następujące oznaczenia:

- „CPU” – wykonanie obliczeń z użyciem CPU,
- „GPU algorytm 1” – wykonanie obliczeń z użyciem GPU i podstawowej wersji algorytmu przedstawionej w podrozdziale 3.1,

[3] Wszystkie parametry karty graficznej są dostępne w [25].

- „GPU algorytm 2” – wykonanie obliczeń z użyciem GPU i algorytmu z dekodowaniem wartości zmiennych przedstawionego w podrozdziale 3.2,
- „GPU algorytm 3” – wykonanie obliczeń z użyciem GPU i algorytmu uwzględniającego optymalizacje sprzętowe, który przedstawiono w podrozdziale 3.3.

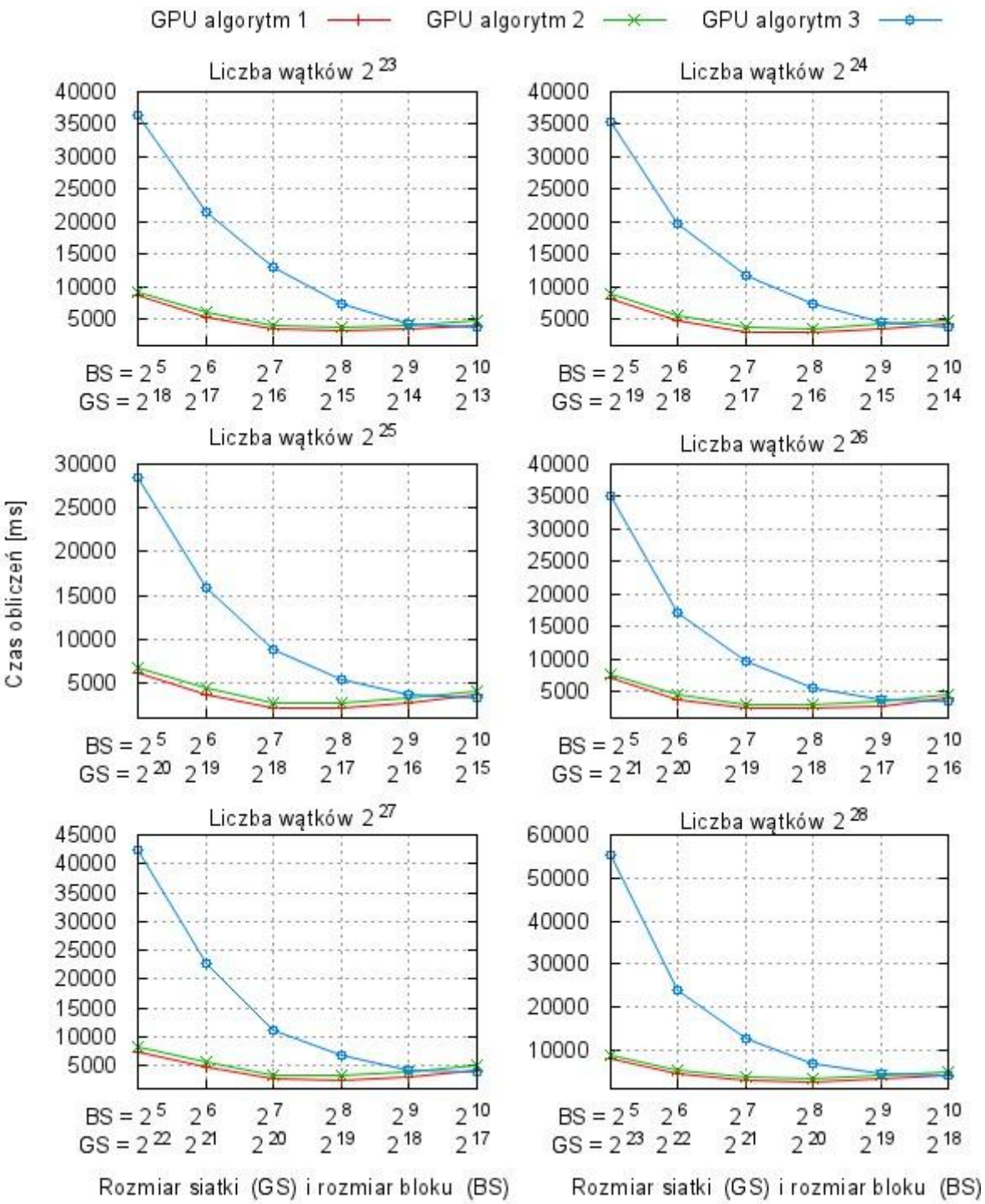
Badania eksperymentalne można podzielić na dwie grupy. Celem pierwszej grupy badań (przedstawionych w podrozdziałach 4.1, 4.2 i 4.3) było określenie zależności czasu obliczeń od organizacji wątków w bloki i siatkę. Celem drugiej grupy badań (przedstawionych w podrozdziałach 4.4 i 4.5) było z kolei określenie zależności czasu obliczeń od liczby zmiennych i maksterm funkcji logicznej. Dodatkowym celem było porównanie czasów obliczeń z użyciem GPU i CPU.

4.1. Zależność czasu wyznaczania rozwiązania od rozmiaru bloku i siatki wątków przy stałej liczbie wątków

Celem niniejszych badań było określenie wpływu rozmiaru bloku wątków oraz rozmiaru siatki na czas wyznaczania rozwiązania przy stałej liczbie wątków. W pojedynczym eksperymencie przyjęto stałą liczbę wątków (przeprowadzono badania dla 6 przypadków, w których liczba wątków wynosiła odpowiednio od 2^{23} do 2^{28}) i zbadano 6 różnych bloków wątków o rozmiarach z zakresu BS = 2^5, …, 2^{10} (ang. *block size*). Ponieważ w ramach pojedynczego eksperymentu liczba wątków była stała, więc zwiększenie rozmiaru bloku BS powodowało zmniejszenie rozmiaru siatki bloków GS (ang. *grid size*).

W badaniach użyto funkcji zawierającej n = 30 zmiennych i k = 500 maksterm. Wyniki przeprowadzonych badań zamieszono na wykresach (rys. 2). W każdym eksperymencie największy czas obliczeń został uzyskany dla bloku o rozmiarze BS = 2^5. Rozmiar ten jest równy rozmiarowi wiązki przetwarzanej w jednym cyklu przez multiprocesor. Ponadto wartość ta jest tylko nieznaczną wielokrotnością liczby procesorów strumieniowych przypadających na multiprocesor, co nieznacznie kompensuje opóźnienie w dostępie do pamięci. Najmniejszy czas obliczeń został natomiast uzyskany dla liczby wątków równej 2^{25}.

Warto zwrócić uwagę na czas obliczeń dla BS = 2^5 z użyciem algorytmu 3. W tym przypadku siatka zawiera największą liczbę bloków. W algorytmie tym korzysta się z danych umieszczonych w pamięci współdzielonej, które są tworzone dla każdego bloku wątków. Użycie takiej liczby bloków powoduje brak możliwości umieszczenia wszystkich danych w pamięci współdzielonej, gdyż ich sumaryczny rozmiar przewyższa dostępną pamięć współdzieloną i z tego powodu są one umieszczane w pamięci globalnej. Fakt ten spowalnia obliczenia ze względu na opóźnienie w dostępie do pamięci globalnej i powoduje, że czas obliczeń jest od 4 (dla liczby wątków 2^{23}) do 7 (dla liczby wątków 2^{28}) razy większy w porównaniu z pozostałymi dwoma algorytmami.



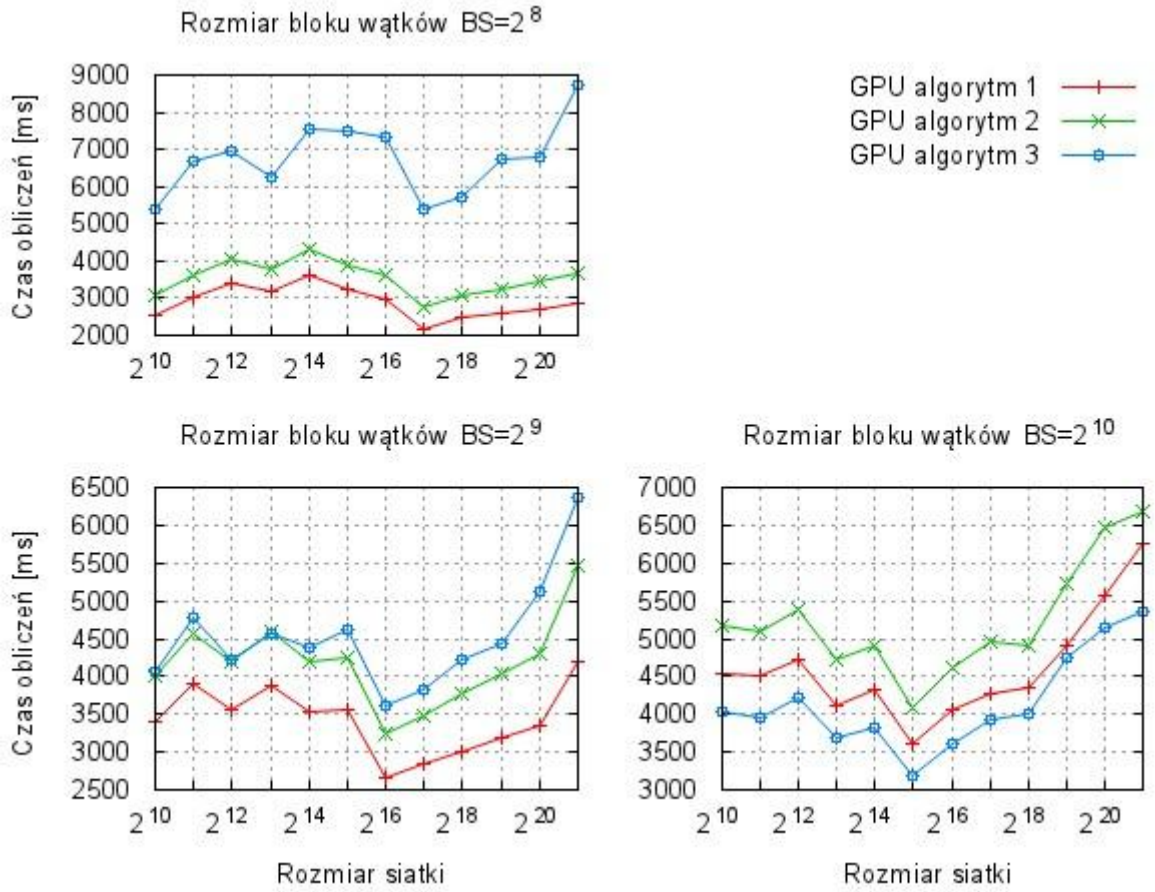
Rys. 2. Zależność czasu wyznaczania rozwiązania od rozmiaru siatki i rozmiaru bloku wątków
Fig. 2. A computation time depending on the grid size and the block of threads size

Algorytm 3 jest jedynym algorytmem, dla którego czas obliczeń maleje wraz ze wzrostem rozmiaru bloku. Wraz ze wzrostem rozmiaru bloku maleje także różnica między czasem obliczeń uzyskanym dla algorytmu 3 i czasem obliczeń uzyskanym dla pozostałych dwóch algorytmów. Dla wartości BS = 2^{10} minimalny czas obliczeń uzyskano właśnie dla algorytmu 3. Dla pozostałych algorytmów minimalny czas obliczeń uzyskano dla wartości BS = 2^8, a dla

wartości BS = 2^8 i 2^{10} można zaobserwować nieznaczny jego wzrost. Warto także zwrócić uwagę na to, że dekodowanie wartości literałów (algorytm 2) nie zmniejsza czasu obliczeń, gdyż czas ten jest większy od czasu obliczeń z użyciem algorytmu 1.

4.2. Zależność czasu wyznaczania rozwiązania od rozmiaru siatki bloków wątków przy stałym rozmiarze bloku wątków

W drugiej serii eksperymentów zbadano zależność czasu obliczeń od rozmiaru siatki przy stałym rozmiarze bloku wątków. Użyto bloków o rozmiarach BS = 2^8, 2^9 i 2^{10}, a dla każdego rozmiaru bloku zbadano siatki o rozmiarach z zakresu GS = 2^{10}, …, 2^{21}. Podobnie jak w przypadku eksperymentów opisanych w podrozdziale 4.1, została użyta funkcja logiczna zawierająca n = 30 zmiennych i k = 500 maksterm.



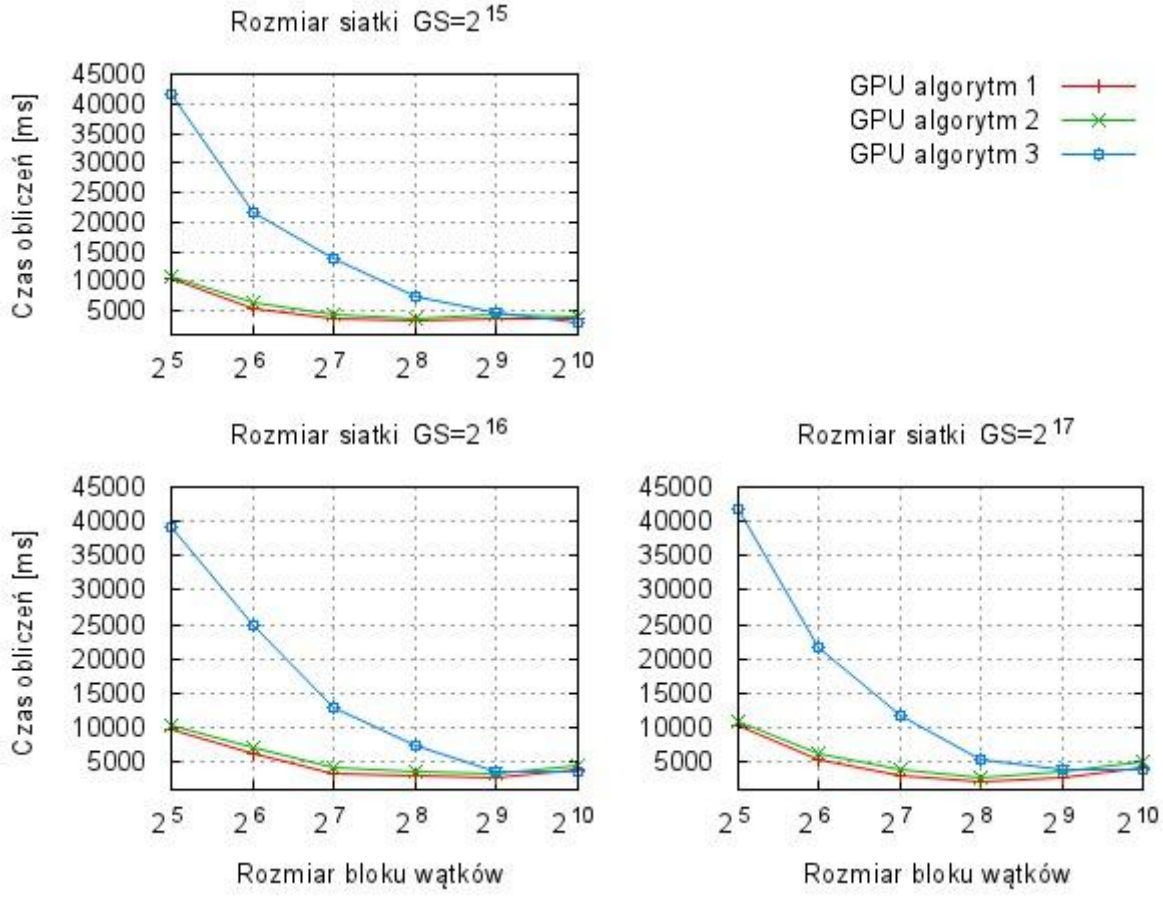
Rys. 3. Zależność czasu wyznaczania rozwiązania od rozmiaru siatki
Fig. 3. A computation time depending on the grid size

Wyniki przeprowadzonych badań przedstawiono na rys. 3; są one zgodne z wynikami przedstawionymi w podrozdziale 4.1. Najmniejszy czas obliczeń został także uzyskany dla liczby wątków równej 2^{25}. Wraz ze wzrostem rozmiaru bloku maleje także różnica między czasem obliczeń dla algorytmu 3 i pozostałych dwóch algorytmów. Dla bloku o rozmiarze BS = 2^8 czas ten jest ok. 2 razy większy dla algorytmu 3 niż w przypadku pozostałych

algorytmów, a dla bloku o rozmiarze BS = 2^{10} jest on najmniejszy spośród wszystkich trzech algorytmów. Podobnie jak w poprzednim eksperymencie, dekodowanie wartości literałów (algorytm 2) pogarsza czas obliczeń w porównaniu z algorytmem podstawowym 1.

4.3. Zależność czasu wyznaczania rozwiązania od rozmiaru bloku wątków przy stałym rozmiarze siatki bloków

W niniejszym eksperymencie zbadano wpływ rozmiaru bloków wątków na czas obliczeń dla stałego rozmiaru siatki. Badania przeprowadzono dla trzech rodzajów siatek o rozmiarach równych odpowiednio GS = 2^{15}, 2^{16} i 2^{17}, dla których zbadano bloki wątków o rozmiarach z zakresu BS = 2^5, …, 2^{10}. W badaniach użyto takiej samej funkcji jak w poprzednich dwóch przypadkach.



Rys. 4. Zależność czasu wyznaczania rozwiązania od rozmiaru bloku wątków
Fig. 4. A computation time depending on the block of threads size

Wyniki badań zostały przedstawione na rys. 4 i są one podobne do wyników przedstawionych w podrozdziałach 4.1, 4.2. Najmniejszy czas obliczeń został uzyskany dla przypadku, w którym łączna liczba wątków jest równa 2^{25}. Dla bloku wątków o rozmiarze BS = 2^5 można zaobserwować znaczną różnicę między czasem obliczeń z użyciem algorytmu

3 i pozostałych dwóch algorytmów. Różnica ta maleje wraz ze wzrostem rozmiaru bloku wątków, a dla bloku o rozmiarze BS = 2^{10} czas ten jest najmniejszy dla algorytmu 3. Wraz ze wzrostem rozmiaru bloku maleje także czas obliczeń dla algorytmu 3.

4.4. Zależność czasu wyznaczania rozwiązania od liczby maksterm

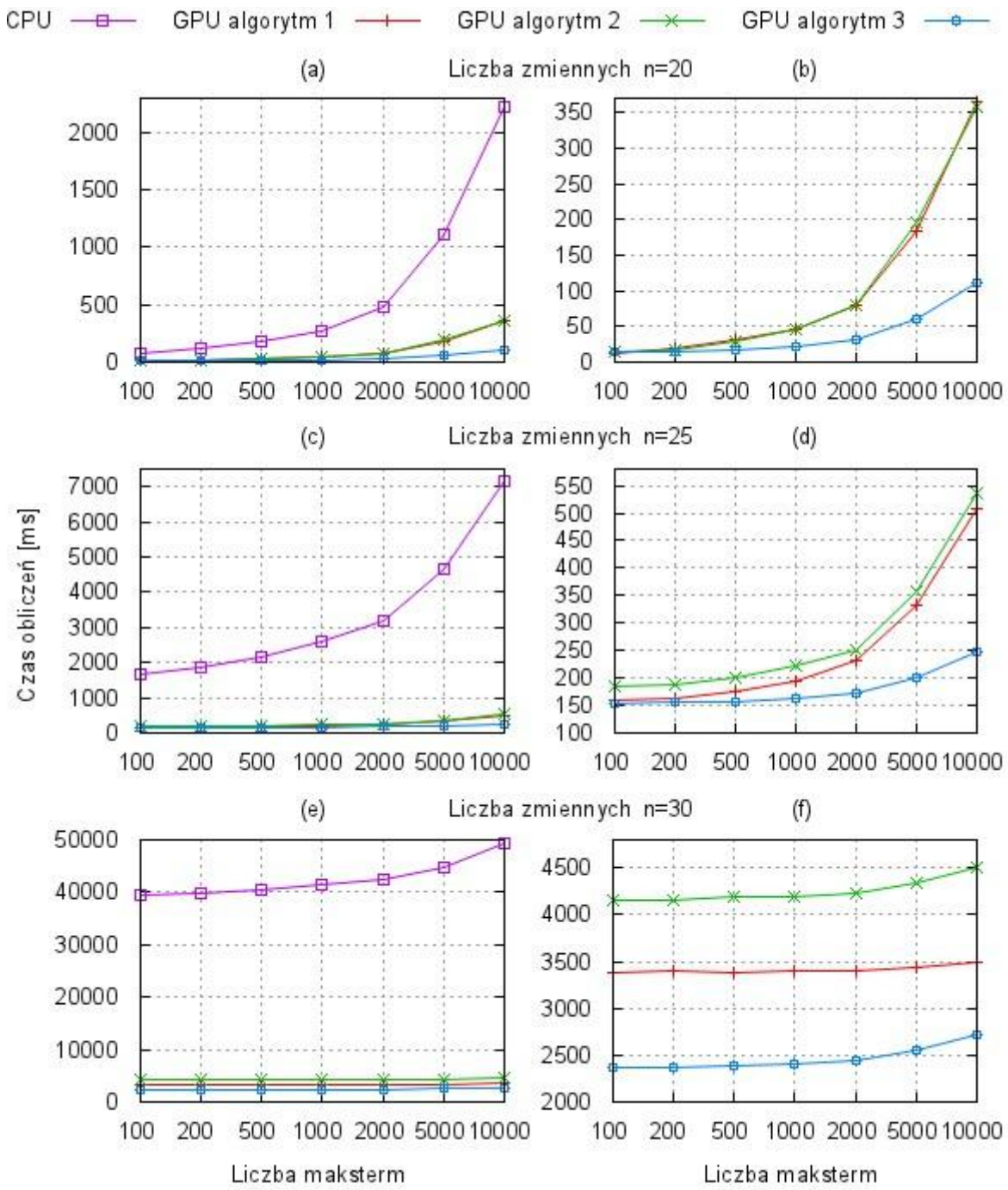
Celem pierwszej grupy badań przedstawionych w podrozdziałach 4.1, 4.2 i 4.3 było określenie wpływu organizacji wątków na czas obliczeń. Celem drugiej grupy badań było określenie zależności czasu obliczeń od postaci funkcji, tj. liczby zmiennych i maksterm, oraz porównanie czasów obliczeń z użyciem GPU i CPU.

Na podstawie pierwszej grupy badań przyjęto liczbę wątków równą 2^{25} pogrupowanych w siatkę o rozmiarze GS = 2^{15} i bloki o rozmiarze BS = 2^{10}. Jako pierwsza została określona zależność czasu obliczeń od liczby maksterm. W badaniach użyto funkcji o liczbie zmiennych równych odpowiednio n = 20, 25, 30. Dla każdego przypadku rozpatrzono liczbę maksterm równą: k = 100, 200, 500, 1000, 2000, 5000 i 10000.

Wyniki przeprowadzonych badań przedstawiono w postaci wykresów na rys. 5. Na wykresach (a), (c), (e) przedstawiono czas obliczeń z użyciem CPU i GPU, natomiast na wykresach (b), (d), (f) wyłącznie czasy obliczeń z użyciem GPU. Dla każdego z trzech badanych przypadków czas wyznaczania rozwiązania rośnie wraz z rozmiarem siatki. Największy czas został uzyskany dla obliczeń z użyciem CPU, a najmniejszy z użyciem algorytmu 3 i GPU. Różnica między czasem obliczeń z użyciem CPU i GPU rośnie wraz ze wzrostem liczby maksterm. Dla k = 100 maksterm czas ten jest ok. 5 razy większy dla n = 20 zmiennych i ok. 11 razy większy dla n = 30 zmiennych. Natomiast w przypadku k = 10000 maksterm czas obliczeń z użyciem CPU jest ok. 20 razy większy dla n = 20 zmiennych i ok. 28 razy większy dla n = 25 zmiennych.

4.5. Zależność czasu wyznaczania rozwiązania od liczby zmiennych

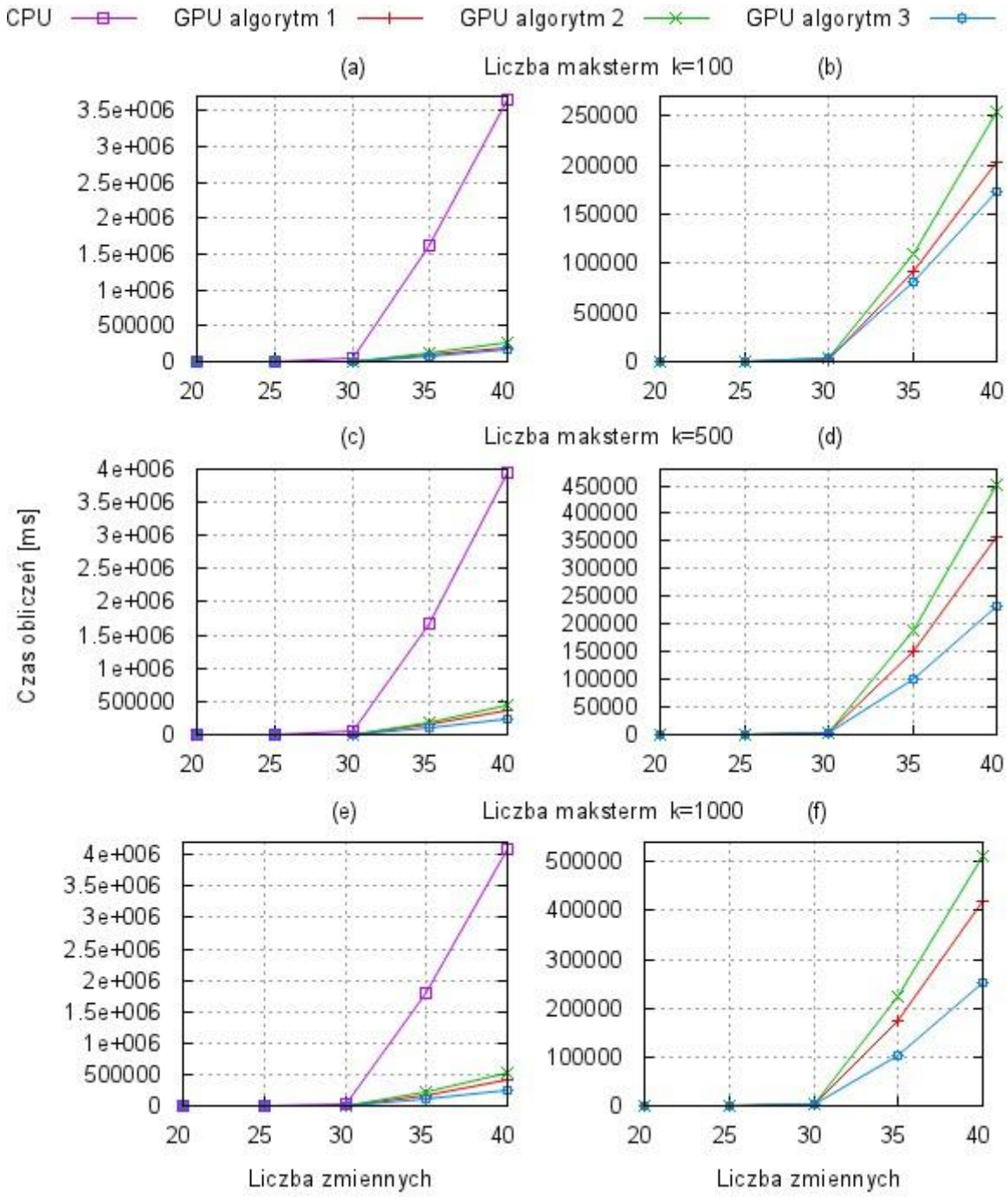
W ostatnim badaniu określono zależność czasu obliczeń od liczby zmiennych. Podobnie jak w przypadku badań opisanych w podrozdziale 4.4, przyjęto 2^{25} wątków pogrupowanych w siatkę o rozmiarze GS = 2^{15} i bloki o rozmiarze BS = 2^{10}. Zbadano funkcje zawierające odpowiednio k = 100, 500 i 1000 maksterm, gdzie dla każdej funkcji przyjęto n = 20, 25, 30, 35 i 40 zmiennych.



Rys. 5. Zależność czasu wyznaczania rozwiązania od liczby maksterm
Fig. 5. A computation time depending on the number of maxterms

Wyniki przeprowadzonych badań przedstawiono w postaci wykresów na rys. 6. Czas obliczeń z użyciem CPU i GPU został pokazany na wykresach (a), (c) i (e), natomiast na wykresach (b), (d) i (f) przedstawiono czas obliczeń wyłącznie z użyciem GPU. Różnicę między czasami obliczeń można zaobserwować, począwszy od liczby zmiennych n = 35, gdyż dla liczby zmiennych n = 20, …, 30 czasy są porównywalne. Spośród badanych algorytmów największy czas obliczeń uzyskano dla CPU. W przypadku n = 40 jest on od 16

(dla k = 1000 maksterm) do 21 razy (dla k = 100 maksterm) większy od czasu obliczeń z użyciem GPU. Przy porównaniu czasów obliczeń dla GPU najmniejszy czas obliczeń został uzyskany dla algorytmu 3.



Rys. 6. Zależność czasu wyznaczania rozwiązania od liczby zmiennych
Fig. 6. A computation time depending on the number of variables

5. Podsumowanie

Problem trójspelnialności formuł logicznych (3–SAT) należy do grupy problemów NP–zupełnych. Dla zadanej funkcji logicznej o n zmiennych będącej w postaci 3–koniunkcyjnej normalnej (3–CNF) należy odpowiedzieć na pytanie, czy istnieje wartościowanie, tj. zbiór wartości zmiennych, dla którego funkcja przyjmuje wartość logicznej prawdy. Liczba wszystkich możliwych wartościowań jest równa 2^n. Jak wykazano w podrozdziale 3.1, część wartościowań może zostać pominięta bez konieczności ich sprawdzania, gdyż funkcja logiczna przyjmuje dla nich wartość logicznego fałszu. Informacja taka jest uzyskiwana na podstawie analizy innych wartościowań.

W artykule przedstawiono zastosowanie architektury CUDA do rozwiązania problemu 3–SAT. Zaproponowane zostały trzy wersje algorytmów rozwiązywania tego problemu: wersja podstawowa, wersja z dekodowaniem wartości zmiennych oraz wersja uwzględniająca optymalizacje sprzętowe. W wymienionych algorytmach jest przeszukiwana przestrzeń rozwiązań, przy czym nie korzysta się w nich z metod heurystycznych. We wszystkich algorytmach dokonuje się przydziału każdemu wątkowi pewnego zbioru wartościowań do sprawdzenia. Część wartościowań, odnośnie których wiadomo, że funkcja przyjmuje dla nich wartość logicznego fałszu, jest pomijanych i nie są sprawdzane, co wpływa na zmniejszenie czasu wyznaczania rozwiązania.

Dla opracowanych algorytmów zostały przeprowadzone dwie grupy badań eksperymentalnych. Celem pierwszej grupy badań było określenie zależności czasu obliczeń od organizacji wątków w bloki i siatkę, natomiast celem drugiej grupy było określenie zależności czasu obliczeń od liczby zmiennych i liczby maksterm funkcji. Dodatkowo zostały przeprowadzone badania z użyciem CPU. W każdym z badanych przypadków najmniejszy czas obliczeń został uzyskany dla liczby wątków równej 2^{25}. Ponadto badania wykazały, że dekodowanie wartości zmiennych pogarsza czas obliczeń w porównaniu z podstawową wersją algorytmu. Wprowadzanie optymalizacji sprzętowych jest opłacalne wyłącznie dla bloków o maksymalnym rozmiarze, zawierających 2^{10} wątków.

BIBLIOGRAFIA

1. Balint A., Fröhlich A.: Improving Stochastic Local Search for SAT with a New Probability Distribution. LNCS, Vol. 6175, Springer, Heidelberg 2010, p. 10-15.
2. Belov A., Järvisalo M., Stachniak Z.: Depth-Driven Circuit-Level Stochastic Local Search for SAT. Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI, 2011), Barcelona, Spain 2011, p. 504-509.

3. Ben–Ari M.: Logika matematyczna w informatyce. WNT, Warszawa 2006.
4. Bhalla A., Lynce I., Sousa J. Marques-Silva J.: Heuristic Backtracking Algorithms for SAT. [In:] International Workshop on Microprocessor Test and Verification, Austin, Texas, USA, IEEE Computer Society, 2003, p. 69–74.
5. Cormen T. H., Leiserson C. E., Rivest R. L.: Wprowadzenie do algorytmów. WNT, Warszawa 2000.
6. Czech Z.: Wprowadzenie do obliczeń równoległych. Wydawnictwo Naukowe PWN, Warszawa 2010.
7. Davis M., Logemann G., Loveland D.: A machine program for theorem proving. Communactions of the ACM, Vol. 5(7), 1962, p. 394–397.
8. Davis M., Putnam H.: A computing procedure for quantification theory. Journal of the ACM, Vol. 7(3), 1960, p. 201–215.
9. Deleau H., Jaillet C., Krajecki M.: GPU4SAT: solving the SAT problem on GPU. In PARA 2008 9th International Workshop on State–of–the–Art in Scientific and Parallel Computing, Trondheim, Norway 2008.
10. Eén N., Sörensson N.: An extensible SAT-solver. In: Giunchiglia E., Tacchella A. (eds.) SAT 2003. LNCS, Vol. 2919, Springer, Heidelberg 2004, p. 502–518.
11. Gallo G., Urbani G.: Algorithms for Testing the Satisfiability of Propositional Formulae. Journal of Logic Programming, Vol. 7, 1989, p. 45-61.
12. Haskins J.: 3SAT Solver, http://www.cs.virginia.edu/~jwh6q/3sat-web/.
13. Kirk D. B., Hwu W. W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 1st edition (February 5, 2010).
14. Łuba T., Rawski M., Tomaszewicz P., Zbierzchowski B.: Synteza układów cyfrowych. WKŁ, Warszawa 2003.
15. Mahajan Y. S., Fu Z., Malik S.: Zchaff2004: An efficient SAT solver. [In:] Hoos H.H., Mitchell D. G. (eds.): SAT 2004, LNCS 3542, Springer, Heidelberg 2005.
16. Marques-Silva J. P., Sakallah K. A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, Vol. 48, No. 5, 1999, p. 506–521.
17. Meyer Q., Schönfeld F., Stamminger M., Wanka R.: 3–SAT on CUDA: Towards a massively parallel SAT solver. Proceedings of High Performance Computing and Simulation (HPCS), 2010, p. 306–313.
18. Papadimitriou Ch.: Złożoność obliczeniowa. Helion, Gliwice 2012.
19. Rauber T., Rünger G.: Parallel Programming for Multicore and Cluster Systems. Springer, 1st edition (March 10, 2010).
20. Sanders J., Kandrot E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 1st edition (July 30, 2010).

21. Zabih R., McAllester D.: A Rearrangement Search Strategy for Determining Propositional Satisfiability. Proceedings of the National Conference on Artificial Intelligence, 1988, p. 155-160.
22. CUDA C Best Practices Guide Version 5.0. NVIDIA Corporation (October, 2012), https://developer.nvidia.com/cuda-downloads.
23. CUDA C Programming Guide Version 5.0. NVIDIA Corporation (October, 2012), https://developer.nvidia.com/cuda-downloads.
24. CUDA Reference Manuals Version 5.0. NVIDIA Corporation (October, 2012), https://developer.nvidia.com/cuda-downloads.
25. GeForce GTX 650 specifications, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650/specifications.

Abstract

The boolean satisfiability problem (SAT) is a well–know decision problem and it belongs to the group of NP–complete problems. A SAT formula is a conjunction of variables disjunctions and it consists of the logical operators AND and OR and NOT on boolean variables. The formula is satisfiable if there exists a boolean assignment of the variables such that the formula is true, otherwise it is unsatisfiable. The number of all assignments is exponential and it equals 2^n, where n equals the number of variables. 3–SAT is a special case of SAT problem where the formula in conjunctive normal form with 3 variables per maxterm (3–CNF) is given.

In the paper three versions of algorithm are presented: a basic algorithm (presented in subsection 3.1), algorithm with decoding of variable values (presented in subsection 3.2) and algorithm with a hardware optimizations (presented in subsection 3.3). The algorithms were implemented using CUDA API 5.0 and tested using NVIDIA GeForce GTX 650. Two groups of tests were carried out. The aim of the first group of tests was to investigate an impact of the threads organization in blocks and grid on the computation time, while the goal of the second group of tests was to determine an impact the number of variables and the number of maxterms of the function on the computation time. The results of tests are presented in section 4.

Adres

Jacek WIDUCH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, jacek.widuch@polsl.pl.
Rafał KRAWCZYK: rafalkrawczyk88@gmail.com.