Włodzimierz BIELECKI, **Marek PAŁKOWSKI**
KATEDRA INŻYNIERII OPROGRAMOWANIA, WYDZIAŁ INFORMATYKI, ZACHODNIOPOMORSKI UNIWESYTET TECHNOLOGICZNY
ul. Żołnierska 49, 71-210 Szczecin

Wyznaczanie równoległości pętli programowych w aplikacjach dedykowanych dla procesorów graficznych

Prof. dr hab. inż. Włodzimierz BIELECKI

Jest kierownikiem Katedry Inżynierii Oprogramowania na Zachodniopomorskim Uniwersytecie Technologicznym. W badaniach koncertuje się na przetwarzaniu równoległym i rozproszonym, kompilatorów optymalizujących, ekstrakcji grubo- i drobnoziarnistej równoległości zawartej w pętlach programowych.

e-mail: wbielecki@wi.zut.edu.pl

Dr inż. Marek PAŁKOWSKI

Stopień doktora uzyskał na Zachodniopomorskim Uniwersytecie Technologicznym z zakresu automatycznego zrównoleglania pętli programowych w oparciu o podział na fragmenty kodu. W badaniach koncentruje się na przetwarzaniu równoległym i jego zastosowaniach w środowiskach sprzętowo-programowym.

e-mail: mpalkowski@wi.zut.edu.pl

Streszczenie

Ekstrakcja równoległości w postaci niezależnych fragmentów kodu pozwala wygenerować równoległe pętle programowe w sposób automatyczny. Kod taki umożliwia wykorzystanie mocy obliczeniowej maszyn równoległych, w tym wieloprocesorowych kart graficznych. W niniejszym artykule poddano analizie zastosowanie algorytmów wyznaczania fragmentów kodu dla aplikacji dedykowanych dla procesorów graficznych. Zbadano przyspieszenie i efektywność obliczeń oraz skalowalność wygenerowanego kodu równoległego.

Słowa kluczowe: automatyczne zrównoleglanie pętli, fragmenty kodu, GPU, CUDA, OpenCL, obliczenia wysokiej wydajności.

Parallelizing program loops for graphics processing in general purpose computing

Abstract

Extracting synchronization-free slices allows automatically generating parallel loops. The code can be executed on multi-processors machines in a reduced period of time. Slicing techniques enable also generating parallel code for graphics processing in general purpose computing. Nowadays, graphic cards support executing multi-threaded applications. GPU systems consist of tens or hundreds of processors. CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. Graphics processing units (GPUs) are accessible to software developers through variants of industry standard programming languages. Using CUDA, the latest NVIDIA GPUs become accessible for computation like CPUs. The model for GPU computing is to use a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. From the user's perspective, the application just runs faster because it uses the high-performance of the GPU to boost performance. In this paper slicing algorithms are examined for generating a parallel code for graphic cards are examined. A short example of the code is presented. CUDA statements and technique are explained. Memory cost and transfer data is considered. Speed-up, efficiency and scalability of the code are analyzed.

Keywords: loop parallelization, GPU, CUDA, OpenCL, slices.

1. Wstęp

Karty graficzne wyposażane są obecnie w kilkadziesiąt lub nawet kilkaset jednostek przetwarzających. Umożliwiają wykonanie kodu równoległego złożonego z niezależnych wątków obliczeń. Dzięki standardowi OpenCL [1] możliwe jest przeprowadzenie dowolnych obliczeń równoległych (nie tylko powiązanych z grafiką) przez procesory kart graficznych (ang. Graphics Processing to General Purpose Parallel Computing) [2].

W celu wygenerowania programu równoległego tożsamego z sekwencyjnym odpowiednikiem niezbędne są techniki przetwarzania kodu, a zwłaszcza zrównoleglanie pętli programowych, w których zawarta jest zdecydowana większość obliczeń aplikacji. Istnieje wiele metod zrównoleglania pętli, w tym transformacje afiniczne [3-5]. Znane są jednak słabości tych transformacji (np. duża złożoność obliczeniowa, brak możliwości znajdowania całkowitej równoległości w pętlac h) przekładające się na ograniczenia ich do zastosowania dla szerokiego spektrum pętli [6].

W artykule opisano technikę zrównoleglania pętli w oparciu o podział jej przestrzeni iteracji na niezależne fragmenty kodu w sposób automatyczny, które przełamują słabości innych technik przekształceń pętli [6-7] oraz zastosowanie tej techniki do tworzenie kodu rnoległego do wykonywania w kartach graficznych NVIDIA.

2. Znajdowanie fragmentów kodu

Zrównoleglenie pętli programowej polega na podziale jej zbioru iteracji na części i rozdystrybuowanie ich do poszczególnych wątków aplikacji, które są następnie przydzielane do różnych jednostek obliczeniowych systemu. Pomiędzy iteracjami pętli jednakże mogą zachodzić **zależności danych**, które nie pozwalają na dowolny podział obliczeń pętli.

Opisane algorytmy stanową rozwinięcie badań nad zrównolegleniem pętli programowych za pomocą znajdowania niezależnych fragmentów kodu (ang. slices). W pracach Pugha i Rossera [7] po raz pierwszy zaproponowano partycjonowanie przestrzeni iteracji pętli (ang. Iteration Space Slicing Framework). Pugh i Rosser nie zaproponowali jednak algorytmu automatycznego wyznaczenia niezależnych fragmentów kodu.

W zrównoleglaniu pętli programowych za pomocą fragmentów kodu wykorzystuje się dokładną analizę zależności opracowaną przez Pugh i Wonnacott'a [8], w której zależności reprezentowane są przez relacje między krotkami wejściowymi i wyjściowymi oraz ograniczeń zawierających formuły Presburgera:

$$\{[krotki\ wejściowe] \rightarrow [krotki\ wyjściowe] : ograniczenia\};$$

Za pomocą zbiorów natomiast można określić przestrzeń i podprzestrzenie iteracji pętli programowej.

Na zbiorach i relacjach przeprowadzić można operacje arytmetyki Presburgera, takie jak : koniunkcja ($\cap$), unia ($\cup$), różnica (-), dziedzina relacji *domain(R)*, zakres relacji *range(R),* aplikacja relacji $R(S)$, tranzytywne domknięcie R^*. W tym celu wykorzystywana jest biblioteka programistyczna *Omega Calculator* [9]. Operacje arytmetyki Presburgera zostały szczegółowo opisane w pracy [10].

Niech dany będzie graf zależności D opisany przez relacje zależności. **Fragment kodu** (ang. *slice*) jest luźno spójnie składową (ang. *weakly connected component*) grafu D. Innymi słowy fragment kodu jest maksymalnym podgrafem grafu D, w którym istnieje nieskierowana ścieżka pomiędzy każdą parą wierzchołków. Fragment kodu jest wolny od synchronizacji (ang. *synchronization-free slice*), jeśli nie istnieją zależności pomiędzy

jego operacjami i operacjami innych fragmentów kodu.

Wyznaczenie fragmentów kodu składa się z znalezienia początków reprezentatywnych fragmentów kodu i pozostałych, zależnych instrukcji fragmentów od ich początków [11].

Początek reprezentatywny fragmentu kodu (ang. *source of slice*) jest to leksykograficznie najmniejszy początek krańcowy zawierający się w tym fragmencie, tj. leksykograficznie minimalna operacja spośród wszystkich operacji należących do fragmentu kodu. Szczegóły obliczeń przedstawiono w [11].

Do podziału przestrzeni iteracji na fragmenty kodu stosuje się operację tranzytywnego domknięcia relacji zależności [7]. Po obliczeniu początków fragmentów kodu i wyznaczeniu zbioru tranzytywnie zależnych instrukcji należy wygenerować kod równoległy. W tym celu opracowany został autorski generator kodu [12], korzystający z technik skanowania instrukcji ze zbiorów – funkcji *codegen* [10]. Możliwe jest jednak wykorzystanie innych rozwiązań skanujących zbiory [13-16].

3. Wykonywanie niezależnych fragmentów kodu za pomocą procesorów graficznych

Ponieważ fragmenty kodu są niezależne, pętla przebierająca ich początki może zostać zrównoleglona bez obawy naruszenia poprawności semantycznej kodu. W ciele tej pętli wykonywane są sekwencyjnie pozostałe instancje instrukcji fragmentów.

Niech dana będzie pętla programowa:

```
for(i=1; i<=N; i++)
  for(j=1; j<=N; j++)
     a[i][j] = a[i][j-1];
```

Relacja zależności uzyskana za pomocą narzędzia Petit [8]:

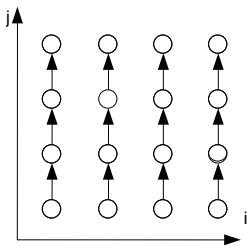
```
R1 := {[i,j]->[i,j+1] : 1 <= i <= n && 1 <= j < n}
```

Początki obliczone za pomocą narzędzia Omega Calculator[10]:

```
{[i,1]: 1 <= i <= n && 2 <= n}
```

Kod przebierający *n* niezależnych fragmentów kodu wygenerowanych automatycznie za pomocą [11] i funkcji *codegen* [10] ma postać:

```
if (n >= 2)
  par for(t1 = 1; t1 <= n; t1++) {
    a[t1][1] = a[t1][0];
    if (n >= t1 && t1 >= 1)
      for(t2 = 2; t2 <= n; t2++)
        a[t1][t2] = a[t1][t2-1];
  }
```



Rys. 1. Przestrzeń iteracji i zależności pętli
Fig. 1. Iteration space and dependences of a loop

W oparciu o ten kod i standard OpenCL [1] można zapisać kod równoległy w języku C jak niżej:

```
// Kernel that executes on the CUDA device
  __global__ void slice(float *a, int N, int pacz-
ka)
{
   int idx = blockIdx.x;
   int t1, t2;
   int lb = idx*paczka+1;
   int ub = ((idx+1)*paczka<N) ? (idx+1)*paczka:N;
   if (N >= 2)
   for(t1 = lb; t1 <= ub; t1++) {
     a[t1*DIM_N+1] = a[t1*DIM_N];
      if (N >= t1 && t1 >= 1)
      for(t2 = 2; t2 <= n; t2++)
         a[t1*DIM_N+t2] = a[t1*DIM_N+t2-1];
    }
}
// main program
...  //cudaAlloc and cudaMemCopy FromHostToDevice
slice <<< N_CPU, 1 >>> (a, N, paczka);
...    // cudaMemCopy FromDeviceToHost
```

Obliczenia równoległe wykonywane są za pomocą jąder (ang. *kernels*). Ponadto do funkcji tej przekazano rozmiar paczki fragmentów kodu, natomiast za pomocą zmiennych *lb* i *ub* określono, które niezależne fragmenty kodu są do wykonania poprzez poszczególny wątek. W programie głównym należy wykonać funkcje *cudaAlloc* i *cudaMemCopy* w celu zapewnienia przesyłu danych pomiędzy pamięcią operacyjną i pamięcią karty graficznej.

4. Badania eksperymentalne

Badania przeprowadzono za pomocą karty graficznej NVIDIA 8800 GTS wyposażonej w 96 procesorów 1.6 GHz i pamięć GDDR3 512 MB o przepustowości 64 GB/s. W celu sprawdzenia wydajności obliczeń równoległych opartych o wyznaczenie niezależnych fragmentów kodu przeanalizowano trzy pętle sparametryzowane z zestawu testowego NAS Parallel Benchmark [16]: FT_auxfnct_2, UA_diffuse_5 oraz UA_transfer_4.

Kod równoległy zapisano za pomocą standardu OpenCL [1] w języku C i uruchomiono za pomocą karty graficznej.

W tabeli pierwszej przedstawiono przyspieszenie i efektywność obliczeń pętli. Czas wykonania każdej pętli za pomocą wielu procesorów jest krótszy niż za pomocą jednego. Czasy wykonania(w sekundach), przyspieszenie S=T(P)/T(1) i efektywność E = S/P obliczeń zbadano dla P = 1, 2, 8 i 64 procesorów GPU oraz różnych wartości parametrów.

W aplikacjach z wykorzystaniem GPU istotny jest też czas przesyłu danych z pamięci operacyjnej do pamięci karty graficznej. Na ten czas składają się trzy elementy: czas alokacji pamięci danych na karcie (*alloc*), czas wysłania danych do pamięci karty (*send*), oraz czas odebrania danych z karty (*fetch*). Warto zaznaczyć, że odbierano tylko te dane, które uległy modyfikacji.

W tabeli drugiej przedstawiono wyniki uwzględniające czas operacji pamięciowych. W kolumnach 4, 6, 10 i 14 podano procentowy udział tego czasu (mc – ang/ *memory copying*) od czasu całego przetwarzania. Warto zauważyć, że przy wykorzystaniu 2 procesorów graficznych czas potrzebny na operacje pamięciowe wynosi zaledwie kilka procent, natomiast przy użyciu 64 procesów czas ten rośnie od 34% do nawet 63%. Wiąże się to z tym, że użycie większej liczby procesorów znacznie skraca czas samego przetwarzania pętli. Choć narzut czasu na przesył danych jest stały, to ogranicza on wartość przyspieszenia obliczeń, ponieważ według prawa Amdahl'a jest to składowa problemu niemożliwa do zrównoleglenia.

5. Podsumowanie

Badania w niniejszym artykule ukazują przydatność techniki wyznaczania równoległości za pomocą fragmentów kodu dla tej technologii. Automatyczne zrównoleglanie pętli z honorowaniem wszystkich jej zależności jest kluczowe do wytwarzania kodu obliczeń potrafiącego w pełni wykorzystać moc obliczeniową maszyn równoległych.

Tab. 1. Czas wykonania, przyspieszenie i efektywność obliczeń zrównoleglonych pętli równoległych wykonanych za pomocą różnej liczby procesorów graficznych
Tab. 1. Calculation time, speed-up, efficiency of parallelized loops executed by various number of GPU processors

Pętla	Parametry	CPU									
		1	2			8			64		
		Czas	Czas	S	E	Czas	S	E	Czas	S	E
FT_ax2	N1,N2,N3=100	0,767	0,467	1,642	0,821	0,117	6,556	0,819	0,015	52,897	0,827
	N1,N2,N3=150	2,587	1,612	1,605	0,802	0,395	6,549	0,819	0,046	56,239	0,879
	N1,N2,N3=200	6,300	3,909	1,612	0,806	0,983	6,409	0,801	0,120	52,500	0,820
UA_d5	N1,N2=64, N3=N4=10	0,171	0,088	1,943	0,972	0,022	7,773	0,972	0,003	57,000	0,891
	N1, N2=64, N3=N4=25	1,069	0,553	1,933	0,967	0,138	7,746	0,968	0,018	59,389	0,928
	N1,N2=64, N3=N4=50	4,276	2,150	1,989	0,994	0,544	7,860	0,983	0,074	57,784	0,903
UA_t4	N1,N2=500	0,101	0,052	1,942	0,971	0,013	7,769	0,971	0,002	50,500	0,789
	N1,N2=1000	0,406	0,207	1,961	0,981	0,052	7,808	0,976	0,007	58,000	0,906
	N1,N2=2000	1,621	0,826	1,961	0,981	0,207	7,830	0,979	0,028	57,882	0,904

Tab. 2. Czas wykonania, przyspieszenie i efektywność obliczeń zrównoleglonych pętli równoległych wykonanych za pomocą różnej liczby procesorów graficznych z uwzględnieniem czasu operacji pamięciowych (mc)
Tab. 2. Calculation time, speed-up, efficiency of parallelized loops executed by various number of GPU processors with memory operations

Pętla	Param.	CPU													
		1		2				8				64			
		czas	mc %	czas	mc %	S	E	czas	mc %	S	E	czas	mc %	S	E
FT_aux	N1,N2,N3=100	0,776	**1,1**	0,476	**1,8**	1,630	0,815	0,126	**7,0**	6,163	0,770	0,023	**38,0**	33,155	0,518
	N1,N2,N3=150	2,615	**1,0**	1,640	**1,7**	1,594	0,797	0,423	**6,6**	6,181	0,773	0,074	**37,9**	35,289	0,551
	N1,N2,N3=200	6,364	**1,0**	3,973	**1,6**	1,602	0,801	1,047	**6,1**	6,077	0,760	0,184	**34,8**	34,556	0,540
UA_d	N1,N2=64 N3,N4=10	0,176	**2,9**	0,093	**5,6**	1,890	0,945	0,027	**19,2**	6,468	0,809	0,008	**63,6**	21,366	0,334
	N1,N2=64 N3,N4=25	1,085	**1,5**	0,569	**2,8**	1,907	0,953	0,154	**10,5**	7,038	0,880	0,034	**47,3**	31,740	0,496
	N1,N2=64 N3,N4=50	4,342	**1,5**	2,216	**2,9**	1,959	0,980	0,610	**10,8**	7,114	0,889	0,140	**47,2**	30,934	0,483
UA_t	N1,N2=500	0,104	**2,6**	0,055	**4,8**	1,896	0,948	0,016	**17,0**	6,617	0,827	0,005	**57,1**	22,217	0,347
	N1,N2=1000	0,413	**1,6**	0,214	**3,2**	1,930	0,965	0,059	**11,7**	7,010	0,876	0,014	**49,6**	29,709	0,464
	N1,N2=2000	1,646	**1,5**	0,852	**2,9**	1,933	0,966	0,232	**10,8**	7,088	0,886	0,053	**47,3**	30,931	0,483

Zaletą rozwiązań układów graficznych jest niski koszt zarządzania wątkami i wsparcie dla podziału obliczeń w porównaniu do tradycyjnych rozwiązań wykorzystujący funkcje systemu operacyjnego. Jak pokazały badania narzut dodatkowych obliczeń nie wpływa znacząco na uzyskiwanie dalszego przyspieszenia wraz ze wzrostem liczby procesorów. Dodatkowym kosztem jest czas niezbędny do przesyłu danych do karty.

Zadaniem na przyszłość jest opracowanie narzędzi generujących automatycznie kod zgodny z standardem OpenCL na podstawie otrzymanego pseudokodu określającego niezależne fragmenty obliczeń.

6. Literatura

[1] OpenCL 1.1, The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/, 2010.

[2] NVIDIA CUDA C Programming Guide, v. 3.1.11, NVIDIA Corportation, 2010. http://developer.nvidia.com/object/cuda_3_1_downloads.html

[3] Lim A. W., Lam M., Cheong G.: An affine partitioning algorithm to maximize parallelism and minimize communication. In ICS'99, s. 228-237. ACM Press, 1999.

[4] Feautrier P.: Some efficient solutions to the affine scheduling problem, part i, ii, one dimensional time, International Journal of Parallel Programming 21. (1992), s. 313-348, 389- 420.

[5] Banerjee U.: Unimodular transformations of double loops. Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing. 1990, s. 192-219.

[6] Beletska A., Bielecki W., Cohen A., Palkowski M..: Synchronization-free automatic parallelization: Beyond affine iteration-space slicing. In Languages and Compilers for Parallel Computing (LCPC'09), LNCS. Springer-Verlag, 2009.

[7] Pugh W., Rosser E.: Iteration Space Slicing and Its Application to Communication Optimization. Proceedings of the International Conference on Supercomputing. 1997, s. 221-228.

[8] Pugh W., Wonnacott D.: An exact method for analysis of value-based array data dependences. In In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer-Verlag, 1993.

[9] The Omega project. http://www.cs.umd.edu/projects/omega

[10] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D.: The omega library interface guide. Technical report, USA, 1995.

[11] Beletska A., Bielecki W., Siedlecki K., San Pietro P.: Finding synchronization-free slices of operations in arbitrarily nested loops. In ICCSA (2), volume 5073 of Lecture Notes in Computer Science, pp. 871-886. Springer, 2008.

[12] Strona projektu Iteration Space Slicing Framework http://sfs.zut.edu.pl

[13] Bastoul C.: Code generation in the polyhedral model is easier than you think. In PACT'2004, s. 7-16, Juan-les-Pins, september 2004.

[14] Bielecki W., Beletska A., Palkowski M., San Pietro P.: Extracting synchronization-free trees composed of non-uniform loop operations, Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science Volume 5022/2008, Springer Berlin / Heidelberg, 2008, s. 185-195.

[15] Bielecki W., Pałkowski M.: Using message passing for developing coarse-grained applications in OpenMP, Proceedings of Third International Conference on Software and Data - ICSOFT 2008, Porto, Portugalia 2008, s. 145-153.

[16] NAS Parallel Benchmarks v.3.2, 2010, http://www.nas.nasa.gov/Resources/Software/npb.html

otrzymano / received: 13.05.2011
przyjęto do druku / accepted: 04.07.2011 *artykuł recenzowany*