Marek PALKOWSKI [1], Wlodzimierz BIELECKI [1]

West Pomeranian University of Technology (1)

Programming NVIDIA cards by means of transitive closure based parallelization algorithms

Abstract. *Massively parallel processing is a type of computing that uses many separate CPUs or GPUs running in parallel to execute a single program. Because most computations are contained in program loops, automatic extraction of parallelism available in loops is extremely important for many-core systems. In this paper, we study speed-up and scalability of parallel code scanning synchronization-free slices and time partitions by means of a 960 CUDA Cores machine, Tesla S1070.*

Streszczenie. *Przetwarzanie równoległe na wielką skalę wykonywane jest za pomocą wielu procesorów (również graficznych) wykonujących jednocześnie instrukcje pojedynczego programu. Ponieważ większość obliczeń zlokalizowana jest w pętlach programowych, automatyczne zrównoleglanie kodu jest ważne dla maszyn wielordzeniowych. W artykule zbadano przyspieszenie i skalowalność równoległego kodu złożonego z niezależnych fragmentów lub harmonogramowania swobodnego za pomocą maszyny Tesla S1070 zbudowanej z 960 rdzeni CUDA.*
Metody tworzenia aplikacji równoległych dla wielordzeniowych komputerów

Keywords: parallel program loops, many-core machines, synchronization-free slicing, free-scheduling.
Słowa kluczowe: równoległe pętle programowe, komputery wielordzeniowe, niezależne fragmenty kodu, harmonogramowanie swobodne.

Introduction

Massively-parallel architectures are able to execute thousands of concurrent threads. These systems are scalable and allow software users to boost computing performance simply by adding processors. Demand for high-computing performance in science and industry requires automated tools permitting for exposing parallelism for such systems.

Because most computations are contained in program loops, automatic extraction of parallelism available in loops is extremely important for many-core systems, allowing us to produce parallel code from existing sequential applications and to create multiple threads that can be easily scheduled to achieve high program performance.

Papers [1, 2] present algorithms to extract coarse-grained parallelism represented with synchronization-free slices consisting of loop statement instances. Those algorithms are base on the Iteration Space Slicing Framework (ISSF) that uses the transitive closure of dependence relations to extract parallelism. Paper [5] introduces an approach to extract fine-grained parallelism representing free scheduling for statement instances of affine loops. It also uses the transitive closure of dependence relations to expose parallelism Under the free schedule, loop statement instances are executed as soon as their operands are available.

In the current paper, we focus on the problem of the speed-up, efficiency, and scalability of parallel loops for a many-core machine, NVIDIA Tesla S1070 being produced by means of slicing and free-scheduling. The Cost of synchronization and communication is also considered. Experimental results are presented

Background

In this paper, we deal with affine loop nests where

- for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters (i.e., parameterized loop bounds),
- the loop steps are known positive constants.

A nested loop is called perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called imperfectly nested.

A statement instance *s(I)* is a particular execution of a loop statement *s* for a given iteration *I*.

Two statement instances $s_1(I)$ and $s_2(J)$ are *dependent* if both access the same memory location and if at least one access is a write. $s_1(I)$ and $s_2(J)$ are called the source and destination of a dependence, respectively, provided that $s1(I)$ is lexicographically fewer than $s_2(J)$ (we write this as $(s_1(I) \prec s_2(J))$, i.e., $s_1(I)$ is always executed before $s_2(J)$).

The approach to extract synchronization-free parallelism in program loops by means of the Iteration Space Slicing Framework requires an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a dependence if and only if it actually exists. To describe and implement algorithms, we choose the dependence analysis proposed by Pugh and Wonnacott [3] where dependences are represented by dependence relations whose constraints are described in the Presburger arithmetic (built of affine equalities and inequalities, logical and existential operators); the Omega library is used for computations over such relations [4].

A dependence relation is a tuple relation of the form {[*input list*] $\rightarrow$ [*output list*] : *constraints};* where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing constraints imposed upon *input list* and *output list*.

We use standard operations on relations and sets, such as intersection ($\cap$), union ($\cup$), difference (-), domain of relation (domain(R)), range of relation (range(R)), relation application (given a relation R and set S, $R(S) = \{[e']: \exists e \in S, e \rightarrow e' \in R\}$), positive transitive closure (given a relation R, $R^+ = \{[e] \rightarrow [e']: e \rightarrow e' \in R \,||\, \exists e'' \, s.t. \, e \rightarrow e'' \in R \,\&\, e'' \rightarrow e' \in R^+\}$), transitive closure ($R^* = R^+ \cup I$, where I is the identity relation). These operations are described in detail in [11].

To permit for applying the operations mentioned above on relations and sets, we have to preprocess them by means of the following two steps.

1. Make the sizes of input and output tuples of dependence relations to be the same by inserting the value ”-1” at the rightmost positions of correspondent tuples
2. Insert identifiers of loop statements in the last position of input and output tuples.

Inserting ”-1” does not introduce any false (not existing) dependence after preprocessing because existing programming languages suppose that loop indices cannot be negative. It can be obviously omitted when we deal with perfectly nested loops.

Inserting loop statement identifiers makes clear which statements originate sources and destinations of dependences. This step of the preprocessing procedure can be skipped when the loop body comprises the only statement.
The formal presentation of the preprocessing procedure is introduced in paper [2].

Extracting coarse-grained parallelism in program loops

Definition 1. Given a dependence graph, *D*, defined by a set of dependence relations, *S*, a *slice* is a weakly connected component of graph *D*, i.e., a maximal subgraph of *D* such that for each pair of vertices in the subgraph there exists a directed or undirected path.

If there exist two or more slices in *D*, then taking into account the above definition, we may conclude that all slices are synchronization-free, i.e., there is no dependence between them.

Definition 2. An *ultimate dependence source(destination)* is a source (destination) that is not the destination (source) of another dependence. Ultimate dependence sources and destinations represented by relation *R* can be found by means of the following calculations: (domain(*R*) - range(*R*)) and (range(*R*) - domain(*R*)), respectively.

Definition 3. The *source(s) of a slice* is an ultimate dependence source(s) that this slice contains.

Definition 4. The *representative* loop statement instance of a slice is its lexicographically minimal source.

Further on in this paper, we refer to representative loop statement instances as to representatives.

The approach to extract synchronization-free slices [1] relies on the transitive closure of an affine dependence relation describing all dependences in a loop and consists of two steps. First, representatives of slices are found in such a manner that each slice is represented with its lexicographically minimal statement instance. Next, slices are reconstructed from their representatives and code scanning these slices is generated.

Given a dependence relation *R* describing all dependences in a loop, we can find a set of statement instances, S_{UDS}, describing all ultimate dependence sources of slices as S_{UDS}=domain(*R*) - range(*R*). In order to find elements of S_{UDS} that are representatives of slices, we build a relation, R_{USC}, that describes all pairs of the ultimate dependence sources that are transitively connected in a slice, as follows:

$$(1)\quad R_{USC} := \{[e] \to [e'] : e, e' \in S_{UDS}, e \prec e', R^*(e') \cap R^*(e)\},$$

where R^* is the transitive closure of relation *R*.

The condition $(e \prec e')$ in the constraints of relation R_{USC} means that *e* is lexicographically smaller than e'. Such a condition guarantees that the lexicographically smallest element from *e* and e' will always appear in the input tuple, i.e., the lexicographically smallest source of a slice (its representative source) can never appear in the output tuple. The intersection $R^*(e') \cap R^*(e)$ in the constraints of R_{USC} guarantees that elements *e* and *e'* are transitively connected, i.e., they are the sources of the same slice.

Set S_{repr} containing representatives of each slice is found as S_{repr}= S_{UDS}- range(R_{USC}).

Each element *e* of set S_{repr} is the lexicographically minimal statement instance of a synchronization-free slice. If *e* is the representative of a slice with multiple sources, then the remaining sources of this slice can be found applying relation (R_{USC})* to *e*, i.e., (R_{USC})*(*e*). If a slice has the only source, then (R_{USC})*(*e*)=*e*. The elements of a slice represented with *e* can be found applying relation R^* to the set of sources of this slice: S_{slice}= R^*((R_{USC})*(*e*)).

Let us illustrate the presented technique by means of the following parameterized loop.

Example 1

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

There is the following dependence relation returned by Petit.

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

The following set

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

Represents sources of slices.

Applying algorithm *Gen_affine* for independent slices extraction [1] and the *codegen* function from the Omega Calculator [4], we generate the following parallel code:

```
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];
  }
```

Next, we manually transform the above code to the parallel CUDA code:

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

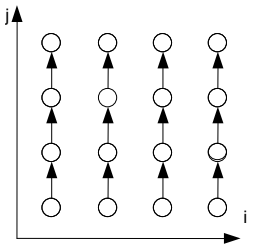

Fig.1. Iteration space and dependences of the loop

The main function of this code runs kernels of parallel loops. The value of variable *n_blocks* represents the number of threads that execute a single block of independent loop statement instances, i.e., the number of

engaged CUDA cores. The value of variable *idx* defines the identifier of a block; the values of variables *lb* and *ub* indicate the lower and upper bounds of the parallel loop, respectively; variable *packet* is to represent the number of iterations in a block.
Figure 1 presents slices for Example 1 when n=4.

Extracting fine-grained parallelism in program loops

Definition 5. The free schedule is the function that assigns statement instances (for execution) as soon as their operands are available, that is, it is mapping $\sigma : LD \to Z$ such that:

$$\sigma(p) = \begin{cases} 0 & \text{if there is no } p' \in LD \text{ s.t. } p' \prec p \\ 1 + \max(\sigma(p')), & p' \in LD, \ p' \prec p \end{cases} \tag{2}$$

Under the free schedule, loop statement instances are executed as soon as their operands are available. An approach for extracting parallel code scanning time partitions is presented in paper [5].

Given a dependence relation *R* describing all dependences in a loop, we create the relation *R'* by inserting variables *k* and and *k*+1 into the first position of the input and output tuples of relation *R*. Variable *k* is to present the time of a partition (a set of statement instances to be executed at time *k*). Next, we calculate the transitive closure of relation *R*', *R*'*, and form the following relation:

$$\begin{aligned} &FS = \{[X] \to [k,Y] : X \in UDS(R) \wedge \\ &(k,Y) \in Range((R')^* \setminus \{[0,X]\}) \wedge \\ &\neg\exists(k > k' s.t. (k',Y) \in Range((R') + \setminus \{[0,X]\})\}. \end{aligned} \tag{3}$$

where (*R*')*\{[0,*X*]} means that the domain of relation (*R*')* is restricted to the set including ultimate dependence sources only (elements of this set belong to the first time partition); the constraint $\neg\exists(k > k' s.t. (k',Y) \in Range((R') + \setminus \{[0,X]\})$ guarantees that partition *k* includes only those statement instances whose operands are available, i.e., each statement instance will belong to one time partition only.

The first element of the tuple representing the set Range(*FS*) points out the time of a partition while the last element of that exposes what is the statement whose instance(iteration) is defined by the tuple elements 2 to *n*-1, where *n* is the number of the tuple elements of a preprocessed relation. Taking the above consideration into account and provided that the constraints of relation *FS* are affine, the set Range(*FS*) is used to generate parallel code applying any well-known technique to scan its elements in the lexicographic order, for example the techniques presented in papers [6-7].

The outermost sequential loop of such code scans values of variable *k* (representing the time of partitions) while inner parallel loops scan independent instances of partition *k*.

Let us illustrate the presented technique by means of the following imperfectly-nested and parameterized loop.

Example 2

```
for(i=1; i<=n; i++){
 a[i][0] = 1; //s1
  for(j=1; j<=n; j++){
   a[i][j] = a[i-1][j] + a[i][j-1]; //s2
  }
}
```

There are the three dependence relations returned by Petit

```
R1 ={[i,-1,1] -> [i,1,2] : 1 <= i <= n};
R2 ={[i,j,2] -> [i+1,j,2] : 1 <= i < n && 1 <= j
<= n};
R3 ={[i,j,2] -> [i,j+1,2] : 1 <= i <= n && 1 <= j
< n}.
```

Applying the presented algorithm, we get the following results being produced by means of the Omega calculator.

```
R' = {[k,i,-1,1] -> [k+1,i,1,2] : 1 <= i <= n &&
0 <= k} union {[k,i,j,2] -> [k+1,i+1,j,2] : 1 <=
i < n && 1 <= j <= n && 0 <= k} union {[k,i,j,2]
-> [k+1,i,j+1,2] : 1 <= i <= n && 1 <= j < n && 0
<= k}.

 R'+ = {[k,i,j,2] -> [k',i',i-k+j-i'+k',2] : 1 <=
i <= i' <= n && 0 <= k < k' && 1 <= j && k+i' <=
i+k' && i+j+k' <= n+k+i'} union {[k,i,-1,1] ->
[k',i',i-k+k'-i',2] : 1 <= i <= i' <= n && k+i' <
i+k' && 0 <= k && i+k' <= n+k+i'}.

FS = {[1,-1,1] -> [k,i',k-i'+1,2] : 1 <= i' <= k,
n && k < n+i'} union {[i,-1,1] -> [0,i,-1,1] : 1
<= i <= n}.

Range(FS) = {[k,i,k-i+1,2]: 1 <= i <= k, n && k <
n+i} union {[0,i,-1,1]: 1
<= i <= n}.
```

The loop scannig elements of the set Range(FS) for k<=0 and being produced by the codegen function of the Omega library is as follows.

```
for(t2 = 1; t2 <= n; t2++) { // parallel loop
 a[t2][0] = 1; // s1(0,t2,-1,1);
}
for(t1 = 1; t1 <= 2*n-1; t1++)
{
 for(t2 = max(-n+t1+1,1); t2 <= min(n,t1); t2++)
 {
   //parallel loop
   a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-
   t2];
   // s1(t1,t2,t1-t2+1,2);
 }
}
```

There is no independent statements in the loop.
The pseudocode above was manually transformed to the parallel code for NVIDIA cards presented below.

```
//Kernel definitions
__global__ void loop1_gpu(float (*a)[n])
{
 int idx = blockIdx.x, t2;
 int packet = (int)ceil(n / blockDim.x);
 int lb = idx*packet+1;
 int ub = ((idx+1)*packet < n) ? (idx+1)*packet :
n;
 for(int t2 = lb; t2 <= ub; t2++)
  a[t2][0] = 1;
 }

__global__ void loop2_gpu(float (*a)[n], t1)
{
 int idx = blockIdx.x, t2;
 int packet = (int)ceil((max(-n+t1+1,1) -
min(n,t1)) / blockDim.x);
 int lb = idx*packet+max(-n+t1+1,1);
 int ub = ((idx+1)*packet < min(n,t1)) ?
(idx+1)*packet : min(n,t1);
 for(int t2 = lb; t2 <= ub; t2++)
  a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-
t2];
}

 int main(int argc, char * argv[]){
 ...
```

```
 int threads_per_block = 1;
 int n_blocks = atoi(argv[1]); // number of CUDA
//cores
// Kernel invocation
 loop1_gpu <<< n_blocks, threads_per_block>>>
((float(*)[n])d_A);
cudaSynchronize();
for(t1 = 1; t1 <= 2*n-1; t1++) {
 loop2_gpu <<< n_blocks, threads_per_block>>>
((float(*)[n])d_A, t1);
...
 }
 cudaSynchronize();
}
```

Figure 2 presents the free schedule for the loop of Example 2 when n=5. The solid lines represent dependences, the dotted lines represent synchronization barriers between time partitions.

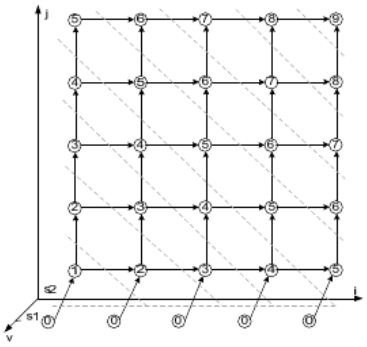

Fig.2. The free schedule for Example 2 when n=5.

Environment of experiments

Experiments have been carried out for a massively-parallel machine, NVIDIA Tesla S1070. The GPU computing system includes four teraflop processors with 240 cores (960 scalar processor cores). It is equipped with 16 GB GDDR3 memory (4Gb for each processor) with ultra-fast access (408 GB/sec total bandwidth) and consumes about 800 watts of power [8].

To connect all four GPUs in a Tesla S1070 to a single host system, the host must have two available PCI Express slots corresponding to the NVIDIA switch. The architecture of a GPU computing system is shown in Figure 3. The Tesla card has been mounted in a computer with Quad-Core Xenon E5504 1.6 Ghz CPU, 8GB RAM and OpenSuse Linux.

Parallel loops were implemented by means of the CUDA library [11]. The CUDA C compiler (nvcc) simplifies many-core programming by enabling code development in a high-level language and optimizing code to run on NVIDIA systems. CUDA applications automatically take advantage of many or few cores in a system, so they can scale from an entry-level notebook GPUs to racks of GPUs. Tesla S1070 is compatible with the CUDA 1.3 version.

Results of experiments

The presented algorithms were implemented by us in a tool by means of the Omega library. It generates C-like pseudo-code scanning synchronization-free slices with defining variables to be privatized. Using this tool, we have experimented with loops of the NAS 3.2 benchmark suite [10].

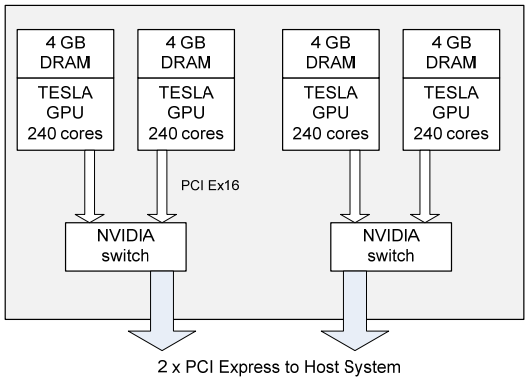

Fig.3. The architecture of the NVIDIA Tesla S1070 [9]

NAS Parallel Benchmarks (NPB) have been developed at the NASA Ames Research Centre to study performance of parallel supercomputers. Benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications [10].

To assess the efficiency of code produced by the presented algorithms, the following criteria were taken into account for choosing NAS loops:

- a loop must be computatively heavy (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified),
- code produced by the algorithm must be parallel (there are NAS loops that cannot be parallelized),
- structures of chosen loops must be different (there are many NAS loops of a similar structure).

Applying these criteria, we selected the following NAS loops: *FT_auxfnct_2* (Fast Fourier Transform Benchmark), *UA_diffuse_4* and *UA_setup_16* (both from Unstructured Adaptive benchmark). The loops are presented in Table 1.

Table 1. Loops for experiments

ua_setup_16

```
for i=1 to N1 do
  for j=1 to N2 do
    for ip=1 to N3 do
      wdtdr(i,j) = wdtdr(i,j) + wxm1(ip)*dxm1(ip,i)*dxm1(ip,j)
    endfor
  endfor
endfor
```

ft_auxfnct_2

```
for i=1 to N1 do
  for k=1 to N2 do
    for j=1 to N3 do
      y(j,k,i)=y(j,k,i)*twiddle(j,k,i)
      x(j,k,i)=y(j,k,i)
    endfor
  endfor
endfor
```

ua_diffuse_4

```
for iz=1 to N1 do
  for k=1 to N2 do
    for j=1 to N3 do
      for i=1 to N4 do
        tm2(i,j,iz) = tm2(i,j,iz)+u(i,k,iz)*wdtdr(k,j)
      endfor
    endfor
  endfor
endfor
```

Table 2. Loops execution times, *s*.

LOOP	ALGORITHM	PARAMETERS	1 GPU (seq. program)	2 GPU	32 GPU	256 GPU	960 GPU	DATA TRANSFER
ua_setup_16	Scheduling	N=500	64.202	36,091	2,237	0,567	0,051	1,373096
		N=1000	512.89	288,789	18,297	7,276	6,541	1,373096
	Slicing	N=500	64.202	33,634	2,06	0,308	0,000074	1,375112
		N=1000	512.89	269,04	17,131	4,898	3,476	1,378965
ft_auxfnct_2	Scheduling	N=200	4,468	2,247	0,138	0,000068	0,000066	0,061973
		N=500	67,428	33,959	2,079	0,543	0,000066	0,803661
	Slicing	N=200	4,468	3,677	0,238	0,000063	0,000068	0,039986
		N=500	67,428	58,872	3,578	0,73	0,000068	0,59415
ua_diffuse_4	Scheduling	N=1024	58,819	29,384	1,982	1,158	0,866	0,429788
		N=2048	238,122	118,651	7,672	3,234	2,311	1,031796
	Slicing	N=1024	58,819	26,384	1,682	0,543	0,384	0,535126
		N=2048	238,122	105,398	6,711	2,013	1,357	0,834596

Table 3. Speed-up and efficiency of parallel loops

LOOP	ALGORITHM	PARAMETERS	2 GPU		32 GPU		256 GPU		960 GPU	
			S	E	S	E	S	E	S	E
ua_setup_16	Scheduling	N=500	1.750	0.875	18.164	0.568	33.800	0.132	46.047	0.048
		N=1000	1.772	0.886	26.144	0.817	59.459	0.232	64.981	0.068
	Slicing	N=500	1.873	0.937	19.090	0.597	38.962	0.152	47.686	0.050
		N=1000	1.902	0.951	27.783	0.868	81.930	0.320	105.926	0.110
ft_auxfnct_2	Scheduling	N=200	1.962	0.981	22.653	0.708	73.016	0.285	73.018	0.076
		N=500	1.963	0.981	23.670	0.740	50.667	0.198	84.894	0.088
	Slicing	N=200	1.213	0.606	16.217	0.507	112.562	0.440	112.548	0.117
		N=500	1.144	0.572	16.304	0.509	51.370	0.201	114.473	0.119
ua_diffuse_4	Scheduling	N=1024	1.987	0.994	24.566	0.768	37.315	0.146	45.724	0.048
		N=2048	1.998	0.999	27.477	0.859	56.063	0.219	71.543	0.075
	Slicing	N=1024	2.205	1.102	26.771	0.837	55.053	0.215	64.577	0.067
		N=2048	2.249	1.125	31.668	0.990	83.915	0.328	109.033	0.114

Table 2 shows loop execution time for 1, 2, 32, 256, and 960 processors). Experiments were carried out for two different values of the upper bounds of loop indices (see column 3). Slicing means that parallel programs were produced by applying the algorithms of extracting synchronization-free slices[1] while Scheduling stands for programs being produced on the basis of the algorithm presented in paper [5]. Both slicing and scheduling permit for good utilization of many GPUs (up 960 under our experiments).

The last column of Table 2 presents the time of data transfer to/from a graphic card. It is worth to note that the time of data transfer does not depend on the number of GPU cores [11]. The time of data transfer comprises the times of [11]: allocation, sending data to the graphic card, and fetching data memory of the graphic card.

Table 3 presents speed-up and efficiency for the studied loops. The results of experiments demonstrate that produced parallel programs are scalable: the time of a loop execution reduces with increasing the number of processors (up to 960). That is the computation power of the many-core GPU system is efficiently used. Figure 4 illustrates the loop execution time for 1, 2, 32, 256, 960 GPUs and the time of data transfer between the host and the graphic card (*dt*).

Related Work

The CUDA parallel hardware architecture is accompanied by the CUDA parallel programming model that provides a set of abstractions that enable expressing fine-grained and coarse-grain data and task parallelism.

Different techniques have been developed to extract parallelism available in loops. In paper [12], an automatic polyhedral compilation for GPGPU is presented for the polyhedral loop parallelizer: LooPo [13].The polytope model is recognized as useful for parallelizing loop programs for massively parallel architectures.

The affine transformation framework, considered in papers [14-16] unifies a large number of previously proposed loop transformations. It is implemented in the tool Pluto - an automatic parallelization]. Version 0.6.2 with support for generating CUDA code is available. However, the affine transformation framework does not exploit all parallelism with synchronization-free slices in some cases [1].

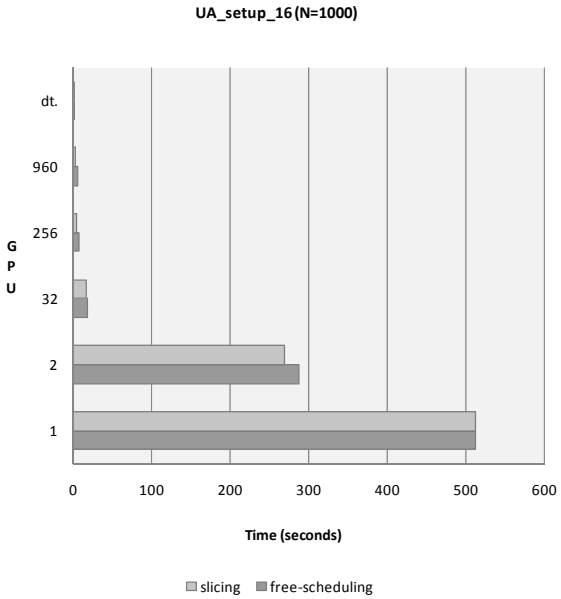

Fig.4. Times for UA_setup_16, n=1000.

Conclusion

Presented in this paper results demonstrate that both slicing and scheduling based on transitive closure calculation of dependence relations can be successfully applied for producing parallel programs foe NVIDIA cards with many GPUs.

In the future work, we intend to develop algorithms of extracting parallelism to be utilized simultaneously by GPUs and CPUs of the same computer.

We thank Piotr Czapiewski from the West Pomeranian University of Technology for providing a machine with the card Tesla S1070.

Wydanie publikacji zrealizowano przy udziale środków finansowych otrzymanych z budżetu Województwa Zachodniopomorskiego.

REFERENCES

[1]] Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K. : Coarse-grained loop parallelization: *Iteration space slicing vs affine transformations*. Parallel Computing, No. 37, (2011), 479-497,

[2] 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, Springer, (2008), 871-886.

[3] 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).

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

[5] Bielecki W., Palkowski M., Using Free Scheduling for Programming NVIDIA Cards, Proceedings of the 2nd Facing the Multicore-Challenge Conference, Karlsruhe, Germany, (2011).

[6] Bastoul C., *Code generation in the polyhedral model is easier than you think,* In IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques PACT'04, (2004).

[7] Verdoolaege S., *Integer Set Library: Manual*, version 0.0.9, http://www.kotnet.org/~skimo/isl/manual.pdf, (2011).

[8] Specification, Tesla S1070 GPU Computing System, http://www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf, (2008).

[9] Tesla Data Center Solutions, S1070 Product Brief, http://www.nvidia.com/object/preconfigured-clusters.html, (2011).

[10] The NAS benchmark suite, http://www.nas.nasa.gov.

[11] NVIDIA CUDA, Programming guide, http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf, version 4.0, (2011).

[12] Lengauer, C. *Loop Parallelization in the Polytope Model*. In Eike Best, editor, CONCUR'93, number 715 in Lecture Notes in Computer Science, Springer-Verlag, (1993), 398–416.

[13] Baghdadi, S., Größlinger, A., Cohen, A. *Putting Automatic Polyhedral Compilation for GPGPU to Work*. In Proc. of Compilers for Parallel Computers (CPC), (2010).

[14] Feautrier, P., *Some efficient solutions to the affine scheduling problem*, part I and II, one and multidimensional time, International Journal of Parallel Programming 21, (1992), pp. 313-348 and 389- 420.

[15] Lim, A., Lam, M., Cheong, G., *An affine partitioning algorithm to maximize parallelism and minimize communication*. In ICS'99, ACM Press, (1999), 228-237,

[16] PLUTO - An automatic parallelizer and locality optimizer for multicores, http://pluto-compiler.sourceforge.net.

***Authors**: prof. dr hab. inż. Włodzimierz Bielecki, Zachodniopomorski Uniwersytet Technologiczny, Katedra Inżynierii Oprogramowania, ul. Żołnierska 49, 71-210 Szczecin, E-mail: wbielecki@wi.zut.edu.pl; dr inż. Marek Pałkowski, Zachodniopomorski Uniwersytet Technologiczny, Katedra Inżynierii Oprogramowania, ul. Żołnierska 49, 71-210 Szczecin, E-mail: mpalkowski@wi.zut.edu.pl;*