Parallel Computing 39 (2013) 867–878

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Improving application behavior on heterogeneous manycore systems through kernel mapping [☆]

Omer Erdil Albayrak, Ismail Akturk, Ozcan Ozturk [*]

Computer Engineering Department, Bilkent University, Ankara, Turkey

ARTICLE INFO

Article history:
Available online 4 September 2013

Keywords:
Mixed Integer Programming
Kernel mapping
Heterogeneous systems

ABSTRACT

Many-core accelerators are being more frequently deployed to improve the system processing capabilities. In such systems, application mapping must be enhanced to maximize utilization of the underlying architecture. Especially, in graphics processing units (GPUs), mapping kernels that are part of multi-kernel applications has a great impact on overall performance, since kernels may exhibit different characteristics on different CPUs and GPUs. While some kernels run faster on GPUs, others may perform better in CPUs. Thus, heterogeneous execution may yield better performance than executing the application only on a CPU or only on a GPU. In this paper, we investigate on two approaches: a novel profiling-based adaptive kernel mapping algorithm to assign each kernel of an application to the proper device, and a Mixed-Integer Programming (MIP) implementation to determine optimal mapping. We utilize profiling information for kernels on different devices and generate a map that identifies which kernel should run where in order to improve the overall performance of an application. Initial experiments show that our approach can efficiently map kernels on CPUs and GPUs, and outperforms CPU-only and GPU-only approaches.

1. Introduction

Today's high performance and parallel computing systems consist of different types of accelerators, such as Application-Specific Integrated Circuits (ASICs) [1], Field Programmable Gate Arrays (FPGAs) [2], Graphics Processing Units (GPUs) [3], and Accelerated Processing Units (APUs) [4]. In addition to the variety of accelerators in these systems, applications that are running on these systems also have different processing, memory, communication, and storage requirements. Even a single application may exhibit different such requirements throughout its execution. Thus, leveraging the provided computational power and tailoring the usage of resources based on the application's execution characteristics is immensely important to maximize both application performance and resource utilization.

Applications running on heterogeneous platforms are usually composed of multiple exclusive regions known as kernels. Efficient mapping of these kernels onto the available computing resources is challenging due to the variation in characteristics and requirements of these kernels. For example, each kernel has a different execution time and memory performance on different devices. It is our goal to generate a kernel mapping system that takes the characteristics of each kernel and their dependencies into account, leading to improved performance.

[☆] A preliminary version of this paper appears in P2S2 2012 workshop proceedings. This submission extends our previous work by presenting a mixed integer programming based mapping implementation and by presenting an extensive experimental evaluation.
[*] Corresponding author. Tel.: +90 3122903444; fax: +90 3122664047.
E-mail addresses: oalbayrak@cs.bilkent.edu.tr (O.E. Albayrak), iakturk@cs.bilkent.edu.tr (I. Akturk), ozturk@cs.bilkent.edu.tr (O. Ozturk).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.08.011

In this paper, we propose a novel adaptive profiling-based kernel mapping algorithm for multi-kernel applications running on heterogeneous platforms. Specifically, we run and analyze each application on every device (CPUs and GPUs) in the system to collect necessary information, including kernel execution time and input/output data transfer time. We, then, pass this information to a solver to determine the mapping of each kernel on the heterogeneous system. Solvers used are a greedy algorithm (GA) based solver, an improved version of the same algorithm (IA), and a Mixed-Integer Programming (MIP) based solver. Our specific contributions are:

- an off-line profiling analysis to extract kernel characteristics of applications.
- an adaptive greedy algorithm (GA) to select the suitable device for a kernel considering its execution time and data requirements.
- an improved version of the greedy algorithm (IA) to avoid getting stuck in local minima.
- a Mixed-Integer Programming (MIP) implementation to determine optimal mapping and to compare it with the greedy approach.

The initial results revealed that our approach increases the performance of an application considerably compared to a CPU-only or GPU-only approach. Furthermore, in many cases, our generated mappings are equivalent to the mappings of MIP implementations, or very close to them. Although our initial experiments are limited to a single type of CPU and GPU, it is possible to extend this work to support multiple CPUs, GPUs, and other types of accelerators.

The remainder of this paper is organized as follows. Related work on general-purpose GPU computing (GPGPU) is given in Section 2. The problem definition and an introduction to the proposed approach are given in Section 3. The details of the greedy algorithm (GA) and the implementation are given in Section 4. The MIP formulation is introduced in Section 5. The experimental evaluations are presented in Section 6. Finally, we conclude the paper in Section 7.

2. Related work

OpenCL is an open standard for parallel programming, targeting heterogeneous systems [5]. It began as an open alternative to Brook [6], IBM CELL [7], AMD Stream [8], and CUDA [9]. It provides a standard API that can be employed on many architectures regardless of their characteristics, and therefore has become widely accepted and supported by major vendors. In this work, we also used OpenCL and evaluated our approach on an OpenCL version of the NAS benchmarks [10].

Recent advancement in chip manufacturing technology has made it feasible to produce power efficient and highly parallel processors and accelerators. This, however, increases the heterogeneity of the computing platforms and makes it challenging to determine where to run the given application. To the best of our knowledge, there are only a few studies targeting this critical challenge. Luk et al. proposed Qilin [11], which uses a statistical approach to predict kernel execution times offline, generate a mapping and perform the execution. Rather than individual kernel mapping, they partition single instruction multiple data (SIMD) operations into sub-operations and map these sub-operations to the devices. In contrast to Qilin, we aim to map the kernels as a whole rather than their sub-operations. Kernel splitting across devices can result in with a better performance; however it cannot be applied to an application directly since it requires deeper analysis. Specifically, splitting a kernel across the CPU and GPU may not be possible due to data objects and their accesses. Therefore, such an approach requires appropriate modifications on the kernel. On the other hand, our approach does not require manual modification and can be applied to any application with multiple kernels.

In addition, while we obtain CPU and GPU execution times (in addition to the data transfer times) through profiling, they use statistical regression model to estimate these values. Therefore, it is orthogonal to our profiling method, and can be integrated into our system when profiling is not possible or when it is costly.

Grewe and O'Boyle [12] proposed a machine learning-based task mapping algorithm. They use a predictor to partition tasks on heterogeneous systems. Their method predicts the target device for each task according to the extracted code features used in the training set of the machine-learning algorithm. Our decision method could be enhanced with such techniques in the future.

Scogland et al. [13] presented a runtime system that can divide an accelerated region across all available resources automatically. This work specifically focuses on OpenMP applications and therefore not applicable to our kernel based applications. Similarly, StarPU [14] proposed a runtime system which is capable of scheduling tasks over heterogeneous, accelerator-based machines. However, StarPU requires that tasks have different implementations for each heterogeneous device. Both of these studies are relevant but have some limitations.

The main difference between these works and ours is that ours is an adaptive profiling-based kernel mapping algorithm.

3. Problem description and overview of our approach

3.1. Preliminaries

A major challenge in a heterogeneous platform is using existing resources while obtaining an application's highest performance. This issue is mainly due to the nature of such systems, because they comprise computing devices with different characteristics and capabilities. Therefore, the main goal of this work is to utilize these devices by capturing tasks' specific characteristics and making task-assignment decisions in a way that tasks are assigned to the device they perform better. Ultimately, the overall performance will be improved and the underlying resources will be utilized. Traditionally, applications run on a single hardware which they perform better. However, when the application is divided into tasks, each task is assigned to a device, thereby improving the overall performance.

Formally, a kernel mapping problem can be defined as a triplet $MAP = \{K, D, S\}$ where K is a set of kernels $K = \{k_1, k_2, k_3, \ldots, k_n\}$, D is a set of devices which kernels can execute on $D = \{d_1, d_2, d_3, \ldots, d_m\}$, and $S = \{s_1, s_2, s_3, \ldots, s_t\}$ is a data object set used by kernels. Each kernel $k_a \in K$ is sequentially executed on a device $d_b \in D$ right after kernel k_{i-1} and the required data objects $s_c \in S$ are copied to local buffers before execution.

For this paper, we use a simple heterogeneous platform with a single type of CPU and GPU. However, in reality, a heterogeneous platform may consist of multiple types of CPUs, GPUs, and APUs from different vendors with different features [15–17]. For example, it is possible to have an NVIDIA GPU and an AMD GPU in the same system. While NVIDIA GPUs are good for simple parallel multi-threaded computations, AMD GPUs are better in vector operations [18,19]. Thus, the characteristics of a task, such as the number of vector operations and the number of threads running in parallel, become critical while deciding where to run the given application.

Similarly, the size of data required by an application is an important issue, because some of the devices, such as GPUs, may have limited memory space. Therefore, even though an application may be developed targeting a GPU, it may not be possible to execute it on a given GPU because of the limitation on memory.

In addition to kernel characteristics and device specifications, dependencies between kernels are also of concern. Running dependent kernels in two different devices requires data movement upon depended kernels to finish the execution. Hence, for certain mappings it is necessary to consider data transfer costs while assessing the performance of an application.

3.2. Our approach

Fig. 1 illustrates, at a high level, how our approach operates. First, we extract the profiling information which is used in kernel mapping.

Specifically, we run and analyze each application on every device (CPUs and GPUs) in the system to collect necessary information, including kernel execution time and input/output data transfer time. As an alternative, we can extract kernel characteristics through compiler analysis and employ a machine-learning-based technique, similar to [12], to predict kernel execution times and data transfer overheads. However, this is left as a future work.

Extracted kernel characteristics is subsequently passed to the solver which determines the mapping of each kernel on the heterogeneous system. Specific solvers used are a greedy algorithm (GA) based solver implemented using Java, an improved version of the greedy solver (IA), and a Mixed-Integer Programming (MIP) based solver implemented on a commercial tool. Our goal in selecting the mapping is to generate a mapping that minimizes the execution cost of each kernel. Note that, depending on the functionality, each device on the system can exhibit different characteristics in terms of data transfer cost, data bandwidth, and performance. This information is crucial for selecting the target device.

4. Greedy mapping algorithm

In this section, we introduce the greedy algorithm (GA) to generate kernel mapping, which minimizes the overall execution cost. Due to the complex data dependencies among kernels, minimizing the execution cost of each kernel may not minimize the overall performance of an application. Therefore, base algorithm may not yield the best results.

In the base algorithm, we try to minimize the execution time of each kernel by selecting the device that will run the given kernel faster. Eventually, we aim to generate a mapping that improves performance to beyond that of CPU-only or GPU-only mapping. We formulate the CPU and GPU execution costs as follows:

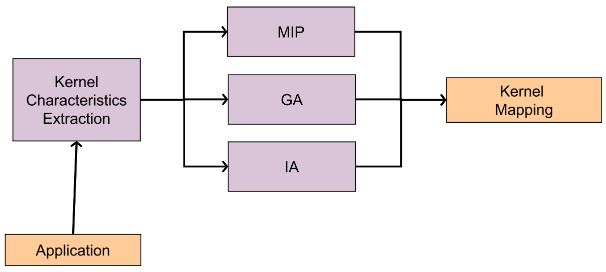

Fig. 1. High level view of our approach.

$$CPUcost_k = CPUrunningtime_k + \sum_{d=1}^{n} DeviceToHost \times InDevice_d \times Required_{k,d} \times size_d. \quad (1)$$

$$GPUcost_k = GPUrunningtime_k + \sum_{d=1}^{n} HostToDevice \times (1 - InDevice_d) \times Required_{k,d} \times size_d. \quad (2)$$

The first part in each equation indicates the execution time, while the second part is the data transfer cost. *HostToDevice* and *DeviceToHost* functions are simply the data transfer costs from device to host and vice versa. Note that, $Required_{k,d}$ is either 1 or 0, and indicates whether kernel k requires data d. Data might already be present on the target device and thus may not need to be moved in. For this purpose, we use $InDevice_d$ to indicate whether data d is already in the target device. Similarly, we express the size of data d using $size_d$.

The aforementioned constants are extracted through profiling except $InDevice_d$. $InDevice_d$ is a binary variable and its value depends on the recent iteration of the algorithm that accessed data d. If data was left in the device after the last access, $InDevice_d$ will be 1, otherwise it will be 0. The algorithm assumes that all the data is initially stored in the CPU, as it is the case in most systems. This assumption can easily be modified to reflect a different system. Algorithm 1 gives the pseudo code for the base algorithm.

Algorithm 1 Greedy Algorithm (GA)

```
procedure BASEALGORITHM
$total\_cost = 0$
for all Kernel $k$ do
  $cpu\_cost = k.CPU\_TIME$ + D2H ($k$)
  $gpu\_cost = k.GPU\_TIME$+ H2D ($k$)
  if $cpu\_cost < gpu_cost$ then
      $k.onCpu \leftarrow true$
      $k.cost \leftarrow cpu\_cost$
      for all Buffer $b \in k$ do
        $b.onCpu \leftarrow true$
      end for
    else
      $k.onCpu \leftarrow false$
      $k.cost \leftarrow gpu\_cost$
      for all Buffer $b \in k$ do
      $b.onCpu \leftarrow false$
      end for
    end if
    $total\_cost += k.cost$
  end for return $total\_cost$
end procedure
procedure H2DKernel k
  $cost = 0$
  for all Buffer $b \in k$ do
    if $b.onCPU == true$ then
      $cost += b.D2H\_transfer\_cost$
    end if
  end for
end procedure
procedure D2HKernel k
  $cost = 0$
  for all Buffer $b \in k$ do
    if $b.onCPU == false$ then
      $cost += b.H2D\_transfer\_cost$
    end if
  end for
end procedure
```

The base algorithm works with most of the tested benchmarks. However, in some cases there is a possibility of getting stuck in local minima due to complex data dependencies among kernels that were not handled in the base algorithm. We proposed the improved algorithm (IA, see Algorithm 2) to avoid such deadlocks. We introduced the notion of *Critical Points*, where the algorithm can select the worst target, for the purpose of assessing alternative mappings that may yield better performance. Table 1 gives a simple example to show the effect of employing the improved algorithm (IA).

Algorithm 2 Improved Algorithm (IA)

```
procedure IMPROVEDALGORITHM
  $total\_cost = 0$
  for all Kernel $k$ do
    $cpu\_cost = k.CPU\_TIME$ + D2H ($k$)
    $gpu\_cost = k.GPU\_TIME$ + H2D ($k$)
    $k\_clone \leftarrow k.clone()$
    $k\_clone.onCPU \leftarrow true$      ▷ set k_clone as if CPU is selected and run BaseAlgorithm to observe the results of CPU selection
    for all Buffer $b \in k\_clone$ do
      $b.onCPU \leftarrow true$
    end for
    $whatif\_cpu\_cost \leftarrow$ BaseAlgorithm ($k$)      ▷ run base algorithm starting from k_clone
    $k\_clone \leftarrow k.clone()$
    $k\_clone.onCPU \leftarrow false$
    for all Buffer $b \in k\_clone$ do
      $b.onCPU \leftarrow false$
    end for
    $whatif\_gpu\_cost \leftarrow$ BaseAlgorithm ($k$)
    if $(cpu\_cost + whatif\_cpu\_cost) < (gpu_cost + whatif\_gpu\_cost)$ then
      $k.onCPU \leftarrow true$
      $k.cost \leftarrow cpu\_cost$
      for all Buffer $b \in k$ do
        $b.onCPU \leftarrow true$
      end for
    else
      $k.onCPU \leftarrow false$
      $k.cost \leftarrow gpu\_cost$
      for all Buffer $b \in k$ do
        $b.onCPU \leftarrow false$
      end for
    end if
    $total\_cost += k.cost$
  end for return $total\_cost$
end procedure
```

When Algorithm 1 is considered for the example in Table 1, the total cost of running the first kernel on a CPU is calculated as the summation of execution time on the CPU and the data movement cost if data is not currently on the CPU. Note that, kernels run sequentially according to the initial kernel order. Moreover, all of the buffer objects are initially assumed to be on CPU. In our example, data is located in the CPU, thereby the total cost of running the first kernel on the CPU is 5 + 0 = 5. Similarly, the total cost of running the first kernel on a GPU is calculated as the summation of the execution time on the GPU and the data transfer cost if the data is not currently on the GPU. Data is initially on the CPU, which leads to a total cost of running the first kernel on the GPU as 4 + 2 = 6. Since the total cost of executing the first kernel on the CPU is lower, the base algorithm would choose the CPU in mapping. Likewise, the second kernel will be mapped to the CPU because the total costs are 3 and 4 for the CPU and GPU, respectively. Similarly, the third kernel will also be mapped to the CPU since the total

Table 1
A simple example to show the difference between GA and IA.

Kernel number	CPU execution latency	GPU execution latency	Data used	CPU to GPU transfer time	GPU to CPU transfer time
1	5	4	A	2	2
2	3	2	A	2	2
3	7	6	A	2	0

costs are 7 and 8 for the CPU and GPU, respectively. Therefore, the total execution time will be 5 + 3 + 7 = 15. However, if the first kernel was executed on the GPU (although the cost is higher than running on the CPU) the second and third kernel would run on the GPU, as well. Because data used by the first kernel (i.e. A) is also used by the second and third kernels, the total cost would be (4 + 2) + 2 + 6 = 14, which is lower than the first mapping (which was CPU only). In this example, the base algorithm got stuck in local minima at the first kernel. The improved algorithm (IA), on the other hand, allows a data transfer to the GPU that seem costly at the beginning but ultimately, it enables other kernels to run on the GPU and thus resulted in a lower total execution time compared to the mapping generated by the base algorithm.

As previously indicated, we aim to avoid getting stuck in local minima by employing Algorithm 2. This algorithm essentially compares two assumptions at each critical point: (i) it assumes the CPU is a better option and performs the remaining decisions according to Algorithm 1, and (ii) it assumes the GPU is a better option and performs the remaining decisions according to Algorithm 1. The best result among (i) and (ii) is selected and the kernel of interest is permanently assigned to that device. This algorithm is applied to each kernel. In addition, for each kernel, Algorithm 2 applies Algorithm 1 to the remaining kernels. Therefore, for kernel i Algorithm 2 calls Algorithm 1, and Algorithm 1 runs a loop of $(n-i)$ iterations. For kernel $(i+1)$ Algorithm 1 runs a loop of $(n-i-1)$ iterations, and so on. For all kernels, a total of $\frac{(n-1)*n}{2}$ executions are performed; therefore our improved algorithm (IA) has a complexity of O (n^2).

5. Mixed-Integer Programming

In this section, our aim is to present a MIP formulation of the kernel mapping to find the optimal mapping and compare it with the mapping generated by our greedy algorithm (GA) presented in the previous section. It is not presented as a separate mapping strategy, instead the results gathered are used as a guideline for our greedy algorithm.

Mixed-Integer Programming provides a set of techniques that solves optimization problems with a linear objective function and linear constraints [20]. We used the *IBM ILOG OPL IDE* [21] tool to formulate and solve our mapping problem. Mixed-integer problems are generally NP-hard problems, yet depending on the algorithms that the solver uses, near-optimal results can be obtained quickly, even if optimal results can not.

In our formulation, the same profiling information is used as in the greedy algorithm (GA). When the MIP solver finishes executing, it generates a mapping with total execution times as well as total data transfer cost.

First, we have some predefined constant values extracted from the profiling data, which are:

- $cpuTime_k$: indicates the running time of kernel k on the CPU.
- $gpuTime_k$: indicates the running time of kernel k on the GPU.
- $required_{b,k}$: indicates if buffer b is required by kernel k.
- $cpu2gpu_{b,k}$: indicates the transfer time of buffer b from the CPU to the GPU for kernel k.
- $gpu2cpu_{b,k}$: indicates the transfer time of buffer b from the CPU to the GPU for kernel k.

To express the location of a kernel and buffer object we have two binary variables:

- T_k : indicates if kernel k runs on the CPU or the GPU. If kernel runs on the CPU, then it is 0; otherwise it is 1.

To keep track of each buffer object during each kernel execution we have another binary variable:

- $V_{b,k}$: indicates if buffer b is on the CPU or the GPU during the execution of kernel k. If buffer b is on the CPU, then it is 0; otherwise it is 1.

In our implementation, each kernel has CPU-to-GPU and GPU-to-CPU data transfer times; the GPU-to-CPU data transfer time is calculated after the execution of kernel k, while the CPU-to-GPU is calculated before. Therefore, if kernel k requires buffer b, its GPU-to-CPU transfer time is bound to the last kernel that used buffer b, so we need to propagate this transfer time to kernel k. To propagate the transfer time to a kernel, we have another decision variable:

- $gpu2cpu_{b,k}$: keeps track of current data transfer time from GPU to CPU for each kernel k.

After describing our constants and binary variables, we define our constraints. Our first constraint is an initialization constraint, which forces each buffer object initially to be on the CPU.

$$\begin{aligned} &V_{b,k} = 0, \\ &\forall b, \quad \text{and s.t.} \quad k = 0 \end{aligned} \tag{3}$$

For each kernel, locations of buffer objects need to be set. In Eq. (4), if buffer b is used in kernel k then its location is set to where kernel k runs. Otherwise, the location of b is set to its previous location.

$$V_{b,k} = required_{b,k} \times T_k + (1 - required_{b,k}) \times V_{b,k-1}, \forall b, k \tag{4}$$

The *htd* variable keeps track of each CPU-to-GPU data transfer cost. If a buffer is previously on the CPU then, transferred to GPU, its *cpu2gpu* cost is updated in the *htd* variable. Note that if buffer *b* is on the GPU during the execution of kernel *k*, then $V_{b,k}$ value is 1, otherwise it is 0. Therefore, subtracting two consecutive *V* values yields the direction of the transfer:

$$htd_{b,k} \geqslant (V_{b,k} - V_{b,k-1}) \times cpu2gpu_{b,k}, \ \forall b, k \tag{5}$$

As explained above, in our implementation, the GPU-to-CPU transfer cost of buffer *b* is bound to the last kernel that used buffer *b*. Therefore, we need to propagate this cost to the following kernels that require buffer *b*. Eq. (6) formulates this operation. If a kernel requires a buffer object, then its GPU-to-CPU data transfer cost is set. If a buffer object is not required, then the last cost is passed on.

$$gpu2cpu_{b,k} = required_{b,k} \times gpu2cpu_{b,k} + (1 - required_{b,k}) \times gpu2cpu_{b,k-1}, \quad \forall b, k \tag{6}$$

To calculate the GPU-to-CPU transfer costs of each buffer object, we used the *dth* variable, which is similar to *htd* in Eq. (5). However, *dth* is calculated as multiplication of two decision variables, because the GPU-to-CPU data transfer cost is present in the *gpu2cpu* variable and the direction of data movement is present in the *V* variable. Multiplying two variables yields a non-linear problem; to cope with it, we employed the big-M method [22]. In the big-M method, an *M* constant that is greater than the largest value in one of the variables is necessary. In Eq. (7), *V* variables are subtracted. If the subtraction of *V* variables is equal to 1, then the *M* constants cancel out and the *dth* value is set to the value of *gpu2cpu*. If this subtraction is not equal to 1, the result of the equation becomes a negative value since the *M* value is greater than the value of *gpu2cpu*. Note that the *dth* variable is 0 or positive; thus if the right hand side of the expression is negative, then the variable is automatically assigned to 0, and eventually has no effect.

$$dth_{b,k} \geqslant (V_{b,k-1} - V_{b,k}) \times M + gpu2cpu_{b,k} - M, \quad \forall b, k \tag{7}$$

Until this point, the *dth* and *htd* variables keep track of the cost of each data transfer. In contrast, Eqs. (8) and (9) define $dev2host$ and $host2dev$, which are the summations of the *dth* and *htd* variables, respectively.

$$dev2host = \sum_{b=1}^{\#buffers} \sum_{k=1}^{\#kernels} dth_{b,k} \tag{8}$$

$$host2dev = \sum_{b=1}^{\#buffers} \sum_{k=1}^{\#kernels} htd_{b,k} \tag{9}$$

Next, the computation cost of each kernel is calculated. Note that, variable *T* keeps track of where each kernel runs. If a kernel is executed on the GPU, then it becomes 1; otherwise it becomes 0 (i.e. it is executed on the CPU).

$$compCost = \sum_{k=1}^{\#kernels} (1 - T_k) \times cpuTime_k + T_k \times gpuTime_k \tag{10}$$

Finally, we minimize the sum of $dev2host, host2dev$, and *compCost* under the constraints (3) through (10). It is important to note that with a few changes to the MIP implementation, it can be extended to generate mapping for a system with more than two devices.

6. Experimental results

6.1. Setup

Profiling each benchmark was done on a heterogeneous system consisting of a six-core AMD CPU and an NVIDIA GeForce GTX 460 GPU. Table 2 shows the details of our experimental system setup.

We assessed our algorithms on OpenCL versions of NAS parallel benchmarks [24]. Details of the benchmarks we used in the experiments are given in Table 3. Each benchmark has different characteristics; some have over 50 kernels and others have only few kernels. Each benchmark differs from others in terms of execution latency, characteristics, and size of input

Table 2
Our experimental setup and hardware components.

	CPU	GPU
Architecture	Phenom II X6 1055T	GeForce GTX 460
Clock	2.8 Ghz	1430 Mhz
# of Cores	6	336 Cuda Cores
Memory Size	4 GB	1 GB
OpenCL	AMD APP SDK v2.6	NVIDIA OpenCL SDK 4.0
OS	Ubuntu 10.04 64-bit	

Table 3
The descriptions and problem sizes of benchmarks used in our experiments [10,23].

Benchmark name	Description	Parameters	Class S	Class W
BT	Solves multiple, independent systems of non-diagonally dominant, block-tridiagonal equations.	grid size	$12 \times 12 \times 12$	$24 \times 24 \times 24$
		no. of iterations	60	200
		time step	0.01	0.0008
CG	Computes an approximation to the smallest eigenvalue of a large, sparse, symmetrically positive definite matrix using a conjugate gradient method.	no. of rows	1400	7000
		no. of nonzeros	7	8
		no. of iterations	15	15
		eigenvalue shift	10	12
EP	Evaluates an integral by means of pseudo random trials.	no. of random-number pairs	2^{24}	2^{25}
LU	A regular-sparse, block (5×5) lower and upper triangular system solution.	grid size	$12 \times 12 \times 12$	$33 \times 33 \times 33$
		no. of iterations	50	300
		time step	0.5	0.0015
SP	Solves multiple, independent systems of non-diagonally dominant, scalar, pentadiagonal equations.	grid size	$12 \times 12 \times 12$	$36 \times 36 \times 36$
		no. of iterations	100	400
		time step	0.015	0.0015

Table 4
Distribution of kernels with different approaches.

Bench. Name	Total # of Kernels	Improved Algorithm (IA)		MIP	
		on CPU	on GPU	on CPU	on GPU
BT-S	54	23	31	13	41
BT-W	54	24	30	24	30
CG-S	19	9	10	9	10
CG-W	19	14	5	12	7
EP-S	2	0	2	0	2
EP-W	2	0	2	0	2
LU-S	26	7	19	7	19
LU-W	26	17	9	11	15
SP-S	69	14	55	4	65
SP-W	69	69	0	69	0

data. It is expected to see a different execution behavior from these benchmarks since each benchmark deals with a completely different problem.

We used different problem sizes (i.e. S and W classes of NAS benchmarks) to determine the effect of data size on kernel mapping. As can be seen in Table 4, the tendency of kernels may change with different problem sizes, which is basically due to the characteristics of that particular kernel. For example, it is better to run benchmark SP with W class data set on only CPU, while it is not the case for the same benchmark with S class data set.

6.2. Results

Mapping occurs in two phases: collecting the profiling information and generating the mapping. In the first step, we profiled the benchmarks on the CPU-only and GPU-only systems separately. We also extracted data access patterns. In the second step, our algorithm generated a mapping based on the profiling data. We tested our simulation on five different NAS benchmarks [25] with smaller and larger data sizes. In addition, we have implemented CPU-only and GPU-only mappings to compare our heterogeneous mappings. Note that, CPU-only mapping runs all of the kernels on CPU one after another. For GPU-only mapping, first, required data objects are transferred to GPU, and kernels are executed on GPU. Finally, required data objects are transferred back to CPU. Our experiments include these data copy costs in our calculations.

The results obtained by GA and IA are given in Table 5. The first column indicates the respective name of the executed benchmark. The second and third columns show the results for CPU-only and GPU-only mappings, whereas the fourth and fifth columns show the base and improved algorithms, respectively. Fig. 2 shows the running times normalized with respect to the best single-device execution with different benchmarks. Based on these results, the base algorithm presented in Algorithm 1 improves the best single-device implementation in nine out of 10 benchmarks. The only exception is the SP-S benchmark, where the GPU-only generates better results. As we discussed above, this indicates that the base algorithm got stuck in local minima and generated worse mapping. Compared to the base algorithm, Algorithm 2 generates the same or

Table 5
Execution times of benchmarks with different approaches.

Bench. name	Execution times in seconds				
	CPU-only	GPU-only	Base Alg.	Impr. Alg.	MIP
BT-S	6.969	3.413	2.457	2.452	2.387
BT-W	32.126	12.616	6.297	6.297	6.297
CG-S	0.308	0.433	0.190	0.188	0.188
CG-W	0.521	2.550	0.278	0.263	0.262
EP-S	693.971	45.301	45.301	45.301	45.301
EP-W	370.042	97.082	97.082	97.082	97.082
LU-S	23.898	2.530	1.747	1.687	1.687
LU-W	66.829	18.916	9.755	9.621	9.518
SP-S	1.522	1.002	1.276	0.998	0.980
SP-W	7.961	12.806	7.961	7.961	7.961

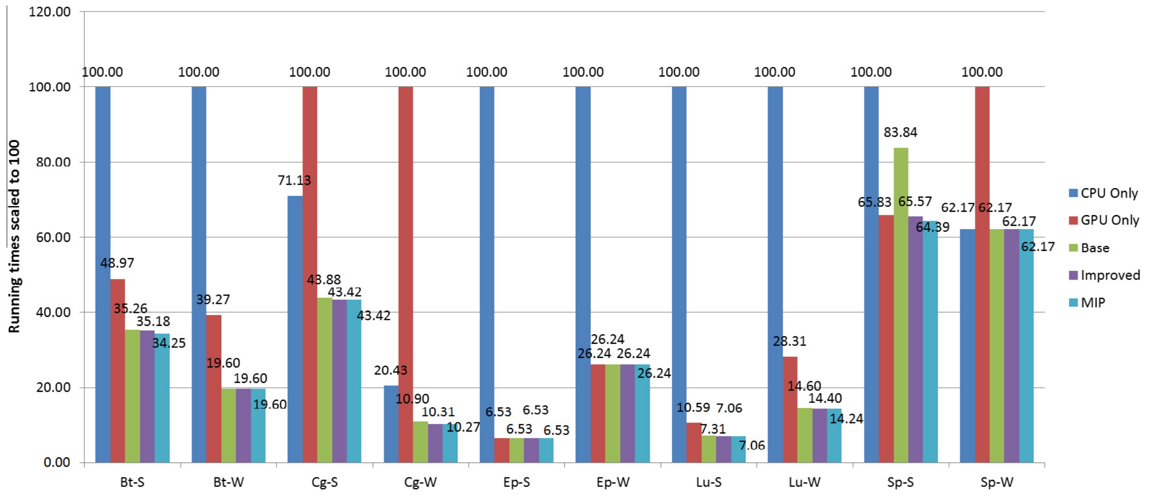

Fig. 2. Comparison of GPU-only, CPU-only, Greedy Algorithm (GA), Improved Algorithm (IA), and MIP implementation mappings.

better mappings for all benchmarks. In some benchmarks, such as EP-S and EP-W, the improved algorithm (IA) generates the same mapping as the GPU-only mapping, since it is faster to run the given kernels of these two benchmarks on a GPU. Similarly, SP-W performs better when executed by CPU-only mapping. The IA generates the same mapping as CPU-only mapping.

The kernel distribution of the benchmarks are presented in Table 4. The first column of the table shows the respective name of the executed benchmark. The second column specifies the total number of kernels each benchmark has. The remaining columns show the number of kernels run on the CPU and the GPU when executed according to the mappings generated by the improved algorithm and by the MIP implementation that represents the optimal mapping, respectively. As can be seen from this table, the majority of the benchmarks take advantage of the heterogeneity available in the system. However, some benchmarks still favor CPU-only and others favor GPU-only mapping due to their processing requirements and data transfer costs.

In order to analyze the behavior of the kernel mapping under different conditions, we have scaled the data transfer costs with some scaling factor ranging from 0.1x to 5x. Note that, our default data transfer costs are obtained through actual measurements for available devices on the system. However, the scaled communication costs are not actual measured values; they are obtained by scaling up/down the already measured default communication latencies. In addition, we used the IA version with our default parameters. In the unscaled case, except for three (EP-S, EP-W, and SP-W), all of the benchmarks use both CPU and GPU resources. While EP-S and EP-W generate the same mapping as GPU-only mapping, SP-W generates the mapping similar to CPU-only. Therefore, we expect that, up to a certain point all of the benchmarks will manage to find a way to utilize both devices. As the data transfer cost increases, all of the benchmarks start to converge to the CPU-only mapping because of the data transfer cost. Respective results are shown in Figs. 3 and 4.

More specifically, Fig. 3 shows the speed up compared to the best single-device mapping (i.e. CPU-only or GPU-only mapping). The EP-S, EP-W, and SP-W benchmarks do not show any improvement, mainly because IA also generates single device mappings for these benchmarks. However, when the data transfer cost is at the lowest case, SP-W generates a better mapping than the CPU-only case. Because, it has speed up more than 1.5 at $0.1x$. Also, for EP-S and EP-W even though data

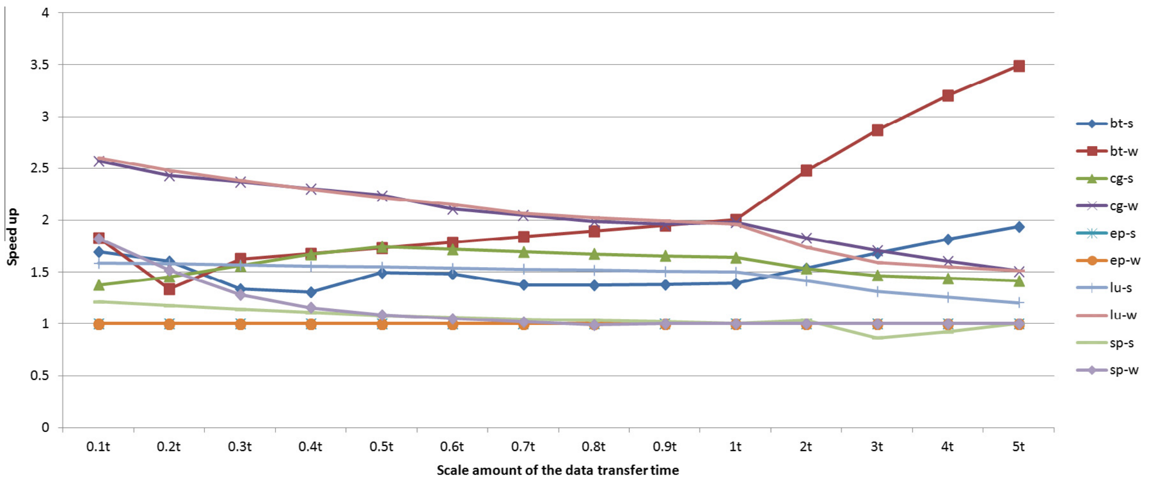

Fig. 3. Speed up of benchmarks normalized with respect to the best single-device execution with different data transfer times. Note that the mapping also changes according to the data transfer times.

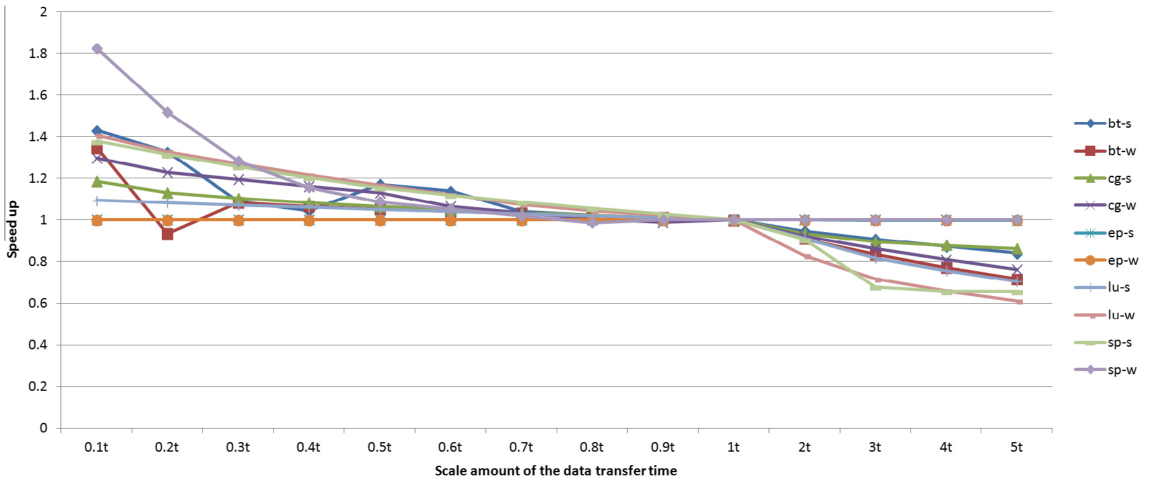

Fig. 4. Speed up of benchmarks normalized with respect to the base case (default data transfer times) with varying data transfer times.

transfer cost is as high as 5x, they persist to generate GPU-only mapping. BT-S and BT-W show interesting behavior, where they continue to increase the speed up. This means their mappings are not as data dependent as their best single device (GPU-only), thus they don't slow down as fast as their best single device and speed up increases. Rest of the benchmarks show the expected behavior.

In Fig. 4, the speed up values for the unscaled mappings and the scaled mappings are compared. When the scale amount is as low as 0.1x all of the mappings are faster than the unscaled versions. This is because of the better utilization of GPU, since the data transfer cost is much lower making GPU an attractive option. Moreover, as data transfer cost gets higher than the default case, the scaled mappings get slower and eventually they saturate at the CPU-only case. This is due to the fact that GPU becomes not profitable, such as in the case of SP-S. In addition, Fig. 4 shows how much data dependent kernel mappings are. The higher change in speed up shows how much the mapping is effected from the change of data transfer cost. Therefore, SP-W is the most data dependent benchmark.

Table 5 shows a comparison of the MIP implementation with GA and IA algorithms. Similarly, Fig. 2 shows the running time of the improved algorithm (IA) and the MIP implementation. Fig. 5 shows the speed up from the best single-device execution of the IA and the MIP formulation. Note that, for some benchmarks, such as BT-S, CG-W, LU-W, and SP-S, the results of MIP formulation differs from IA, because of a change in kernel mapping. For six out of 10 cases, MIP implementation and the IA generate the same mapping. For the other four cases, the results of the MIP implementation is better than the IA, yet the difference is minor. When speed up values are compared, the gap gets even smaller.

Note that, although the MIP implementation finds the optimum mapping, mapping generation usually takes considerably longer. However, as mentioned in Section 4, the proposed algorithm is lightweight but efficient. The MIP solver uses branch

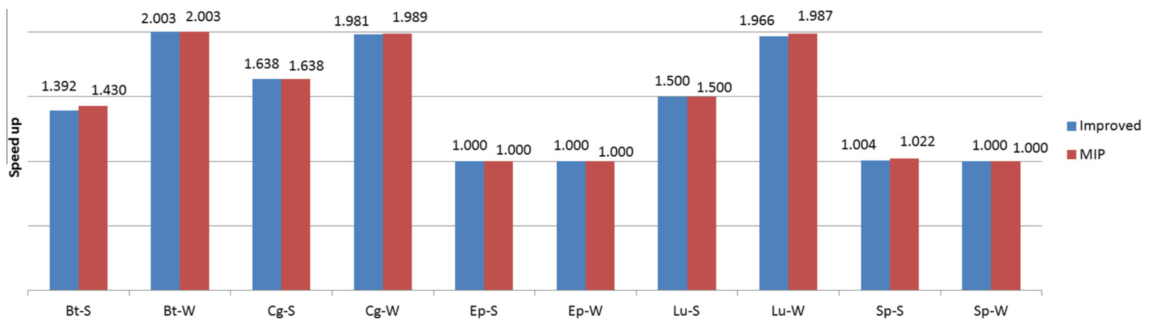

Fig. 5. Speed up of benchmarks normalized with respect to the best CPU-only or GPU-only.

and bound [26] techniques, search recursively through a space obtained from the formulation and progressively set an upper bound for the search tree whenever it finds a better result. Then, the subsets with lower bounds greater than the upper bound of the best case are eliminated. Therefore, MIP does not search the whole recursion tree. However, compared to MIP, our algorithm looks for only the best device for each kernel. In addition, when a kernel is set, the algorithm does not backtrack to a previously set kernel and search a better result for it, therefore it reduces the search space considerably. Hence, the proposed algorithm is much faster than the branch-and-bound techniques used in MIP solvers.

7. Conclusion and future work

Efficient kernel mapping for multi-kernel applications on heterogeneous platforms is important to exploit the provided computational resources and to obtain higher performance. In this paper, we introduce an efficient mapping algorithm for multi-kernel applications. We first employ a greedy approach to select the most suitable device for a specific kernel by using profiling information; then we enhance it to avoid getting stuck in local minima. Our initial experiments show that our approach generates better mappings compared to CPU-only and GPU-only mappings. Moreover, we also formulate the mapping problem and solve it by a Mixed-Integer Programming-based technique, which allowed us to compare mappings generated by the proposed algorithm with optimal mappings. Our approach generates the same mappings with the MIP for six of the ten benchmarks tested. The remaining benchmarks also result in with very close results to MIP. As a future work, we plan to extend this work to support multiple CPUs, GPUs, and possibly other types of accelerators. Moreover, the algorithm can be enhanced using machine-learning-based techniques to predict the execution times of kernels and the data transfer cost for available devices instead of using profiling information obtained beforehand.

Acknowledgments

The authors thank Prof. Kagan Gokbayrak of Bilkent University for his much-appreciated help in developing the MIP formulation, and also for the solver environment. This research is supported in part by TUBITAK grant 112E360, by a grant from NVidia, and by a grant from Turk Telekom under Grant Number 3015-04.

References

[1] R. Gupta, G. De Micheli, Hardware-software cosynthesis for digital systems, IEEE Design Test of Computers 10 (3) (1993) 29–41.
[2] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, Architecture of field-programmable gate arrays, Proceedings of the IEEE 81 (7) (1993) 1013–1029.
[3] C.J. Thompson, S. Hahn, M. Oskin, Using modern graphics architectures for general-purpose computing: a framework and analysis, in: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, Ser. MICRO 35, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 306–317 ([Online]. Available: <http://dl.acm.org/citation.cfm?id=774861.774894>).
[4] M. Daga, A. Aji, W. chun Feng, On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing, in: 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC), july 2011, pp. 141–149.
[5] Khronos group, OpenCL – the open standard for parallel programming of heterogeneous systems. [Online]. Available: http://www.khronos.org/opencl/.
[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for gpus: stream computing on graphics hardware, ACM Transaction on Graphics 23 (3) (2004) 777–786 ([Online]. Available: http://doi.acm.org/10.1145/1015706.1015800).
[7] Ibm, CELL. [Online]. Available: http://www.research.ibm.com/cell/.
[8] AMD, Accelerated parallel programming SDK. [Online]. Available: http://www.amd.com/stream.
[9] NVIDIA, CUDA. [Online]. Available: http://www.nvidia.com/cuda.
[10] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS parallel benchmarks summary and preliminary results, in: Proceedings of the ACM/IEEE Conference on Supercomputing '91, November 1991, pp. 158–165.
[11] C.-K. Luk, S. Hong, H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, Ser. MICRO 42, ACM, New York, NY, USA, 2009, pp. 45–55 ([Online]. Available: http://doi.acm.org/10.1145/1669112.1669121).
[12] D. Grewe, M.F.P. O'Boyle, A static task partitioning approach for heterogeneous systems using opencl, in: Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, Ser. CC'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 286–305 ([Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987259).

[13] T. Scogland, B. Rountree, W. chun Feng, B. De Supinski, "Heterogeneous task scheduling for accelerated openmp," 2012 IEEE 26th International in Parallel Distributed Processing Symposium (IPDPS), (2012) pp. 144–155.
[14] C. Augonnet, S. Thibault, R. Namyst, StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines, INRIA, Rapport de, recherche RR-7240, March 2010.
[15] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198 ([Online]. Available: http://dx.doi.org/10.1002/cpe.1631).
[16] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, Y. Robert, Scheduling strategies for master-slave tasking on heterogeneous processor platforms, IEEE Transactions on Parallel and Distributed Systems 15 (4) (2004) 319–330.
[17] C. Augonnet, R. Namyst, Euro-par 2008 workshops – parallel processing, in: E. César, M. Alexander, A. Streit, J.L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, S. Jha (Eds.), A Unified Runtime System for Heterogeneous Multi-core Architectures, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 174–183.
[18] M. Daga, T. Scogland, W. chun Feng, Architecture-aware mapping and optimization on a 1600-core gpu, in: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), December 2011, pp. 316–323.
[19] AMD, Accelerated parallel processing OpenCL programming guide. [Online]. Available: http://developer.amd.com/sdks/AMDAPPSDK/assets/AMDAcceleratedParallelProcessingOpenCLProgrammingGuide.pdf.
[20] G.L. Nemhauser, L.A. Wolsey, Integer and Combinatorial Optimization, Wiley-Interscience, New York, NY, USA, 1988.
[21] Ibm, ILOG. [Online]. Available: http://www.ibm.com/software/websphere/products/optimization/.
[22] H. Crowder, E.L. Johnson, M. Padberg, Solving large-scale zero-one linear programming problems, Operations Research 31 (5) (1983) 803–834 ([Online]. Available: http://www.jstor.org/stable/170888).
[23] NAS parallel benchmarks problem sizes. [Online]. Available: http://www.nas.nasa.gov/publications/npbproblemsizes.html.
[24] S. Seo, G. Jo, J. Lee, Performance characterization of the nas parallel benchmarks in opencl, in: 2011 IEEE International Symposium on Workload Characterization (IISWC), November 2011, pp. 137–148.
[25] NASA, Nas parallel benchmarks. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html.
[26] A.H. Land, A.G. Doig, An automatic method of solving discrete programming problems, Econometrica 28 (3) (1960) 497–520 ([Online]. Available: http://www.jstor.org/stable/1910129).