2012 41st International Conference on Parallel Processing Workshops

Effective Kernel Mapping for OpenCL Applications in Heterogeneous Platforms

Omer Erdil Albayrak, Ismail Akturk, Ozcan Ozturk
Computer Engineering Department
Bilkent University
Ankara, Turkey
{*oalbayrak, iakturk, ozturk*}*@cs.bilkent.edu.tr*

***Abstract*—Manycore accelerators are being deployed in many systems to improve the processing capabilities. In such systems, application mapping need to be enhanced to maximize the utilization of the underlying architecture. Especially in GPUs mapping becomes critical for multi-kernel applications as kernels may exhibit different characteristics. While some of the kernels run faster on GPU, others may refer to stay in CPU due to the high data transfer overhead. Thus, heterogeneous execution may yield to improved performance compared to executing the application only on CPU or only on GPU. In this paper, we propose a novel profiling-based kernel mapping algorithm to assign each kernel of an application to the proper device to improve the overall performance of an application. We use profiling information of kernels on different devices and generate a map that identifies which kernel should run on where to improve the overall performance of an application. Initial experiments show that our approach can effectively map kernels on CPU and GPU, and outperforms to a CPU-only and GPU-only approach.**

***Keywords*-Heterogeneous, OpenCL, kernel, mapping, GPU**

I. Introduction

Today's high performance and parallel computing systems consist of different types of accelerators such as ASICs [1] (Application Specific Integrated Circuits), FPGAs [2] (Field Programmable Gate Arrays), GPUs [3] (Graphics Processing Unit), APUs [4] (Accelerated Processor Unit). In addition to the variety in accelerators in these systems, applications that are running on these systems have also different processing, memory, communication, and storage needs. Even a single application may exhibit different processing, memory, communication, and storage requirements throughout its execution. Thus, leveraging the provided computational power and tailoring the usage of the resources based on the applications' execution characteristics have an immense importance 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 platforms. It is our goal to generate a kernel mapping that takes these characteristics of each kernel and dependencies into account and leads to improved performance.

In this paper, we propose a novel profiling-based kernel mapping algorithm for multi-kernel applications running on heterogeneous platforms. Our specific contributions are:

- an off-line profiling analysis to extract kernel characteristics of applications.
- a greedy algorithm to select the most suitable device for certain kernel considering its both execution time and data dependencies.
- an improved version of the algorithm to avoid getting stuck in local minima.

The initial results revealed that our approach increases the performance of an application compared to a CPU-only and GPU-only approach. Although our initial experiments are limited to a single type of CPU and GPU, it is possible to extend this framework to support multiple CPUs, GPUs, and other types of accelerators.

The remainder of this paper is organized as follows. The related work on GPUs and kernel execution is given in Section II. Problem definition and introduction the proposed approach are given in Section III. The details of the algorithm and the implementation are given in Section IV. The experimental evaluation is presented in Section V. Finally, the paper is concluded in Section VI.

II. Related Work

OpenCL is an open standard for parallel programming, especially targeting heterogeneous systems [5]. It was initially started as an open alternative to Brook [6], IBM CELL [7], AMD Stream [8], and CUDA [9]. It provides a standard API that can be used on many different architectures regardless of architecture specific characteristics. Therefore it has become widely accepted and supported by major vendors. In this work we also use OpenCL version of the NAS benchmarks [10].

Recent advancement in chip manufacturing technology makes it feasible to produce power efficient and highly parallel processors and accelerators. This, in turn, increases the heterogeneity of the computing platforms and increases the options of where to execute provided applications. To do the best of our knowledge, there are a few studies targeting this critical problem. Especially, Luk et al. proposed Qilin [11]

1530-2016/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPPW.2012.14

which uses a statistical approach to predict the kernel execution times offline. Based on the predicted execution times, they generate a mapping and perform the execution. Also, rather than individual kernel mapping, they partition 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 the sub-operations of the kernels. Also, their statistical regression model is orthogonal to our profiling method. We obtain CPU and GPU execution times in addition to the data transfer times through profiling. It is possible to integrate such model into our system in case profiling is not possible or costly. Furthermore, Grewe and O'Boyle [12] proposed a machine learning-based task mapping. They use a predictor to partition tasks on heterogeneous systems. Their predictor predicts the target device for each task according to the extracted code features that are used in training set of machine learning algorithm. Our decision method can be enhanced with such machine learning-based techniques in the future.

The main difference between the prior works targeting heterogeneous systems and ours is that the latter is a profiling-based kernel mapping algorithm.

III. Problem Definition

A major challenge in a heterogeneous system is the utilization of existing computing devices while obtaining the uttermost performance of an application. This is mainly due to the nature of such systems as they provide computing devices with different characteristics and capabilities. Therefore, the main goal of this work is to utilize these devices by capturing specific characteristics of tasks and making task assignment decisions accordingly.

For this paper, we use a simple heterogeneous system with only a single type of CPU and GPU. However, in reality, a heterogeneous system may consist of multiple types of CPUs, GPUs and APUs from different vendors with different features [13], [14], [15]. It is possible to have both NVIDIA GPU and AMD GPU in the same system. While NVIDIA GPUs are good for simple parallel multi-threaded computations, AMD GPUs support vector operations [16], [17]. Thus, characteristics of a task such as number of vector operations and number of threads running parallel become crucial in the decision of where to run the given application.

The size of data being required by an application is an important issue, since some of the devices may have limited memory space such as GPUs. Therefore, even though an application is developed targeting GPU in mind, it may not be possible to execute it on a certain GPU since data may not fit in the memory of the given GPU.

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

In this paper, we consider both kernel execution times and data transfer overheads obtained through profiling, thereby we map kernels onto devices according to the data dependency requirements. As an alternative, we can extract the kernel characteristics through compiler analysis and employ machine learning-based technique, similar to [12], to predict the kernel execution times and data transfer overheads. This is left as a future work.

IV. Mapping Algorithm

We first analyze each application through executing applications on different devices individually. Specifically, we use CPU and GPU to collect necessary information including input data transfer time, execution time, and output data transfer time. These statistics are collected for each kernel on all devices (i.e. CPU and GPU). We use a greedy algorithm to generate a mapping that minimizes the execution cost of each kernel. However, we realized that minimizing the execution cost of each kernel may not minimize the overall performance of an application due to the complex data dependencies among kernels. In other words, we may get stuck in a local minimum, so we enhanced our algorithm to avoid this problem.

In the base algorithm, we try to minimize the execution time of each kernel by selecting the device that runs the given kernel faster. Eventually, we aim to generate a mapping that improves the performance compared to CPU-only or GPU-only mapping. We can formulate how we obtain the CPU and GPU execution times as follows:

$$\begin{aligned} CPUcost_k &= CPUrunningtime_k + \\ &\quad \sum_{d=1}^{n} DeviceToHost \times InDevice_d \times \\ &\quad Required_{k,d} \times size_d. \end{aligned} \tag{1}$$

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

In the above equations, the first part (in each equation) is indicating 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 that indicates whether kernel k requires data d. Data might already be present on target device and may not be required to be moved in. For this purposed $InDevice_d$ is used and it indicates whether data d is already being in the target device. Similarly, we express the size of data d using $size_d$.

Aforementioned constants are all extracted through profiling and source code analysis except $InDevice_d$. $InDevice_d$ depends on the previous iteration of the algorithm that

Algorithm 1 Base algorithm

```
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 ← true
            k.cost ← cpu_cost
            for all Buffer b ∈ k do
                b.onCpu ← true
            end for
        else
            k.onCpu ← false
            k.cost ← gpu_cost
            for all Buffer b ∈ k do
                b.onCpu ← false
            end for
        end if
        total_cost+ = k.cost
    end forreturn total_cost
end procedure

procedure H2D(Kernel k)
    cost = 0
    for all Buffer b ∈ k do
        if b.onCPU == true then
            cost+ = b.D2H_transfer_cost
        end if
    end for
end procedure

procedure D2H(Kernel k)
    cost = 0
    for all Buffer b ∈ k do
        if b.onCPU == false then
            cost+ = b.H2D_transfer_cost
        end if
    end for
end procedure
```

Algorithm 2 Improved algorithm

```
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 ← k.clone()
        k_clone.onCPU ← true                ▷ set k_clone as if CPU is selected and run BaseAlgorithm to observe the results of CPU selection
        for all Buffer b ∈ k_clone do
            b.onCPU ← true
        end for
        whatif_cpu_cost ← BaseAlgorithm(k)   ▷ run base algorithm starting from k_clone
        k_clone ← k.clone()
        k_clone.onCPU ← false
        for all Buffer b ∈ k_clone do
            b.onCPU ← false
        end for
        whatif_gpu_cost ← BaseAlgorithm(k)
        if (cpu_cost + whatif_cpu_cost) < (gpu_cost + whatif_gpu_cost) then
            k.onCPU ← true
            k.cost ← cpu_cost
            for all Buffer b ∈ k do
                b.onCPU ← true
            end for
        else
            k.onCPU ← false
            k.cost ← gpu_cost
            for all Buffer b ∈ k do
                b.onCPU ← false
            end for
        end if
        total_cost+ = k.cost
    end forreturn total_cost
end procedure
```

accessed the data d. If data was left in the device after this last access, $InDevice_d$ will be 1, otherwise it will be 0. The algorithm assumes all of the data is initially stored in the CPU. Algorithm 1 gives the pseudo code for it.

For each kernel, we compare the respective costs associated with each candidate target and select the lowest one. This greedy algorithm works fine with most of the tested benchmarks. However, in some cases there is threat of getting stuck in local minima due to the complex data dependencies among kernels that can not be considered in the base algorithm. We introduced the improved algorithm (see Algorithm 2) to avoid getting stuck in such local minima. Notice that it has ability to accept worse decisions at *Critical Points*. Table I gives a simple example to show the effect of using the improved algorithm.

When Algorithm 1 is considered for the example given in Table I; the total cost of running the first kernel on CPU is calculated as the summation of execution time on CPU and data movement cost if data is not currently on CPU. Since data is currently on CPU, the total cost of running the first kernel on CPU is 5 + 0 = 5. Similarly, the total cost of running the first kernel on GPU is calculated as the summation of execution time on GPU and data movement cost if data is not currently on GPU. Since data is initially

Kernel Number	CPU Execution Latency	GPU Execution Latency	Data Being 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

Table I
A SIMPLE EXAMPLE TO SHOW THE DIFFERENCE BETWEEN BASE AND IMPROVED ALGORITHMS.

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

Table II
OUR SIMULATION SETUP AND HARDWARE COMPONENTS.

on CPU, the total cost of running the first kernel on GPU is 4 + 2 = 6. Since the total cost of executing the first kernel on CPU is lower, the base algorithm would choose CPU in mapping. Likewise, the second kernel will be mapped to CPU because the total costs are 3 and 4 for CPU and GPU, respectively. Similarly, the third kernel will be mapped to CPU as well because the total costs are 7 and 8 for CPU and GPU, respectively. This will result the total execution time of being 5 + 3 + 7 = 15. However, if the first kernel would run on GPU, although the cost is higher than CPU, it would let the second and third kernel to run on GPU also. Since data being used by the first kernel (i.e. A) is also used by the second and third kernels the total cost would become (4 + 2) + 2 + 6 = 14 that is lower than the CPU-only mapping. For this example, the main problem of the base algorithm was that it gets stuck in local minima at the first kernel. However, improved algorithm allowed to perform the data transfer that increased the cost initially and it caused other kernels to run on also GPU, having lower total execution time compared to the mapping generated by the base algorithm.

As indicated before, we aim to avoid getting stuck in local minima through Algorithm 2. This approach essentially compares the two possible options: (i) it assumes CPU is a better option and performs the remaining decisions according to Algorithm 1, and (ii) it assumes GPU is a better option and performs the remaining decisions according to Algorithm 1. Among the results of (i) and (ii), the best one is selected and that kernel is permanently assigned to that device. This algorithm is applied to every single kernel. In addition, for each kernel algorithm 2 applies algorithm 1 to all the remaining kernels. Therefore, for kernel i algorithm 2 calls algorithm 1, and algorithm 1 runs a loop of $(n - i)$. For kernel $(i + 1)$ algorithm 1 runs a loop of $(n - i - 1)$, and so on. For all the kernels, in total $\frac{(n-1)*n}{2}$ executions are performed, therefore our Improved Algorithm has a complexity of O(n^2).

V. EXPERIMENTAL RESULTS

A. Setup

The profiling of each benchmark was carried on a heterogeneous system consists of a six-core AMD CPU and an NVIDIA GeForce GTX 460 GPU. Table II shows the details about our system.

We have tested our algorithm on OpenCL versions of NAS parallel benchmarks [19]. which are first ported on OpenCL. Details of the benchmarks that we have used in experiments are given in Table III. Each benchmark has different characteristics, while some of them include over 60 kernels, others have only 2 kernels. These kernels are device implementations of independent tasks in these benchmarks. Each kernel has been implemented and tailored for the target device.

We have used different problem sizes to see the effect of data size and other metrics on mapping. As can be seen in Table IV, the tendency of kernels may change with different problem sizes, this is basically due to the characteristics of that particular kernel. For example, benchmark SP on class W has a tendency to run on only CPU, while the kernels of benchmark SP on class S have different tendencies.

B. Results

The mapping is done in two phases, collecting the profiling information and generating the mapping. In the first step, we profile the benchmarks on both CPU-only and GPU-only systems separately. We also extract data access patterns. In the second step, our algorithm generates a mapping based on the profiling data. We tested our simulation on 5 different NAS benchmarks [20] with smaller and larger data sizes.

Benchmark Names	Description	Parameters	Class S	Class W
BT	Solves multiple, independent systems of non diagonally dominant, block tridiagonal equations.	grid size no. of iterations time step	12x12x12 60 0.01	24x24x24 200 0.0008
CG	Computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix by using a conjugate gradient method.	no. of rows no. of nonzero no. of iterations eigenvalue shift	1400 7 15 10	7000 8 15 12
Ep	Evaluates an integral by means of pseudorandom trials.	no. of random-number pairs	2^{24}	2^{25}
LU	A regular-sparse, block (5 x 5) lower and upper triangular system solution.	grid size no. of iterations time step	12x12x12 50 0.5	33x33x33 300 0.0015
SP	Solves multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations.	grid size no. of iterations time step	12x12x12 100 0.015	36x36x36 400 0.0015

Table III
SHOWS THE BENCHMARK DESCRIPTIONS AND PROBLEM SIZES TAKEN FROM [10], [18].

Benchmark Name	Execution Times				Number of Kernels		
	CPU Only	GPU Only	Base Alg	Improved Alg	Total	on CPU	on GPU
BT-S	6,969	3,413	2,457	2,452	54	23	31
BT-W	32,126	12,616	6,297	6,297	54	24	30
CG-S	0,308	0,433	0,19	0,188	19	9	10
CG-W	0,521	2,55	0,278	0,263	19	14	5
EP-S	693,971	45,301	45,301	45,301	2	0	2
EP-W	370,042	97,082	97,082	97,082	2	0	2
LU-S	23,898	2,53	1,747	1,687	26	7	19
LU-W	66,829	18,916	9,755	9,621	26	17	9
SP-S	1,522	1,002	1,276	0,998	69	14	55
SP-W	7,961	12,806	7,961	7,961	69	69	0

Table IV
EXECUTION TIMES AND KERNEL DISTRIBUTIONS OF BENCHMARKS TESTED WITH DIFFERENT APPROACHES.

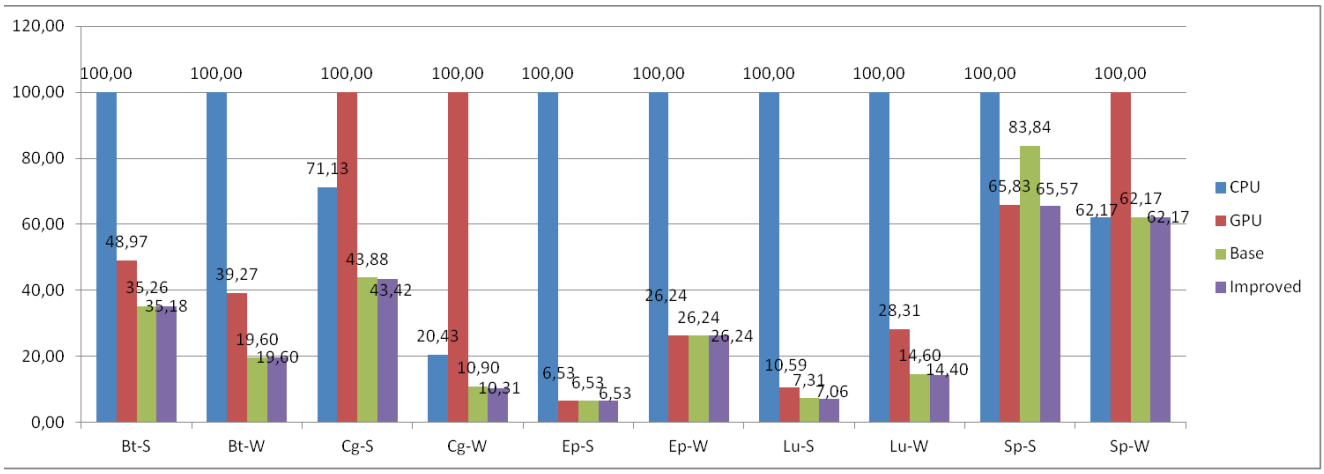

Figure 1. Execution times for GPU-only, CPU-only, Base, and Improved algorithms.

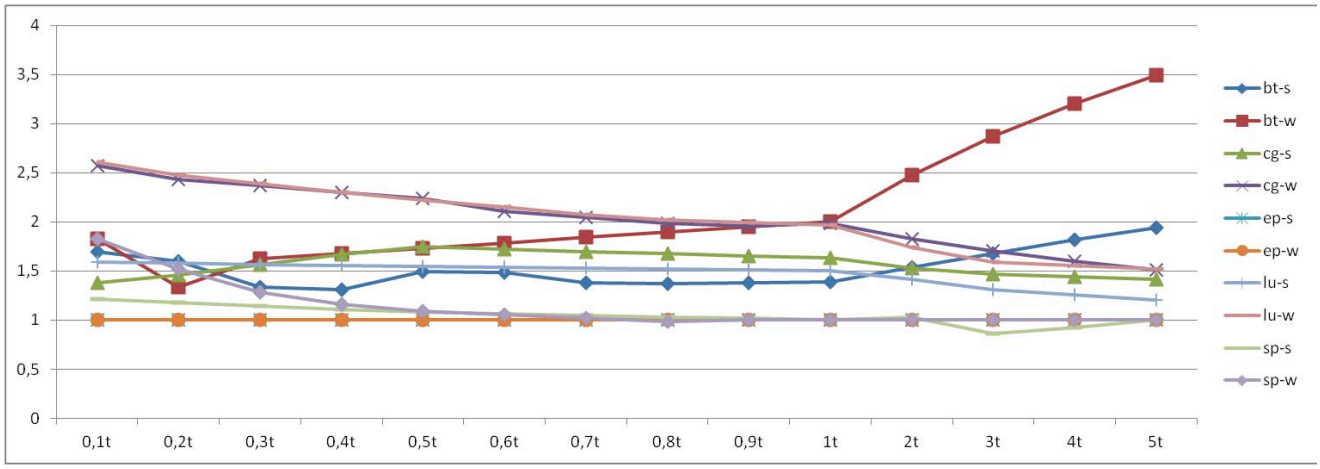

Figure 2. Speed up normalized with respect to the best single device execution with different data transfer times. Note that, mapping is also changing according to the data transfer times.

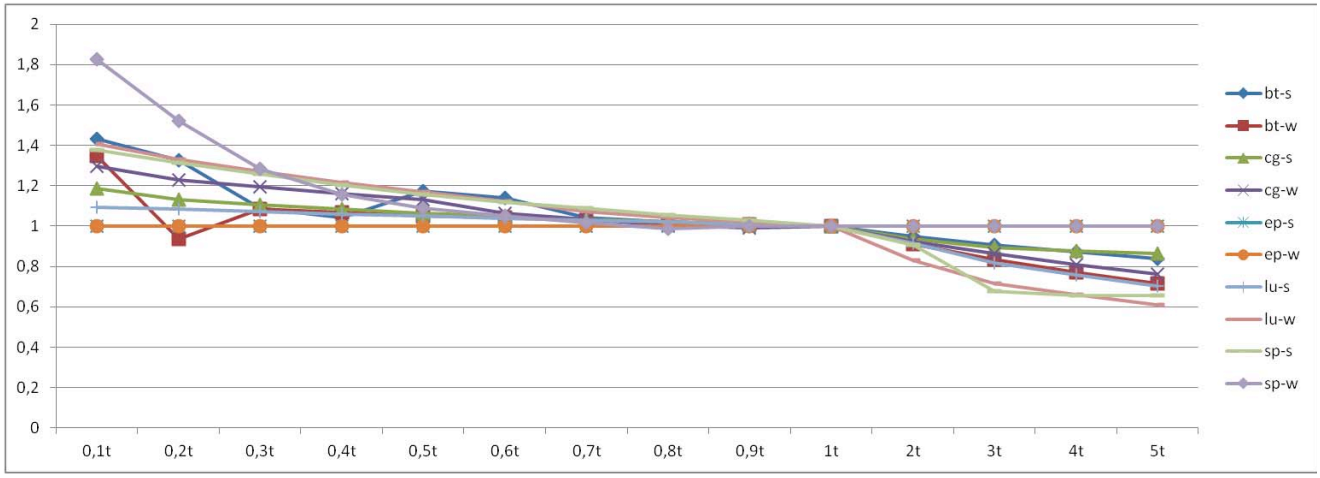

Figure 3. Speed up normalized with respect to the base case (default data transfer times) with varying data transfer times.

Collected statistics and the results obtain our base and improved algorithms are given in Table IV. The second and third columns show the results for CPU-only and GPU-only mappings, whereas the fourth and fifth columns show our base and improved algorithms, respectively. Figure 1 shows the speed up normalized with respect to the best single device execution with different data transfer times. Based on these results, our base algorithm Algorithm 1 improves the best single device implementation in 9 out of 10 benchmarks. The only exception is the SP-S benchmark, where GPU-only generates better results. As discussed above, this can be eliminated through the improved algorithm, Algorithm 2. Improved algorithm outperforms Algorithm 1 in all benchmarks tested as it already compares the result generated by Algorithm 1. In some applications, such as EP-S and EP-W , algorithm gives the same result as GPU-only mapping, since it is faster to run the kernels of these two benchmarks on GPU. Similarly, SP-W performs best when executed on CPU-only mapping, and therefore, our algorithms return the same mapping as the CPU-only mapping.

Last three columns of Table IV give the kernel distributions when executed according to improved algorithm. As can be seen from this table, majority of the applications take advantage of the heterogeneity available in the system. However, some of the benchmarks still favor CPU-only and some others favor GPU-only mapping due to their processing requirements and data dependencies.

Except the three benchmarks (EP-S, EP-W and SP-W) mentioned earlier, all the benchmarks use both CPU and GPU resources. Figure 2 and 3 shows the results for the same algorithm with the same data but with scaling the data transfer times between CPU and GPU, and vice versa. In these figures, the effects of data transfer overhead on total

kernel execution time can be seen. Note that, the x-axis shows the normalized data transfer times with respect to the original data transfer time. For example, the first point assumes that it takes 10x less amount of time to transfer the data to the device and vice versa.

In addition, Figure 2 shows the speed up compared to the best CPU-only or GPU-only mapping. EP-S, EP-W and SP-W do not show any improvement. This is mainly due to the fact that our algorithm also generates single device mappings for these benchmarks.

It is expected to see that when data transfer time is increased too much, all the kernels will tend to run on CPU as the cost of running on GPU will outweigh the CPU. Therefore, after a certain threshold, data transfer times will dominate and our approach will only generate CPU-only mappings.

In Figure 3, the speed up decreases continuously as the data transfer cost is increased. This is because of the fact that when GPU data transfer costs are really low, it is profitable to run these benchmarks on the GPU with lower execution times. However, as the data transfer cost increases, GPU is becoming less attractive.

VI. CONCLUSION AND FUTURE WORK

Effective kernel mapping for multi-kernel applications on heterogeneous platforms has significant importance to exploit the provided hardware resources and obtain higher performance. In this paper, we introduce an effective algorithm to map the kernels of multi-kernel applications written in OpenCL. We first use greedy approach to select the most suitable device for a specific kernel by using profiling information and enhanced it to avoid getting stuck in local minima. Our initial experiments show that our approach generates better mappings compared to a CPU-only and GPU-only mapping. Although we used a single type of CPU and GPU, we plan to extend this work to support multiple CPUs, GPUs, and other accelerators. We also would like to implement an Integer Linear Programming-based (ILP) technique to compare our results with the optimal mapping. Moreover, we plan to enhance the extraction of kernel characteristics phase of our algorithm in a way that it can generate a mapping on the fly. The algorithm will be enhanced by using the machine learning-based techniques to predict the execution times of kernels and data transfer cost for available devices instead of using profiling information.

REFERENCES

[1] R. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *Design Test of Computers, IEEE*, vol. 10, no. 3, pp. 29 –41, sept. 1993.

[2] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013 –1029, july 1993.

[3] C. J. Thompson, S. Hahn, and 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. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 306–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=774861.774894

[4] M. Daga, A. Aji, and W. chun Feng, "On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, 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, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, aug. 2004. [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, and S. Weeratunga, "The NAS parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, nov. 1991, pp. 158 –165.

[11] C.-K. Luk, S. Hong, and 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. New York, NY, USA: ACM, 2009, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669121

[12] D. Grewe and 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. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 286–305. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987259

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[14] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Scheduling strategies for master-slave tasking on heterogeneous processor platforms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 4, pp. 319 – 330, april 2004.

[15] C. Augonnet and R. Namyst, "Euro-par 2008 workshops - parallel processing," E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. A Unified Runtime System for Heterogeneous Multi-core Architectures, pp. 174–183.

[16] M. Daga, T. Scogland, and W. chun Feng, "Architecture-aware mapping and optimization on a 1600-core gpu," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, dec. 2011, pp. 316 –323.

[17] "AMD, Accelerated Parallel Processing OpenCL Programming Guide." [Online]. Available: http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[18] "NAS parallel benchmarks problem sizes." [Online]. Available: http://www.nas.nasa.gov/publications/npb_problem_sizes.html

[19] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, nov. 2011, pp. 137 –148.

[20] "NASA, NAS parallel benchmarks." [Online]. Available: http://www.nas.nasa.gov/publications/npb.html