PERFORMANCE AND ENERGY OPTIMIZATION OF HETEROGENEOUS CPU-GPU SYSTEMS FOR EMBEDDED APPLICATIONS

By

Abdullah Siddiqui

Bachelor of Engineering

Ryerson University, 2015

A thesis

presented to Ryerson University

in partial fulfilment of the

requirements for the degree of

Master of Applied Science

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2018

Author's Declaration

Performance and Energy Optimization of Heterogeneous CPU-GPU Systems for Embedded applications

Abdullah Siddiqui

Master of Applied Science

Electrical and Computer Engineering
Ryerson University

2018

Abstract

One of the most critical steps of embedded systems design is Hardware-Software partitioning. It is characterized by distributing the components of an application between hardware and software such that the user defined system constraints are satisfied. Heterogeneous computing platforms consisting of CPUs and GPUs have tremendous potential for enhancing the performance of embedded applications. The challenge of application partitioning for CPU-GPU mapping is much greater on such platforms due to their unique and diverse characteristics. In this thesis, an optimization algorithm is devised and presented for partitioning and mapping computational tasks on CPU-GPU platforms while keeping a check on the power consumption. Our methodology also uses parallelism in applications and their tasks by utilizing the architectural capabilities of the GPU. The optimization algorithm was tested with a MJPEG decoder, several benchmarks and synthetic graphs.

Acknowledgment

I would like to first of all thank the Almighty Allah for giving me the inspiration and the perseverance to carry out the work detailed in this thesis document.

I am very grateful to my supervisor, Dr. Gul Khan, for giving me the opportunity to work for him. I was initially fascinated with the subject of embedded systems but thanks to him, that initial fascination has grown into a deep passion for this subject. I always had his encouragement and support throughout the duration of my master’s studies and could not have asked for a better supervisor than him to oversee my research efforts. My friend and colleague, Muhammad Obaidullah, generously shared his knowledge of object-oriented programming and design with me. It was invaluable for my research and I shall always be grateful to him for that. I am grateful to Mr. Jason Naughton for providing timely help with the CMC system and uncomplainingly resetting it each time after it crashed. I also want to thank Mr. Jochem Bonarius for giving me feedback for the implementation of the MJPEG decoder. I am also thankful to Ryerson University and the Canadian Microsystems Corporation (CMC) for providing funding and equipment for my research.

Last but not the least, my mother, father and brother Rashed have been a constant source of support and encouragement for me throughout my life. They always pushed me to do my best and were consoling and supportive during my periods of frustration and difficulties. None of my accomplishments would have been possible without the unrelenting dedication and sacrifices of my parents since my childhood up to now. I dedicate this thesis to them.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Traditional processors are easy to program and can process any application. However, they are only capable of performing sequential instruction execution and need considerable amount of time to process general applications. Embedded systems especially do not have powerful cores because of power consumption, area, costs, etc. Consequently, their micro-processors are even slower than those in computers at executing sequential code. Dedicated hardware units are faster than software in executing the same functions.

Hardware/Software co-design emerged in the 1990's as a methodology for combining the advantages of fast but expensive dedicated hardware units with slow but inexpensive software-based solutions. It is a process of meeting system-level goals by exploiting the trade-offs involving hardware and software throughout their concurrent design. It mainly targets embedded systems with strict constraints on performance, area and energy expenditure. In current embedded systems, software has become the rigid component while hardware is evolving at a much faster rate. Moreover, increased emphasis on better performance, *green computing* and Moore's law are driving the evolution of computer architectures [1] and [2].

The ambitions of companies such as ARM and Xilinx confirm that the embedded industry is headed towards heterogeneous multicore systems. ITRS Road Map 2007 showed that the paradigm of heterogeneous massively parallel computing should be adopted to meet the demands of next generation's applications. A heterogeneous massively parallel computing platform consists of numerous specialized cores around multiple general-purpose processors which are not independent either from a programming or an implementation perspective. In these systems, homogeneous many core architectures have been compounded with task specific and specialized cores called accelerators. On such systems, control intensive and general-purpose software is executed sequentially whereas data intensive parts of the application are offloaded to corresponding tailored architectures [2].

A typical modern embedded system has a workload consisting of a mix of tasks which have varied performance and power consumption characteristics depending on the processing element where they will be mapped. This makes them ideal for heterogeneous architectures. Platforms consisting of CPUs, GPUs and FPGAs are becoming increasingly prominent, partly due to the energy, performance and flexibility requirements of modern applications [3].

GPUs, especially, have prompted researchers to explore novel methods of exploiting their immense computational power. General purpose GPUs are now being used in the automation industry, robotics, medicine, etc. The speed-up achieved by the GPUs depends not only on the system hardware but also on how the functionality accesses the resources (shared or global memory, number of registers per thread, etc.). Using GPUs can considerably enhance performance at a fraction of the power needed by conventional CPUs [4].

The process of Hardware/Software Co-design generally consists of the following stages [5]:

- System specification: Co-design starts with a description of the system functionality along with their timing constraints. A specification language is used to keep away from the bias towards hardware or software.
- Functional simulation: The system is then simulated to verify the correctness of the input data sets. Data related to completion times, communication and concurrency is acquired during this step and constitutes the system’s dynamic behavior.
- Co-Analysis: System information is extracted in this step through static analysis of the system specification.
- Co-Estimation: This step generates the timing information for every step of the specification for every processor class belonging to a set. Other characteristics such as area consumption are also accounted for in this step.
- Load and bandwidth estimation: The profiling data acquired in the Co-Estimation phase is used to estimate the load for the system functionality execution along with the timing constraints. The bandwidth is estimated after combining the timing and communication data of the application.

- Design Space Exploration: This phase consists of two major tasks: Hardware/Software Partitioning with the architecture definition and timing co-simulation. The definition of the architecture along with HW/SW partitioning perform design space exploration to determine the optimal mapping of system functionality on hardware and software to meet user-defined constraints. Timing co-simulation is performed to satisfy the timing constraints of the system architecture and mapping which were already generated. The purpose of DSE is to minimize the design space exploration time and generate high-quality solutions.
- Implementation: Once satisfactory results for the mapping of tasks and the target architecture are obtained, system implementation can be initiated. Software components of the system will be implemented in C while hardware components will be transformed into HDL code which will be synthesizable.

General purpose computing on the GPU (GPGPU) offers the facility of load sharing between the CPU and the GPU in compute-intensive applications such as matrix multiplication, collision detection, ray tracing and so on. Hardware/Software Co-design and GPGPU are analogous to each other at a higher level of abstraction. Hence, the strategies involved in Co-design can also be applied in GPGPU. Design Space Exploration plays a crucial role in Hardware/Software Co-design and consequently also in GPGPU.

Design Space Exploration in GPGPU has its challenges. It is not optimal in all the application scenarios to move everything to the GPU. Moreover, offloading certain functions to the GPU may be of little benefit for the application if those functions do not follow the data-parallel nature of the GPU. In certain applications such as 2-D FFT, it was shown that the application performance can be optimized by appropriately distributing 1-D FFT calculations between the CPU and the GPU. Hence, it is possible to partition both functionality and data and this can be effectively utilized [6].

1.2 Problem Definition

Partitioning an application between the hardware and software is a NP-hard problem. This means there is no known algorithm for this problem which runs efficiently for all possible inputs and always gives the correct answer. Furthermore, after an application is distributed between the hardware and software, deciding the optimal implementation of the application parts on hardware is also difficult to do manually.

We present a co-synthesis tool which tackles these challenges. The hardware of choice in our study is GPU. The application is first specified in a modular fashion prior to being fed to the proposed co-synthesis tool. Our tool partitions the application between the software and the GPU. In addition to that, it leverages the features of both the application and the GPU to arrive at an optimal implementation of the application parts assigned to the GPU in terms of power consumption and execution time.

1.3 Thesis Organization

Chapter 2 gives a detailed account of each of the stages of hardware/software co-design discussed this chapter. It contains an extensive review of the work done so far in the different aspects of hardware/software co-design. Additionally, it contains a thorough assessment of the work done in the domain of CPU-GPU systems to optimize and improve various important parameters such as power consumption and execution times of the target applications. The characteristics of important heterogeneous programming frameworks such as OpenCL and CUDA are also discussed in this chapter. A detailed description of the proposed design space exploration methodology is presented in Chapter 3. Chapter 4 depicts the results produced upon applying the proposed methodology to optimize real-world and synthetic applications. The conclusion and suggestions for future work are presented in chapter 5.

Chapter 2

Overview of Hardware/Software Co-Design and CPU-GPU computing

2.1 Hardware/Software Co-Design

2.1.1 Introduction

A great number of research methods addressed design techniques for software and hardware and not much was known about the joint design of hardware and software. Hardware/software Co-design emerged to address the convergence of problems in integrated system design. Microprocessors were regularly used at that point, but microprocessor-based systems were mainly board-level systems. A class of designers was solely responsible for integrating standard hardware components on the board. It become apparent by the 1990s that microprocessor based system design would become an important discipline for system designers as well. When embedded processors were small and executed only a few hundred lines of code, manual design techniques were sufficient to satisfy functional and performance goals. Modern embedded systems demanded thousands of lines of code and needed to execute at high speeds to meet performance deadlines. Large 32-bit processors were already being used in board-level designs and it was apparent that Moore's law would lead to chips that would include both a CPU and other hardware [7].

Motivated by these developments, researchers began to develop introductory approaches for the design of embedded software running on multiple CPUs and these early efforts formed the basis of the codesign methodology. The predictability of embedded system design was sought to be developed through synthesis methods which indicate if constraints of power, performance, etc.

have been met or not. These methods should also allow the designer to incrementally refine a design over multiple levels of abstraction and create a first working implementation [8].

2.1.2 Early works in Hardware/Software Co-Design

The first major step towards co-design was taken by Prakash and Parker when they formulated the co-synthesis problem as a mixed integer-linear program which could determine a multiprocessor topology while scheduling and allocating tasks on the target architecture [9]. The objective function is composed of functions which can be linearized such as system cost and performance. Constraints can be expressed as timing variables and binary variables. The non-linear constraints are linearized and converted to a MILP formulation [9,10]. This instigated active research on automatically partitioning task graphs into hardware and software [10].

In Hardware-software partitioning, a given functional specification is mapped on a hardware platform which initially consisted of a single CPU and an ASIC where both parts communicate via a bus (Fig. 2.1). As a result, the target platform was already fixed and it was required only for determining which functionalities would be implemented on the ASIC.

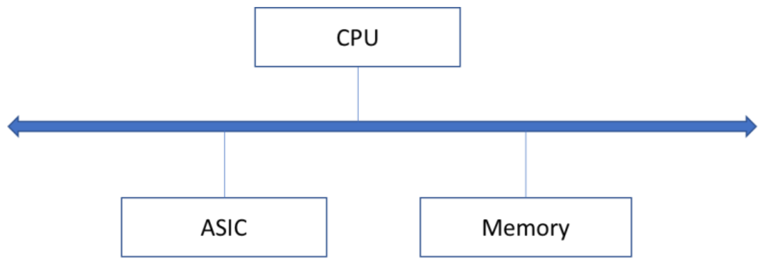


Fig 2.1: Target architecture for Hardware/Software Partitioning.

Gupta et al. proposed a partitioning methodology that starts with a hardware-only solution and migrates tasks to software while satisfying performance constraints and reducing system cost [11]. The COSYMA system took the opposite approach and started with a software-only partition and then transferred tasks to hardware to satisfy performance constraints while minimizing the cost of the resulting hardware [12]. Both approaches were based on the assumption that the implementation was single-threaded and the CPU and ASIC worked independently without interacting with one another [11 and 12].

Distributed task allocation algorithms are based on the assumption that processes have been allocated beforehand and the topology of the distributed computing engine is given. Stone presented the first algorithm for allocating processes to processors on distributed systems [13]. Shen and Tsai used a graph matching heuristic for allocating processes [14]. In their work, inter-process communication was minimized and the load was balanced.

2.1.3 Traditional System-level design flow

The design process of embedded systems varies considerably with the complexity of the application. However, the study of a typical design flow shows that hardware and software components have common abstractions. This principle was utilized for efficient hardware/software codesign of systems [1].

Fig. 2.2 depicts a traditional design flow of embedded systems.

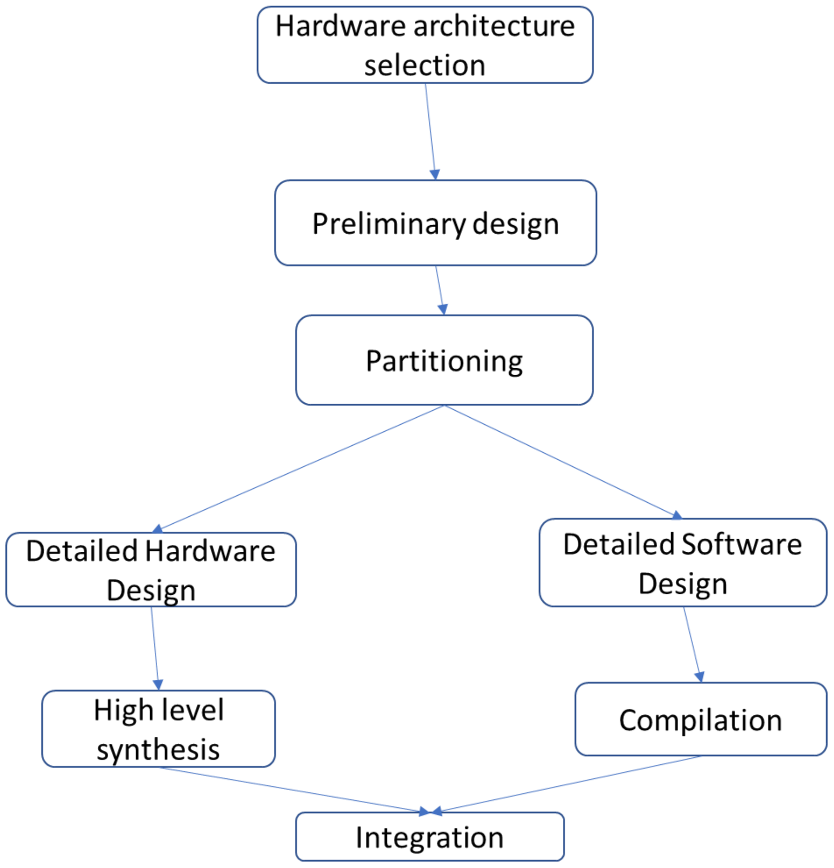


Fig.2.2: Typical design flow.

Hardware architecture selection and partitioning are performed later than in a traditional design. This allows the design decisions to be based on more complete and accurate information [15].

Wolf argued that embedded systems design starts with the creation of a specification [7]. A system specification should include functional requirements (operations to be performed by the system) as well as non-functional (speed, power and manufacturing cost) requirements. Embedded system design methodologies should be able to support incomplete specifications or changes to the spec during the design [8]. Axelson outlined the following steps in the co-design of real-time systems [15]:

- System behavioral description: Executable specification of what a system is supposed to do.
- Hardware Architecture Selection: The description of the hardware components to be used and how they should be connected.
- Task scheduler design: The scheduler determines how the different computational resources of the hardware architecture are shared between the tasks of the behavioral description.
- Hardware/Software partitioning: This step determines how the tasks will be distributed among different processing elements.

2.1.4 Models of Representation

Formal representations are important to meet the goals of performance, cost and reliability. An embedded system can generally be represented with two models: the system requirements model and a system architectural model. The requirements model can be a data-flow/control-flow diagram with response time specifications. The architecture model includes an architecture flow diagram (which allocates functional elements of the requirements model to physical units in the architecture) and an interconnection diagram. A model is different from the language used to specify a system. A model of computation can be based on several languages and there are some description languages which manage different models.

A variety of models have been developed and are used to represent heterogeneous systems. A computational model should comprehend concurrency, sequential behavior and communication methods. Some models are meant for data-intensive systems and others are more suitable for control-oriented systems. There are others which combine data and control to represent systems [16].

Data-flow graph

Data-flow graphs are popular in modeling multimedia and similar data-oriented systems. Computationally intensive systems can be denoted by directed graphs where the nodes describe computations and the arcs represent the order in which computations are performed. Fig. 2.3 represents a typical data-flow graph.

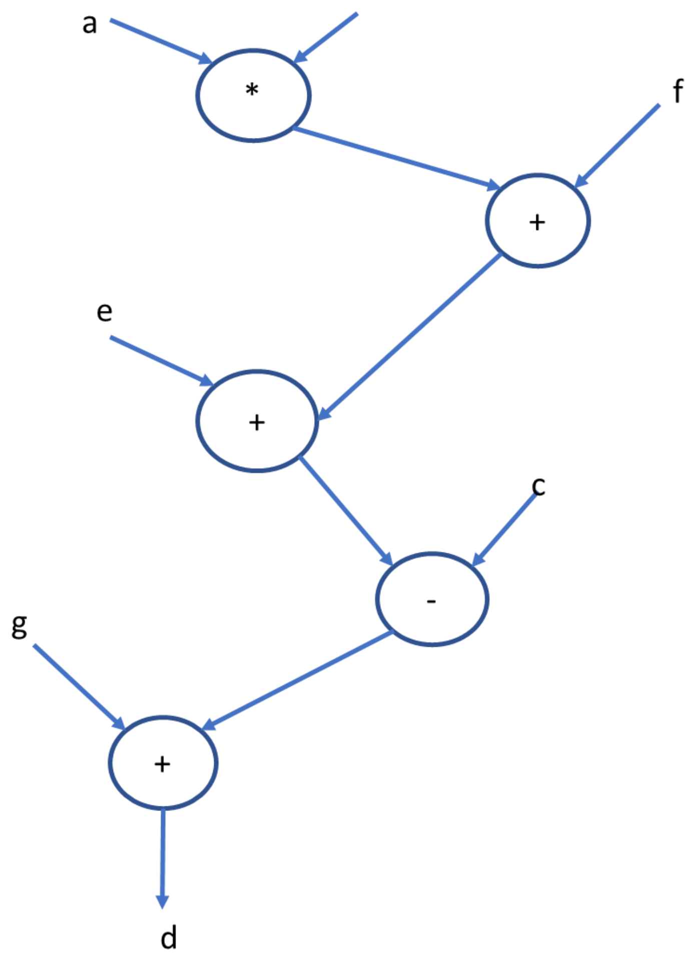


Fig. 2.3: A typical data-flow graph

Petri-net

Petri-nets are composed of a set of places, a set of transitions, an input function which maps transitions to places and an output function which maps from transitions to places. Fig. 2.4 represents a simple petri-net.

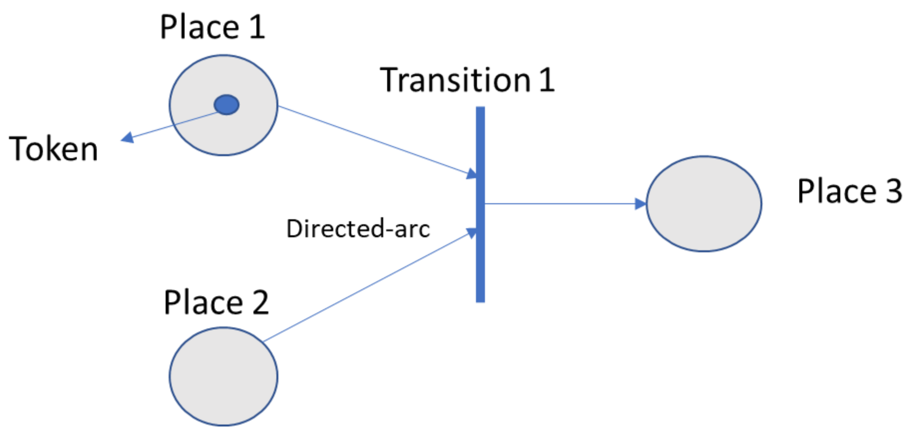


Fig. 2.4: A simple petri-net

Finite State Machine

A finite state machine consists of a set of states, a set of transitions between states and a set of actions associated with these states and transitions [16]. Fig. 2.5 depicts a simple finite state machine. The nodes represent the state of the application and the arrows/arcs represent transitions from one state to another.

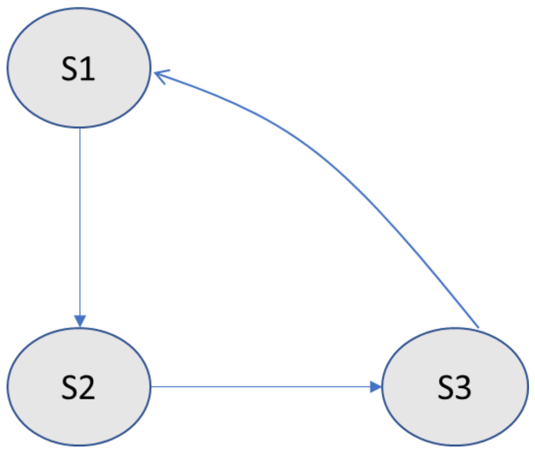


Fig. 2.5: A simple FSM

2.1.5 Limitations of traditional design flow and the need for alternate design strategies

The architectural assumptions of earlier works are no longer valid, given the complex SoC architectures that have emerged recently. Most codesign strategies relied on the premise that a single executable description of the system can be compiled into either silicon or CPU code. This created the possibility of uniformly describing the behaviour of a system, which will be implemented in a combination of application-specific hardware and software. The description is partitioned, with the aid of fully/semi-automatic tools, into separate hardware and software parts. The result produced is passed to a high-level synthesis tool (a compiler) for producing the final implementation. The development of such automated tools is however outside the scope of this thesis.

There are many forms of hardware and software and differentiating between them is no longer straight forward. For example, a Field-Programmable Gate Array (FPGA) is a hardware circuit which can be reconfigured to a user-specified netlist of digital gates. The program for an FPGA is a bitstream and is used to configure the netlist topology. A bitstream of an FPGA can even be used to implement a softcore processor that can execute C programs. This has convinced scientists and engineers that the creation of software requires intimate familiarity with the hardware.

The demand for energy efficiency and performance has instigated a preference for hardware (parallel, fixed) implementation over CPU implementation (sequential, flexible). The most popular argument for dedicated hardware has been greater performance gains i.e. more work done in minimal time. Higher performance can be obtained by decreasing the flexibility of the system and specializing the architecture. However, it must be noted that almost every electronic device carries a battery, which in turn imposes energy constraints on the device. To improve energy efficiency, consumer devices are being implemented by using a combination of embedded CPU and dedicated hardware components. Moreover, parallel computer architectures, despite their utility, cannot cover all the applications. As a result, software designers cannot disregard the computer architecture during design and development.

There is considerable evidence to support the claims for the inclusion of on-chip software as well. Simulations need to be performed extensively to test the designs prior to the implementation phase. Since software bugs are easier to address than hardware, there is a tendency for increasing the amount of software. Compact design schedules require multiple tasks to be performed at the same time. Consequently, hardware and software need to be developed concurrently. Software development begins as soon as the characteristics of the hardware platform are decided and much before a hardware prototype becomes available [15, 17].

Finding the correct balance among all the objectives is a daunting challenge. Adding hardware to a software solution might increase the performance of the overall application but it will also increase the total number of resources. Optimization techniques involving minimization of hardware costs and/or power under performance constraints are too restrictive in their approach and not feasible because each product has different and varied objectives to fulfill. Hence, it is critical for a designer to not only be able to implement the criteria into a CAD environment for codesign, but also to simultaneously account for multiple objectives [10, 17].

The following section surveys the techniques devised to date to address the challenges described here.

2.2 Design Space Exploration strategies and techniques

The trade-offs mentioned in the previous section ought to be made in the context of the system design space. For any given application, there are many possible system design solutions. The design space is given by a set of all possible permutations of allocations, mappings and schedules. When system parameters collectively satisfy both functional and non-functional constraints related to cost, performance, temperature, etc., a feasible solution is said to have been found [10]. System-level design space exploration is the task of exploring the set of feasible implementations efficiently and finding not only one but many optimal solutions.

Typical single-objective based techniques, which were proposed in the early days of co-design needed to optimize only a single objective function. Therefore, if a system solution that satisfies multiple objectives has to be found, a weighting function combining all the objectives has to be employed. Moreover, weights to every objective should be properly assigned according to the preferences of the designer. In order to perform a thorough exploration and allowing the designer to determine which trade-offs are achievable, multi-objective exploration techniques are recommended. The number of objectives and evaluation functions need to be chosen by the user.

Any renowned search/optimization technique could be applied to obtain feasible implementation candidates like randomized search techniques, techniques reliant on iterative search improvement or even exact techniques such as ILP [10]. Wolf presented a heuristic algorithm which simultaneously synthesized the hardware and software architectures for systems consisting of several heterogeneous processors and communication links to satisfy the performance goals at a minimal cost. Wolf's algorithm was unique because unlike the previous algorithms, it did not assume that the hardware topology was provided beforehand. It could synthesize the computer system with an arbitrary topology. In this algorithm, processes are first allocated to PEs such that the specifications deadlines are satisfied and then scheduled to determine communication rates. They are then re-allocated to PEs to minimize PE costs. Reallocation occurs again to minimize the communication costs. This is followed by the allocation of communication channels and then devices either to PEs or communication channels. Scheduling occurs during several of these steps to test the feasibility of the designs. The algorithm refines the design from feasible to one with a minimal cost [18].

A graph based approach was presented in 1997 which offered the capability of modeling heterogeneous target architectures and their interconnections. A specification graph consists of a task graph with data dependencies and a target architecture which specifies the available hardware components and their communication facilities. Edges between tasks and resources in this model denote mapping possibilities and contain further details such as power, cost, etc. Communication tasks were used in the place of edges to represent timing delays caused by transferring data (via communication links) between data-dependent tasks. This work specified the problem of allocation (architecture selection) and binding (mapping the application on the selected architecture) as an optimization problem and used an evolutionary algorithm to solve it. Furthermore, it was proved through this work that the problem of allocation and binding is NP-complete. This technique motivated the development and application of sophisticated algorithms for finding feasible solutions for target architectures [10].

2.2.1 Genetic Algorithm

Genetic Algorithm (GA) is one of the most popular optimization algorithms for process assignment and scheduling. It maintains a set of solutions and these solutions evolve in each generation during the execution of the GA. Solution are improved in every generation by randomized changes and interchange of information between solutions. “Chromosome”, in the context of GA, refers to a candidate solution for a problem. The “Mutation” operation randomly picks a location in the solution array and swaps it with a new value. The other operator, “Crossover”, takes two solutions and picks out two locations in these solutions and interchanges parts of solutions between them. After the crossover operation, a new pair of solutions is evolved. This algorithm is capable of handling complex problems which consists of multiple NP-hard problems [19].

Dick and Jha devised a multi-objective optimization strategy, called MOGAC (Multi-Objective Genetic Algorithm for Co-synthesis) which partitions and schedules functional specifications which consist of multiple periodic task graphs. It employs a multi-objective GA based technique to optimize the cost and power and targets real-time heterogeneous architectures and meets hard real-time constraints. MOGAC accepts a library, which specifies the performance and power consumption of each task for every available PE. The edges of task graphs are assigned to communication links for which the criteria of communication time and power consumption are taken into account [20].

Chakraverty et al. proposed a stochastic model called ESCORTS (Evolutionary Scheme for Co-synthesis of Real-Time Systems) for hardware software co-synthesis which treats timing parameters as random variables. A hierarchical GA was used to optimize the task allocations and the resources needed for building the architecture. A stochastic scheduling algorithm was used to generate task schedules. The stand-out feature of this model is that the effective processor pools are evolved, cloned and re-evolved to generate better solutions whereas ineffective processor selections are discarded. Communication buses and edge allocations are similarly optimized and evolved by the cloning process and ultimately low-cost and high-performance architectures are generated [21].

2.2.2 Simulated Annealing

Simulated Annealing (SA) is also a widely used optimization algorithm for process assignment and scheduling problems. In SA, solution changes occur by random “moves” (moving tasks from hardware to software and vice-versa). Eles et al. used SA and Tabu Search for automatic hardware software partitioning of system level specifications. They sought to minimize communication costs between the hardware and software partitions and enhance parallelism in the eventual performance. They developed a cost function which guided partitioning towards the objective. It was formulated as a graph partitioning problem and implemented SA and Tabu Search on it and found Tabu Search to be superior in performance [22].

Advancements in system design space exploration have also been implemented in MPSoC targets. Xie and Hung proposed a Multiprocessor System-on-Chip (MPSoC) task allocation and scheduling algorithm which relied on temperature. Temperature is a critical factor and affects performance, reliability and cost of the embedded system. Their cosynthesis algorithm takes a task graph and technology library and generates a MPSoC architecture with minimum cost and effectively maps tasks on the generated architecture. It meets the deadline requirements of an application and reduces the average temperatures. Thermal aware heuristics are used and a temperature aware floor planning is employed to reduce the peak temperature and achieve a thermally aware distribution while meeting the real-time constraints. They investigated both power and thermal aware approaches for task allocation and scheduling. It was observed that the thermal aware approach outperformed the power aware schemes in reducing the average and maximal temperatures [10, 23].

2.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population based stochastic optimization technique which was developed by Eberhart and Kennedy. Inspired by the fish schools and bird flocks, PSO is very similar to GA in its approach. But unlike GA, it has no genetic operators such as crossover and mutation. The initial input for PSO is a population of random solutions and optimal solutions are looked for by updating generations. In PSO, potential solutions are referred to as “particles”. These particles fly through the solution space by following the current optimum particles [24].

Bhattacharya et al. compared the effectiveness of the PSO algorithm with Integer Linear Programming (ILP), Ant Colony Optimization (ACO) and Genetic Algorithm (GA) by using some standard benchmarks. They found that ILP worked best for graphs with as many as a thousand nodes and produced optimal solutions while PSO produced sub-optimal solutions on an average. In terms of run-time requirements, the performance of PSO was superior than the other tested algorithms [25].

Li et al. sought to devise a solution for the time-consuming nature of PSO to enhance the safety and performance of embedded industrial applications. They proposed a HW/SW co-design architecture for implementing PSO on a FPGA. The Particle Updating Accelerator Module was implemented on the hardware. The other available processors operate in conjunction with the particle updating accelerator module to perform fitness evaluation and speed up the execution performance of PSO. Their design is flexible and various applications can be optimized without redesigning the hardware [26].

Today's exploration tools need to be flexible as any organization using such tools may have to contend with different optimization objectives for different products to be developed. For evaluation, the exploration model could be annotated and used to write user-specific cost functions. Evaluation functions to assess the various user-defined objectives have to be developed [10].

2.3 Co-simulation and Co-verification

After a co-synthesis mechanism has produced an architecture as well as mapped and scheduled the system activities, concurrent hardware and software synthesis is performed. These design steps transform the system specification, which has been split between hardware and software, into a physical implementation. System parts which should be mapped on hardware are designed using high-level, logic and layout synthesis tools. System parts that have been mapped on software programmable processors (CPUs, DSPs) are compiled into assembler and machine code, using either standard or specialized compilers and assemblers. Concurrent hardware and software synthesis offers the advantage of co-simulating both system parts, with the aim of finding errors in the design as early as possible to avoid expensive re-designs.

Once the hardware and software components have been implemented, they should be separately tested, integrated and tested again. Both hardware (manufacturing testing) and software testing (system validation) are necessary and both types of elements should be executed together to ensure the system meets its specifications [27].

2.4 Heterogeneous Computing: CPU-GPU platforms

There are several challenges involved in the collaborative computing of CPUs and GPUs due to their vastly different architectures. To optimize both performance and energy efficiency, the characteristics of both processors should be taken into account. Conventional optimization techniques which focus solely on either the CPU or the GPU are no longer effective in heterogeneous systems. As a result, novel techniques are required to harness the potential and promise of heterogeneous computing.

At the architecture level, a chip which has both the CPU and GPU integrated on the same chip is called an Accelerated Processing Unit (APU) or Single-Chip Heterogeneous Processor (SCHP) (Fig. 2.6). This is also known as a fused or integrated system. APUs are characterized by significantly shorter communication paths and a unified address space across the CPU and the GPU. Communication occurs via the shared memory in APUs [28].

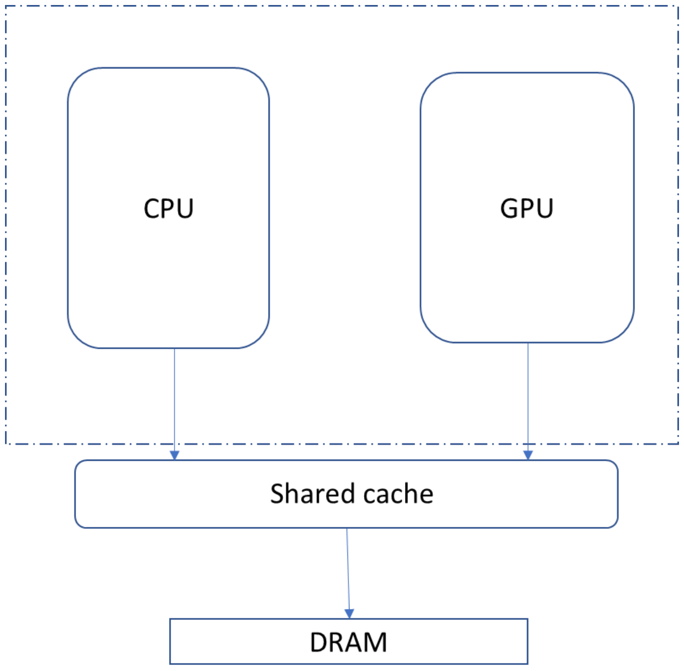


Fig. 2.6: Fused GPU + CPU

A conventional discrete system, on the other hand, has a CPU and GPU on different boards, connected with a Peripheral Component Interconnect Express (PCIe) bus as shown in Figure 2.7 [28]. Communication occurs across the PCI-express connection.

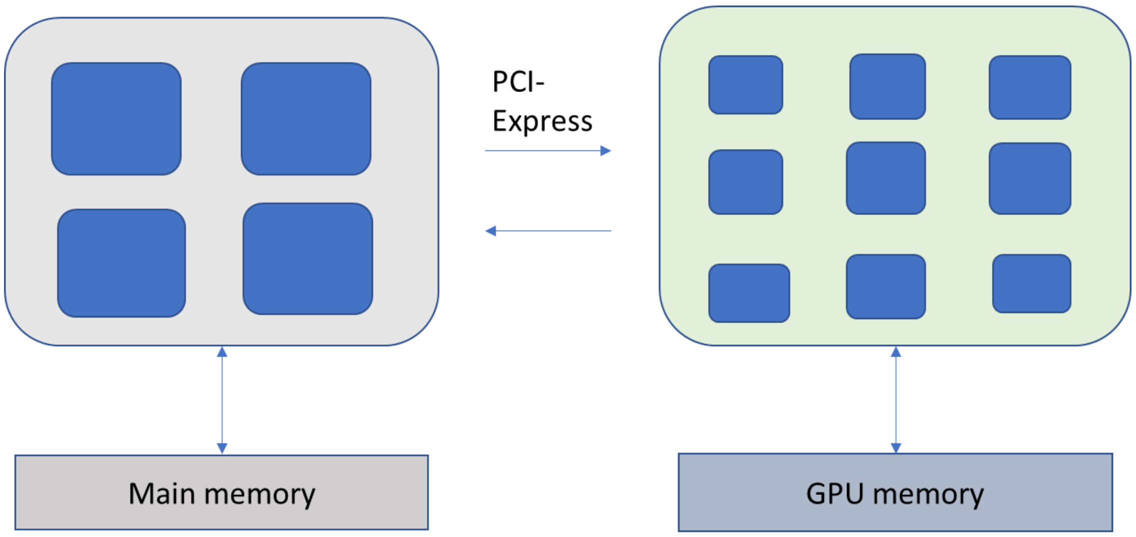


Figure 2.7: System overview of the CPU-GPU architecture.

Both discrete and fused systems have their advantages and disadvantages. Fused systems provide better bandwidth and latency. They bypass the PCI-e bus which is a critical bottleneck for discrete systems. However, fused systems can only serve specific categories of applications such as consumer electronics in the mobile and embedded sector.

Modern multicore CPUs use anywhere from 4 up to 64 cores, which are normally out-of-order, multi-instruction issue cores. These cores also run at a high frequency and utilize large caches to minimize the latency during the execution of a single thread. This makes them suitable for latency-critical applications. GPUs, on the other hand, have a larger number of cores, which are in-order cores that share their control unit. They operate at a lower frequency and have smaller caches. Hence, GPUs are more suitable for applications that have critical throughput requirements. Hence, a system consisting of both a CPU and a GPU can provide high performance for a variety of applications and usage scenarios than single CPU or GPU systems [28].

2.4.1 Energy efficiency and Resource Utilization

Even though GPUs have the potential to speed-up the execution of many applications, they do so at the expense of significant power and energy consumption. Moreover, with the increasing number of installations of GPUs in supercomputers and data centers, their electricity costs bring power and environmental concerns. Dynamic Voltage and Frequency Scaling (DVFS) is a technique used for saving the energy of modern computers. It allows the processors to achieve better energy efficiency with proper voltage and frequency settings. While CPU DVFS techniques are mature, GPU DVS techniques are still in their infancy. Additionally, the CPU techniques cannot be applied to the GPU because the frequency domains of the GPU make it more complex [29].

Wang and Chu proposed a power estimation model based on a support vector regression (SVR) machine which is applicable to real GPU hardware. It can estimate the average runtime power of a GPU kernel by using a set of profiling parameters under different core and memory frequencies. They experimented with 931 samples obtained from 19 GPU kernels running on a real GPU platform with the core and memory frequencies ranging between 400 MHz to 1000 MHz. They further demonstrated that in conjunction with a performance prediction model, the optimal GPU frequency settings could be determined. These frequency settings led to considerable energy savings across 12 kernels [29].

Boyer claims that optimizing applications for systems with a great level of heterogeneity is quite challenging. GPU-enabled systems, despite offering many advantages, deal with applications which are wasteful of performance and energy. This wastefulness stems from a lack of resource utilization. General techniques were proposed to reduce overheads and improve GPU utilization and performance. The underutilization of GPUs was first addressed by reducing CPU-GPU interaction. Their strategies were tested with a leukocyte tracking case study. A dynamic scheduling algorithm was also proposed to balance an arbitrary GPU kernel across multiple devices and to keep each device fully utilized. A DVFS algorithm was also proposed to slow down the under-utilization of unutilized resources to increase effective utilization and energy efficiency [30].

Wang and Song proposed a technique for saving energy in HCSs [31]. This technique models the problem of workload division and voltage scaling as an integer linear programming problem, with the objective of minimizing energy consumption for a given performance constraint. Voltage scaling and workload division affect each other and also have an effect on the performance and energy consumption of both CPUs and GPUs.

Hamano et al. proposed a task scheduling scheme to optimize the overall energy consumption for heterogeneous clusters consisting of CPUs and accelerators. This scheme has both static and dynamic aspects and is characterized by an acceleration factor. The acceleration factor is used to denote the speed-up a task can achieve upon being implemented on either of the CPU or an accelerator. The scheme uses the acceleration factor as well as other parameters to minimize the EDP (Energy-Delay Product) of each task. EDP enables one to determine the optimum supply voltage and allows flexible power-performance trade-off. This results in the optimization of both the processing time and the energy [32].

Timm et al. proposed a GPGPU-based design space exploration strategy for a biosensor which is used for both in-situ and real-time detection of viruses. Their strategy placed special emphasis on energy reduction. They proved that increasing the number of active cores does not necessarily decrease the application's energy consumption [33]. Komoda et al. presented an efficient power-capping technique by coordinating DVFS and task mapping in a single computing node equipped with GPUs. Power capping is a method used for limiting the power consumption of a system to a predetermined level. To implement the settings of DVFS and task mapping for avoiding power violation and load balance, empirical models for power and performance of a CPU-GPU heterogeneous system were developed. The empirical models predict the execution time and the maximum power consumption in a hybrid power consumption by using both the CPU and the GPU. Performance and power consumption are usually dependent on the CPU and GPU frequencies and the task mapping. The frequencies of the CPU and the GPU and the percentages of tasks mapped to the CPU or the GPU are determined at the start of the application's execution. The outcome of this method is an optimal set of device frequencies and task mapping [34].

Cebrian et al. claim that not all the applications to be ported to GPUs can make use of the available resources due to bandwidth requirements, data dependencies, etc. which in turn reduces the performance per watt. They proposed several strategies involving voltage and frequency scaling and SM (Streaming Multiprocessor) power gating on GPUs. Their work confirmed the presence of under-utilization and indicated that resource optimization can increase the energy efficiency of GPU-based computation [35].

Lin et al. proposed analytical energy and performance models for auto-tuning GPGPU applications [36]. The objective of their strategy was to find an optimal configuration for a kernel which meets the energy and performance constraints set by the user. Their work consists of an analytical model for estimating the performance and power consumption of kernels and an auto-tuning framework for automatically obtaining the near-optimal configuration for kernel computation. Their strategy for obtaining the optimal kernel configuration involved Simulated Annealing (SA) or Genetic Algorithm (GA). They performed their experiments with the Intel Core i7-2600 CPU and used five NVIDIA GPUs having either the Kepler or Fermi micro-architectures. They chose some of their applications from popular GPU-based benchmarks and the classic matrix multiplication example from the NVIDIA's SDKs. They experimented with several iteration values, termination conditions, annealing rates and compared their search space results with the brute-force algorithm.

According to Park et al., the following optimization strategies are used for modern GPU architectures:

- Mapping threads to Streaming Multi-processors (SMs) and Streaming Processors (SPs) of GPU
- Coalesced global memory accesses
- Shared memory access devoid of bank conflicts
- Divergence avoidance in conditional statements
- Loop Unrolling or Algorithmic Cascading
- Using Single Precision
- Using Fast math

Park et al. claimed that the most effective optimization strategy is to find the optimal mappings of the application threads to the SMs and the SPs of the GPU. To affirm that claim, they performed design space exploration with a commonly used GPU (NVIDIA GTX 660) to investigate the best kernel grid structure of the GPU for optimal power or energy consumption.

They performed their experiments on the dot-product application and sought to identify the thread and block sizes, which lead to the least expenditure of power/energy.

Rethinagiri et al. proposed a two-part methodology consisting of a system-level power/energy estimation strategy and optimization techniques for heterogeneous CPU-GPU platforms. The power estimation technique is devised using functional parameters such as external memory access, bus access rate, cache miss rate, frequency of the processor, etc. Moreover, a system-level simulation prototype was devised to accurately assess the activities of the power model. They leveraged the power estimation methodology to develop novel power optimization techniques involving inter-task DVFS and work-load balancing. The effectiveness of their methodologies was tested using the CARMA kit, consisting of an ARM quad-core processor and a NVIDIA 96 core GPU processor for industrial benchmarks [38].

Timm et al. sought to prove that adding an additional GPU in an embedded system will lead to reduced energy consumption and also speed up applications. They found from their studies that to optimize the benefits of incorporating a GPU in an embedded system, the structure and features of the application (such as parallelism and GPU resource utilization) should be fully exploited. Their methodology consisted of modeling and profiling applications at a coarse-grain level and then performing a profit analysis to confirm the benefits of including a GPU to improve energy efficiency. They used an Intel Atom 270 as their CPU and the NVIDIA 8400 GS as their GPU. They highlighted the necessity of minimizing the idle time, the communication over-head and the incorporation of DVFS techniques to optimize the usage of GPUs [33].

Mapping applications with deadlines onto heterogeneous systems consisting of two or more resources is an NP-hard problem. It calls for designing efficient mapping heuristics which can speed-up the execution of applications to meet their deadlines. Liu et al. sought to devise power-efficient mapping techniques which can meet the timing requirements of applications while reducing power and energy consumption by using the DVFS methodology. Their technique mainly consists of two steps. In the first step, their technique maps the application to either a CPU or GPU to satisfy the timing constraints. The mapping can occur both online and offline. The second step involves using DVFS to both the CPU and the GPU to minimize energy expenditure. They also exploit the fact that average-case execution times are less than the worst-case execution times. Additionally, they leverage the slack from the early completion of tasks by using DVFS. This leads to considerable energy savings. They used the Intel Xeon 5160 CPU and an AMD Radeon HD 5770 GPU for their experiments [40].

Wang sought to optimize the leakage power of caches in GPUs by dynamically switching L1 and L2 caches in low power modes during periods of inactivity to reduce leakage power. Two DFS techniques were also proposed to improve the performance and the dynamic power of GPUs. One uses a feed-back controlling algorithm to regulate the frequency of parallel processors and memory channels based on the occupancy of memory buffering queues. The other technique is based on maximizing the throughput of all parallel processors under dynamic power constraints. This was formalized as a linear programming problem and solved at run-time [41].

2.4.2 Profiling Techniques for GPUs

Timm et al. measured the overall power consumption at two different points during the execution of the benchmarks [39]. The power consumption of the GPGPU-equipped graphics systems was measured by inserting probes between the supply lines of the PCI Express bus. For measurements, a 0.1 Ω resistor is embedded into the 12 V and 19 V supply lines and a 0.01 Ω resistor into the 3.3 V supply line. They measured the voltage drop across these resistors. The power values for the system and the GPU were determined by using the values of the current. The energy consumption for the system alone and the system with the graphics card were found by using the Reimann Sum (approximation method which uses finite sums).

Martin Peres claims that estimating the power consumption can be done in real-time by using two different methods. The first method involves reading the voltage drop across a shunt resistor mounted in series with the chips power-line. The instantaneous power consumption on the chip is equal to the voltage delivered by the voltage controller times the measured current. This method, he notes, also has its flaws. It requires an ADC converter and dedicated circuitry. The setup is expensive and requires dedicated hardware on the PCB of the GPU and a fast communication channel between the ADC and the chip. The other solution requires monitoring the block-level activity inside the chip. Power consumption can be estimated by monitoring the activity of different blocks on the chip, giving them a weight depending on the number of gates they contain, sum all the values, low-pass filter them then integrate over the refresh period of the power estimator. This is done by the hardware itself [42].

Lin et al. do power profiling by considering the fact that power consumed by a kernel is related to the instructions in its assembly code. When a kernel is executed, every instruction consumes several watts. The power consumed by each instruction is measured and the information is used to profile the power consumption of a kernel. The kernel is compiled by NVCC into object code and it dis-assembled into assembly code by a dis-assembler. The power consumption of the given assembly code is analyzed by power-profiling. For their experiments, they used a current clamp to measure the GPU current. The values obtained were used to calculate the power and energy. The electric current was measured from the PCI-e bus to the GPU device. It is then multiplied by the voltage of the PCI-e bus to get the power consumption. The power consumption was multiplied by the execution time of the kernel to get the energy consumption of the kernel on the GPU [36].

NVIDIA has introduced a tool called Nvidia System Management Interface (`nvidia-smi`) which provides management and monitoring capabilities for NVIDIA's GPU devices. It allows the user to control the power state of the GPUs and monitor memory usage and power consumption. It is specifically targeted towards NVIDIA's Tesla, Quadro, GRID and GeForce devices [43].

2.4.3 Task Mapping for CPU-GPU platforms

Paone proposed a novel analytic technique for fast pruning of the design space [45]. The methodology exploits the concurrency in an application task graph for efficient mapping on heterogeneous parallel platforms. During the tuning phase, the technique generalizes the previous analysis of interdependent parameters by exploiting a constraint solver to efficiently identify an initial set of task configurations which are compliant with the platform constraints. The second step is a mapping phase where inter-task parallelism is improved while accounting for the overhead of host-to-device and device-to-host memory transfers. The design flow was validated with a stereo-matching application that was implemented in OpenCL.

Vilches et al. presented a novel adaptive partitioning strategy that was geared specifically towards irregular applications running on heterogeneous CPU-GPU chips. This strategy involves exploiting parallel loops in irregular applications running on CPU-GPU chips to dynamically find appropriate chunk sizes for CPU and GPU cores. This strategy was evaluated on the quad-core Intel Core i7-4770 processor which also features the HD-4600 on-chip GPU. They used applications from the Rodinia and SHOC suites in addition to other regular and irregular applications to test their strategy [45].

Wachter et al. proposed a temperature-aware mapping and partitioning methodology for CPU-GPU MPSoCs. They have done profiling by measuring the temperatures of the CPU and GPU cores while executing different applications at different partitions. The temperature data obtained from evaluated to partition their applications between the CPU and the GPU. They tested their methodology with all the applications in the Polybench benchmark on the Odroid XU-3 platform [46].

2.5 GPGPU programming

Many programming options are available to speed up the applications using GPGPU platforms. CUDA, OpenCL and DirectCompute are the most prominent Application Programming Interfaces (APIs) for heterogeneous systems. DirectCompute is a GPGPU API which was developed by Microsoft and it uses the High-Level Shader Language (HLSL). It is easy to use for programmers but does not provide sufficient support and documentation.

Nvidia has developed the CUDA environment. It is a mature framework and has a C like syntax which makes it user-friendly for C programmers. It provides support for GPU optimized libraries and can be used easily with existing solutions. Nevertheless, CUDA is only supported by NVIDIA GPUs and does not fall back to the CPU when the GPU is not detected.

OpenCL (Open Computing Language), on the other hand, is an open environment and has been adopted by several vendors. It facilitates both task parallelism and data parallelism. For each PE, a command queue can be created through which tasks can be submitted. Use of parallel programming can be achieved on each PE individually and on all the PEs collectively. It provides a higher abstraction programming framework and enables the programmer to write portable programs for a wide variety of processing units. This does not necessarily mean that OpenCL can achieve the highest possible performance for heterogeneous systems [47].

2.5.1 Introduction to CUDA architecture

In GPUs, most of the die area is used for ALUs unlike CPUs, where majority of the area is devoted to the memory cache. GPUs, on the other hand, have relatively small caches. The previous generations partitioned resources into vertex and pixel shaders. The CUDA architecture (Nvidia uses the same name for its GPUs' architecture and their programming environment) includes a unified shader pipeline which allows every arithmetic logic unit (ALU) to be harnessed by the program to perform general purpose computations. NVIDIA intended this new family of processors to be used for general purpose computing. In order to meet this goal, the ALUs were designed to use an instruction set tailored for general computation rather than for graphics. Execution units were allowed arbitrary read and write accesses to memory and to a software managed cache called shared memory. These features enable the CUDA architecture to perform computations and traditional graphics tasks [48].

NVIDIA GPUs consist of several Streaming Multiprocessors (SMs) and they serve as abstractions for the underlying hardware. Every Streaming Multiprocessor has a set of Streaming Processors (SPs), which are commonly called CUDA cores. These cores are solely responsible for executing instructions in a program. The number of SPs in a SM and the number of SMs vary with every GPU device.

The CUDA cores in a SM are called SIMT cores (Single Instruction Multiple Threads) cores. Groups of 32 cores called Warps operate simultaneously to execute the same instructions with multiple data. Every SM has warp schedulers for executing the work performed by the 32 cores.

GPUs have their own memory on board. GPU memory can range from 768 MB to 6 GB of GDDR5 Memory. All CUDA capable cards have a fully coherent L2 cache. A GPU's L2 cache is considerably smaller than a CPU's L2 cache but offers much larger bandwidth. The L1 caches in such CUDA capable cards are not coherent, unlike the L1 caches in CPUs. As a result, if two different SMs are reading from and writing to the same memory location, changes made by one SM will not be visible to the other SM. This complicates the task of debugging errors. Every SM is equipped with its own L1 cache [48, 50].

2.5.2 CUDA's Programming Model

CUDA's parallel programming model has three key abstractions: a hierarchy of thread groups, a hierarchy of shared memories and barrier synchronization. These abstractions provide the facilities of fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The programmer can partition the problem into coarse sub-problems which can be solved independently in parallel, and then into finer pieces which can be cooperatively solved in parallel. This allows threads to cooperate when solving each sub-problem. At the same time, it enables transparent scalability since each sub-problem can be scheduled to be solved on any number of processor cores and only the run-time system needs to know the processor count. As a result of this model, the CUDA architecture spans a wide range of applications by simply scaling the number of processors and memory partitions [49].

2.5.3 CUDA's Execution Model

A CUDA device is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendental operations, a multithreaded instruction unit, an on-chip shared memory. The multiprocessor creates, manages and executes threads concurrently in hardware with little scheduling overhead. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support fine-grained parallelism.

To manage hundreds of threads running several different programs, the multiprocessor employs a SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one SP, and each scalar thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates and manages threads in groups of 32 parallel threads called warps. Individual threads composing a SIMT warp start together at the same program address but are free to branch and execute independently [50].

2.5.4 OpenCL Application Programming Interface

OpenCL is an open-source API developed to allow co-processors to work in tandem with CPUs. It is maintained by the Khronos Group. It is like CUDA and has a C like syntax which can be integrated easily. OpenCL supports multiple devices such as multi-core CPUs, multi-socket CPU, GPUs and cell processors. This allows the programmer to change the hardware architecture without any changes to the code as OpenCL is the standard from which vendors are expected to derive abstractions to support their heterogeneous devices. This provides it the capability to support devices like FPGAs and mobile hardware in the future. Since OpenCL can provide support for a general heterogeneous environment and industry support is available for it, this API is being preferred over others.

Every OpenCL implementation defines platforms which enable the host system to interact with OpenCL-capable devices. The software architecture of all implementations can be described by:

- Platform Model
- Execution Model
- Memory Model

Platform Model

This model consists of the host connected to one or more OpenCL devices. A device is divided into one or more compute units. Compute units, in turn, are divided into processing elements.

Execution Model

The Execution Model consists of two parts: kernels which execute on one or more OpenCL devices and a host program which executes on the host. Both the platform and the execution model are visually represented by Fig. 2.8.

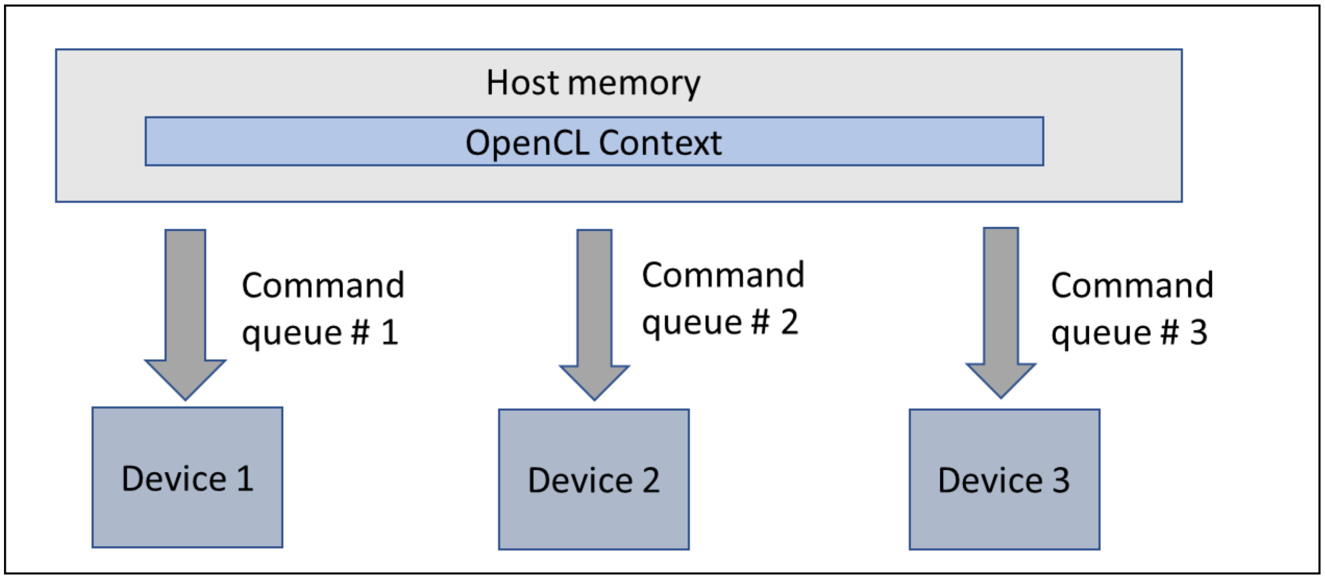


Fig. 2.8: OpenCL platform and execution model.

The host program creates the context for the kernels and coordinates their execution. An index space is defined whenever a kernel is submitted to the host for execution. The index space is called ND-Range and is an N-dimensional index space (where N can be 1,2 or 3).

An instance of a kernel executes for each point in the index space. In OpenCL terminology, a kernel instance is called a work-item and is identified by its point in the index-space, which provides a global-id for the work-item. Every work-item executes the same code but the data that every work-item operates upon is different. Work-items are organized into work-groups. Work-groups are assigned a unique work-group ID with the same dimensions as the index space used for the work-items. All work-items in a work-group execute on the same compute unit and are capable of synchronization.

A context is defined for the execution of the kernels. It is created by the host using OpenCL API functions. A context is composed of the following parts:

- Devices: Collection of OpenCL devices used by the host.
- Kernels: OpenCL functions which run on OpenCL devices.
- Program: Source and executable for the implementation of kernels.
- Memory objects: These objects are visible to the host and the OpenCL devices.

Commands are placed in the command queue which are scheduled on the devices within the context. They are of the following types:

- Kernel execution commands which are executed on the processing elements of a device.
- Memory commands are used for transferring data between host and device as well as for mapping and un-mapping objects from the host address space.
- Synchronization commands regulate the order of execution. The command queue regulates the order of execution on a device. Commands can be launched in-order, where they execute in the order they appear in the command queue or out-of-order, where the following commands begin executing before their completion [47, 50].

Memory Model

Memory objects can be accessed by all the devices only when they are defined in the same context. Consequently, the work-items which are defined have access to four distinct memory regions as shown in Fig. 2.9:

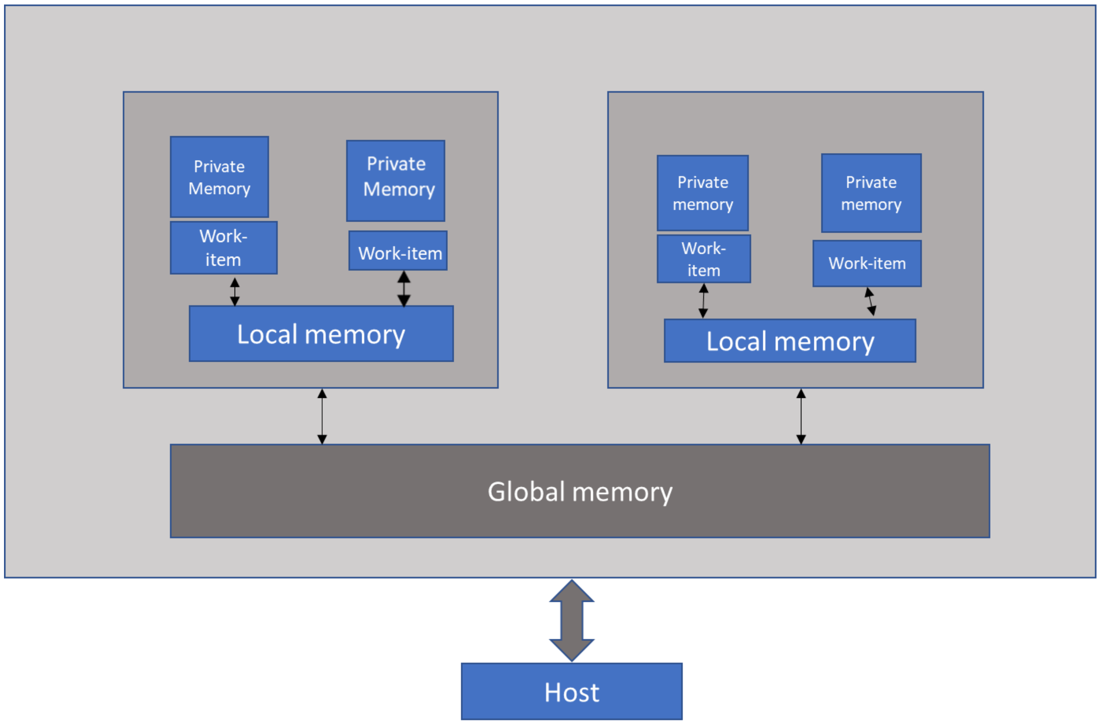


Fig. 2.9: OpenCL memory model.

1. Global Memory: This region permits read/write access to all the work-items in all work-groups.

2. Constant Memory: It is a region in global memory and remains constant during the execution of a kernel. Memory objects placed into constant memory are allocated and initialized by the hosts.

3. Local Memory: This memory region is only accessible to its corresponding work-group. This memory region can be used to allocate variables that are shared by all the work-items of that work-group. It may be implemented as dedicated regions of memory on the OpenCL device.

4. Private Memory: This memory region corresponds to work-items. The variables in a work-item's private memory are not visible to other work-items.

Chapter 3

System-level Optimization for CPU-GPU platforms

3.1 Introduction

The proposed co-synthesis methodology is created with the objective of obtaining optimal assignment of application segments on a target architecture so that user-defined constraints for the application are fulfilled. The methodology, visually represented by Fig. 3.1, consists of three major steps. It closely follows the methodology described in Chapters 1 and 2. Section 2 of this chapter provides a generic description of the proposed co-synthesis algorithm (Step 1 in Fig. 3.1). The second step, described in detail in section 3, involves expanding this algorithm to cater to specific target architectures and their peculiarities. Section 4 delves into step 3. In step 3, the algorithm developed in Sections 2 and 3 is further modified to fully harness its potential to produce optimal solutions.

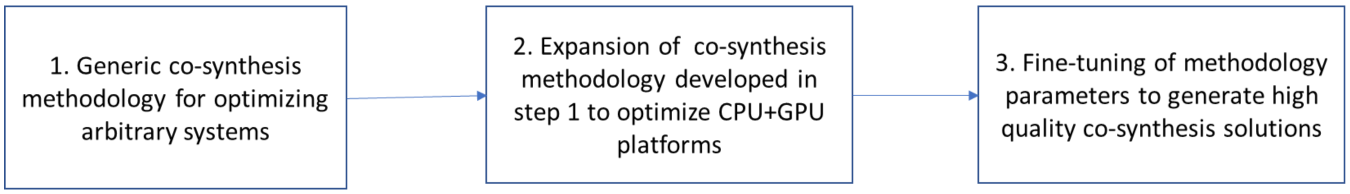

1. Generic co-synthesis methodology for optimizing arbitrary systems
2. Expansion of co-synthesis methodology developed in step 1 to optimize CPU+GPU platforms
3. Fine-tuning of methodology parameters to generate high quality co-synthesis solutions

Fig. 3.1: Three stages of the proposed co-synthesis methodology

3.2 General Design Flow

The objective of this co-synthesis methodology is to obtain near-optimal solutions which satisfy timing and power constraints for partitioning problems. These constraints are provided by the user. It uses a Genetic Algorithm (GA) to obtain near-optimal solutions. The GA is chosen due its proven ability of producing optimal solutions in a considerably short amount of time.

The target application is first specified in the form of a Directed Acyclic Graph (DAG) of nodes or sub-tasks. The Processing Elements and Communication Links and their specific characteristics are specified in a resource library. These elements are used to construct the target architecture on which the application will be executed. The algorithm starts with the random assignment of nodes of the application on the Processing Elements and the edges on the Communication Links. All the nodes and edges of the application are then assigned priorities depending upon the architectural elements they have been mapped to. After all the sub-tasks and edges are assigned priorities, they are scheduled according to the priorities they were assigned. The scheduled application is then evaluated according to the metrics selected by the user. The GA takes the scheduled application as the input and uses it to generate application configurations that meet user-defined targets.

We begin with the generic co-design methodology which can obtain optimal hardware/software co-synthesis solutions for generic problems. The flowchart in Fig. 3.2 is a visual depiction of our co-synthesis methodology. After describing the strategy on a generic level, we gradually incorporate more features into the generic strategy to exploit unique features of both the application and the architecture. The following sub-sections describe each of the blocks of the generic algorithm in more detail.

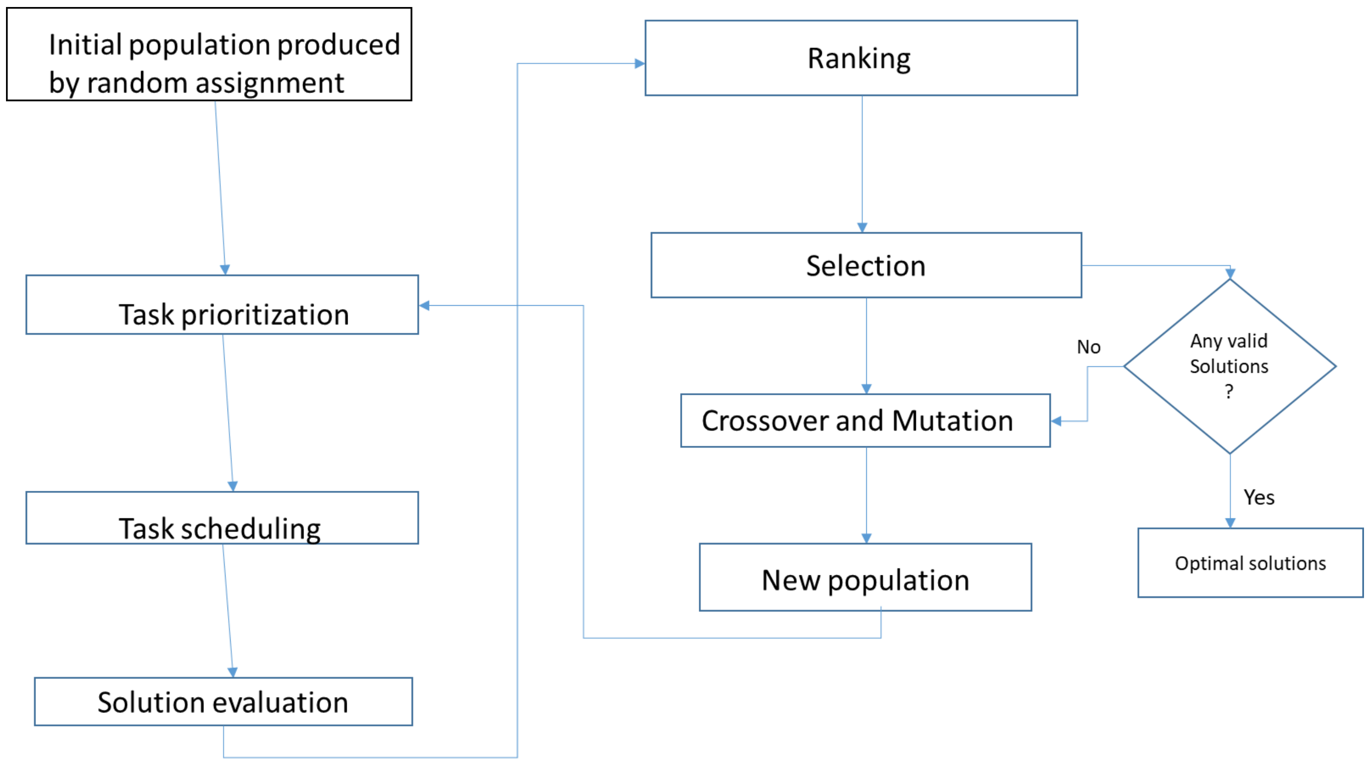


Fig. 3.2: Co-synthesis methodology.

3.2.1 Input Specification

The inputs to the co-design tool are an application specified as a DAG and a library of the processing elements being used. The relevant information about the DAG and the target architecture is contained in a text file which is fed to the tool before it starts to execute.

Directed Acyclic Graph (DAG): The DAG is characterized by nodes and edges. It is derived from a sequential C specification of the application. The nodes correspond to processes (sub-tasks) of the application and the edges correspond to the amount of data travelling from one node to another. A DAG is also characterized by a period (time elapsed between consecutive executions of the graph) and deadlines. All the sink-nodes (nodes with no outgoing edges) have deadlines, which is the latest time by which the node must complete its execution. Fig. 3.3 shows a typical DAG. The words sub-task, node and process are used interchangeably in the rest of this document.

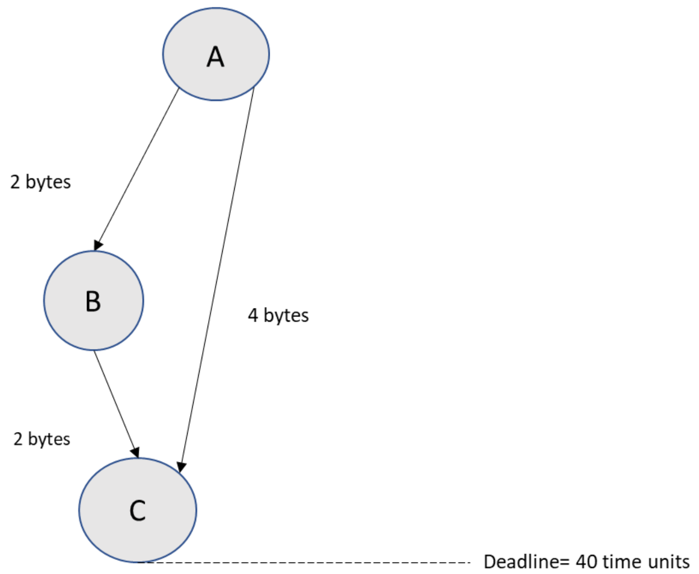


Fig.3.3: An example of a typical DAG. Node C is the sink node.

In our co-design methodology, each process of the DAG has the following characteristics: *ID, name*, *deadline*, *start-time*, *finish-time* and *priority*. Every edge of the DAG is characterized by its *ID, data-transfer rate, start-time* and *finish-time*.

The fields *deadline, priority*, *data-transfer, start-time* and *finish-time* and their significance will be described in the subsequent sections in more detail.

Processing Elements and Communication Links (Resource library): The resource library consists of processing elements (PE) and communication links. It also contains specific information about the architectural elements' features which are relevant to the methodology. The processes of the DAG are mapped to the processing elements whereas the edges are mapped to the communication links. An example of a target architecture can be seen in Fig. 3.4.

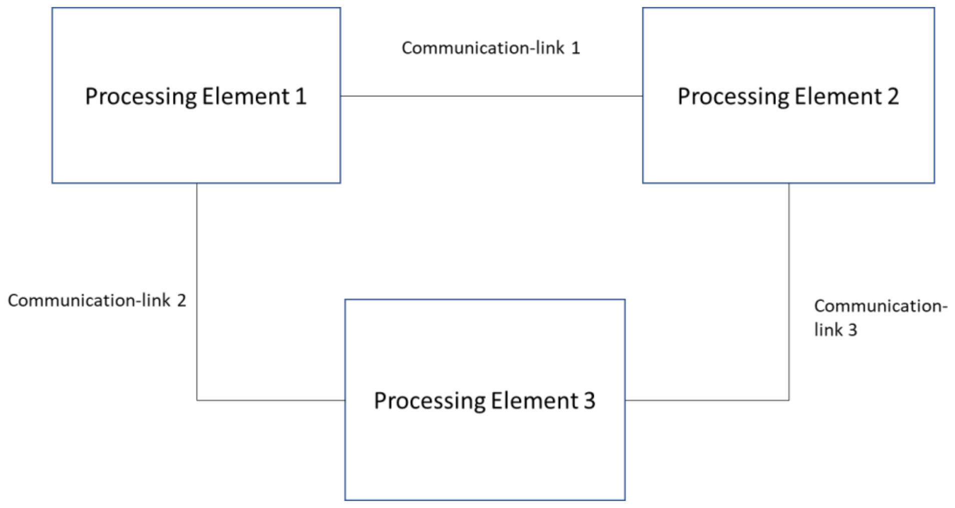


Fig. 3.4: Example of a target architecture.

In the proposed methodology, each PE is characterized by the following rules:

- *ID:* The unique ID of the PE.
- *Cost:* The monetary cost associated with the PE.
- *Execution time* of every process of the DAG which can be executed on that PE.

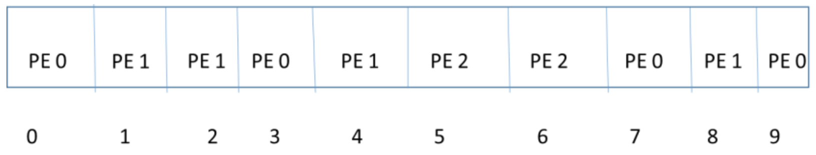
Each communication link is characterized by these features:

- *ID:* The unique ID of the communication link.
- *Data-transfer:* Amount of data of every edge that can be mapped to it.

This above information about the PEs and the communication links is provided in the resource library. The following sections will explore how the GA uses the above information about the resource library and the DAG to meet its objectives.

3.2.2 Creating a population

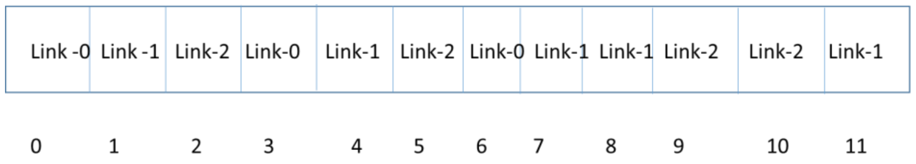
Population creation is the first step of the algorithm. The “population” consists of several “individuals”/ “solutions”. A “solution” is characterized by an array of elements where every element corresponds to one of the processing elements and the index of the element corresponds to the ID of one of the task graph’s processes. The length of the array is equal the number of processes in the task graph. Every solution in the initial population is generated by randomly assigning processes to Processing Elements as shown in Fig. 3.5.



PE 0	PE 1	PE 1	PE 0	PE 1	PE 2	PE 2	PE 0	PE 1	PE 0
0	1	2	3	4	5	6	7	8	9

Fig. 3.5: Process Assignment array.

Similarly, an array is formed for the edges of the task graph. In this array, the elements correspond to the communication links whereas the indices of those elements denote the edges which have been assigned to those communication links. It is important to note that edge assignment is dependent upon the process assignment array produced above. The edges of the task-graph are then assigned to communication resources based on the PE assignment solution generated earlier. The edge assignment array is depicted in Fig. 3.6.



Link -0	Link -1	Link-2	Link-0	Link-1	Link-2	Link-0	Link-1	Link-1	Link-2	Link-2	Link-1
0	1	2	3	4	5	6	7	8	9	10	11

Fig. 3.6: Edge assignment array.

If the source process of an edge is on one PE and the sink process is on another PE and there is a link connecting those PEs, the edge is then assigned to that link, as shown in Fig. 3.7. If the source and sink processes of an edge are on the same PE (see Fig. 3.8) after random assignment of processes to PEs, the source/sink process is shifted to a PE which is on the other end of a communication link attached to the PE being considered. The edge is then assigned to the communication link. The pseudo-code in Fig. 3.9 describes this activity in more detail.

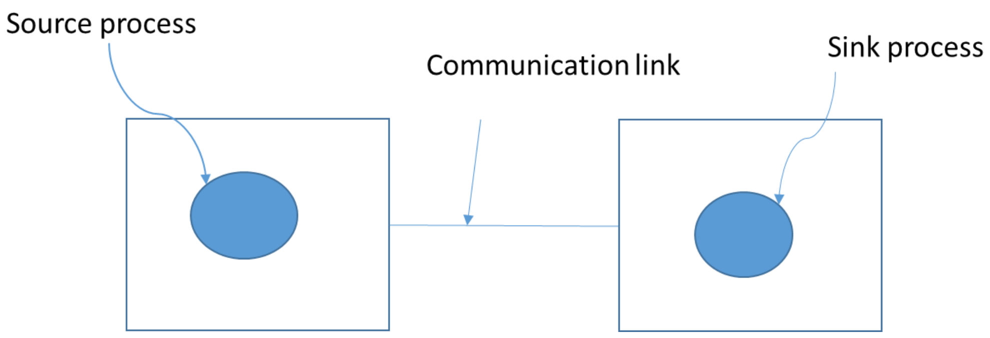


Fig. 3.7: Source and sink processes on different PEs.

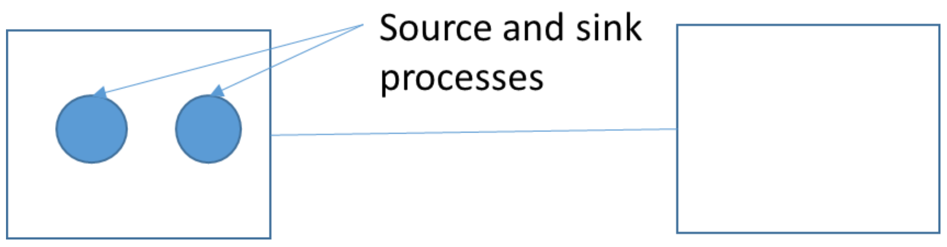


Fig. 3.8: Source and sink processes on the same PE.

```
Int[] Edge_Assignment(){
Edge_Assignment_Array[all_edges_of_DAG];
for each (Edge in all_edges_of_DAG)
{
  1. Get the PE the preceding node is assigned to
  2. Get the PE the following node is assigned to
  3. Determine if there is a link between the PEs found in steps 1 and 2

    if (link_found)
    {

       4. Push back link into Edge_Assignment_Array

    }

    Else if (link_not_found) {

        5. Replace PE in step 2 with any of the other available PEs

        6. Determine if there is link between PEs of step 1 and step 5

      if (link_found)
      {
             // Link found, Processes are on seperate PEs now
            7. Push back link into Edge_Assignment_Array
       }
    }

  }
}
```

Fig. 3.9: Pseudo-code for Edge-assignment array

3.2.3 Priority Assignment

After the initial population of a user-defined number of solutions is generated, all the nodes and communication edges of the task graph are assigned priorities on the basis of task deadlines and execution times. Please note that the final node of the task graph, the sink process, has no outgoing edges. This node has a “real” deadline. The deadline represents the maximum allowable time by which the entire application must complete its execution.

The priority of this sink process is calculated by using equation 1:

$$Priority\ (sink\ process) = Processing\ time - Deadline \qquad (1)$$

The “Processing time” of a certain process is the sum of its execution time on the PE it is assigned to and also the maximum amount of data transfer time of all its incoming edges.

$$Processing\ time =$$
$$Execution\ time\ on\ the\ PE + \max(Data\ transfer\ time\ of\ incoming\ edges) \qquad (2)$$

The data transfer times of all edges in the task graph are calculated using the following equation:

$$Data\ transfer\ time\ of\ an\ edge =$$
$$Data\ transfer\ rate\ of\ the\ communication\ resource * amount\ of\ data \qquad (3)$$

The first node of the task graph i.e. the source process, has no incoming edges. Every other node, apart from the source and sink nodes, in the task graph has nodes that precede it and follow it. For any given node, the preceding nodes are also called “parent processes” and the nodes which follow it are known as “child processes” [19].

The priorities of these nodes are calculated by using the following equation:

$$Priority\ (non-sink\ process) =$$
$$Processing\ time + \max(highest\ child\ priority, -Deadline) \qquad (4)$$

Communication edges are also assigned priorities. A communication edge whose destination process has a higher priority is assigned a higher priority. The pseudo-code for deadline and priority assignments can be seen in Fig. 3.10.

```
void RecursiveDeadline&PriorityAssignment()
{
  for(all_processes_in_DAG)
 {
      if (process_deadline_not_yet_assigned)
      {
          Get child processes of the process

         if (children.size() == 0)
        {
                  // Last Node (Sink Process)
          process.priority= maximum_incoming_Data_Transfer +
process.exec.time - process.deadline;
         }
      }

      Else if (process_deadline_assigned)
      {
             Get child processes of the process

            if (deadlines_assigned_to_child_processes)
            {
                  // All of it's children have deadlines assigned
            process.deadline = get_minimum_children_deadline;
            process.priority = maximum_incoming_Data_Transfer +
process.exec.time +   max(max_child_priority,-process.deadline);

            }

            else
            {
                  for (int i = 0; i < children.size(); i++)
                  {
                     //assign deadlines recursively to child processes
                       RecursiveDeadline&PriorityAssignment();
                   }
              }
         }
     }
   }
```

Fig. 3.10: Pseudo-code for recursive deadline and priority assignments

3.2.4 Scheduling of Processes and Communication Events

After the processes and edges of a task graph are assigned to the PEs and edges respectively and prioritized, they are scheduled. In the process of scheduling, starting times and finishing times are assigned to all processes and edges.

All processes and communication events of the task graph are assigned start times and completion times according to their priorities. The source process, with no incoming edges, has a *start time* of zero. Its *finish time* is its processing time on the PE it was assigned. The outgoing edges of the source process are retrieved and assigned start and finish times. The *start time* of each of these outgoing edges is equivalent to the *finish time* of its source process. All processes and edges are scheduled on the resources they have been assigned to. If the process or node is not a source process, the finish times of its incoming edges are compared and the highest *finish time* is selected as the *start time* of the process. The start and finish times of the outgoing edges of all the non-source processes are assigned in the same way as those of the source process. The *finish time* of the source process is assigned as the *start time* of each of its outgoing edges. The *finish time* of an outgoing edge is then the sum of the *start time* of the edge and its processing time on the communication link it has been assigned to. The processing times of the edges can be determined as the product of the bandwidth of the resource and the rate of data transfer. The details of this routine can be seen in the pseudo-code of Fig. 3.11. The pseudo-codes in Figs. 3.12, 3.13 and 3.14 respectively describe how edges, a single process and all the processes of the application are scheduled. The visual representation of the scheduled edges and nodes can be seen in Fig. 3.15.

```
double TimeTakenForDataTransfer(comedge edge, vector<comresource> links)
{
      resource foundLink = GetLinkForEdge(edge, links);

      //If a link is found
      if (foundLink)
      {
            double com_speed = foundLink.com_rate;

            double data_transfer = edge->rate/com_speed;
            return data_transfer;
      }

      return -1.00;
}
```

Fig. 3.11: Pseudo-code for the processing time of communication edges

```
void calculate_Edge_Start_And_End_Times()
{
      vector<edges> outgoing_Edges = Get_Outgoing_Edges(Scheduled_Process,
      all_Edges);

      for (int i = 0; i < outgoing_Edges.size(); i++)
      {
            outgoing_Edges[i].start_time = Scheduled_Process.finish_time;

            outgoing_Edges[i].finish_time = ScheduledProcess.finish_time +
TimeTakenForDataTransfer(outgoing_Edges[i], links);

      }
}
```

Fig. 3.12: Pseudo-code for scheduling of outgoing edges.

```
void ScheduleOneProcess(){
if (highest_priority || p_is_source_process)
       {
          p.start_time = 0;
              p.finish_time = p.exec_time;

       }
       else
       {
              // get the incoming edges of the process
              GetIncomingEdges(p.id);

               // get the edge schedule of all my edges
              for (int j = 0; j < all_incoming_edges; j++)
              {
                     IncomingEdgeTimes.push_back(all_incoming_edges[j].finish_time);
              }
              for (int i = 0; i < edges.size(); i++)
              {
                     if (p == following node of all_incoming_edges)
                     {
                          // pick the maximum edge time
                            max_time = max_element(IncomingEdgeTimes.begin(),
IncomingEdgeTimes.end());
                            break;
                     }
                     else
                     {
                            max_time = 0;
                     }
              }
                     // this maximum time will be equal to the start time of the process
                     p.start_time = max_time;

                     // the finish time will be equal to start_time + exec_time
                     p.finish_time = max_time + p.exec.time;

       //Calculate the start and end times of the following edges of p
         calculateEdgeStartAndEndTimes(p.id, following_edges);

       }
}
```

Fig. 3.13: Pseudo-code for scheduling a single process

```
void scheduleAllProcesses()
{

    for (int i = 0; i < allSortedProcesses; i++)
    {
         //Get parent processes
        parents = GetParentProcesses(processToSchedule);

        if (processToSchedule == source_process)
        {
            // pToSchedule is a source process
    Helper::scheduleOneProcess(processToSchedule);
        }
        else
        {
            if (all_parent_process ! scheduled)
            {
                // This process has an incoming parent process which
has not yet been scheduled.

                for (int k = 0; k < troubleProcesses.size(); k++)
                {
                    //Schedule all the troublesome processes
                    scheduleOneProcess(troubleProcesses[k]);
                }
            }
            else if (ifAllPreviousEdges != scheduled)
            {
                // This process has an incoming edge which has not
yet been scheduled.
                // Problem
                //cout << "Edges: accounting for unscheduled
edges...." << endl;
                vector<process*> troubleProcesses =
Helper::getUnscheduledProcesses(&parents, edges);
                for (int k = 0; k < troubleProcesses.size(); k++)
                {

    Helper::scheduleOneProcess(troubleProcesses[k]);
                }
            }

            // This process does not have highest priority but all it's
parent process and incoming edges have been scheduled.
            scheduleOneProcess(pToSchedule);
        }
    }

}
```

Fig. 3.14: Pseudo-code for scheduling all the processes

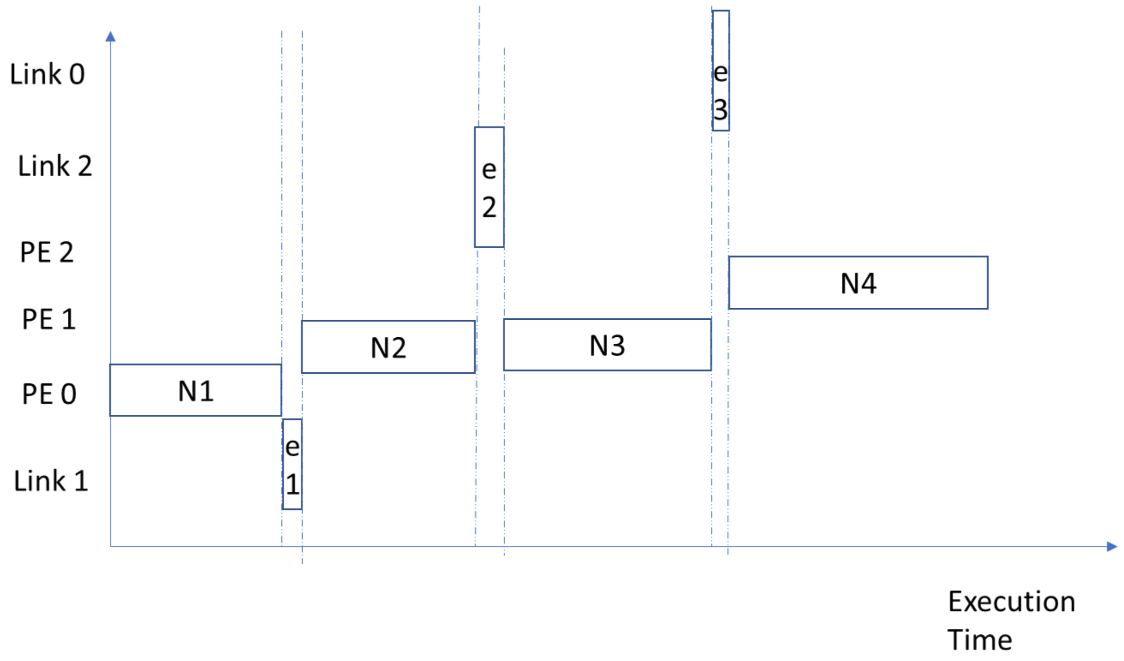


Fig. 3.15: Visual representation of scheduled nodes and edges

3.2.5 Evaluation of Solutions

Each solution in a population is evaluated on basis of its system cost, total execution time and the amount of by which it violates the deadlines. The deadline violation of a process is calculated by comparing its completion time with its deadline. The deadline violation of each process is added to obtain the total deadline violation time. The pseudo-code in Fig. 3.16 provides the details of this routine.

```
double total_deadline_violation(vector<process>* Scheduled_Processes)
{
      double deadline_violation = 0;

//Calculate the deadline violation time of every process after it has been
scheduled


  for (int i = 0; i < Scheduled_Processes->size(); i++)
      {

      if ((Scheduled_Processes->at(i).finish_time - Scheduled_Processes-
>at(i).start_time) > Scheduled_Processes->at(i).process_deadline){

              deadline_violation += (Scheduled_Processes->at(i).finish_time -
Scheduled_Processes->at(i).start_time) - Scheduled_Processes-
>at(i).process_deadline;
      }

      }

      return deadline_violation; //Deadline violation time of the entire
application


}
```

Fig. 3.16: Pseudo-code for obtaining deadline violation

The system cost of a solution is the sum of the cost of the PEs and communication resources it is made of. Every solution is essentially going to be evaluated with the cost function in (5):

$$Total\ system\ cost = M_1 * \sum_{i\in process} ExecutionTime_i + M_2 * \sum_{i\in process} PowerConsumption_i + M_3 * \sum_{i\in process} (Execution - time - Deadline)_i + M_4 * \sum_{j\in PE} ProcessorCost \quad (5)$$

The pseudo-code in Fig. 3.17 represents the function used to evaluate the solutions:

```
void evaluate_solutions(){

//Evaluate every solution in the population

for (int i = 0; i < generated_solutions.size(); i++)
{


//Total execution time of the application after it is scheduled

Generated_solutions[i].total_time= schedule_length();

//Amount of time by which the scheduled application violates its deadline

Generated_solutions[i].violation_time= total_deadline_violation();


//Total cost of the resources used by the application

Generated_solutions[i].resource_cost = System_cost();


//Final weighting function to determine the quality of the solution. M1, M2
and M3 represent the weights of each of the parameters

Generated_solutions[i].final_score = M1 * Normalized_ScheduleLength + M2 *
Normalized_violation + M3 * Normalized_SystemCost;

}

}
```

Fig. 3.17: Pseudo-code for calculating a solution's final score

A solution is invalid if it either violates the resource constraints or fails to meet its hard deadline.

3.2.6 Producing new solutions using the Genetic Algorithm

The Genetic Algorithm is allowed to run for a pre-determined number of times where every generation has a fixed number of solutions. It consists of the following steps:

Ranking

The solutions in the GA are ranked according to their system costs, total execution times and the amount by which they violate their deadlines. A solution with a lower system cost, shorter deadline violation time and execution time has a higher rank. The pseudo-code for sorting the solutions is shown in Fig. 3.18.

```
bool arrange(solutionarray &a, solutionarray &b)
{
      //solution with a smaller final score is superior
            return (a.final_score < b.final_score);
}

vector<solutionarray> Rank(vector<solutionarray>* population)
{
      //Ranking the individuals in the population
      sort(population->begin(), population->end(), arrange);

      return *population;
}
```

Fig. 3.18: Pseudo-code for sorting the solutions in a population

Selection

The solutions which satisfy the pre-established constraints are preserved in a separate array. This array consists exclusively of the best solutions and is regarded as the elite club. The solutions which have been put in the elite club are removed from the original population. If none of the solutions in the initial population matches the pre-established constraints, the best three solutions are picked to be placed in the elite club. These elite club solutions are subsequently removed from the original population. The very best solution in the elite club is added back into the original population. The remaining solutions of the elite club are passed on to the subsequent stages of the algorithm. This routine is described in Fig. 3.19.

```
vector<solutionarray> Selection(vector<solutionarray>* population)
{
      vector<solutionarray> elite_club;

      int best_out_of_elite = 0;

   for (int e = 0; e < population->size(); e++){

      if (population->at(e).final_score <= target_score){

            //Include this solution in the elite club
                  elite_club.push_back(population->at(e));
            }

      }


      //Remove the solutions which are now in the elite club from the
original population

      for (int chosenID = 0; chosenID < elite_club.size(); chosenID++)
      {
            for (int populationID = 0; populationID < population->size();
populationID++)
            {
                  if (population->at(populationID).ID ==
elite_club[chosenID].ID)
                  {
                        //remove the same values from the population vector
                        population->erase(population->begin() +
populationID);

                        break;
                  }
            }
      }

      //If none of the solutions in the population matched the target score

      if (elite_club.size() == 0)
      {
            //push back the 2 best solutions in elite_club
            for (int others = 0; others <= population->size(); others++)
            {
                  elite_club.push_back(population->at(others));

                  if (elite_club.size() == 3)
                  {
                        break;
                  }
            }
//pick the absolute best solution from elite_club and push it back in the
original population
            for (int i = elite_club.size() - 1; i > 0; i--)
            {
                  if (elite_club[i].final_score < elite_club[i -
1].final_score)
                  {
                        best_out_of_elite = i;

                  }
            }

            this->bestOfAll = elite_club[best_out_of_elite];

            elite_club.erase(elite_club.begin() + best_out_of_elite);

            //push back the best solution in the original population
            population->push_back(this->bestOfAll);


            (this->best_population) = elite_club;

            //elite club contains the second and third best solutions

            return (this->best_population);
      }

}
```

Fig. 3.19: Pseudo-code of selection strategy

Crossover and Mutation

After the solutions are selected, they undergo mutation and crossover. A PE for an arbitrary process is exchanged with another PE on which the process can be executed. This action is referred to as mutation. For example, if process 5 has currently been assigned to PE 1, it will be randomly assigned to PE 2(if it can be run on PE 2) as a result of mutation. Fig. 3.20 shows how mutation occurs.

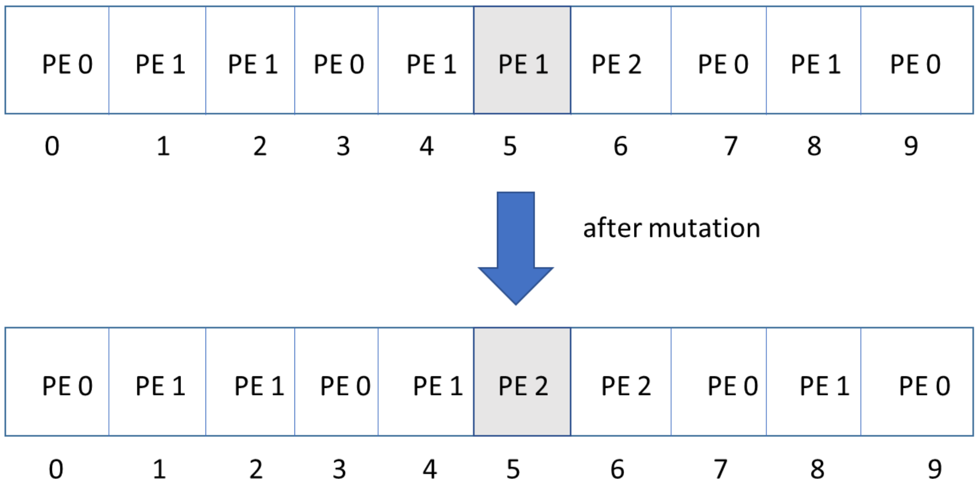


Fig. 3.20: Process 5 is now on PE 2 after mutation.

The pseudo-code in Fig. 3.21 illustrates how mutation is performed. In this code, the solution is “mutated” at random points i.e. processes of the target DAG are randomly picked and assigned to random processes.

```
void mutation()
{
//iterate over the solutions of the best population obtained after selection
  for (int i = 0; i < best_population->size(); i++)
{

//iterate over the process assignments of the solutions
   for (int offset = 0; offset < best_population->at(i).PE_array.size();
offset++)
  {
     //for the first half of the total number of generations
      if (before_halfway_point) {
       if ((rand() % 100) < 70) {
        //Solution is mutated at random positions with 70 % probability
         best_population->at(i).PE_array[offset].elementid = rand() % 2 + 1;
      }
      }

       //for the latter half of the total number of generations
      else {
       if ((rand() % 100) < 30) {
        //Solutions are mutated with 30 % probability
      best_population->at(i).PE_array[offset].elementid = rand() % 2 + 1;
      }
      }

    }

  }

}
```

Fig. 3.21: Pseudocode for mutation.

In crossover, two solutions are randomly selected. Two offsets on those solution arrays are then randomly chosen and portions of solutions between those offsets are swapped. This leads to the generation of two new solutions. Fig. 3.22 illustrates how crossover occurs between two solutions. Two random positions, namely offset 1 (process no. 3) and offset 2 (process no. 6), are picked for the two solutions. The two arrays swap all the array elements from indices 3 to 6 with each other. In other words, processes 3 to 6 in the two chosen solutions exchange PE assignments. Fig. 3.23 shows the pseudo-code for crossover.

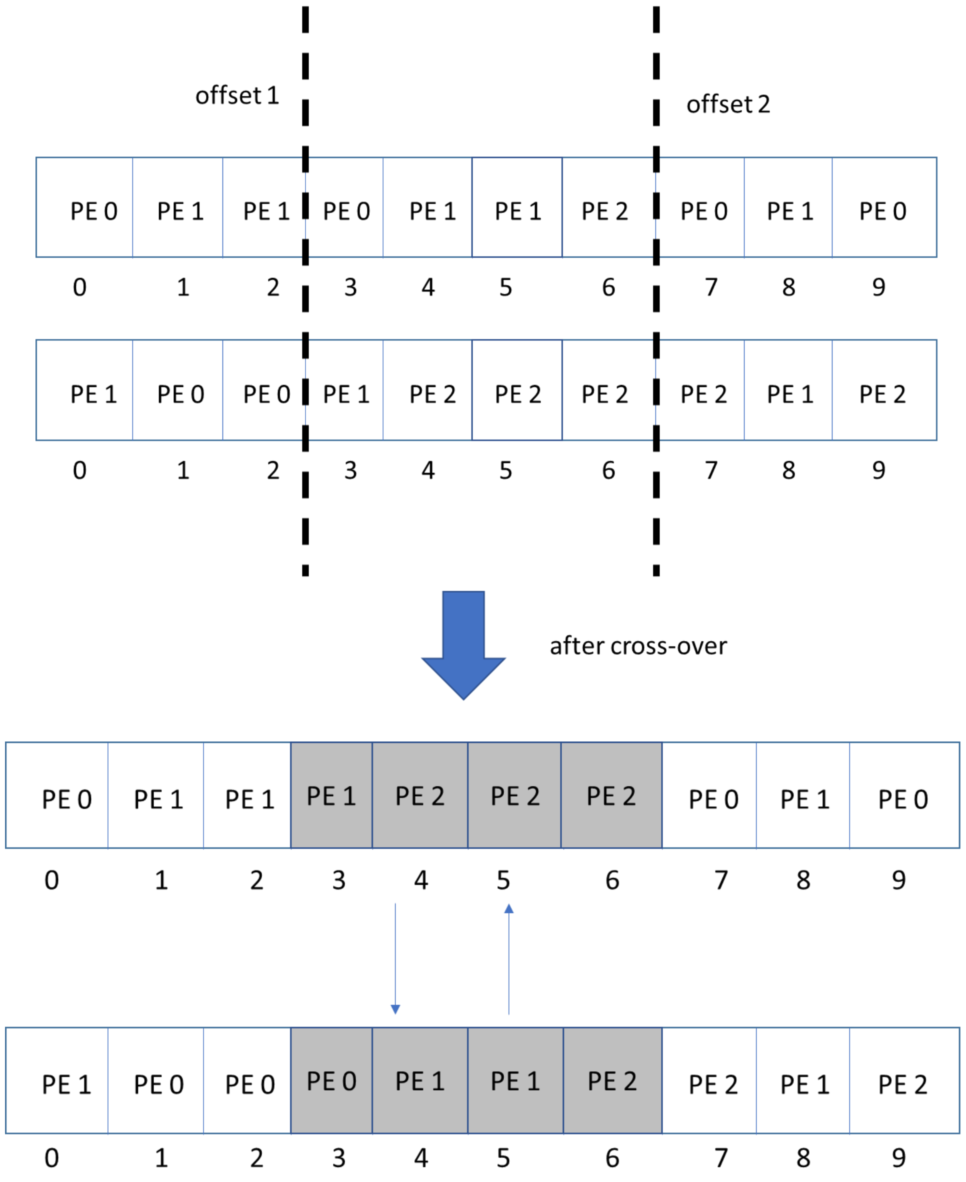

offset 1
offset 2
PE 0 PE 1 PE 1 PE 0 PE 1 PE 1 PE 2 PE 0 PE 1 PE 0
0 1 2 3 4 5 6 7 8 9
PE 1 PE 0 PE 0 PE 1 PE 2 PE 2 PE 2 PE 2 PE 1 PE 2
0 1 2 3 4 5 6 7 8 9
after cross-over
PE 0 PE 1 PE 1 PE 1 PE 2 PE 2 PE 2 PE 0 PE 1 PE 0
0 1 2 3 4 5 6 7 8 9
PE 1 PE 0 PE 0 PE 0 PE 1 PE 1 PE 2 PE 2 PE 1 PE 2
0 1 2 3 4 5 6 7 8 9

Fig. 3.22: Illustration of crossover of solutions

```
vector<solutionarray> crossover()
{

      int random_number = rand() % 100;
      int offset1, offset2;


      for (int i = 0; i < best_population->size(); i++)
      {
           //pick two random solutions from best_population
            int pair1 = rand() % best_population->size();
            int pair2 = rand() % best_population->size();


            if (before_halfway_point)
            {
                  if (random_number <= 70)
                  {
                        //pick two points randomly from 0 to length of
solution array
                        offset1= rand() % (best_population-
>at(0).PE_array.size());
                        offset2 = rand() % (best_population-
>at(0).PE_array.size());

                        while (offset1 < offset2){
                        //interchange all elements from offset1 to offset2
                        swap(best_population->at(pair1).PE_array[offset1],
best_population->at(pair2).PE_array[offset1]);
                             offset1++;
                        }
                  }
            }

      //after first-half of the total number of generations elapse
            else
            {
                  if ((random_number) <= 30)
                  {
                        //pick two points randomly from 0 to length of
solution array
                        offset1= rand() % (best_population-
>at(0).PE_array.size());
                        offset2 = rand() % (best_population-
>at(0).PE_array.size());

                        while (offset1 < offset2) {
                             //interchange all elements from offset1 to
offset2
```

```
    swap(best_population->at(pair1).PE_array[offset1], best_population-
    >at(pair2).PE_array[offset1]);
                            offset1++;
                            }
                      }
                }
          }


  //The modified solutions are put back in the original population from
  best_population
        for (int a = 0; a < (best_population->size()); a++)
        {
              population->push_back(best_population->at(a));
        }


        return *population;

  }
```

Fig. 3.23: Pseudo-code for crossover

The cross-over and mutation rates are considerably lower after the first half of the user defined number of generations have passed. This is to ensure that the best solutions that have been obtained up to that point are preserved. The modified solutions are returned to the original population from the *elite_club*.

At the end of crossover and mutation, a new generation of hardware/software partitioned solutions is produced. The solutions of the newly evolved generation are prioritized, scheduled, evaluated and ranked as described in the preceding sections. In every iteration, during the selection stage, the very best solution is left unaltered and the worst solutions are discarded. The purpose of keeping the best solution out of the subsequent processes of GA is to not lose that solution. On the other hand, other good solutions with desirable characteristics are allowed to go through the subsequent stages of the GA to improve their characteristics. The objective of doing so is to obtain solutions which can replace the currently existing best solution. This process has been visually represented by Fig. 3.24.

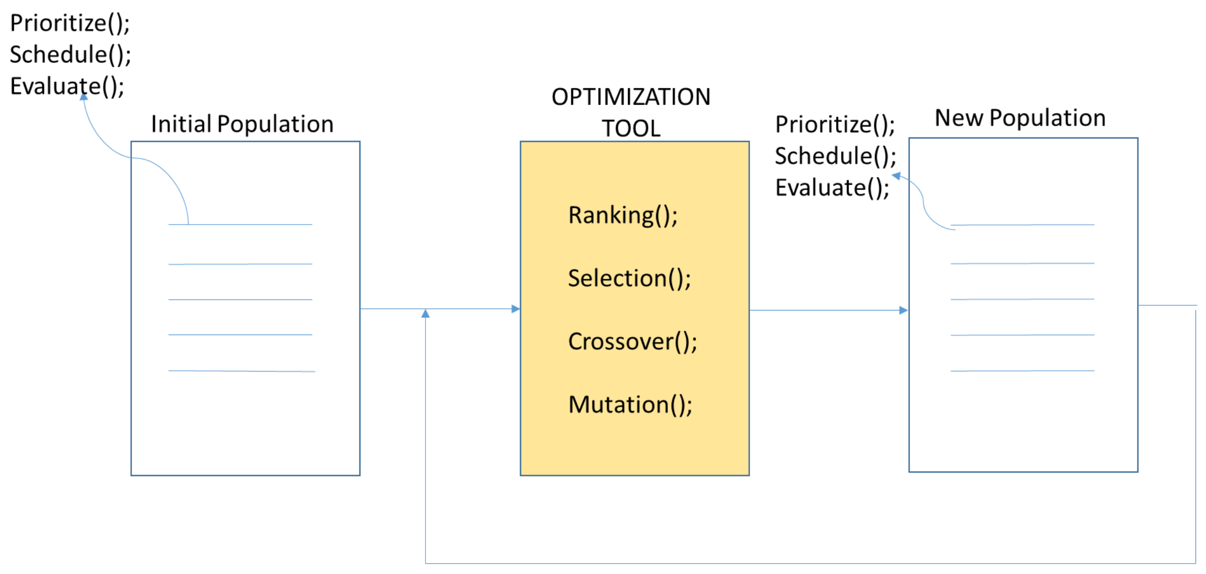


Fig. 3.24: Newly evolved solutions are prioritized, scheduled and evaluated prior to being passed to the GA.

3.3 HW/SW Partitioning methodology for CPU-GPU platforms

GPUs are being widely used as accelerators for scientific and computing-intensive applications and their merits as accelerators have been discussed in the 2nd chapter. However, designers face several challenges in partitioning applications on CPU-GPU platforms. Some of those challenges are addressed in this thesis by expanding the GA-based partitioning methodology presented in section 3.2 to cater to CPU-GPU platforms.

In addition to identifying the portions of the target application that will either be offloaded to the GPU or remain on the CPU, we seek to inform the designer "how" the application should be mapped on the GPU in an optimal fashion. Hence, to address the challenge of obtaining an optimal mapping of the application on the GPU, it becomes imperative to also consider the characteristics of heterogeneous programming frameworks. It is however important to note that the focus of our work is not on programming applications heterogeneously by using these frameworks.

Among the programming frameworks discussed before, CUDA and OpenCL are suitable for "lower-level" programming. They enable greater control over the architectural features of the platforms they are mapped on. Both frameworks are similar in their programming approach but CUDA is compatible only with NVIDIA GPUs. OpenCL, on the other hand, is vendor-independent and can be used to program applications for other accelerators as well such as FPGAs. We have picked OpenCL for our purposes due to its greater versatility. The role of OpenCL in our algorithm will be discussed in more detail after we describe the concepts of task and data parallelism. The following section briefly explains these concepts.

3.3.1 Task and Data Parallelism

Different kinds of parallelism are hidden in various applications. A traditional compiler is capable of exploiting Instruction Level Parallelism (ILP) but parallelism is also available at the coarser level of applications. The most popular kinds of coarse parallelism are as follows:

- Task-level parallelism: Application computation is divided into multiple tasks that operate in parallel on different data-sets. Tasks may be dependent upon one another, but a task whose data is ready can be executed in parallel with the tasks that are already running.
- Data-level parallelism: The computation is replicated into several equal tasks which operate in parallel on different data-sets.
- Pipeline-level parallelism: A computation is broken into a sequence of tasks that are repeated for different data sets. It is believed to be a general form of software pipelining [51].

In this work, emphasis was mainly on identifying task-level parallelism and data-level parallelism in applications and exploiting them to optimize application mapping (Fig. 3.25 and Fig. 3.26).

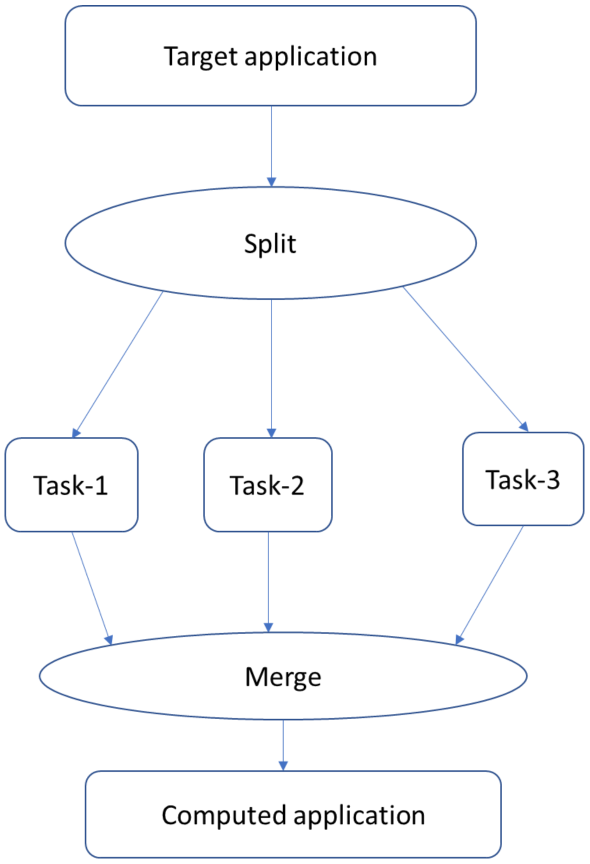


Fig. 3.25: Task parallel computing.

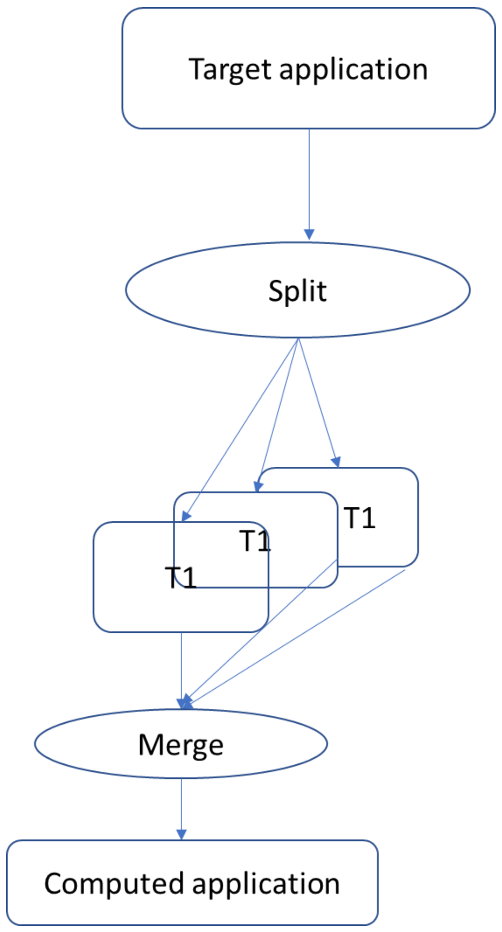


Fig. 3.26: Data-parallel computing.

GPUs are SIMD engines. Hence, it will be beneficial to execute data-parallel application parts on GPUs. This is however not a straight-forward decision in most cases as CPUs are also capable of executing data-parallel tasks. This makes it difficult for the designer to manually partition the application between the CPU and the GPU to achieve optimal performance. Hence, it is essential to develop a tool which can harness data and task parallelism present in applications for partitioning them. In addition to that, the tool should also be able to predict the performance of such partitions. The methodology required for evaluating partitions has already been developed. We need to adapt it for CPU-GPU platforms. The characteristics of the data structures, namely the processes and PEs were described earlier at the generic level. It is essential to now re-define them for CPU-GPU platforms.

3.3.2 Input Specification

Resource library: There are only two processing elements present in our target architecture, namely the CPU and the GPU. A typical discrete CPU-GPU platform has a PCI-e bus interconnection.

In addition to the characteristics discussed in the previous section, the structure concerning Processing Elements has the following additional characteristics:

- *maximum no. of work-items*
- *total available global memory*
- *total available local memory*

This information can be obtained for any device compatible with the OpenCL framework.

The PCI-e bus has the same characteristics that were described in the previous section. Those characteristics are now definite. In a typical CPU-GPU platform, the *contact_head* is the CPU and the *contact_tail* is the GPU. The *com_rate* of the device is related to the bus' bandwidth and the amount of data being transferred across it.

Application characteristics (Processes and Edges): The application to be optimized is specified as a DAG. Application processes have the same characteristics as discussed earlier in addition to a few key additions.

Every process now has a Boolean variable *SIMD* which indicates if a process has data-parallel capabilities. If *SIMD* for a process is true, that process can be executed on both the CPU and the GPU. If SIMD is false, it means the process can only be executed on the CPU.

In the random assignment process when initial solutions are being created, a process could be assigned either to a CPU or a GPU. From a different perspective, computational work for such platforms can be partitioned in two different ways: through functional decomposition and domain decomposition. In functional decomposition, the entire computational work is partitioned into functions/jobs. In domain decomposition, the data needed for solving the problem is partitioned. Both approaches are dependent upon one another. The subsequent text explains how domain decomposition was accounted for in our partitioning algorithm.

When a data-parallel task is identified (*SIMD*= true), it is split into “n” number of tasks where each task works independently on its own set of data. In Fig. 3.27, Task D is a SIMD task and can be split into a number of copies depending on the application constraints (mentioned later) imposed prior to the algorithm’s execution.

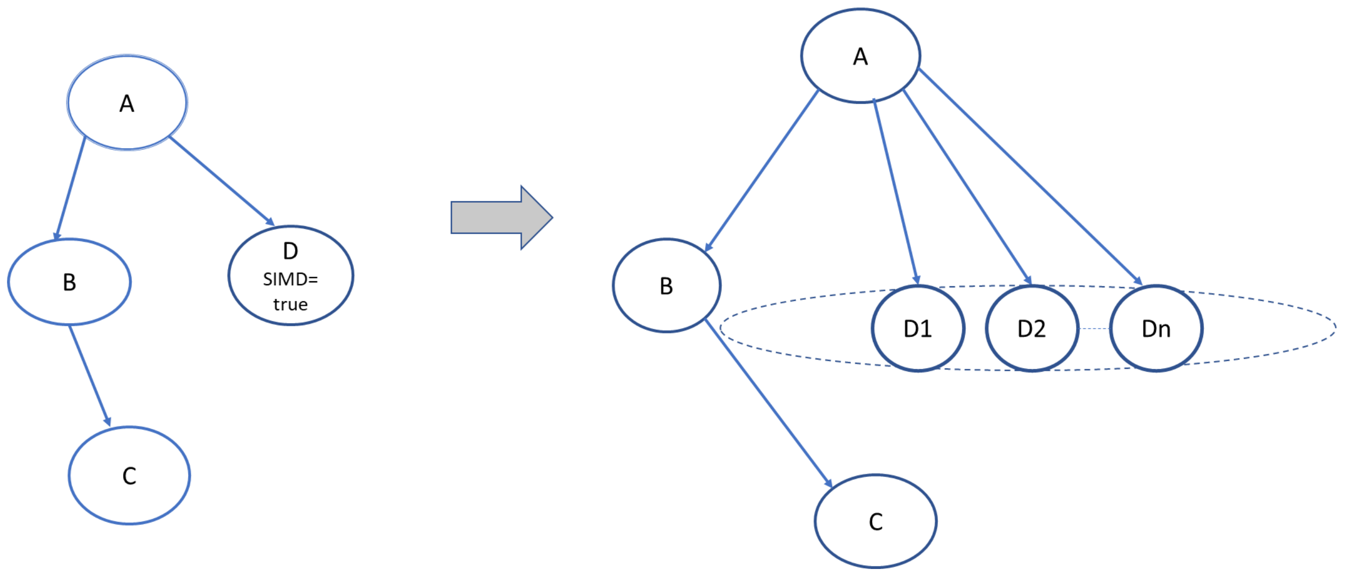


Fig. 3.27: SIMD task is split into a number of concurrent copies for offloading to the GPU

As mentioned earlier, in addition to specifying which processes will be offloaded to the GPU and which ones will remain on the CPU, we seek to describe how the processes offloaded to the GPU can be mapped on it. A method is needed to account for the various ways of implementing processes on the GPU.

A new data structure called *gpu_details* is also introduced to address additional properties of properties of processes. This structure essentially denotes the manner is which a process should be mapped on the GPU. Each SIMD process consists of a dynamic array of *gpu_details* called *process_config*. The properties of *gpu_details* are as follows:

- *number of concurrent copies*
- *work-items*
- *work-groups*
- *shared memory*

- *power consumption*
- *execution time of the process*

The significance of the properties mentioned above will be discussed in the subsequent section. If the process is assigned to a GPU, its relevant data (about number of concurrently executing copies, work-items, power consumption and so on) is extracted from a text file. This data is then used for the subsequent processes of the algorithm.

3.3.3 System-level Constraints

Every DAG node has processing times corresponding to the CPU and the GPU. Furthermore, if a process is being executed on the GPU, it will have additional information such as global memory usage, local memory usage, number of concurrently executing copies and number of cores used.

Every node contains a Boolean variable called *SIMD* which indicates if the sub-task is data parallel. If *SIMD* is true, the sub-task will have data-parallel capabilities and it won’t if *SIMD* is false.

- OpenCL can be used to obtain the values of global memory size, local memory size and the maximum number of work-items for every OpenCL-compliant device.
- The number of concurrently running copies of any data-parallel task is governed by the total number of iterations of the task in the sequential code, as given in Equation 6.

$$1 \leq \textit{No. of task copies} \leq \textit{No. of task-iterations} \qquad (6)$$

- The number of concurrently running task copies will determine the number of OpenCL work-items needed to process the copies. The total no. of work-items cannot exceed the maximum number of work-items that an OpenCL device can manage (Equation 7). It is necessary to select the optimal number of work-items because too few or excessive number of them will adversely affect the resource utilization of the device.

$$1 \leq \textit{No. of work-items} \leq \textit{Max. no. of work-items} \qquad (7)$$

- The number of created work-items are automatically put by OpenCL into work-groups. The work-group size is associated with the local memory size of the device. The collective size of all the work-groups in a program cannot exceed the device's local memory size (Equation 8). Similarly, the collective size of all the generated work-items corresponds to the amount of the device's global memory being used (Equation 9).

$$\textit{Memory}_{\textit{work-groups}} \leq \textit{Local memory size of the device} \tag{8}$$

$$\textit{Memory}_{\textit{work-items}} \leq \textit{Global memory size of the device} \tag{9}$$

- A data-parallel process' description also contains a Boolean variable called local-memory. If local-memory is true, the data relevant for the SIMD process will be placed in the PE's local memory and the computations will occur there. If local-memory is false, the SIMD process' data will be directly used from the global memory.

3.3.4 Characteristics of solutions and populations

The solutions of CPU-GPU co-design are represented in the same way as discussed in Section 3.2. A population of solutions is generated by randomly assigning processes to PEs. A solution or individual of a population is represented with an array where the array indices correspond to process indices and array elements correspond to the PEs i.e. either CPU or GPU.

The case concerning the edges is slightly unique. In a typical CPU-GPU platform (described in Chapter 2), the only mode of data transactions is the PCI-e bus. For determining the initial population, random assignment is done as explained in the previous section. This routine is described in the pseudo-code in Fig. 3.28.

```
Void ProcessAssignment(){
for (int i= 0; i < All_Processes.size(); i++)
       {
              //check which PE the process has been assigned to

                     //if PE is a CPU

                     if (PEId == 0 || process_config.size() == 0)
                     {
                            return process_execTime;

                     }

                     //if PE is a GPU
                     else if (PEId == 1)
                     {

                            //pick random process configuration
                            int gpuPick = rand() % (process_config.size());


                                   return (Random_pick.time_taken);

                     }
              }
       }
```

Fig. 3.28: Pseudo-code for CPU-GPU based process assignment

3.3.5 Priority assignment

The priority assignment scheme developed in the previous section is used for CPU-GPU based partitioning as well. The execution times, process deadlines and data transfer times are all accounted for to assign priorities to processes.

3.3.6 Scheduling

All the processes and edges are scheduled on their respective hardware resources after the priorities are assigned. If a source process and its sink process are both on the same PE, the data transfer time between those nodes is considered to be zero. If the source and sink processes are on different nodes, the data-transfer time is obtained by dividing amount of data to be transferred from the source to the sink node with the bandwidth of the bus. Fig. 3.29 shows the schedule of a simple application running on a CPU-GPU platform.

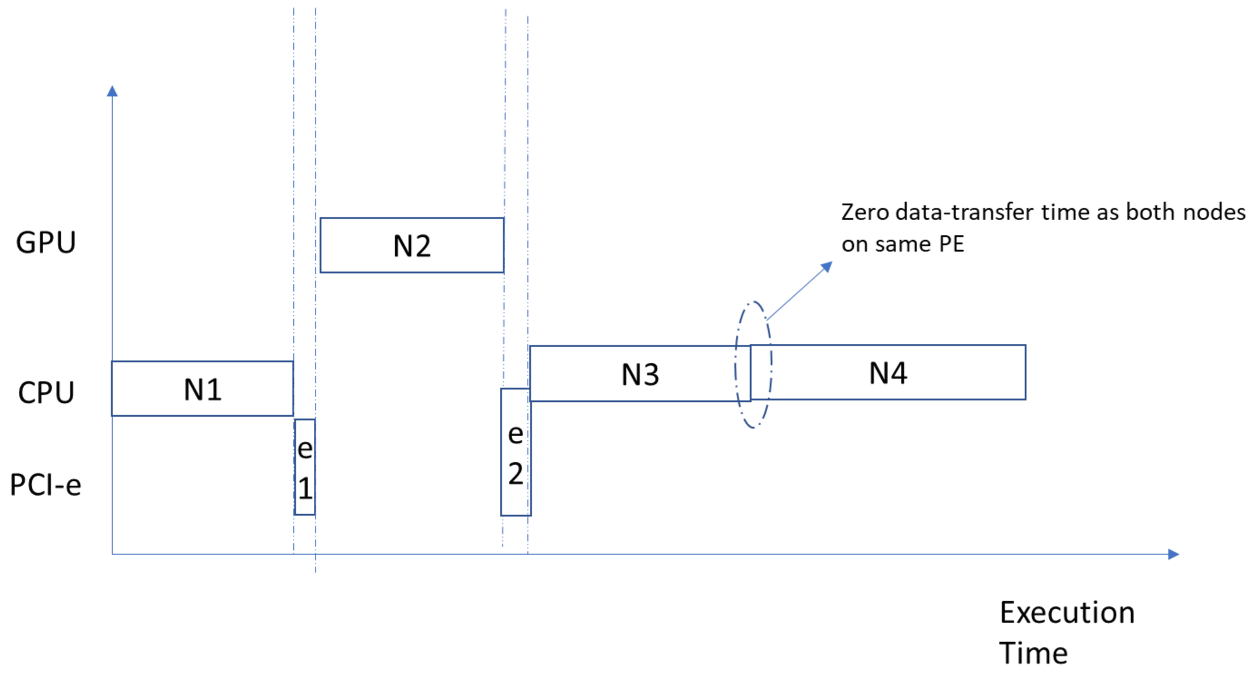


Fig. 3.29: Visual depiction of application scheduled on CPU-GPU platform

3.3.7 Solution Evaluation

When evaluating solutions for CPU-GPU platforms, the energy consumption must be taken into account as it is a very important characteristic of these platforms. The energy consumed is the product of a process' execution time and power consumption. The sum of the energy consumption of all the processes is used to calculate the total energy consumption of the application. The pseudo-code in Fig. 3.30 shows the routine for calculating the energy consumption of the application.

```
double total_Energy()
{
       double totalEnergy = 0;

       for (int i = 0; i < SortedProcesses.size(); i++)
       {

       totalEnergy += (SortedProcesses[i].finish_time - SortedProcesses[i].start_time) *
SortedProcesses[i].power;
       }

       return totalEnergy;

}
```

Fig. 3.30: Pseudo-code for calculating energy consumption

We have taken the power consumption of the CPU to be zero in this work because the power consumption of CPUs is insignificant when compared to GPUs.

Equation 10 describes the new function to evaluate the quality of solutions:

$$Total\ system\ cost = M_1 * \sum_{i \in process} Execution_{Time_i} + M_2 * \sum_{i \in process} Energy_{Consumption_i} + M_3 * \sum_{i \in process} (Execution - time - Deadline)_i \quad (10)$$

The values are normalized for subsequent processing by the Genetic Algorithm.

3.3.8 Partitioning GA for CPU-GPU platforms

The CPU-GPU partitioning GA has largely the same characteristics as the one described in Section 3.2. We review it here to explain how the algorithm works to produce CPU-GPU partitioning solutions desired by users. The characteristics of GA-based partitioning are summarized below:

Ranking

The solutions in the population are ranked according to the weights assigned to each of the target parameters. The solution with the smallest final score in the population is regarded as the best solution whereas the solution with the highest final score is considered as the worst solution.

Selection

The best solution is preserved and the second and third-best solutions are passed to the subsequent stages of the GA.

Mutation and Crossover

In mutation, a few processes from the chosen solutions are picked randomly and assigned to different PEs than the ones they were initially assigned. For crossover, two random points are picked along the length of the solution arrays and contents of both the arrays are swapped.

After these steps, all the solutions are re-evaluated. After mutation and crossover occur, a process may often get assigned to a PE it cannot be executed on. Hence, during the evaluation phase the solutions are re-assigned to a PE they can run on (Fig. 3.31).

```
for (int b = 0; b < mynodes->size(); b++)
            {
                  //if process cannot run on the GPU
                  if (mynodes->at(b).SIMD == false)
                  {
                        //but has been put there any way
                        if (solution->at(sourceProcess).elementid == ‘GPU’)
                        {
                              //put the process on the CPU
                              solution->at(sourceProcess) = ‘CPU’;
                        }
                  }
            }
```

Fig 3.31: Pseudo-code for rectifying PE assignment.

The over-all methodology for CPU-GPU partitioning we developed in this section can be visually represented in Fig. 3.32. The text file contains information about the various GPU-based configurations and parameters of the processes.

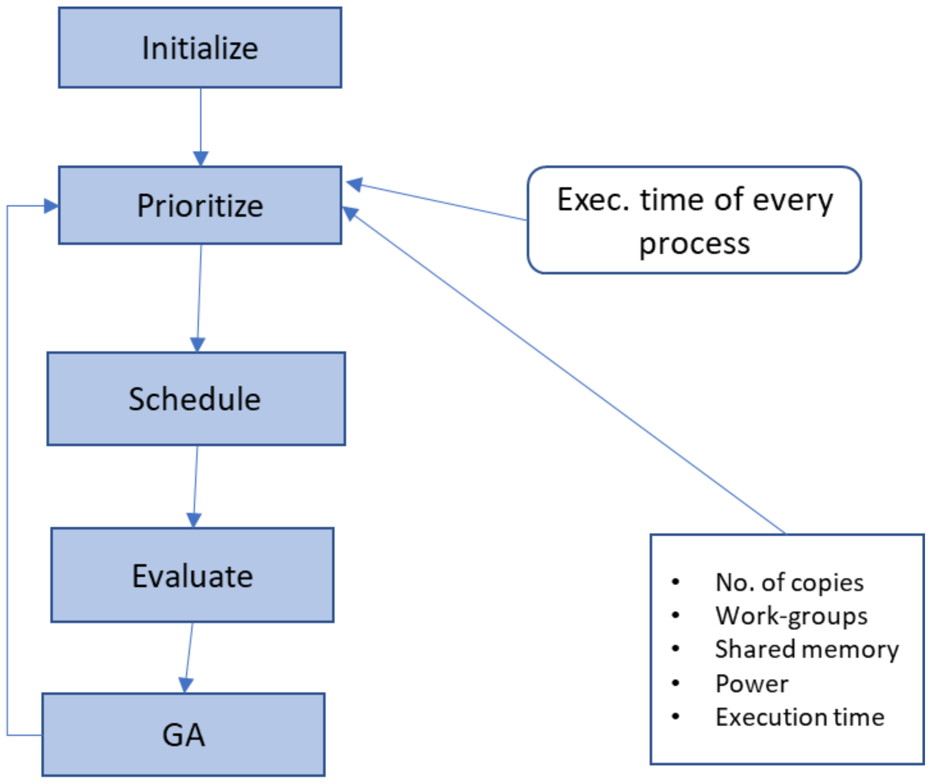


Fig. 3.32: Co-design methodology for CPU-GPU platforms

3.4 Parameter tuning for Performance Improvement

3.4.1 Introduction

The successful application of Genetic algorithms depends greatly upon the parameters of the algorithms which vary with the context and the problems. An extensive body of work exists which sheds light on different techniques of controlling the parameters of evolutionary algorithms to exploit their potential.

Some parameter control methods involve adjusting the parameter values based on user-defined rules. Other strategies are characterized by "blending" the parameters with the solutions so that they evolve with them. In some approaches, the behavior of the GA is monitored in each run and the parameter values are adapted accordingly. These approaches are characterized by the fact that the probability of applying a parameter value is proportional to the quality of that value.

Aleti et al. proposed a method for adapting real-valued parameter ranges. The continuous-valued parameters are partitioned into two equal intervals. The best-performing interval is subdivided by splitting in the middle and the poorly-performing intervals are merged with their poorly performing neighbors.

A strategy involving Bayesian networks was proposed for parameter control in genetic algorithms. A user-defined number of algorithm instances are run independently and the results from every iteration are reported to an algorithm manager. After every iteration, the performance of the current solutions is checked against those of previous iterations. Solutions that cross a user-defined threshold are deemed successful. Bayesian networks were used as a parameter effect assessment strategy and to model the relationship between algorithm parameters and their effect on the performance of the algorithm [52].

We were inspired by Aleti's work and devised our own parameter tuning methodology to harness the full potential of genetic algorithms.

3.4.2 Proposed parameter tuning strategy

The solutions that were generated had values corresponding to the following targets:

- The amount by which the solution violates the deadline.

- Total execution time of the solution.
- Total cost of the resources used in the system (Processors + Communication links).
- Total energy of the application.

The most desirable solution has the smallest sum of parameter values among all the solutions in the set. Each solution had a total value associated with it and these values were used to compare solutions and select the most suitable solution. The initial plan was to save the best solution and apply crossover and mutation to the remaining solutions which meet the selection criteria. However, when the selection criteria are strict, only one solution is selected and crossover cannot be performed.

The solution set was assessed in several ways:

(i) Initial plan: Instead of saving the best solution, all the good solutions (including the best one) were selected for mutation and crossover. The resulting solutions produced poorer total values and the good solutions were lost in the process.

(ii) The undesirable solutions were selected for mutation and crossover. No major improvement was detected when compared to (i).

(iii) Only the best solutions were chosen for mutation and crossover. Crossover was not being done because in majority of the iterations, only one solution was being chosen. A small solution set and a large number of iterations produced the best results.

(iv) A large solution set was used for mutation and crossover. No change in the quality of the solutions was observed when compared with (i), (ii) and (iii).

In all cases, the best solutions weren't consistently producing a single value towards the end of the pre-defined number of generations. Hence, a parameter tuning strategy was developed to overcome the disparities that result from the various aspects of both the genetic algorithm such as the mutation and crossover rates, the length of the chromosome/solution, the threshold criteria/score which solutions should exceed/beat and the population of solutions.

The factors which seem to influence the quality of the GA most are the number of iterations, the threshold score and the crossover and mutation probabilities.

To reiterate, the GA consists of the following steps:

(i) Rank all the solutions in the population in decreasing order of quality (The first solution is the best and the last solution is the worst in the given population).

(ii) Pick the solutions whose final scores are lower than or equal to the threshold score.

(iii) The best among these good solutions is preserved and left untouched.

(iv) The remaining solutions in this elite group undergo mutation and crossover according to user-defined probabilities.

(v) All the solutions are evaluated and ranked again.

(vi) If the newly generated solutions are better than some of the existing solutions in the population, they replace those solutions. The inferior solutions are discarded from the new population.

(vii) The process is repeated with the newly formed population.

In our parameter tuning strategy, every GA runs for a fixed number of iterations. Additionally, the population size of each GA is very small and is the same across GAs. The adjustable parameters for each GA (mutation rate, crossover rate and target score) are different. Every GA is allowed to run independent of the other GAs. Every GA stores saves its best solution (and its properties) after every iteration in a text file (Fig. 3.33). The saved values of all the GAs are compared with one another, the absolute best solution is picked and its associated parameters are saved. It is checked if the best solution meets the constraints set by the user. If the constraints relating to the energy consumption, total execution time and deadline violation time are met, the program ends and the best solution is picked and implemented. Otherwise, a GA is run again with a much larger population size and for a larger number of iterations (Fig. 3.34).

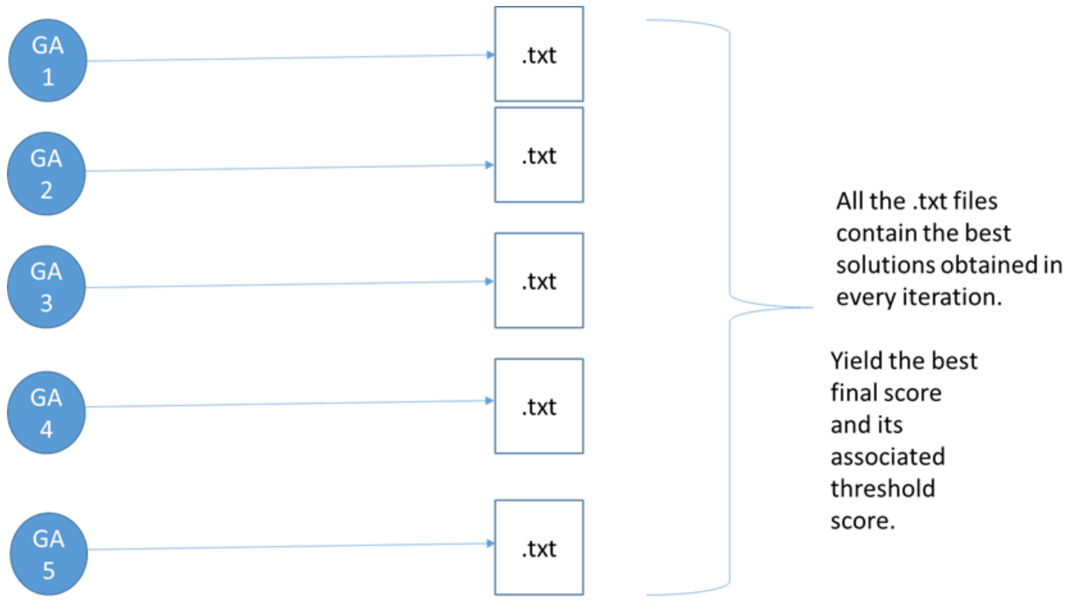

GA 1
GA 2
GA 3
GA 4
GA 5
.txt
.txt
.txt
.txt
.txt
All the .txt files contain the best solutions obtained in every iteration.
Yield the best final score and its associated threshold score.

Fig. 3.33: Writing the best scores of each GA to text files.

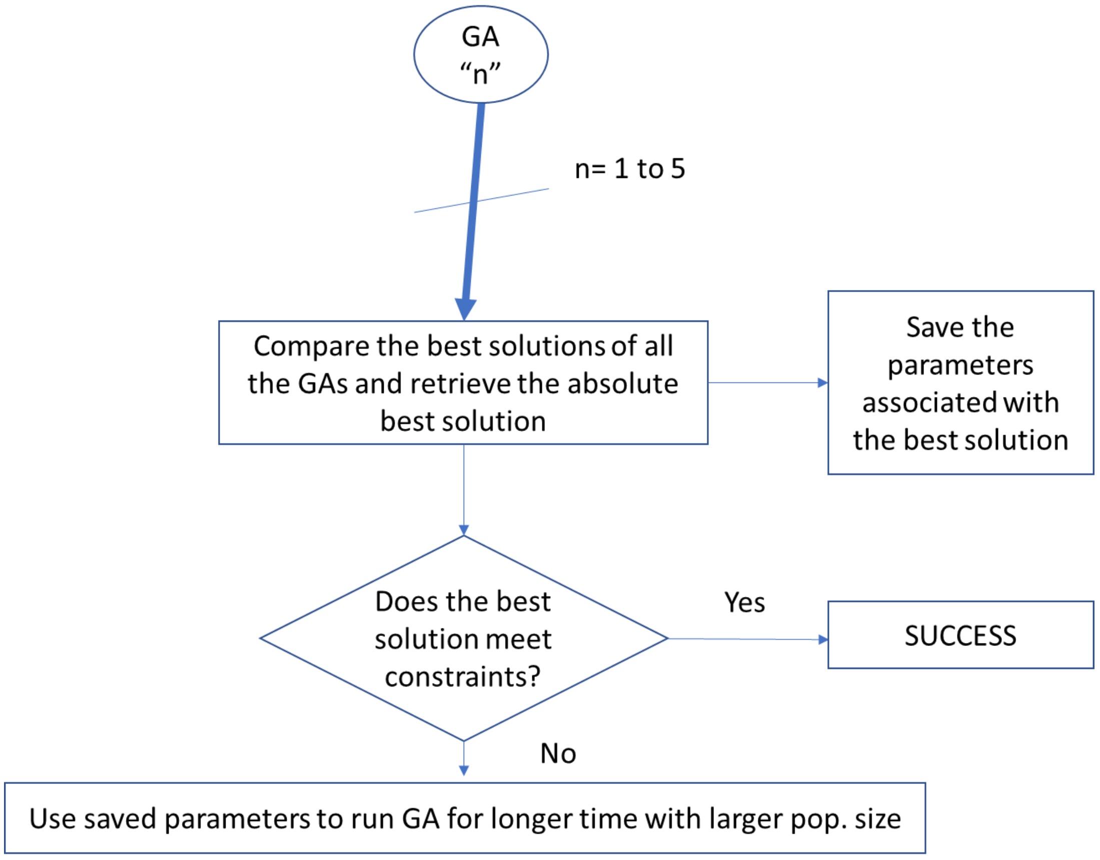

GA "n"
n= 1 to 5
Compare the best solutions of all the GAs and retrieve the absolute best solution
Save the parameters associated with the best solution
Does the best solution meet constraints?
Yes
SUCCESS
No
Use saved parameters to run GA for longer time with larger pop. size

Fig. 3.34: Flowchart of our parameter tuning strategy

It is essential to allow the GA to run for a reasonably long period of time to see its true potential. There was one constant pattern in all the trails which were performed: a low threshold score does not give favourable results. More favorable results will be obtained if the threshold score corresponds to intermediate scores. This enhances the probability of more solutions to get chosen and altered. This will in turn lead to solutions of a better quality.

Crossover and mutation for all scenarios were carried out in the same way. The probability of crossover and mutation were higher during the first half of the total number of generations. After the first half of the total number of generations elapse, the probabilities of crossover and mutation are set slightly lower than the first half to settle at the solutions found near the end of the GA run.

Chapter 4

Experiments and Results

4.1 Target Architecture- Details

The applications highlighted in the case studies were run on the heterogeneous computing platform provided by CMC Microsystems. It is referred to as the Heterogeneous Parallel Platform (HPP). It is highly customizable and extensible and can be described as a single node computing system integrating a variety of different types of computational units. The HPP consists of multi-core CPUs, many-core GPUs, application-specific many-core CPUs and FPGAs in a single pre-validated platform.

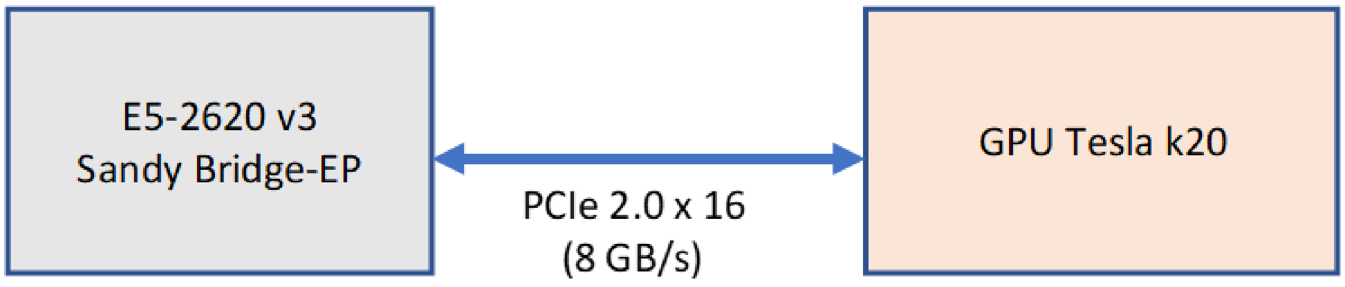


Fig. 4.1: Target architecture

This platform consists of 2 Sandy-Bridge CPUs, the Xeon Phi CPU, Altera's Nallatech FPGA, NVIDIA's Tesla k20 and k620. The two GPUs, the FPGA and the Xeon Phi CPU are meant to be used as accelerators on this platform [53].

For the purposes of our research, the focus was exclusively on the Sandy-Bridge processor connected the Tesla k20c via the PCI-bus (Fig. 4.1).

The Sandy-Bridge Processor (Intel® Xeon® Processor E5-2620) was operated as the host processor where the Tesla k20 GPU functioned as the accelerator. The Tesla k20 GPU has 2,496 CUDA cores, its max. memory size is 5 GB and its bandwidth is 208 GB/sec. PCIe 2.0 x 16 acts as its host interface and its maximum power is 225 W.

4.2 Synthetic Graphs

4.2.1 Synthetic graph with ten nodes

The methodology developed in Chapter 3 has been used to test the partitioning of the synthetic graph depicted in Fig. 4.2.

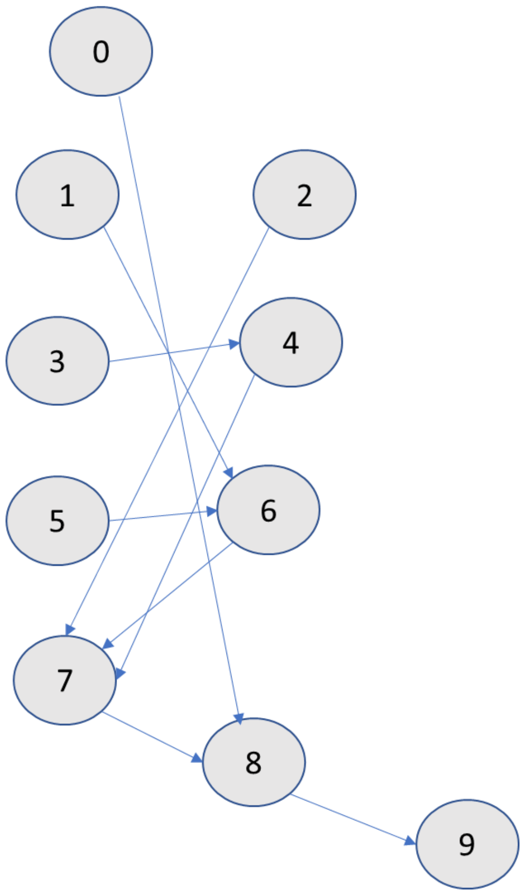


Fig. 4.2: Synthetic graph with 10 nodes. Deadline= 300 units

The library for the graph in Fig. 4.2 is shown in Table 4.1.

Table 4.1: CPU and GPU specification details for the graph in Fig. 4.2

Node	CPU Exec. Time (sec.)	Copies	Work-items	Work-groups	GPU Exec. Time (sec.)	Power (W)	Memory (shared)
0	41	--------	--------	--------	--------	-------	---------
1	34	1	4	2	7	42	1
2	69	--------	--------	-------	--------	--------	--------
3	78	3	12	6	30	206	0
4	62	--------	---------	--------	--------	--------	---------
5	5	2	16	8	6	35	0
		1	4	2	1	173	0
		2	16	4	1	47	1
6	81	--------	---------	--------	--------	--------	---------
7	61	1	10	5	20	211	0
		2	6	3	19	92	0
8	95	--------	--------	--------	---------	---------	---------
9	27	3	9	3	3	196	1

The results indicate that as expected, the execution time is inversely proportional to the energy consumed by the processes of the synthetic task graph. The application developer can choose from a variety of solutions, depending upon his/her preferences and requirements. Furthermore, in this section, the synthetic graphs for different configurations of the GA are shown separately to emphasize and depict the impact of minute changes in the GA's configurations on the quality of the solutions.

The graph in Fig. 4.2 was fed to the algorithm, executed and "all" the best solutions were picked for mutation and crossover. Fig. 4.3 illustrates the changes undergone by the absolute best solution over a span of 100 iterations.

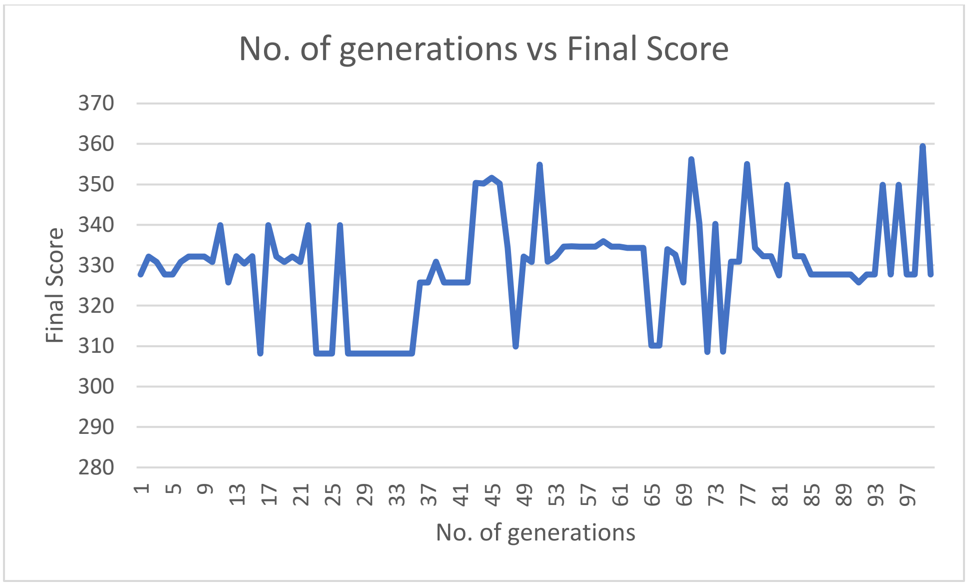


Fig. 4.3: Variations in the quality of the absolute best solutions

As stated in chapter 3, a lower final score corresponds to a superior solution. As Fig. 4.3 illustrates, our algorithm found its best possible solution as early as the 13th generation but is lost in the following iterations, before being recovered briefly and then lost again. This fluctuation in the quality of the best solutions is both detrimental and wasteful. It is plausible that the fluctuations are preventing the search for an even better solution than the one obtained in the 13th iteration. If the solution obtained in the 13th iteration is indeed the best possible solution, the number of generations can be drastically cut down. This will consequently improve the overall execution time of the algorithm.

This was immediately observed with a change in the strategy of the algorithm. Unlike the previous instance where all the desirable solutions were undergoing mutation and crossover, the best of the desirable solutions was preserved and the remaining desirable solutions were allowed to undergo mutation and crossover. The quality of the best solutions over a period of 100 iterations with this change is depicted in Fig. 4.4.

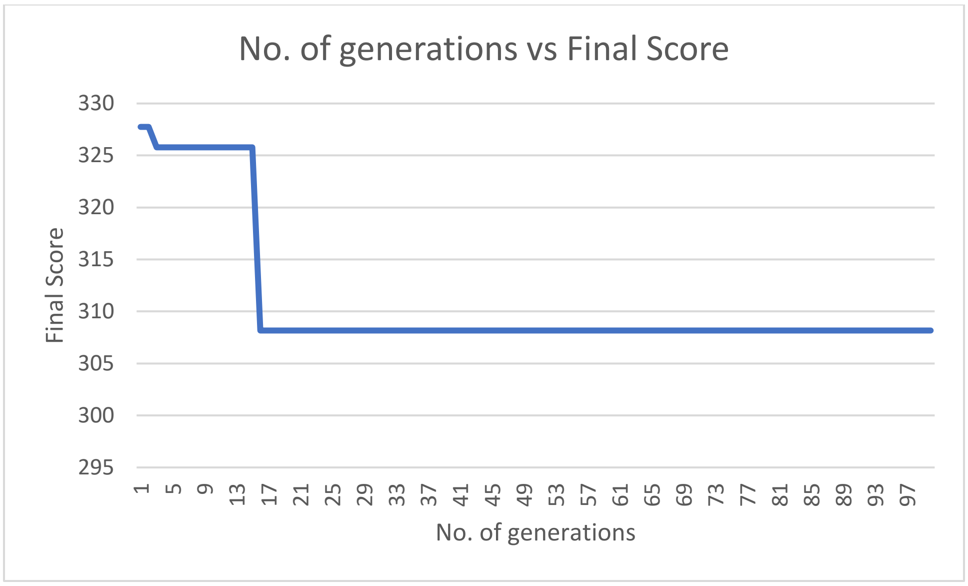


Fig. 4.4: Absolute best solution was retained after every iteration

As depicted in Fig. 4.4, the best possible solution (as in the previous case) was found in the 13th iteration itself and there is no need for executing the algorithm for 100 iterations.

The effect of this change in the algorithm's output prompts one to think that genetic algorithms have vast potential for optimization which is often left untapped by not properly using them. Genetic algorithms are complex and are comprised of multiple parameters, namely the population size, the mutation rate and the crossover rate. There are also parameters associated with the application itself, such as the threshold score. We observed the impact of a few other variables, namely the threshold/target score and the population size.

The threshold score in the previous two instances was 450. Fig. depicts how the solutions space was altered when the target score was changed to 370. The best solution is the same as the previous two cases but was determined in iteration 17, which is a few iterations more than the previous case. This shows that as the target score is varied, the solution space undergoes a change.

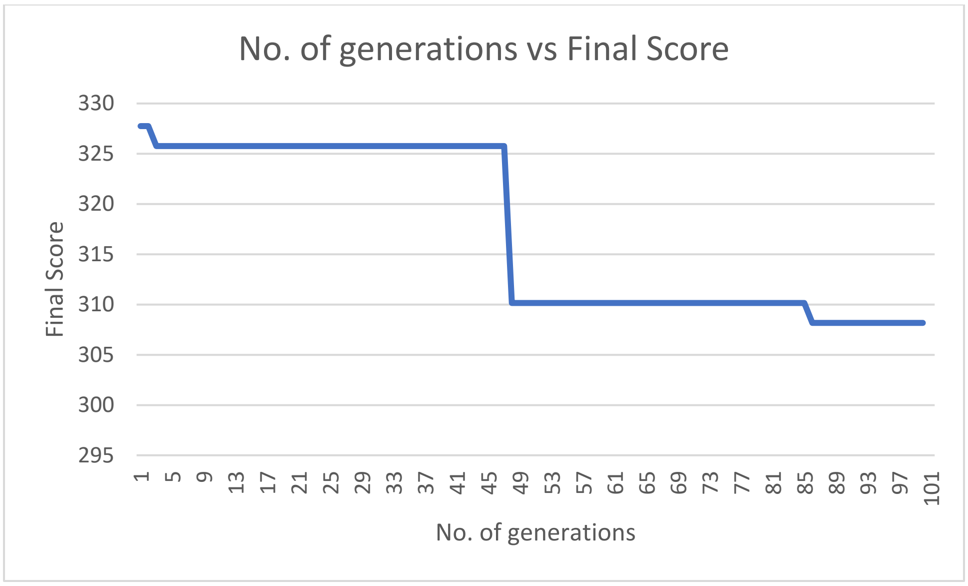


Fig. 4.5: The best solutions when the target score is made stricter.

The impact on the solution space was observed by altering the population size. The population size was 5 in the previous instances and it was changed to 10. It can be argued that a greater population size will enable greater diversity in the solutions. A greater diversity could allow faster arrival at the optimal solution. The following figures demonstrate the impact of population size on the solutions space. When the target score is 450 and the population size is 10, the optimal score is arrived in approximately 7 iterations. When the target score is 370 and the population size is 10, the optimal score is reached even quicker within 5 iterations. Through experiments and multiple GAs running in parallel, it was concluded that a target score of 370 leads to the quickest arrival at the best solution for this particular application. Table 4.2 depicts two of the solutions produced by the algorithm after processing the graph with a target score of 370.

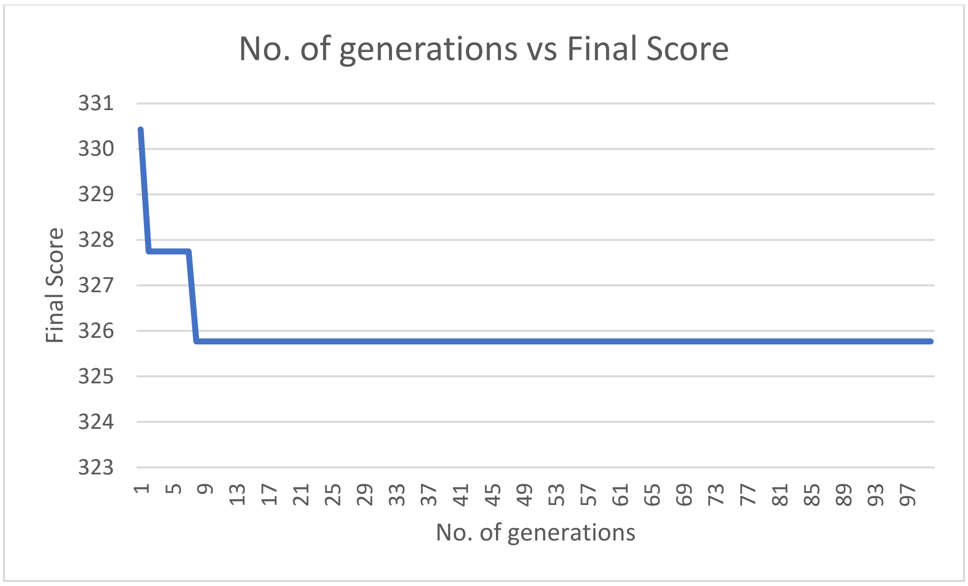


Fig. 4.6: Variation in quality of best solutions for a population size of 10 and a target score of 450.

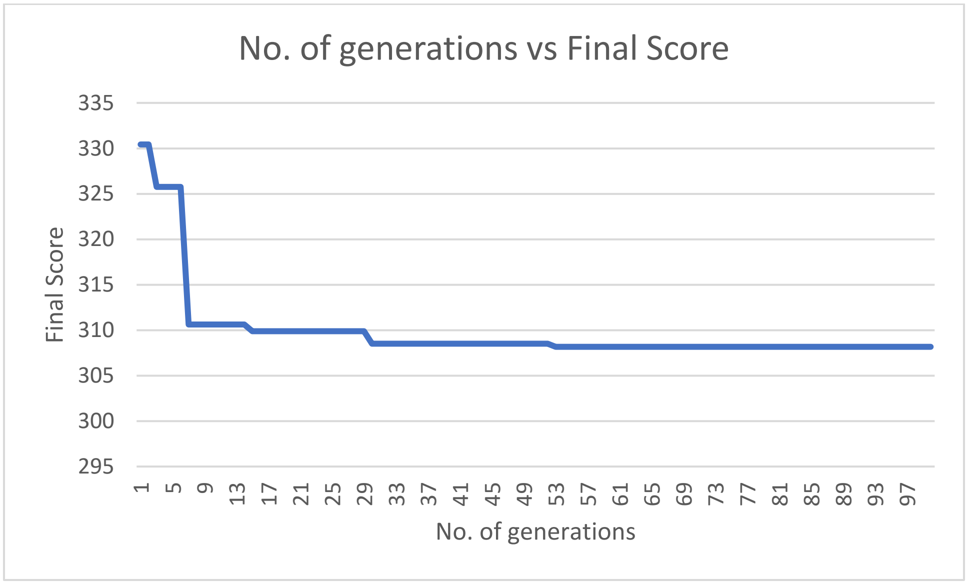


Fig 4.7: Best solutions in every iteration for a population size of 10 and a target score of 370

Table 4.2: Representative solutions for Fig. 4.7 (without normalization)

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
308.17	0000000101	283	2336	0
372.13	0101000000	301	7074	1

These results reiterate the claim made in an Chapter 3 that changes in the independent variables of the algorithm induce changes in the quality of the solutions. The results presented also prove that the parameters are dependent on each other rather than independent. Furthermore, the results emphasize that the performance of the algorithm is difficult to intuitively predict and warrants examination from a more unique perspective. The utility of the parameter tuning strategy outlined in one of the previous sections is thus confirmed.

Changes were also observed when the number of processes and edges in the task graph are varied. It was observed through repeated experiments that mutation and crossover rates impact the quality of solutions. Mutation and crossover rates influence the length of the chromosome and vice-versa. Higher mutation and crossover rates lead to more aggressive and drastic changes in the chosen solutions. The methodology dictates that as the first half of the predetermined number of iterations are completed, the mutation and crossover rates should be lowered (100 – original mutation/crossover rate). This ensures slower changes in the solution space and facilitates settling upon an optimal solution. As a result, the mutation and crossover rates were set between 50 % and 70 %. Population size is an obvious tunable parameter because a greater population size ensures greater diversity. A greater population size could ease the process of finding optimal solutions. The designer's intuition ought to be used to determine a range of values and combinations for the parameters influencing the quality of solutions.

4.2.2 Synthetic graph with 30 nodes

Four Genetic algorithms were executed with varying target scores for different parameter constraints. The population size in each of the GAs was set to 5 because it was found through experimentation that this population size was sufficient in producing desirable solutions. Through repeated trials, it was also discovered that each GA should be allowed to run for 100 iterations to fulfill its potential. The algorithm was used to find the optimal process allocation of a 30-node randomly generated task graph (Fig. 4.8) on the CPU-GPU platform. The input specifications of the graph in Fig. 4.8 are presented in Table 4.3. In the tables containing the representative solutions, a process mapped to the GPU is denoted by 1 and a process mapped to the CPU is denoted by 0. The sequence of 1's and 0's corresponds to the sequence of processes in the task graph.

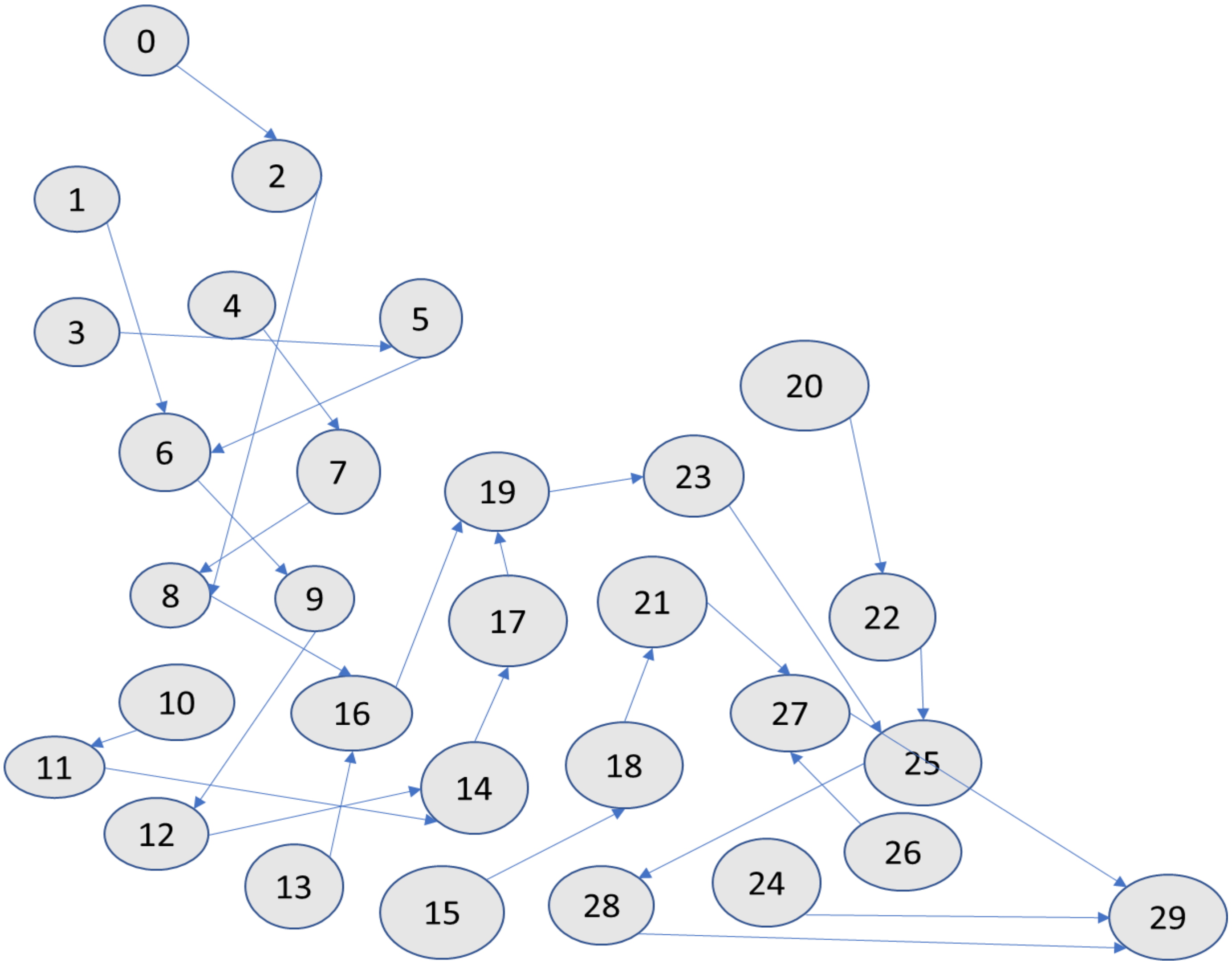


Fig. 4.8: 30-node task graph. Deadline= 480 seconds.

Table 4.3 depicts the specification library for Fig. 4.8.

Table 4.3: CPU and GPU specification details for the graph in Fig. 4.8

Node	CPU Exec. Time (sec.)	Copies	Work-items	Work-groups	GPU Exec. Time (sec.)	Power (Watts)	Memory (shared)
0	41	-----	-----	-----	-----	------	------
1	34	3	12	6	9	30	0
2	69	-----	-----	-----	-----	------	------
3	78	3	6	3	3	160	0
		4	10	2	25	95	0
		2	12	2	23	124	0
4	62	-----	-----	-----	-----	------	------
5	5	10	10	2	2	204	0
6	81	-----	-----	-----	-----	------	------
7	61	2	9	3	9	150	0
		3	8	4	14	158	0
		5	20	2	5	163	1
		3	6	1	30	100	0
8	95	-----	-----	-----	-----	------	------
9	27	2	4	1	8	80	0
		4	4	2	13	90	0
		11	22	2	5	146	1
10	91	-----	-----	-----	-----	------	------
11	2	3	18	9	1	200	1
12	92	-----	-----	-----	-----	------	------
13	21	3	10	5	9	157	0
		6	50	5	3	170	1
		3	6	1	13	46	0
14	18	-----	-----	-----	-----	------	------
15	47	3	10	5	14	94	0

16	71	-----	-----	-----	-----	------	------
17	69	3	20	10	22	100	0
18	67	-----	-----	-----	-----	------	------
19	35	3	9	6	12	100	0
20	3	-----	-----	-----	-----	------	------
21	22	3	10	5	10	39	0
		2	9	3	1	182	1
		2	6	3	17	35	0
22	73	-----	-----	-----	-----	------	------
23	41	3	12	6	13	118	0
		2	10	5	13	118	0
		4	10	2	4	150	1
24	53	-----	-----	-----	-----	------	------
25	47	2	10	5	20	66	0
26	62	-----	-----	-----	-----	------	------
27	37	2	10	2	12	70	0
28	23	-----	-----	-----	-----	------	------
29	29	3	10	5	13	50	0
		3	20	4	8	105	1
		2	20	2	2	195	1

Fig. 4.9 shows how the impact of different threshold scores on the performance of the GA when all the target parameters are assigned equal weights. Table 4.4 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.9.

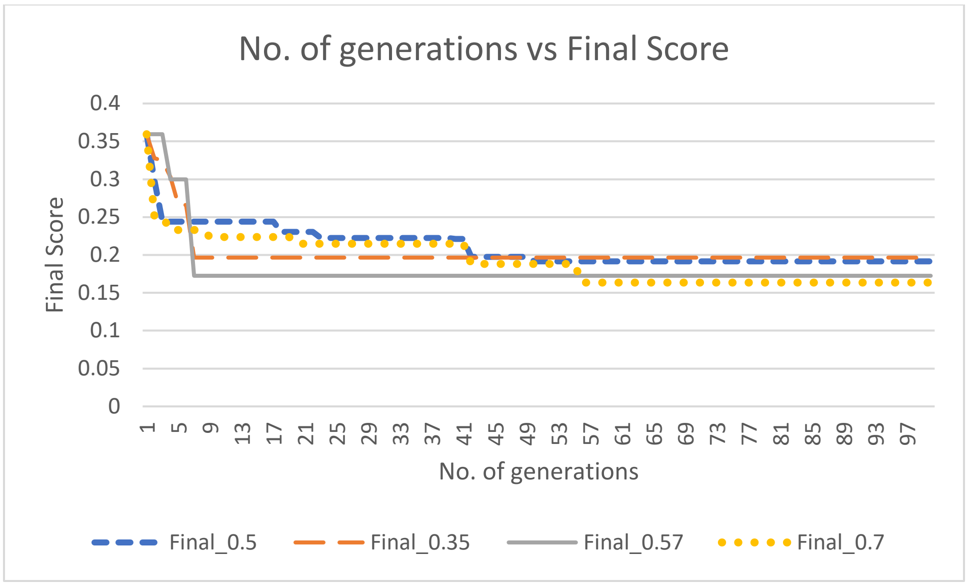


Fig. 4.9: Energy- 33%, Exec. time- 34% and Deadline violation- 33%

Table 4.4: Representative solutions for Fig. 4.9

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.16	0101010101010001010100010100001	466	11766	0
0.17	0101010101000001010101010010001	475	10755	0

Fig. 4.10 depicts the relationship between the energy and the execution time of the solutions for one of the cases of Fig. 4.9(Threshold score= 0.5). As Fig. 4.10 shows, the energy consumption of the solutions is inversely proportional to their execution times.

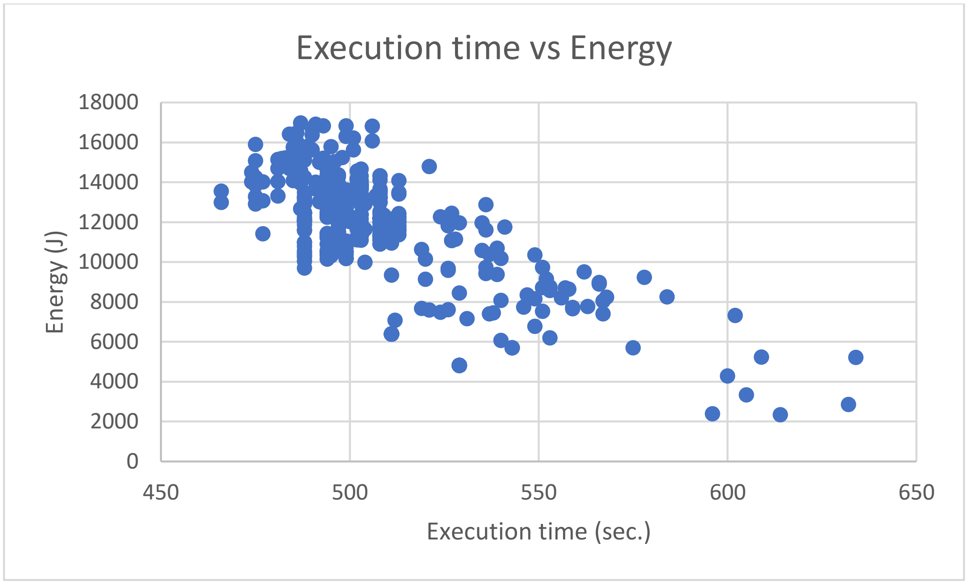


Fig. 4.10: Relationship between Energy and Execution time

In Fig. 4.11, "mxcx" corresponds to mutation and crossover rates. For example, m50c50 corresponds to a mutation rate of 50 % and a crossover rate of 50 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The representative solutions of the GAs of Fig. 4.11 can be seen in Table 4.5.

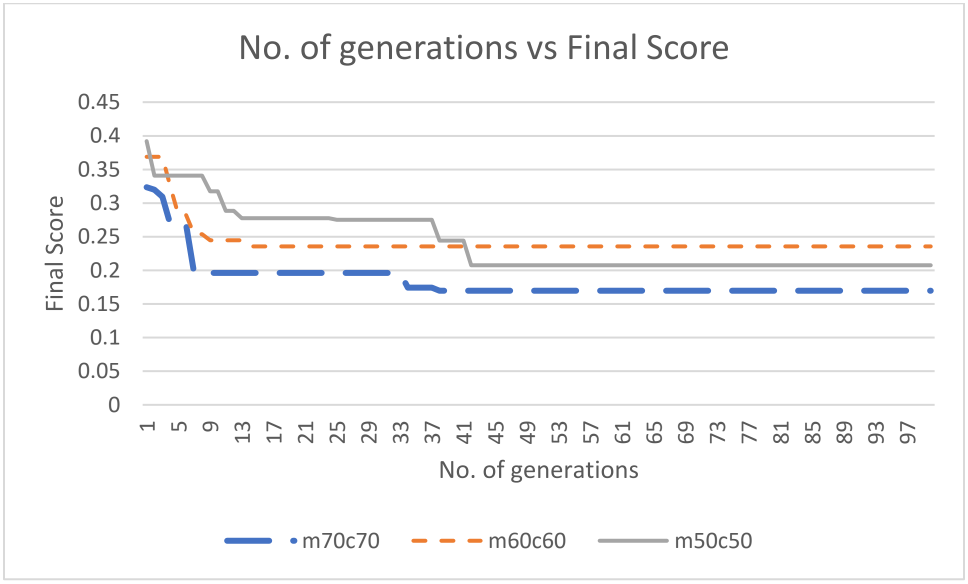


Fig. 4.11: Energy- 33%, Exec. time- 34% and Deadline violation- 33%. Target Score= 0.7

Table 4.5: Representative solutions for Fig. 4.11

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.17	010100010100000001010101010101	466	8597	0
0.28	010101010101000101010101010101	480	11721	0

Fig. 4.12 shows how the impact of different threshold scores on the performance of the GA when energy is assigned a weight of 80 %. Table 4.6 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.12.

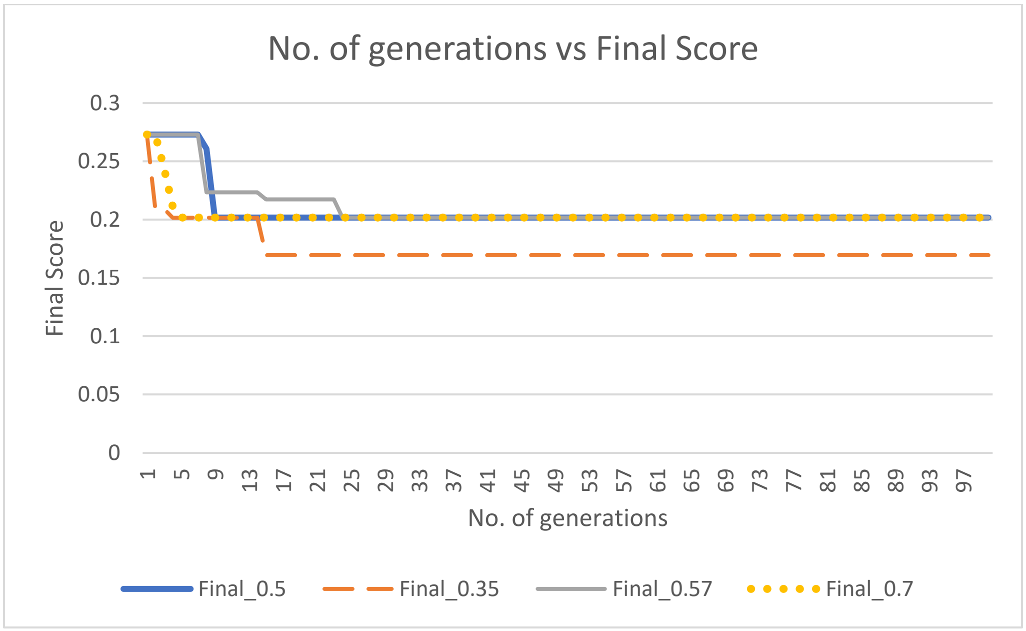


Fig. 4.12: Energy- 80%, Exec. time- 19% and Deadline violation- 1%

Table 4.6: Representative solutions for Fig. 4.12

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.17	010100000000000000000000000000	642	750	162
0.20	000000000000000000000000000000	717	0	237

Fig. 4.13 depicts how the mutation and crossover rates and population sizes impacted the solution space when energy was assigned a weight of 80 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The representative solutions of the GAs of Fig. 4.13 can be seen in Table 4.7.

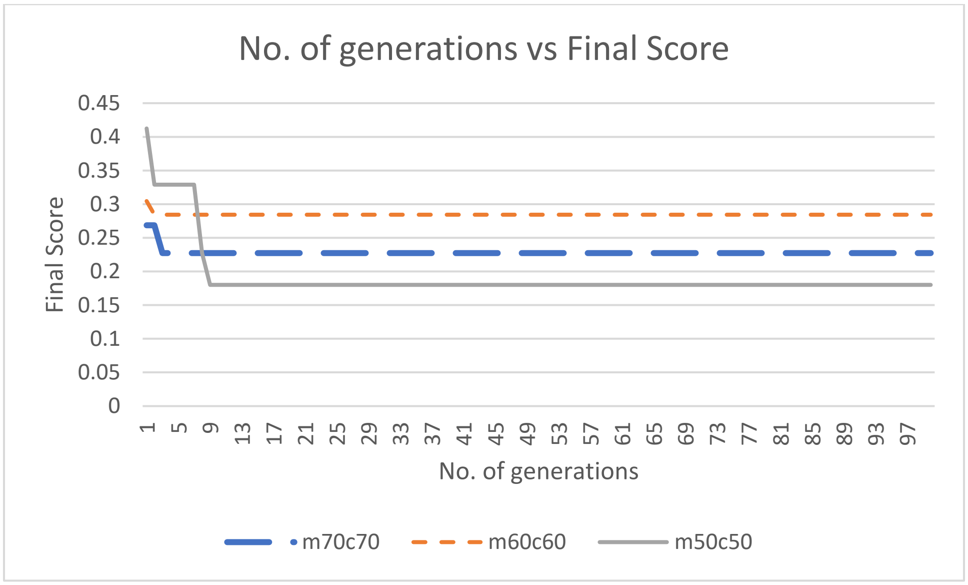


Fig. 4.13: Energy- 80%, Exec. time- 19% and Deadline violation- 1%. Target Score= 0.7

Table 4.7: Representative solutions for Fig. 4.13

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.24	000100010100000001010101010000	502	7305	22
0.23	000100000000000000000001000100	623	1920	143

Fig. 4.14 shows how the impact of different threshold scores on the performance of the GA when exec. time is assigned a weight of 80 %. Table 4.8 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.14.

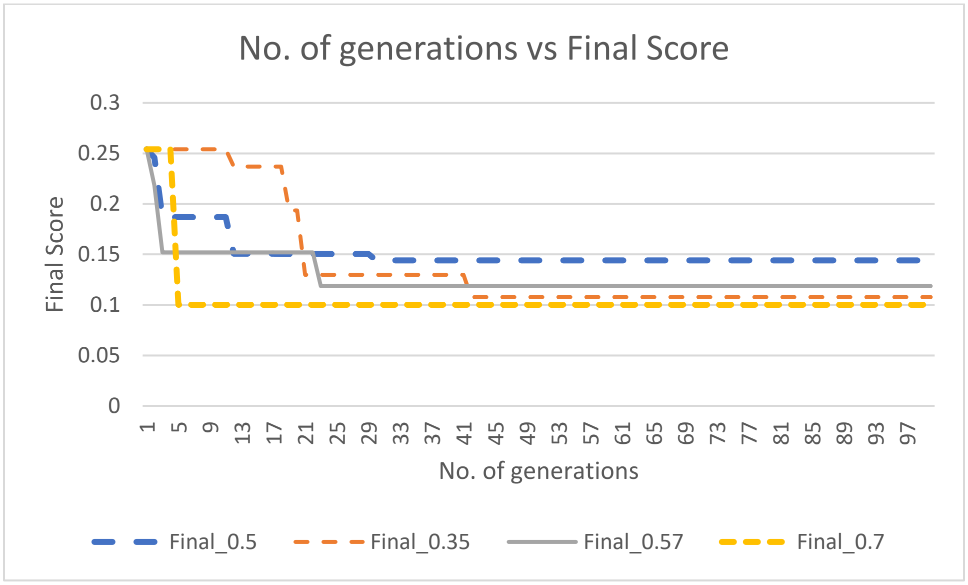


Fig. 4.14: Energy- 19%, Exec. time- 80% and Deadline violation- 1%

Table 4.8: Representative solutions for Fig. 4.14

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.10	0101000101000001010101010010001	466	11258	0
0.14	0101010101010101010101010101010101	477	12188	0

Fig. 4.15 depicts how the mutation and crossover rates and population sizes impacted the solution space when exec. time was assigned a weight of 80 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate

was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.15 are shown in Table 4.9.

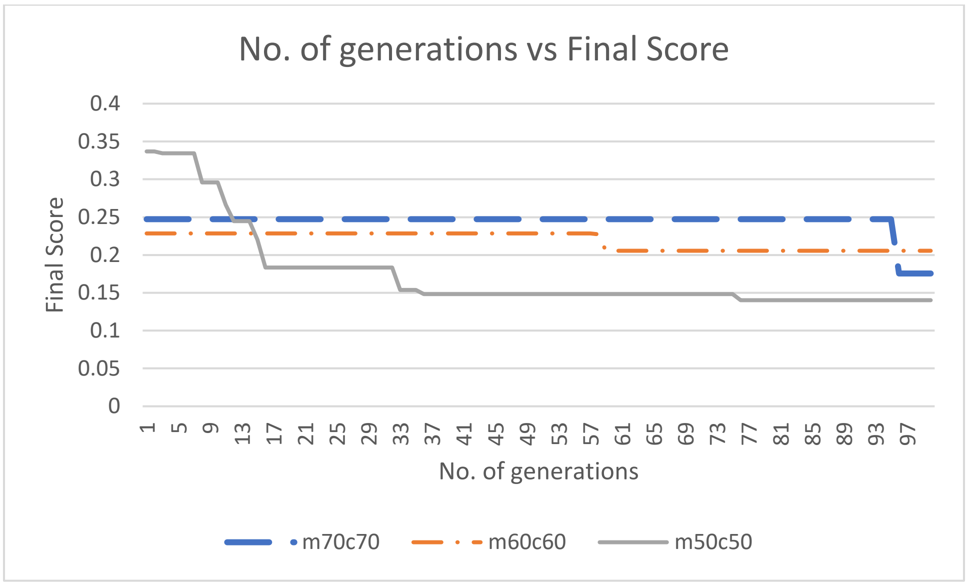


Fig. 4.15: Energy- 19%, Exec. time- 80% and Deadline violation- 1%. Target Score= 0.5

Table 4.9: Representative solutions for Fig. 4.15

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.14	010100010100000101010101010001	466	11119	0
0.20	000100010100000001010101010000	502	7097	22

The results in the tables (Tables 4.5-4.9) and figures (Figs. 4.9-4.15) above confirm that minor changes in establishing the threshold score, mutation rate, crossover rate and population size have a considerable impact on the performance of the genetic algorithm and the quality of the solutions produced.

4.3 Benchmarks for Heterogeneous Computing

Special benchmarks are needed to gain insight into the characteristics and capabilities of heterogeneous computing systems. Several benchmark suites have been created to fulfill the need of comparing the architectures and programming environments of such systems against similar systems.

Parboil and Rodinia are two of such benchmark suites. The applications in Rodinia have been implemented for both GPUs and multicore CPUs using OpenCL, CUDA and OpenMP. The applications span the domains of medical imaging, bioinformatics, data mining and molecular dynamics. Every application is unique in terms of inherent architectural characteristics, parallelization capabilities and power consumption. Parboil covers throughput computing applications in the fields of image processing, biomolecular simulation, astronomy and fluid dynamics. The applications have been implemented serially (in C and C++), OpenMP, OpenCL and CUDA. It provides multiple versions of the applications with varying levels of optimizations. This gives compiler writers the facility of evaluating source and compiler optimizations on different architectures [28, 54 and 55].

We picked Parboil benchmarks to test our methodology because both serial and parallel versions of these benchmarks were readily available and an extensive profiling system was built into the programs. Additionally, these benchmarks were tested with varying input sizes.

The benchmarks we selected for evaluating our algorithm are SGEMM, Stencil and SAD. Details of these applications can be found in [54].

A task graph was created by using the benchmarks discussed above (Fig. 4.16). In our task graph, two or more implementations of a benchmark with data-sets of different sizes were treated as separate processes. For example, SAD was executed with a data-set of medium size and with a data-set of large size. Each of these SAD implementations was treated as a separate process. The same approach was used for SGEMM and SAD. The values for the power consumption of each of these processes were obtained by using `nvidia-smi`. Table 4.10 denotes the values obtained for power consumption and execution time for each of these benchmarks through profiling.

Four different genetic algorithms were allowed to run with different threshold scores. Each GA consisted of 5 individuals and ran for 100 iterations. The same settings as section 4.2.2 were used for running these algorithms as these settings were producing desirable solutions. The threshold scores were selected according to the quality of solutions generated in the first iteration. The threshold scores ranged from strict to lenient to identify the intermediate score which would lead to the best solution in the shortest amount of time. Figs. 4.17 and 4.19 - 4.23 illustrate the performance of the GAs as a function of every iteration. In the PE assignment section of the tables, 1 represents the GPU and 0 represents the CPU.

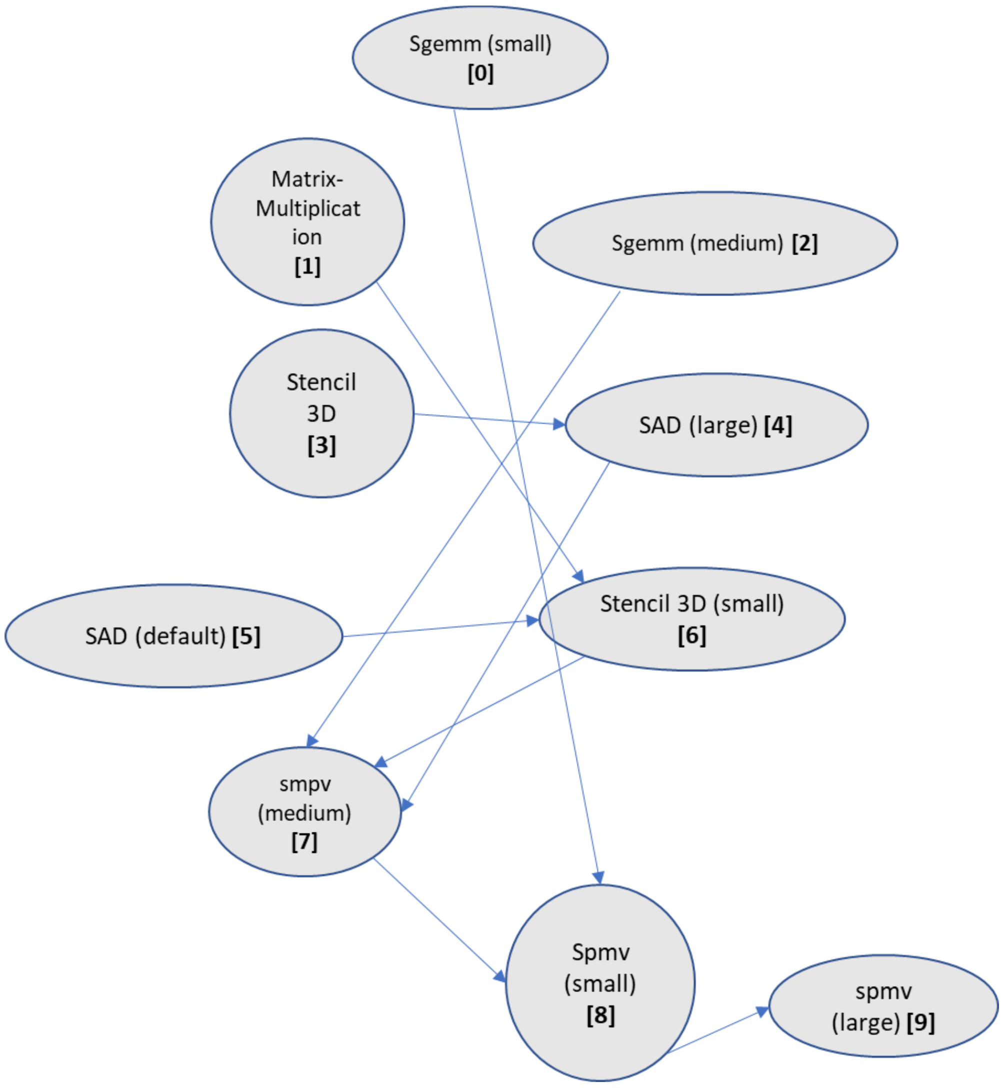


Fig. 4.16: Task-graph made from Parboil's benchmarks. Deadline= 8 seconds.

Table 4.10: PE library for the task graph in Fig. 4.16

Process	**CPU time (sec.)**	**GPU time (sec.)**	**Power**
Matrix-multiplication	2.13	0.26 | 1.03 | 0.32 (priv.)	42 W
SGEMM(medium)	6.8	0.004	32 W
Stencil (default)	213.6	0.52	32 W
SAD (large)	19.5	0.42	34 W
SAD (default)	0.13	0.43	35 W
Stencil (small)	1.02	0.44	33.7 W
Spmv (medium)	0.07	0.41	34 W
SPMV (small)	0.2	0.43	34 W
Spmv (large)	1.814	0.4	30 W

Fig. 4.17 shows how the impact of different threshold scores on the performance of the GA when energy is assigned a weight of 69 %. Table 4.11 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.17.

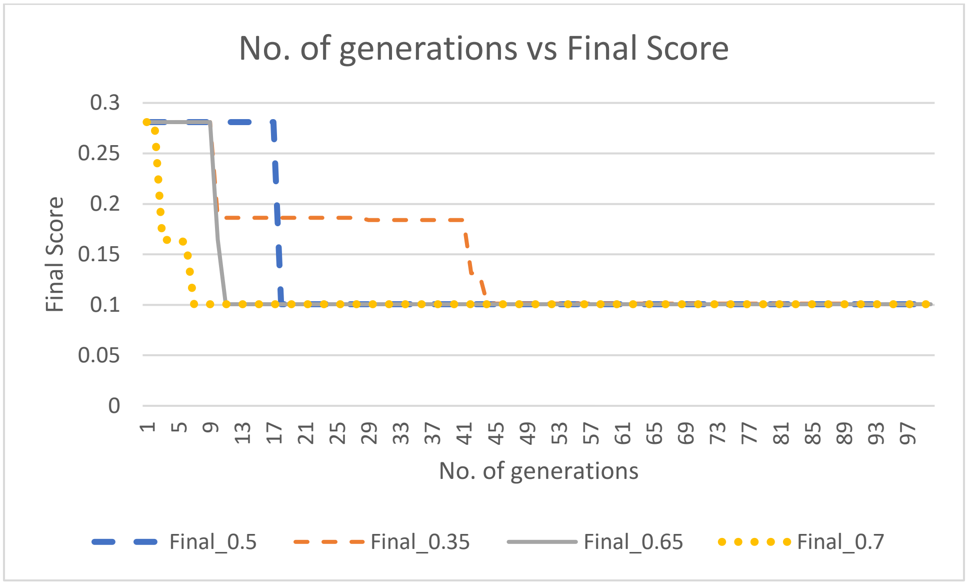


Fig. 4.17: Energy-69 %, Exec. time- 30 % and Deadline violation- 1%.

Table 4.11: Representative solutions for the case in Fig. 4.17

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.10	0001000000	22.116	16.64	10.1
0.28	0111100111	2.696	87.4	0

Fig. 4.18 depicts the relationship between the energy and the execution time of the solutions for one of the cases of Fig. 4.17 (Threshold score= 0.5). As can be seen in the figure, the energy consumption of the solutions is inversely proportional to their execution times.

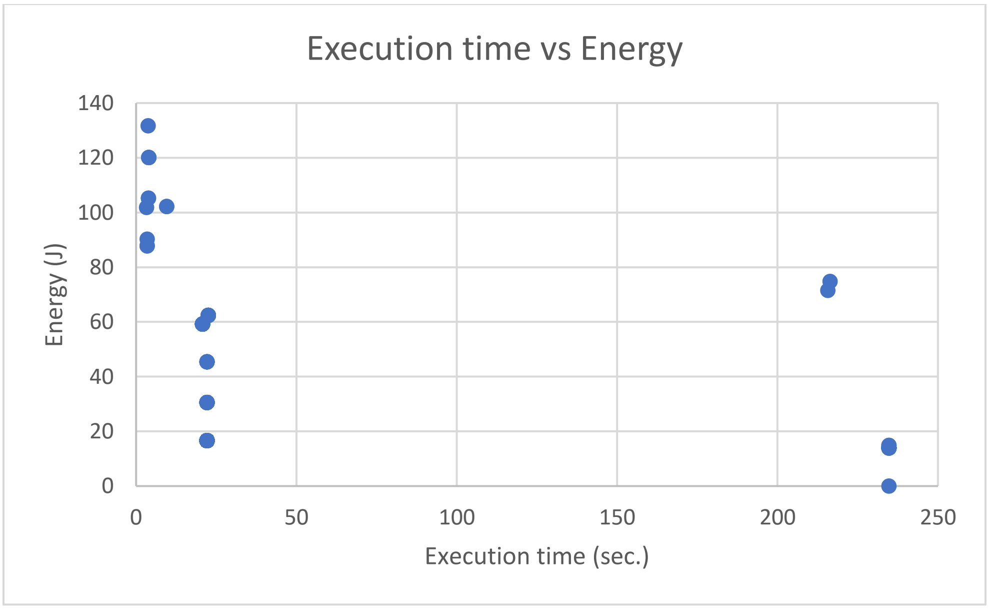


Fig. 4.18: Relationship between Energy and Execution time

Fig. 4.19 depicts how mutation and crossover rates and population sizes impacted the solution space when energy was assigned a weight of 69 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.19 are shown in Table 4.12.

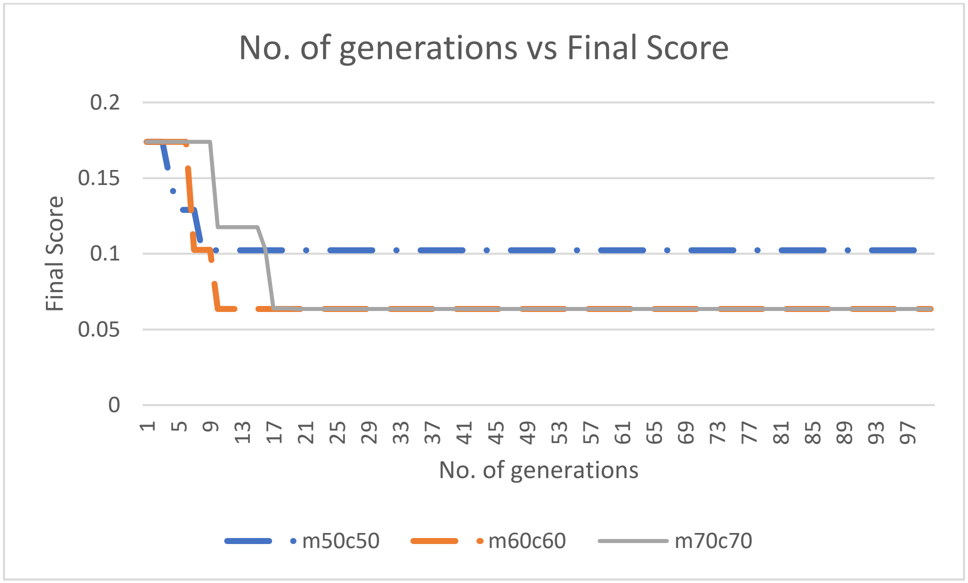


Fig. 4.19: Energy-69 %, Exec. time- 30 % and Deadline violation- 1%. Target Score= 0.5

Table 4.12: Representative solutions for the case in Fig. 4.19

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.17	0001000100	22.546	33.64	10.5
0.28	0001000010	22.346	31.26	10.3

Fig. 4.20 shows how the impact of different threshold scores on the performance of the GA when all the target parameters are assigned equal weights. Table 4.13 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.20.

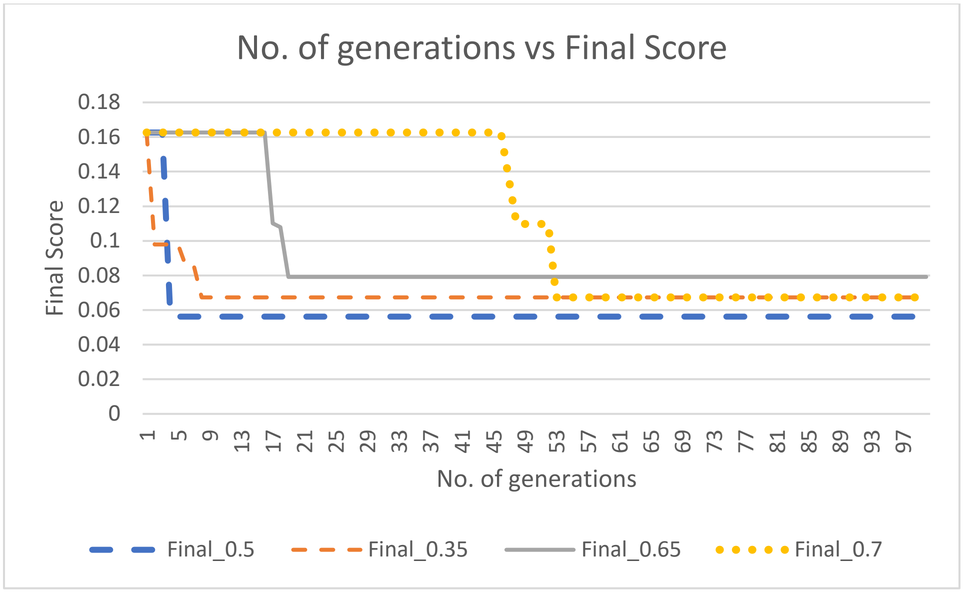


Fig. 4.20: Energy-33 %, Exec. time- 34 % and Deadline violation- 33%.

Table 4.13: Representative solutions for the case in Fig. 4.20

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.056	0011100000	5.246	31.05	0
0.067	0001100000	8.9	30.92	0

Fig. 4.21 depicts the how the mutation and crossover rates and population sizes impacted the solution space when all aspects (energy, time and deadline violation) were assigned equal weights. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.21 are shown in Table 4.14.

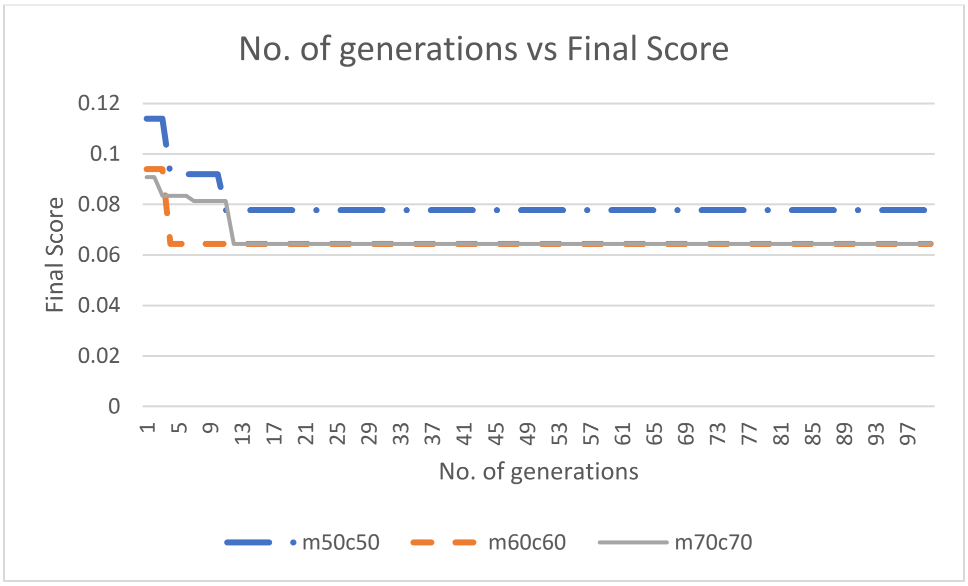


Fig. 4.21: Energy-33 %, Exec. time- 34 % and Deadline violation- 33%. Target Score= 0.5

Table 4.14: Representative solutions for the case in Fig. 4.21

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.13	0001111010	9.12	74.368	0
0.08	0001000000	22.116	16.64	10.116

Fig. 4.22 shows how the impact of different threshold scores on the performance of the GA when exec. time is assigned a weight of 80 %. Table 4.15 shows the characteristics of two of the solutions which were generated by the GAs of Fig. 4.22.

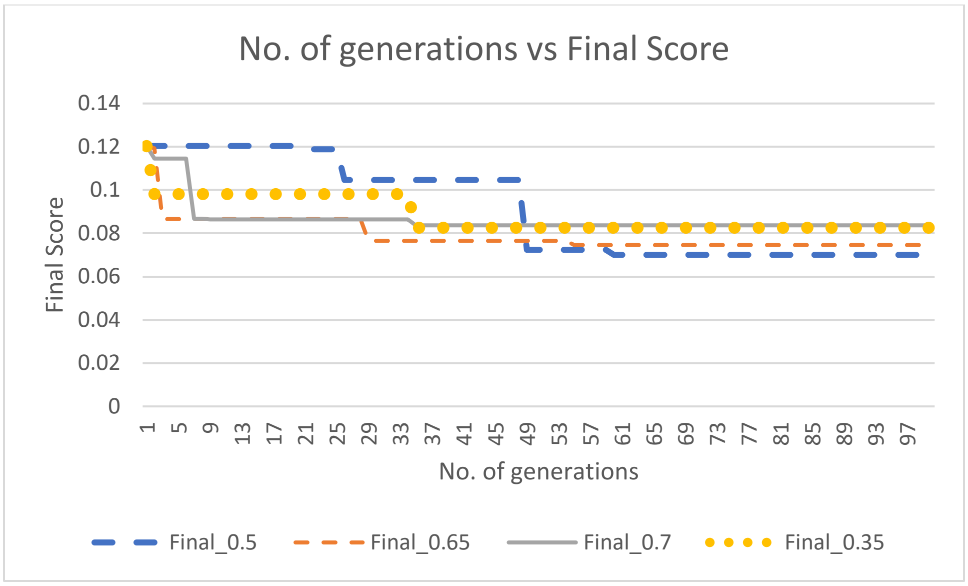


Fig. 4.22: Energy-19 %, Exec. time- 80 % and Deadline violation- 1%.

Table 4.15: Representative solutions for Fig. 4.22

Final Score	PE Assignment	Exec. time (sec.)	Energy (J)	Deadline violation (sec.)
0.07	0111110000	4.146	55.968	0
0.08	1011100100	5.676	62.0025	0

Fig. 4.23 depicts how the mutation and crossover rates and population sizes impacted the solution space when exec. time was assigned a weight of 80 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.23 are shown in Table 4.16.

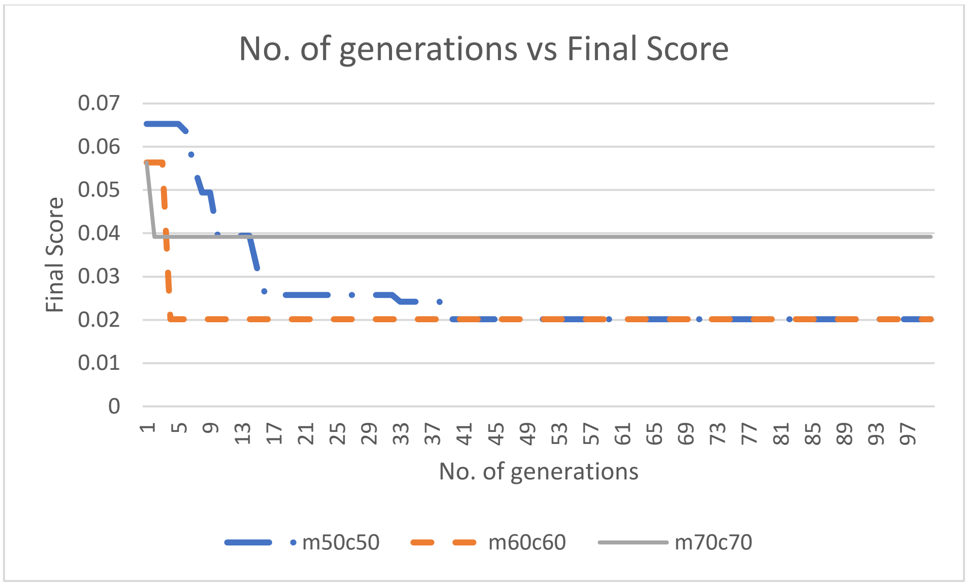


Fig. 4.23: Energy-19 %, Exec. time- 80 % and Deadline violation- 1%. Target Score= 0.5

Table 4.16: Representative solutions for Fig. 4.23

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.07	0001110000	8.89	44.92	0
0.06	0011101000	4.666	45.876	0

The results shown above illustrate the impact of mutation rates, crossover rates and population sizes on the performances of the GAs and the quality of the solutions. In certain cases, the solution space was mostly constant and better solutions were not obtained until the final few generations of the algorithm. Nevertheless, the solutions produced with each configuration are different and offer the developer useful alternatives for implementation.

Moreover, it can also be seen from Fig. 4.18 that in general, energy consumption is inversely proportional to the execution time. It was observed in some unusual cases that a higher execution time does not always lead to lower power consumption. This is because some processes take longer time to execute on the GPU. Hence, care must be taken to ensure that a process or application has the characteristics needed to execute more efficiently on GPUs than CPUs.

4.4 Case Studies- Motion-JPEG Decoder

The JPEG standard (ISO/IEC 10918-1 ITU-T Recommendation T.81) defines compression techniques for image data. As a consequence, it allows to store and transfer image data with reduced demand for storage space and bandwidth. The first step to a video decoder is implementing a still image decoder [56].

The decoder reads a stream of JPEG images, called motion-JPEG, from an input peripheral and writes pixels into an output random access memory digital-to-analog converter. The specification of motion-JPEG is a C program that reads compressed images from a file and writes their pixel maps to a file. Its functionality can be depicted by a task graph (Fig. 4.24) [57].

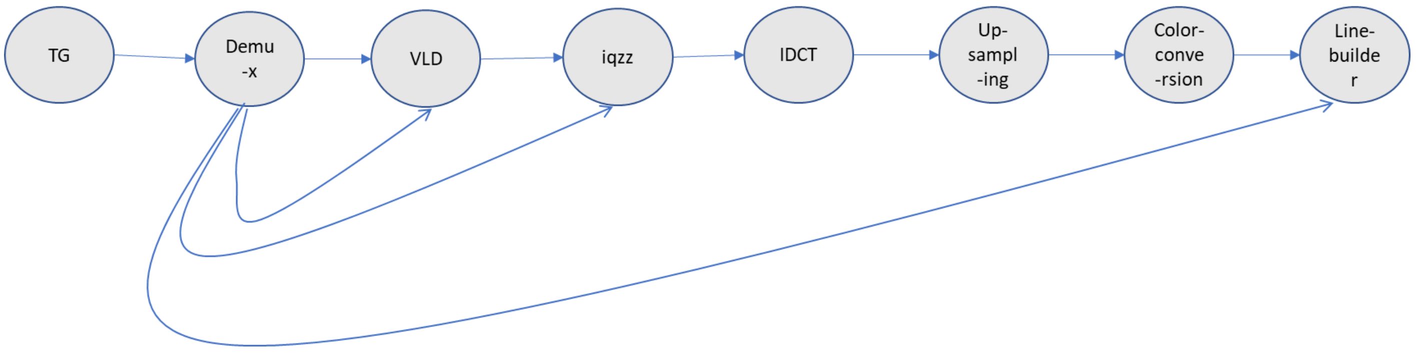


Fig. 4.24: MJPEG decoder- Task graph.

The first step in the decoding process is Entropy decoding. The information needed for run length decoding and variable length decoding is decoded by applying Huffman's algorithm. Run-length decoding and variable length decoding can then be applied. In encoding, the coefficients corresponding to the image in the frequency domain are divided according the corresponding entry in the quantization table. The result is rounded to the next integer value and information is lost. The dequantization operation reverses the division that had occurred during the encoding phase. During the encoding phase, the 8x8 blocks are reordered into an array in a zig-zag order. IDCT needs the coefficients in the original order. Inverse zig-zag is responsible for reverse mapping these blocks. IDCT then transforms the image in the spatial domain. Up-sampling is then performed to interpolate the color components to their original resolutions. YCbCr2RGB then transforms the data from YCbCr to RGB.

Decoding Motion JPEG involves sequentially processing several different images and reading the header of each of them. Hence, the header information is not read out just once at the first decoding cycle but for every subsequent cycle [56].

Fig. 4.25 depicts a sequential single-threaded implementation of the MJPEG decoder. In this implementation, every process (from VLD to color-conversion) processes one 8x8 block at a time.

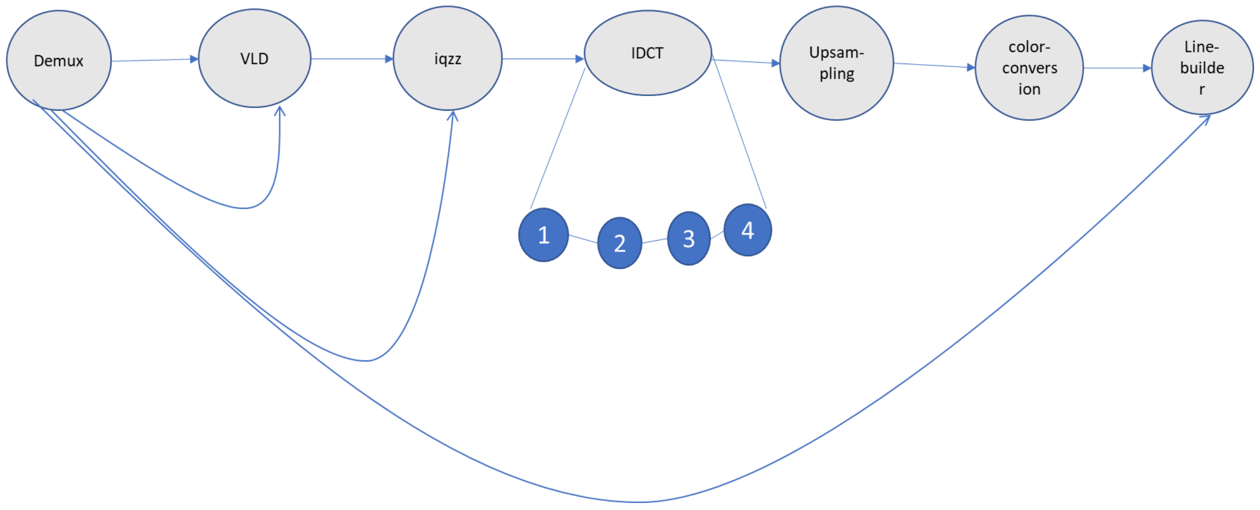


Fig. 4.25: MJPEG decoder-Sequential implementation

The single-threaded code for the MJPEG decoder (visually depicted in Fig. 4.25) was refactored and parallelized to implement it more efficiently on the GPU and subsequently, improve its performance. The parallel version of the MJPEG decoder processes all the 8x8 blocks in a frame at a time. The task graph for the parallel version of the MJPEG decoder is shown in Fig. 4.26.

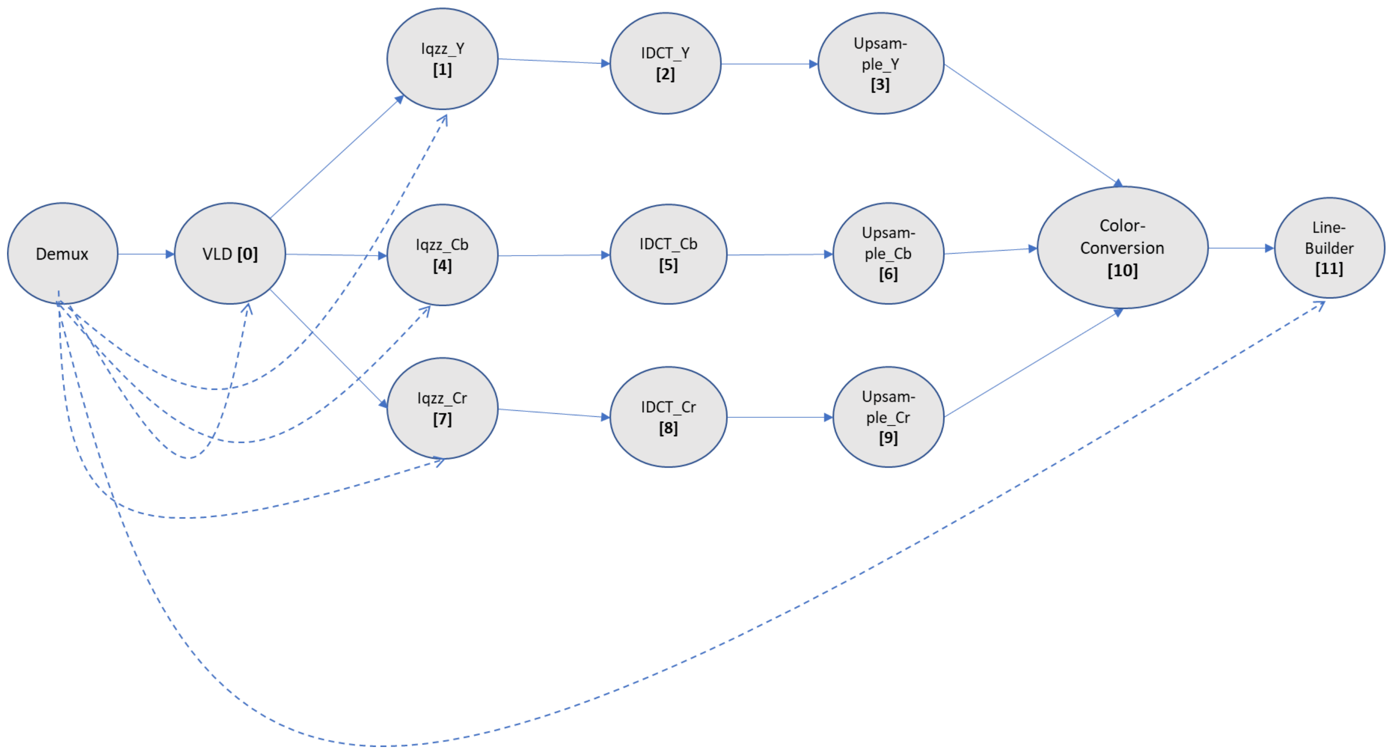


Fig. 4.26: MJPEG decoder-Parallel implementation. Deadline= 0.02 seconds.

Table 4.17: Resource library for MJPEG decoder (single frame)

Process	**CPU (sec.)**	**GPU (sec.)**	**Power Draw- GPU (Watts)**
VLD	0.002608	--------------	-----------------
iqzz_Y	0.000158	0.000112	48.5
IDCT_Y	0.015076	0.000062	48.5
Upsample_Y	0.000053	0.000095	48.6
iqzz_Cb	0.000158	0.000112	48.5
IDCT_Cb	0.015076	0.000062	48.5
Upsample_Cb	0.000053	0.000095	48.6
iqzz_Cr	0.000158	0.000112	48.5
IDCT_Cr	0.015076	0.000062	48.5
Upsample_Cr	0.000053	0.000095	48.6
Color-conversion	0.000891	0.00015	48.4
Line-builder	0.00008	--------------	-------------------

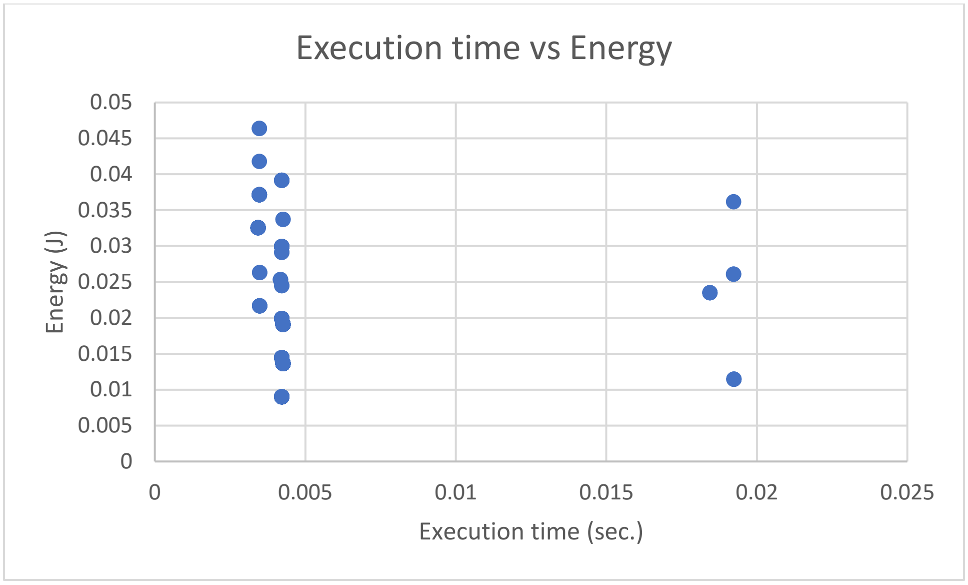


Fig. 4.27: Relationship between Execution time and Energy

Fig. 4.27 shows the relationship between the energy consumption and the execution times of various solutions of the MJPEG decoder on the CPU-GPU platform. As can be seen in the resource library in Table 4.17, the execution times of most processes are similar for both CPUs and GPUs. As a result, close to optimal solutions can be achieved by offloading only a few of the processes to the GPU. Fig. 4.28 depicts how mutation and crossover rates and population sizes impacted the solution space when all the criteria are assigned the same weights. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.28 are shown in Table 4.18.

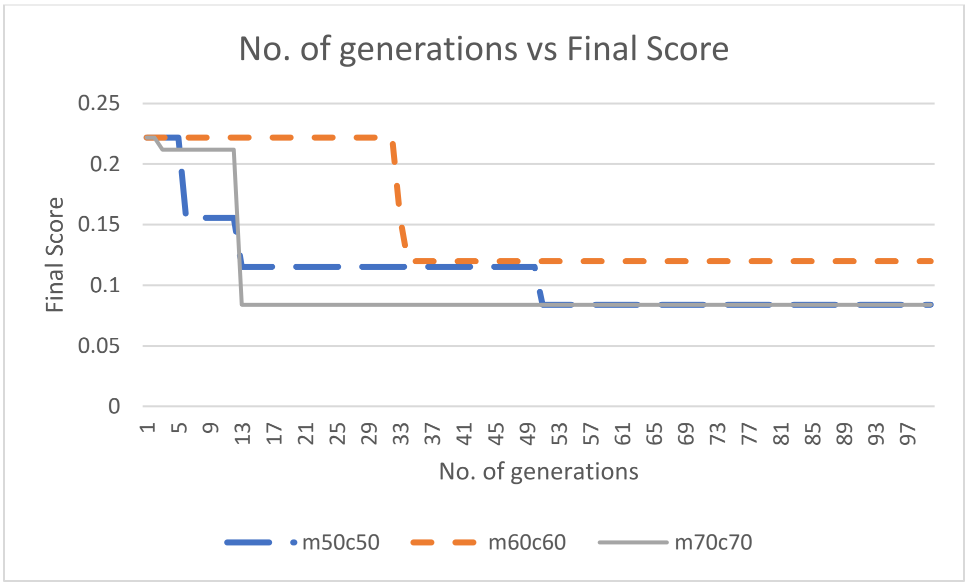


Fig. 4.28: Energy-33 %, Exec. time- 34 % and Deadline violation- 33%.

Table 4.18: Representative solutions for Fig. 4.28

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.083	001001001000	0.00422064	0.009021	0
0.119	011001001000	0.00422064	0.014453	0

Fig. 4.29 depicts how the mutation and crossover rates and population sizes impacted the solution space when energy consumption was assigned a weight of 80 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.29 are shown in Table 4.19.

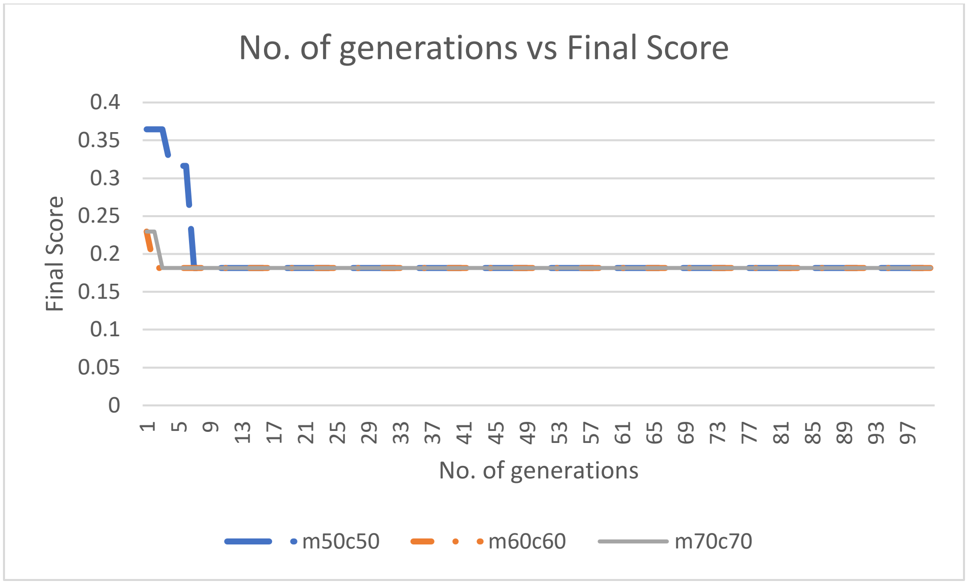


Fig. 4.29: Energy-80 %, Exec. time- 19 % and Deadline violation- 1%.

Table 4.19: Representative solutions for Fig. 4.29

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.23	001000000000	0.0192346	0.003007	0
0.360	001010001100	0.0191886	0.016063	0

Fig. 4.30 depicts how the mutation and crossover rates and population sizes impacted the solution space when exec. time was assigned a weight of 80 %. When the mutation rate was 50 % and the crossover rate was 50 %, the population size was set at 5. When the mutation rate was 60 % and the crossover rate was 60 %, the population size was set at 10. When the mutation rate was 70 % and the crossover rate was 70 %, the population size was set at 15. The characteristics of two of the solutions produced by the GAs of Fig. 4.30 are shown in Table 4.20.

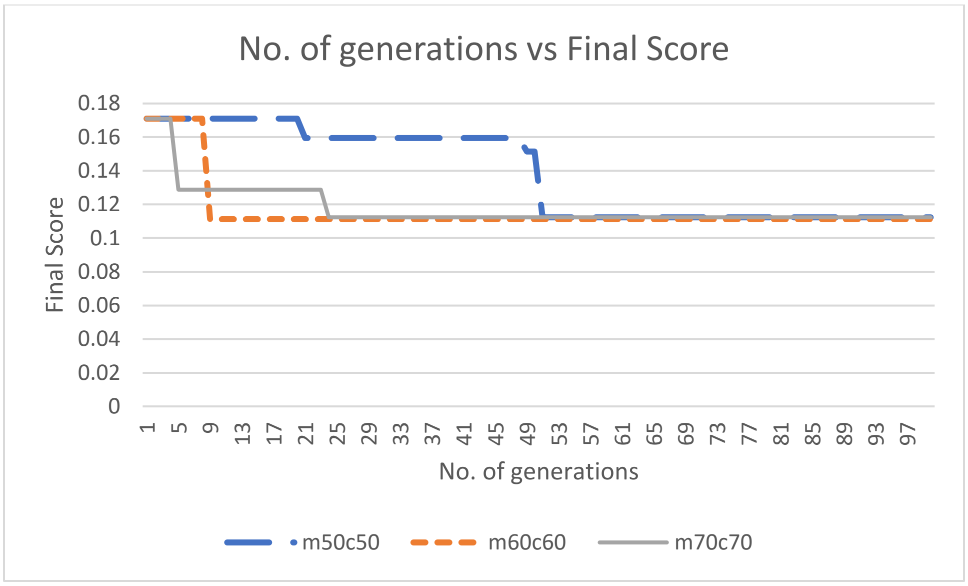


Fig. 4.30: Energy-19 %, Exec. time- 80 % and Deadline violation- 1%.

Table 4.20: Representative solutions for Fig. 4.30

Final Score	**PE Assignment**	**Exec. time (sec.)**	**Energy (J)**	**Deadline violation (sec.)**
0.111	001101001000	0.00426264	0.013638	0
0.112	001001011000	0.00422064	0.014453	0

It can be observed in from Figs. 4.28-4.30 that even for this application, the algorithm produces high quality results. In every case, the algorithm begins with a sub-optimal solution and as generations pass, it settles at an optimal solution. Moreover, the scenarios from Figs. 4.28-4.30 also confirm that different cross-over rates, mutation rates and population sizes produce more diverse and often more desirable results.

Chapter 5

Conclusion and Future Work

Most works concerned with GPUs focus mainly on task scheduling and load balancing of parallel algorithms. There is limited research on using task and data-level parallelism to partition entire applications between the CPU and the GPU. We presented a methodology for hardware/software partitioning of applications which can also effectively partition applications on CPU-GPU platforms. The methodology is based on Genetic algorithms and finds optimal partitioning solutions to satisfy the user-defined constraints. A generic methodology was first presented which does hardware/software partitioning for any embedded application and architecture. The inputs of the methodology are the application, specified as a directed-acyclic graph, the architectural elements and a library which contains information necessary for partitioning the application on the given architecture. In the second stage, additional features were added to the generic methodology so that it could leverage the architectural capabilities of GPUs to effectively partition applications on CPU-GPU platforms. Lastly, a parameter-tuning strategy was presented to use GAs more efficiently to acquire desirable solutions.

The utility of the presented methodology was verified with a 10 and 30-node synthetic graphs, a hypothetical graph composed of benchmark applications and a MJPEG decoder. In every case, the methodology is capable of generating the partitioning solutions under energy constraints. In the way, the effectiveness of using genetic algorithms to do hardware/software partitioning on CPU-GPU platforms is confirmed.

In this thesis, system execution time, energy consumption and deadline violation were the targets of optimization. Resource occupancy is also an important optimization target for GPUs and the presented methodology can be further expanded to account for this aspect. Communication overhead is a critical challenge in CPU-GPU platforms and the scheme for predicting and modeling communication overhead can be further improved. Lastly, FPGAs can be used to supplement CPU-GPU platforms and can also replace GPUs as accelerators. System developers can easily adapt and scale our methodology to find optimal partitioning solutions for applications on architectural platforms consisting of various accelerators.

References

[1] H. Dutta, “Synthesis and Exploration of Loop Accelerators for Systems-on-a-Chip”, Ph.D. dissertation, University of Erlangen-Nuremberg, 2011.

[2] K. Bertels, “Introduction”, in *Hardware/Software Co-design for Heterogeneous Multicore Platforms: The hArtes Toolchain*, 1st ed. Dordrecht: Springer Netherlands, 2012, pp. 1-8.

[3] A.A. Lifa, “Hardware/Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms”, Ph.D. dissertation, Linköping, 2015.

[4] G. Campeanu, J. Carlson and S. Sentilles, "Component Allocation Optimization for Heterogeneous CPU-GPU Embedded Systems," *in Proc. 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 229-236, August 2014, Verona, Italy.

[5] L. Pomante, P. Serri and S. Marchesani, “System-Level Design Space Exploration for Heterogeneous Parallel Dedicated Systems”, in *Proc. 2013 World Congress on Computer and Information Technology (WCCIT)*, pp. 1-6, June 2013, Sousse, Tunisia.

[6] Z. Mann, "GPGPU: Hardware/Software Co-Design for the Masses", *Computing and Informatics*, vol. 30, no. 6, 2011, pp. 1247–1257.

[7] W. Wolf, “A decade of hardware/software codesign,” *IEEE Computer*, vol. 36, no. 4, 2003, pp. 38-43.

[8] W. Wolf, “Hardware-Software Co-Design of Embedded Systems”, *Proceedings of the IEEE*, vol. 82, no. 7, 1994, pp. 967- 989.

[9] S. Prakash and A. Parker, "Synthesis of application-specific multiprocessor systems including memory components", *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 8, no. 2, 1994, pp. 97-116.

[10] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future," *Proceedings of the IEEE*, vol. 100, 2012, pp. 1411-1430.

[11] R. K. Gupta and G. De Micheli, “Hardware-software cosynthesis for digital systems,” *IEEE Des. Test Comput.*, vol. 10, no. 3, 1993, pp. 29–41.

[12] R. Ernst, J. Henkel, and T. Benner, “Hardware-software cosynthesis for microcontrollers,” *IEEE Des. Test Comput.*, vol. 10, no. 4, 1993, pp. 64–75.

[13] H.S. Stone, “Critical Load Factors in Two-Processor Distributed Systems,” *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, 1978, pp. 254-258.

[14] C.-C. Shen and W.-H. Tsai, “A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion,” *IEEE Transactions on Computers*, vol. c-34, no. 3, 1985, pp.197-203.

[15] J.Axelsson, “Hardware/software partitioning of real-time systems”, *IEE Colloquium on Partitioning in Hardware-Software Codesigns*, pp. 5/1-5/8, Feb. 1995, London, UK.

[16] L.A. Cortes, P. Eles, Z. Peng. “A Survey on Hardware/Software Codesign Representation Models”, Dept. of Computer and Information Science, Linköping Univ., Linköping, SAVE Project Rep. 10, 1999.

[17] P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, 1st ed. Springer, 2010, pp. 3-31.

[18] W. Wolf, “An architectural co-synthesis algorithm for distributed, embedded computing systems,” *IEEE Transactions on VLSI Systems*, vol. 5, pp. 218-229, June 1997.

[19] G. N. Khan and A. Awwal, "Codesign of Embedded Systems with Process/Module Level Real-Time Deadlines," in *Proc. 2009 International Conference on Computational Science and Engineering*, pp. 526-531, Aug. 2009, Vancouver, BC, Canada.

[20] R.P. Dick and N.K. Jha, “MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems”, *IEEE Transactions Computer-Aided Design*, vol. 17, no. 10, 2006, pp. 920-935.

[21] S. Chakraverty, C. Ravikumar, and D. Choudhuri, “An evolutionary scheme for cosynthesis of real-time systems”, in *Proceedings Design Automation Conference*, pp. 251-256, Jan 2002, Bangalore, India.

[22] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Design Automation for Embedded Systems,* vol.2, no.1, 1997, pp.5-32.

[23] Y. Xie and W. Hung, "Temperature-Aware Task Allocation and Scheduling for Embedded Multiprocessor Systems-on-Chip (MPSoC) Design", *J VLSI Sign Process Syst Sign Image Video Technol*, vol. 45, no. 3, 2006, pp. 177-189.

[24] M. B. Abdelhalim, A. E. Salama and S. E. D. Habib, "Hardware Software Partitioning using Particle Swarm Optimization Technique," in *Proc. 2006 6th International Workshop on System on Chip for Real Time Applications*, pp. 189-194, Dec. 2006, Cairo, Egypt.

[25] A. Bhattacharya, A. Konar, S. Das, C. Grosan, and A. Abraham, "Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm", in *Proc. International Conference on Complex, Intelligent and Software Intensive Systems,* pp. 171-176, Mar. 2008, Barcelona, Spain.

[26] S. Li, C. Hsu, C. Wong and C. Yu, "Hardware/software co-design for particle swarm optimization algorithm", *Information Sciences*, vol. 181, no. 20, 2011, pp. 4582-4596.

[27] M.T. Schmitz, B.M. Al-Hashimi, P. Eles, *System-level Design Techniques for Energy-Efficient Embedded Systems.* 1st ed. Springer, 2004.

[28] S. Mittal and J.S. Vetter, *"*A Survey of CPU-GPU Heterogeneous Computing Techniques*", ACM Computing Surveys,* vol. 47, no. 4, 2015, pp. 1-35

[29] Q. Wang and X. Chu., "GPGPU Power Estimation with Core and Memory Frequency Scaling", *ACM SIGMETRICS Performance Evaluation Review,* vol. 45, no. 2, 2017, pp. 73-78.

[30] M. Boyer, "Improving Resource Utilization in Heterogeneous CPU-GPU Systems", Ph.D. dissertation, Dept. Computer Eng., Univ. Virginia, Charlottesville, Virginia, 2013.

[31] G. Wang and W. Song, "Communication-aware task partition and voltage scaling for energy minimization on heterogeneous parallel systems", in *Proc. of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies,* pp. 327-333, Oct. 2011, Gwangju, South Korea.

[32] T. Hamano, T. Endo and S. Matsuoka, “Power aware dynamic task scheduling for heterogeneous accelerated clusters”, in *Proc. of the IEEE International Symposium on Parallel and Distributed Processing*, pp. 1-8, May 2009, Rome, Italy.

[33] C. Timm, F. Weichert, P. Marwedel and H. Müller, “Design space exploration towards a realtime and energy-aware GPGPU-based analysis of biosensor data,” *Computer Science-Research and Development*, vol. 27, no. 4, 2012, pp. 309-317.

[34] T. Komada, S. Hayashi, T. Nakada, S. Miwa and H. Nakamura, “Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping” in *Proc. 2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 349-356, Oct. 2013, Asheville, NC, USA.

[35] J. Cebri´an, G.D. Guerrero and J.M. Garcia, “Energy Efficiency Analysis of GPUs”, in *Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 1014-1022, May 2012, Shanghai, China.

[36] C.S. Lin, S.M. Teng, and P.A. Hsuing, “Auto-tuning for GPGPU applications using performance and energy model”, *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 62, no. C, 2016, pp. 40-53.

[37] H. Park, Y.W. Ko, J. So, and J. Lee, “Performance/Power Design Space Exploration and Analysis for GPU Based Software”, *International Journal of Control and Automation,* vol. 6, no. 6, 2013, pp. 371-380.

[38] S.K. Rethinagiri, O. Palomar, J.A. Moreno, G. Yalcin, O. Unsal and A. Cristal, “System-level power and energy estimation methodology and optimization techniques for CPU-GPU based mobile platforms”, in *Proc.2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pp. 118-127, Oct. 2014, Greater Noida, India.

[39] C. Timm, A. Gelenberg, P. Marwedel and F. Weichert, *Reducing the Energy Consumption of Embedded Systems by Integrating General Purpose GPUs,* tech. report, Department of Computer Science, TU Dortmund University, Dortmund, 2010.

[40] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight and X. Lin, “Power-Efficient Time-Sensitive Mapping in Heterogeneous Systems”, *21st Int’l Parallel Architectures and Compilation Techniques Conf.* (PACT 12), pp. 23-32, Sept. 2012, Minneapolis, MN, USA.

[41] Y. Wang, “Performance and Power Optimization of GPU Architectures for General-purpose Computing”, Ph.D. dissertation, University of Rhode Island, 2014.

[42] Martin Peres. Reverse Engineering Power Management on NVIDIA GPUs - Anatomy of an Autonomic-ready System. ECRTS, Operating Systems Platforms for Embedded Real-Time applications 2013, Jul 2013, Paris, France. 2013.

[43] NVIDIA, nvidia-smi.txt, pp. 1-34. Internet: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[44] E. Paone, “Design Space Exploration of OpenCL applications on Heterogeneous Parallel Platforms”, Ph.D. dissertation, Politecnico Di Milano, 2014.

[45] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran and M. Garzaran, “Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips,” *Procedia Computer Science,* vol. 51, no. C, 2015, pp. 140-149.

[46] E. Wachter, G. Merrett, B. Al-Hashmi and A.K. Singh, “Reliable mapping and partitioning of performance-constrained OpenCL Applications on CPU-GPU MPSoCs”, in *Proceedings of 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia (ESTIMEDIA ’17)*, pp. 78-83, Oct. 2017, Seoul, Republic of Korea.

[47] K. Shetti, “Optimization and Scheduling in a Heterogeneous CPU-GPU Environment”, master’s thesis, School of Computer Eng., Nanyang Technical Univ., 2014.

[48] *NVIDIA Cuda C Programming Guide,* NVIDIA, Santa Clara, Cal., 2012. URL: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[49] “CUDA Memory and Cache Architecture”. URL: http://supercomputingblog.com/cuda/cuda-memory-and-cache-architecture, Sept. 10, 2011 [April 15, 2017].

[50] *OpenCL Programming Guide for the CUDA Architecture*, NVIDIA, Santa Clara, Cal., 2010. URL: http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf

[51] J. Castrillon, "Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap," Ph.D. dissertation, RWTH Aachen University, 2013.

[52] A. Aleti, "Designing automotive embedded systems with adaptive genetic algorithms," *Automated Software Engineering*, vol. 22, no. 2, pp. 199-240, June 2015.

[53] "Heterogeneous Processing Platform (HPP)". URL: https://www.cmc.ca/en/WhatWeOffer/Prototyping/HighPerformance/HPP.aspx., May 7, 2018 [December 20, 2017].

[54] Parboil Benchmark Suite. URL: http://impact.crhc.illinois.edu/parboil/parboil.aspx

[55] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Shaeffer, S. Lee and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing", in *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44-54, Oct. 2009, Austin, TX, USA.

[56] *Development and Implementation of an MotionJPEG Capable JPEG Decoder in Hardware,* https://github.com/freecores/mjpeg-decoder/blob/master/mjpeg.pdf.

[57] I. Auge, F. Petrot, F. Donnet and P. Gomez, "Platform-based design from parallel C specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12. 2006, pp. 1811-1826.