2021 International Conference on Computer, Information and Telecommunication Systems (CITS) | 978-1-6654-4913-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/CITS52676.2021.9618353

Workload Distribution on Heterogeneous Platforms

Mahmoud Alasmar
Electrical and Electronics Engineering Dep.
Middle East Technical University
Ankara, Turkey
https://orcid.org/0000-0002-7862-7271

Cüneyt F. Bazlamaçcı
Computer Engineering Dep.
Izmir Institute of Technology
Izmir, Turkey
cuneytbazlamacci@iyte.edu.tr

***Abstract*—This paper targets the problem of finding an efficient distribution of a computational task on a heterogeneous computing platform. The heterogeneity of the processing elements arise due to differences in computation speed and memory capacity of the processors. We first consider using a discrete functional performance model that integrates processing speed and capacity of processing elements and then develop a mathematical model and propose a heuristic mapping algorithm for distributing a given total workload of size N on p processing elements such that the total computation time is minimized. Computational results show that the proposed method provides a significant improvement in reducing the computation time in comparison to equal distribution approach.**

***Index Terms*—Heterogeneous Computing Platform, High Performance Computing, Workload Distribution, Task assignment, Functional Performance Model**

I. Introduction

Many modern computer platforms are built in the form of a heterogeneous architecture, in which a multi-core CPU system is integrated with accelerators composed of GPUs, FPGAs, ASICs, etc. [1]–[3] in order to meet high performance computing applications' requirements. The heterogeneity arises from having processing elements of different speed, memory size, memory access time and communication cost [4]. One of the significant challenges in the field of parallel computing is to optimize the performance of a parallel application on a heterogeneous platform in order to utilize the processing elements in the best possible way while satisfying certain conflicting criteria.

Typical parallel software applications consist of computing routines known as *kernels* that vary in terms of their individual characteristics and requirements [5]. Depending on the available computing resources, the execution time and memory usage of such kernels may differ considerably.

This work aims to propose a compile-time efficient mapping algorithm for parallel applications that will run on a heterogeneous platform such that the total execution time of the overall application is minimized. The success of the algorithm is measured in terms of the speed-up and cost reduction. The contributions of the paper can be summarized as follows:

- Functional performance model is used for the purpose of profiling the tasks on different processing elements.
- A set of workload types based on the profiling step are generated.
- A priority value is used for each processing element based on the performance model encountered. Highest priority indicates the fastest processing element.
- A priority based recursive tree search algorithm is proposed for distributing the existing total workload among the processing elements using a generated pool of tasks (kernels).

The rest of the paper is organized as follows: The related work is discussed in section II. The problem definition and our proposed method are presented in sections III and IV, respectively. Computational experiments, evaluation results and discussions are given in section V. Section VI concludes and points out some possible future work.

II. Related Work

Workload distribution and task assignment problems have been investigated widely in the field of parallel and distributed computing systems research. Different approaches have been proposed either to tackle a certain aspect of the problem or to adapt to a certain characteristic of the environment and the associated hardware. Proposed algorithms and solutions can be classified as static or dynamic [6]. Static algorithms require prior information about the parallel program and the platform. This information is usually assumed to be an accurate performance model in order to predict the future execution of the application. The downside of such algorithms is that the size of the workload is required to be fixed during run-time and hence they become ineffective if the load changes over time. Dynamic algorithms, on the other hand, are more efficient in case the workload changes during run-time and they tend to move fine-grained tasks between processors so that balancing can also be achieved (e.g. work-stealing algorithm). Dynamic algorithms do not require prior information about the application, however, they may suffer from communication overhead due to frequent migration of the data.

Balanced distribution is one of the methods that have been developed and used widely for the purpose of distributing the workload on heterogeneous environment [7]. The methods relies on distributing the workload such that all processing elements runs and executes their associated tasks using same amount of time. Such approaches aim to avoid having under-loaded or over-loaded processing elements. Another technique used for distributing the workload is the one proposed by [8], which is a branch and bound based method adapted for finding a near optimum solution for the workload distribution problem.

There exist different models that are used to represent the performance of a processing element while executing a workload. The simplest model is to use a positive constant to characterize the application and the computing device; this number could be task computation time, average execution time, normalized processor speed, etc. This approach is independent of the workload size and known as the constant performance model (CPM) [9]. Another but more advanced model used for characterizing the performance of a processor is the functional performance model (FPM) [8], in which the speed of the processor is modeled as a continuous function of the problem size.

A. Advanced Performance Model

Within the scope of this work, functional performance model (FPM) is adapted for the purpose of profiling and characterizing the performance of the processing elements. However, instead of using a continuous function of the problem size, a discrete model is proposed. In order to create the discrete model, a minimal entity of the workload or the problem size need to be defined and this minimal workload, in the rest of the paper, will be named as $kernel$ and denoted as λ. Other discrete workload sizes are chosen as multiples of λ. Figure 1 shows the functional performance model for both ARM A15 and DSP C66 for matrix multiplication application measured on a TI platform. The application is modeled as two matrices A and B multiplied with each other yielding matrix C. Within this setup matrix B is considered as shared data, that is the whole matrix is accessible to all other processors. While matrix A is the candidate for partitioning, each processing element will be assumed to be able to access only a certain partition of the matrix. The horizontal axis in Figure 1 labeled as "A partitions" shows how matrix A is being partitioned, the partitions being multiples of λ. The minimal workload λ in this case is chosen as a single row of matrix A. The range of the workload size goes from one row to all rows of the matrix. Figure 2 illustrates the partitioning of matrix A. The vertical access in Figure 1 is the speed measured in terms of MAC (Multiply-Accumulate) operations per second. The speed of the i-th processing element is evaluated according to equation 1:

$$S_i(x) = \frac{x}{T_i(x)} \tag{1}$$

- x: Problem size (number of MAC operations)
- $T_i(x)$: Average execution time of size x work on i-th processing element

It is observed in Figure 1 that FPM monotonically increases within $[\lambda, n\lambda]$ where n is a positive integer, and stays almost constant after $n\lambda$. For example, $n = 8$ for ARM A15 case.

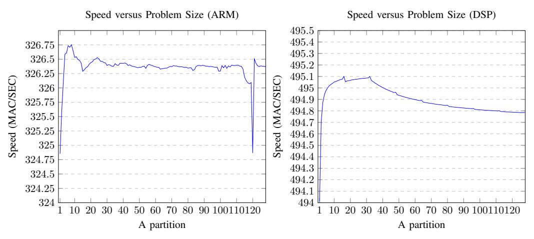


Fig. 1. Advanced performance model for matrix multiplication on ARM A15 and DSP C66.

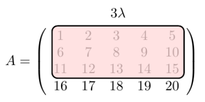


Fig. 2. Example partitioning of matrix A for 3λ

III. Problem Definition

Given a total workload of size N, our aim is to distribute this workload among a set of processing elements such that the total execution time is minimized. We assume that the total workload can be partitioned into smaller sub-tasks, which we call workload types, of different sizes that can run in parallel. The following are the definitions of problem variables:

P: set of processing elements
T: set of tasks
W: set of generated workload types
(w_i, a_{ij}, D_i): 3-tuple associated with task t_i
w_i: workload type of task t_i
a_{ij}: average execution time of task t_i on p_j
D_i: partition of the workload assigned to t_i

$$u_{ij} = \begin{cases} 1 & \text{if } t_i \text{ is assigned to } p_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Then the problem is to minimize f

$$f = \min \ (\max_{j \in P} \sum_{i \in T} a_{ij} \cdot u_{ij} \ ) \tag{3}$$

subject to

$$\sum_{j \in P} u_{ij} = 1, i \in T \tag{4}$$

$$\sum_{i \in T} \sum_{j \in P} w_i \cdot u_{ij} = N \tag{5}$$

$$\bigcap_{i \in T} D_i = \emptyset \tag{6}$$

Constraint 4 states that each task must be assigned to a single processing element. Constraint 5 ensures that the total workload of all assigned tasks sum up to the original total workload size while constraint 6 ensures that partitions assigned to tasks do not overlap. u_{ij} is a binary utilization value and the cost function f, aimed to be minimized, is indeed the total computation time of the workload based on a certain assignment.

IV. Proposed Method

A. Workload Type Generation

Our approach requires a certain number of task types (workload types) be generated based on apriori timing measurements. W_j is a set of workload types associated with the $j-th$ processing element and it is made up of entities where each represents a type (size) of workload. Set W_j is created using the functional performance model (FPM) of p_j. The set contains all workloads within the range starting from the minimal workload size up to and including workload having the maximum speed. This set will then be used later during the process of dispatching the tasks. When a task is created as a solution to a partition of the whole task, and dispatched to a processor, the task's associated type is to be chosen from set W_j.

B. Task Distribution Algorithm

In the rest of the paper, the following example is to be used to demonstrate and clarify the basics of our approach. Consider $N = 16$, $W = \{2, 4, 8\}$, $P = \{0, 1\}$ and

$$A = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 6 \end{bmatrix} \tag{7}$$

where rows and columns of the matrix correspond to workload types and processing elements, respectively and a_{ij} is the average execution time of workload type w_i on p_j.

First step is to order the workload types for each processing element based on their speed where speed is found using equation 1. In the given simple example, let's assume that the task categories are ordered as w_2, w_1, w_0 for both processing elements. A computed priority value is then assigned to each processing element using the following equation:

$$priority_j = \frac{\max_{i \in W} S_{ij}}{\sum_{j \in P} \max_{i \in W} S_{ij}} \times N \tag{8}$$

where S_{ij} is the speed of workload type w_i on processing element p_j. In the given example case, $N = 16$ and the total number of workload types and processing elements are 3 and 2, respectively. Using equation 8, one can find the priority of p_0 as 9.6 and p_1 as 6.4.

Figure 3 shows a snapshot of the search process, the value at each node represents the remaining size of the unassigned workload, each edge represents a creation and an assignment of a task to a processing element. The terminals of the tree are feasible solutions, the path from the root to a terminal is the assignment associated with the solution. The value in each rectangular box is the cost of the corresponding solution. For this particular example, the root node has a value 16, which is equal to the total size of the workload.

Our search algorithm picks a processing element and a workload type and creates a task that is assigned to the selected processing element. In this step, the processing element with

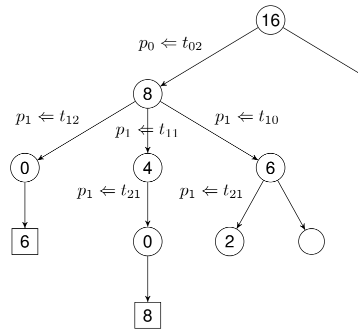


Fig. 3. Snapshot for the searching process of the algorithm.

the highest priority and the workload type with the highest speed having capacity fit for the remaining workload are selected. After assigning the generated task to a processing element, the priority of that processing element is decremented. In figure 3, the notation $p_j \Leftarrow t_{ik}$ means that task number i and of workload type w_k is created and assigned to $j-th$ processing element. For our example, the algorithm creates and assigns t_{02} to p_0, the remaining workload size is made 8, priority of p_0 is modified by subtracting from it the assigned workload size hence new priority being 1.6 and the cost accumulated at that point being 4 (average execution time a_{02}).

Since p_1 now has the highest priority, our algorithm creates and assigns another task t_{12} with workload 8 to p_1. Then the remaining workload size reduces to 0 and the algorithm reaches a terminal for which the cost is 6. When a terminal is encountered, the algorithm backtracks and starts investigating other possible paths. Eliminating or cutting a branch in the search tree is also considered in order to reduce the search space. The case where the algorithm performs the following assignment, $p_0 \Leftarrow t_{02}$, $p_1 \Leftarrow t_{10}$, $p_1 \Leftarrow t_{21}$ as shown in figure 3, makes the remaining workload size 2, however, the current cost becomes 7 which is larger than the current global cost, hence the algorithm may stop expanding that branch.

The pseudo-code of the described tree search is presented in Algorithm 1. The following are global variables and objects that needs to be initialized before starting the algorithm:

- $RemSize$: remaining workload size, initially N.
- P: set of processing elements, $InsertProc(pe)$ adds new processing element to the set.
- pe: a processing element with three attributes; ID, set of workload types, initial priority.
- sol: initially empty solution object implemented as a linked list of nodes, each containing information about creating and assigning a task to a certain processing element.
- $fsol$: initially empty feasible solution corresponding to minimum cost found so far.
- $Cost$: Global cost that is updated every time a feasible solution is reached, initially a large value.
- $BackTrack$: a variable to control and limit the number of times the search tree is pruned.

The algorithm works as follows: i) first check the remaining size of the workload ii) if the remaining size is larger than zero then choose a processing element from set P, create a task and assign it to the chosen processing element iii) otherwise a feasible solution or a terminal is reached hence update the global cost and store the current solution.

The function $GetProcessor()$ picks the processing element with the highest priority from P, if highest priority is negative, priority values of all processing elements in P gets reset using $ResetPriority()$. $GetProcessor()$ returns a processing element object pe. The workload types associated with a processing element can be accessed using $GetWorkloadTypes()$, which returns a set of workload types suitable to be created as a new task. The workload types for each processing element (note that processing elements are heterogeneous) are ordered in descending order according to the speed of each type. The algorithm iterates over workload types starting with the highest speed. Once a workload type is chosen and the remaining size fits into this type of workload, $InsertNode(w.size, pe)$ is invoked so that a new task can be created with size $w.size$ and assigned to pe.

When a task assignment is done, a node carrying this information will be added to the list of sol object. When a new node is added to the list, the sol object automatically updates its cost and compares current cost with the global one and if the former is larger, then the algorithm removes the newly added node from the list using $RemoveTopNode()$, meaning that the current branch of the solution tree is pruned. If the current cost is less than the global one then the algorithm proceeds to expand the current branch of the solution recursively after updating the remaining workload size as long as the $BackTrack$ is larger than zero.

Algorithm 1: Proposed Tree Search Algorithm

```
main():
    $RemSize = N$
    $P = \emptyset$
    for $i \leftarrow 0$ to $k - 1$ do
        $pe(i, WorkloadTypes, Priority)$
        $P.InsertProc(pe)$
    end
    $sol = null$
    $fsol = null$
    $Cost = \infty$
    $BackTrack = LargeValue$
    $TS(RemSize)$

GetProcessor():
    $x, PE = max(P)$
    if $x < 0$ then
        $P.ResetPriority()$
        $x, PE = max(P)$
    end
    return $PE$;

TS($RemSize$):
    if $RemSize > 0$ then
        $pe = GetProcessor()$
        $wt = pe.GetWorkloadTypes()$
        for $w \in wt$ do
            if $w.Size \leq RemSize$ then
                $sol.InsertNode(w.Size, pe)$
                if $sol.GetCost() > Cost$ then
                    $sol.RemoveTopNode()$
                else
                    if $BackTrack > 0$ then
                        $TS(RemSize - w.Size)$
                        $BackTrack - -$
                    end
                    $sol.RemoveTopNode()$
                end
            end
        end
    else
        $Cost = sol.GetCost()$
        $fsol = sol$
    end
```

V. Computational Analysis

In order to verify our approach, three use case applications are deployed: Matrix Multiplication, Fast Fourier Transformation (FFT) and Convolution.

Reference Methods: The performance of our method is compared with three different methods for workload distribution: Trivial equal distribution (in which the workload is distributed equally), balanced distribution and HPOPTA algorithm proposed by [8].

Test Platform: 66AK2H12, which consists of ARM-Cortex A15 and DSP C66, is used as a multi-core platform in our practical experiments. ARM and DSP cores runs at 1.4 GHz and 1.22 GHz, respectively. Data is placed on a shared memory of size 4 MBytes. The algorithm is also tested on a simulation framework consisting of Intel Haswell multicore CPU, Nvidia K40c GPU, and Intel Xeon Phi 3120P as three different processing elements. HPOPTA algorithm in [8] used FFTW profiles on these three elements as shared data in our experiments.

A. Experiments and the implementation model

Two different experiments are conducted in order to observe (i) the execution time for different problem sizes (this test is conducted as a simulation where the workload is distributed among three processing elements) (ii) the speed up (this test is conducted on 66AK2H12 platform by varying the number of DSP cores while the problem size is kept constant).

B. Measurements and results

To measure the speed up, both sequential execution time on a single processing element (in this case ARM core) and parallel execution time are measured. Each case is repeated for 100 times and the average of execution time is recorded.

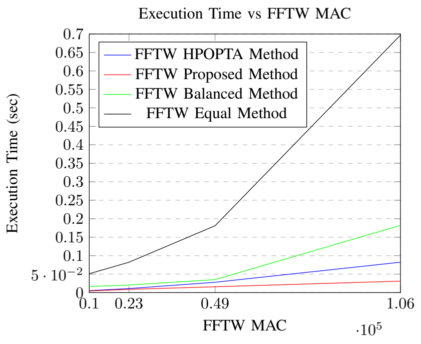
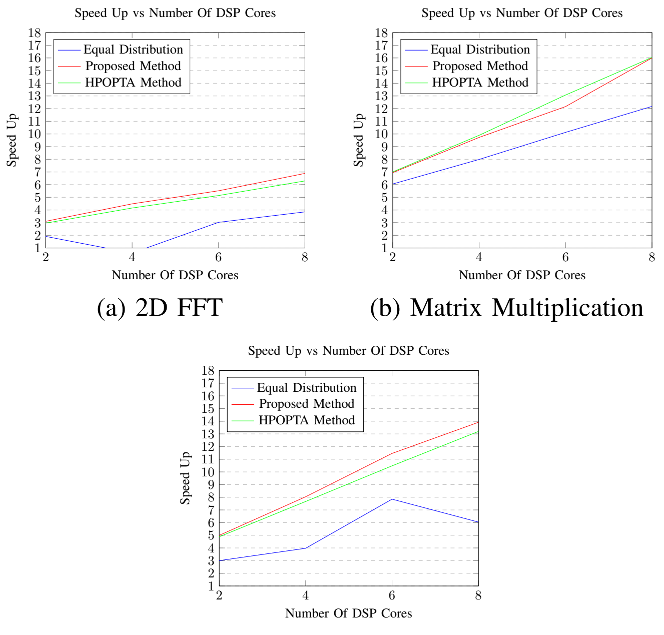


Fig. 4. Execution time wrt FFTW MAC



(a) 2D FFT (b) Matrix Multiplication

(c) 2D FIR

Fig. 5. Speed up wrt number of cores

C. Discussion

It is observed that our approach yields better performance in comparison to equal distribution and balanced distribution, and as good as the HPOPTA method. Figure 4 illustrates the execution time of simulating parallel FFT on CPU, GPU and Xion Phi, FFTW library is used for creating the performance model of the processing elements. During the simulation, the problem size is measured in terms of Multiply-Accumalate (MAC). In all tested cases it has been shown that the proposed method yields the smallest execution time in comparison to other methods. Another observation in figure 4 is that the effect of using the proposed method becomes more significant for problems of large size.

Figure 5 shows the speed up for three different functions (FFT, Matrix Multiplication and Convolution) as the number of DSP core is varied on TI 66AK2H12 platform. The speed up for the proposed method is compared with HPOPTA and equal distribution methods. It is observed that our proposal has better performance in terms of speed up in comparison to equal distribution as expected while yielding a speed up that is as good as HPOPTA. The results show that both the proposed and HPOPTA methods can exploit the heterogeneity unlike the equal distribution case. Although our proposal yields a comparable performance comparable with HPOPTA, the novelty in our approach is that, unlike balanced distribution and HPOPTA methods, it does not require an execution profile that spans whole problem sizes. Indeed, our method is designed in such a way that using only a profile for small problem sizes, the algorithm should still be able to generate a near optimum solution for larger problem sizes.

VI. Conclusion and Future Work

This work introduced a new method for distributing a given total workload on multi-core heterogeneous devices. Our work has focused initially on applications that can be modeled in single program multiple data (SPMD) form. Although the proposed approach is shown to be promising with an preliminary evaluation, further evaluations are needed and planned as follows:

Since we are choosing in all our selections max entries from ordered sets, our solution is not guaranteed to be optimal. The mathematical model we developed can be solved exactly either by i) an exhaustive search (for small problems) or ii) a suitably designed branch and bound algorithm combining appropriate upper bounding and lower bounding mechanisms and branching strategy (for moderate size problems). For the original problem, the complexity of finding an optimum solution is in the order of $O(n^p)$, where n is the number of workload types and p is the number of processors, hence using heuristics is reasonable for large problems. As future work, the complexity and running time of the algorithm will be analyzed and compared with other methods in literature.

References

[1] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[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, 1993.

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

[4] M. Cierniak, M. J. Zaki, and W. Li, "Compile-time scheduling algorithms for a heterogeneous network of workstations," *The Computer Journal*, vol. 40, no. 6, pp. 356–372, 1997.

[5] O. E. Albayrak, I. Akturk, and O. Ozturk, "Effective kernel mapping for opencl applications in heterogeneous platforms," in *41st International Conference on Parallel Processing Workshops*, 2012, pp. 81–88.

[6] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 1–35, Jul. 2015.

[7] A. Becker, G. Zheng, and L. V. Kalé, *Load Balancing, Distributed Memory*. Boston, MA: Springer US, 2011, pp. 1043–1051. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4504

[8] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.

[9] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 520–535, 2001.