A Study of Time and Energy Efficient Algorithms for Parallel and Heterogeneous Computing

Thesis submitted in accordance with requirements of the University of Liverpool for the degree of Doctor in Philosophy

by

Jude-Thaddeus Ojiaku

September 2016

Degree of Doctor of Philosophy

Abstract

This PhD project is motivated by the need to develop and achieve better and energy efficient computing through the use of parallelism and heterogeneous systems. Our contribution consists of both theoretical aspects, as well as in-depth and comprehensive empirical studies that aim to provide more insight into parallel and heterogeneous computing.

Our first problem is a theoretical problem that focuses on the scheduling of a special category of jobs known as deteriorating jobs. These kind of jobs will require more effort to complete them if postponed to a later time. They are intended to model several industrial processes including steel production, fire-fighting and financial management. We study the problem in the context of parallel machine scheduling in an online setting where jobs have arbitrary release times. Our main results show that List Scheduling is $(1 + b_{max})$-competitive and that no deterministic algorithm is better than $(1 + b_{max})^{1-\frac{1}{m}}$, where b_{max} is the largest deteriorating rate. We also extend our results to online deterministic algorithms and show that no deterministic online algorithm is better than $(1 + b_{max})$-competitive.

Our next study concerns the scheduling of n jobs with precedence constraints on m parallel machines. We are interested in the precedence constraint known as chain precedence constraint where each job can have at most one predecessor and at most one successor. The jobs are modelled as directed acyclic graphs where nodes represent the jobs and edges represent the precedence constraints between jobs. The jobs have a strict deadline that must be met. The parallel machines are considered to be unrelated and a communication network connects each pair of machines. Execution of the jobs on the machines as well as communication across the network incurs costs in the form of time and energy. These costs are given by cost matrices that covers processing and communication. The goal is to construct a feasible schedule that minimizes the total energy required to execute the chain of jobs on the machines, such that all deadlines are met. We present a dynamic programming solution to the problem that leads to a pseudo polynomial time

algorithm with running time $O(nm^2 d_{max})$, where d_{max} is the largest deadline. We show that the algorithm computes an optimal schedule where one exists.

We then proceed to a similar problem that involves the scheduling of jobs to minimize flow time plus energy. This problem is based on a dynamic speed scaling heuristic in literature that is able to adjust the speed of a processor based on the number of *active jobs*, called AJC. We present a comprehensive empirical study that consists of several job selection, speed selection and processor allocation heuristics. We also consider both single processor and multi processor settings. Our main goal is to investigate the viability of designing a fixed-speed counterpart for AJC, that is not as computationally intensive as AJC, while being very simple. We also evaluate the performance of this fixed speed heuristic and compare it with that of AJC.

Our fourth and final study involves the use of graphics processing unit (GPU) as an accelerator for compute intensive tasks. The GPU has become a very popular multi processor for heterogeneous computing both from an economical point of view and performance standpoint. Firstly, we contribute to the development of a Bioinformatics tool, called GapsMis, by implementing a heterogeneous version that uses graphics processors for acceleration. GapsMis is a tool designed for the alignment of sequences, like protein and DNA sequences, and allows for the insertion of gaps in the alignment. Then we present a case study that aims to highlight the various aspects, including benefits and challenges, involved in developing heterogeneous applications that is vendor-agnostic. In order to do this we select four algorithms as case studies including GapsMis and the algorithm presented in our second problem. The other two algorithms are based on the Velocity-Verlet integration and the Fruchterman-Reingold force-based method for graph layout. We make use of the Open Computing Language (OpenCL) and C++ for implementation of the algorithms on a range of graphics processors from Advanced Micro Devices (AMD) and NVIDIA Corporation. We evaluate several factors that can affect performance of these applications on each hardware. We also compare the performance of our algorithms in a multi-GPU setting and against single and multi-core CPU implementations. Furthermore, several metrics are defined to capture several aspects of performance including execution time of application kernel(s), execution time of application including communication times, throughput, power and energy consumption.

Acknowledgements

Firstly I would like to show my deepest gratitude to my supervisors Dr. Prudence Wong and Prof. Leszek Gąsieniec for their support and advice throughout the duration of my PhD studies. I have learned a lot during this period and they have always guided me in the right direction.

I am very grateful to my family for their unending love, encouragement and support, and for giving me the means to ensure that I complete my PhD studies.

I would also like to thank all my friends and colleagues for their help and support including Dr. O. Nwamadi, whose help, discussions and advice benefited me a great deal.

Contents

List of Figures

List of Tables

Dedicated to my family, for providing me with the means to pursue my dreams. And to my supervisor, Dr. Prudence Wong, for making those dreams a reality.

Chapter 1

Introduction

1.1 Overview

The study of scheduling dates back as far as the 1950s when researchers in operations research, industrial engineering and management were faced with the problem of managing various activities occurring in a workshop [64]. An organization can lower production costs in its manufacturing processes thereby enabling it to stay competitive. Later on in the 1960s computer scientists also encountered scheduling problems during the development of operating systems. During this time period computer resources such as CPU, memory and I/O devices were considerably scarce so it was crucial that they had to be efficiently utilized in order to minimize the cost of running these computer systems. Therefore an economic reason to study scheduling was established and eventually various classes of scheduling problems have been developed to take into account the different scenarios they aim to address. Even in present times, new scheduling problems arise from various sources such as the introduction of a new technology in various fields like systems design, automated industrial processes and so on.

Advances in the technology of microprocessor development and chip fabrication process means that the density of transistors that make up these chips continue to grow. In addition, the computational power and processing capability of these chips somewhat doubled with each new design as manufacturers are able to ramp up the clock speeds with the introduction of subsequent generations of microprocessors. Consequently, operating at high clock speeds usually comes at the expense of incurring exponential increase in power consumption and in order to mitigate this issue, manufacturers resorted to shrinking chip sizes but this process has been restricted by available technology. As a result, the growth in clock speeds began to slow down [103] and the gains in the performance levels of these chips began to diminish. Chip manufacturers pursued other

means of achieving higher performance and this brought about the advent of multi-core technology and mainstream parallel computation. Multi-core processor technology was particularly attractive and promising especially because manufacturers were able to more than double performance without necessarily increasing the operating frequency by simply adding more processing cores. This means that devices are able to do more as more processing power became readily available and this lead to an age of ubiquitous computing as these chips powered almost everything ranging from small devices such as mobile phones and tablets to our home computers to enterprise server systems and super-computers. However, the problem of energy consumption soon re-surfaced and it became highly imperative that system designers and developers tackle the issue for both technical and economic reasons in order to prolong the sustainability of multi-core and parallel systems.

On the economic side, apart from the costs of powering these computer systems, extra costs are incurred as a result of cooling systems required to keep them within their optimal operating conditions. Since significant amount of the energy drawn by these systems is dissipated as heat, the life span of a system can be greatly shortened due to the adverse effects of high temperatures such as degraded transistor performance and damage to components like soldering which can cause permanent damages. Therefore, the problem of managing energy has become a critical topic in both industrial and academic research and various approaches have been considered leading to innovations in algorithm designs, software and hardware. Among the several approaches of managing energy consumption, a growing trend is the adoption of heterogeneous computing to deliver high performance. This is evident from the recent rankings of the 500 most powerful and energy-efficient supercomputers where the 17 most energy-efficient supercomputers [38] as well as the most powerful supercomputer [106] are heterogeneous systems utilizing graphics processing units (GPU) and other co-processors.

The development of heterogeneous and parallel systems offer exciting prospects in the quest to find a balance between energy consumption and performance of computer systems. The GPU has shot to the forefront as most used co-processor in heterogeneous systems quite simply because it was already part of existing systems where it is used for visual and rendering tasks making its adoption very easy. As a part of this thesis, we will present a detailed study to demonstrate the benefits of a heterogeneous system that includes the GPU with respect to saving energy while achieving high computational performance.

This chapter is organized as follows; in Section 1.2, we discuss some of the basic concepts related to scheduling problems. The problems we studied along with related works are

introduced in Section 1.3. Finally, in Section 1.4, we outline the contribution of the thesis.

1.2 Background on Scheduling

Scheduling can be generally considered as dealing with allocation concerns involving scarce resources and tasks or operations that demand them with the goal of optimizing one or more performance measures of interest in a given setting. These resources could refer to a number of entities depending on the situation being considered. These may include cores in a multi-core CPU, CPUs in a multiprocessor system, servers in a server farm or cluster, memory, I/O devices, machines in an assembly line, airport runways, train stations, personnel assignment in workplaces, just to mention a few. Operations, on the other hand, may also refer to train calls at stations, airport landing and take-offs, an operation in a manufacturing process, execution of a computer program, manning workstations in an industrial setting or call centres.

1.2.1 Inputs and outputs

The inputs in a scheduling problem includes a set consisting of a number of jobs to be executed on a set consisting of a number of machines and each job or machine can be uniquely referenced by a subscript. The time at which a job becomes available for processing is known as the *release* or *arrival* time. The time it takes for a job j to completely execute on a machine i is known as its *processing time*, and this value is assumed to be finite and non-negative unless explicitly stated. The point in time at which job j finishes its execution is known as its *completion time*. In some cases a job is required to finish execution at a particular time or deadline, however, a job might finish at a time greater than the time specified as its deadline. The term *tardiness* measures how late a job completes past its deadline and is expressed as $\max\{\textit{completion time} - \textit{deadline},\ 0\}$, while *earliness*, expressed as $\max\{\textit{deadline} - \textit{completion time},\ 0\}$, measures how much a job completes before its deadline. Hence, when a job completes before its deadline its tardiness is 0, and when a job completes after its deadline then its earliness is 0.

In order to represent each scheduling problem in a concise way including the inputs, constraints and objective function(s), I would like to mention the well-known three-field $\alpha|\beta|\gamma$ notation first introduced by Graham et al. [35]. However, we will be describing the notation that was later introduced to incorporate machine availability constraints [95].

1.2.2 The $\alpha|\beta|\gamma$ scheduling notation

The first field $\alpha = \alpha_1\alpha_2\alpha_3$ in the notation describes the machine environment in a problem setting. Parameter $\alpha_1\epsilon\{\emptyset, P, Q, R, F, J, O\}$ is used to characterize the machines in single machine, identical, uniform and unrelated parallel machines, flow shop, job shop and open shop problem settings respectively. Parameter $\alpha_2\epsilon\{\emptyset, m\}$, where m is a positive integer, indicates that the number of machines in a parallel machine environment or number of stages for dedicated machines is assumed to be the variable m. Parameter $\alpha_3\epsilon\{\emptyset, h_{i,k}\}$ describes unavailability intervals which occur on the machines otherwise referred to as *holes*. In this notation $\alpha = \emptyset$ represents a problem setting with no holes and $h_{i,k}$ specifies the number of holes and the machine(s) on which they occur. However $\alpha = h_{i,k}$ represents a problem setting with an arbitrary number of holes on each machine. If i is replaced by a positive integer it means that holes only occur on machine M_i otherwise holes will occur on all machines but if k is replaced by a positive integer it denotes the number of holes occurring on the corresponding machine. For instance, $\alpha = h_{1,k}$ denotes a problem with an arbitrary number of holes on M_1 only; $\alpha = h_{i,1}$ represents a problem with one hole on each machine while $\alpha = h_{1,1}$ represents a problem with one hole on machine M_1 only.

The second field $\beta = \beta_1, ..., \beta_5$ describes characteristics or constraints associated with operations (jobs) and resources. Parameter $\beta_1\epsilon\{\emptyset, t{-}pmtn, pmtn\}$ denotes the kind of preemption constraint in place which is either non-preemption, operation preemption or arbitrary preemption respectively. An operation is said to be preempted if its processing on a particular machine is interrupted at any time and resumed later at any time and restarted at no cost. It must either remain on the same machine until it can be continued later (*operation preemption*) or it can be shifted to another machine (*arbitrary preemption*). However, there are several studies that use $\beta_1 = r{-}a$, $\beta_1 = nr{-}a$ and $\beta_1 = sr{-}a$ to represent resumable, non-resumable and semi-resumable availability constraints respectively [68]. In the resumable case preemption is allowed so if an operation cannot be completed before the unavailability period of a machine it can resume later when the machine becomes available again. The non-resumable case does not allow preemption so the disrupted operation has to completely restart instead of continue. However, in the semi-resumable case, operations can only be restarted partially after the machine becomes available again. Parameter $\beta_2\epsilon\{\emptyset, r_j\}$ indicates release time for operations (jobs), which can either be zero or differ respectively. Parameter $\beta_3\epsilon\{\emptyset, d_j\}$ denotes deadline constraints on the job set where $\beta_3 = \emptyset$ indicates no assumed deadlines, however, due dates may be defined if necessary. On the other hand, $\beta_3 = d_j$ indicates a deadline constraint imposed in the job set. Parameter $\beta_4\epsilon\{\emptyset, q_j\}$ indicates the absence or presence of tails in the jobs while $\beta_5\epsilon\{\emptyset, online\}$ represents an offline or online problem. A

problem is said to be *offline* if we have full knowledge regarding job data before building a schedule or *online* if a scheduling decision is required once a job arrives without any information about jobs that are yet to arrive in the future [68]. The β field can also be left blank to denote no constraints on the job set and that all the jobs are available before the construction of the schedule begins.

The third field γ represents the performance measure or objective function to be optimized. Some of the commonly studied objective functions include maximum completion time of all jobs or *makespan* (C_{max}), minimum completion time (C_{min}), maximum lateness (L_{max}), maximum tardiness (T_{max}), total completion time ($\sum C_i$), total weighted completion time ($\sum w_i C_i$), number of tardy jobs ($\sum U_i$) and weighted number of tardy jobs ($\sum w_i U_i$).

1.2.3 Classes of scheduling problems

Single machine problems.

The single machine problem simply involves scheduling a set of jobs on one machine only which can either be assumed to be continuously available throughout the processing period of the job set or have periods of unavailability or holes. It is the simplest form of the scheduling problems.

Parallel machine problems.

The parallel machines can be further subdivided into *identical* (P_m), *uniform* (Q_m) and *unrelated* (R_m) parallel machines. The variable m usually indicates the number of parallel machines in the problem setting which can also be omitted to denote an arbitrary number of machines. Identical parallel machine problem involves a set of identical machines running at the same speed, therefore, a given job will take the same amount of time to process on all the machines. For the case of uniform parallel machines each machine runs at a different speed. For instance machine M_i, $1 \leq i \leq m$, runs at speed s_i. The processing time p_{ij} that job J_j spends on M_i is given by p_j/s_i assuming J_j is completely processed by M_i. Finally, in unrelated parallel machines, the jobs are processed at different speeds on the machines so even if two machines run at the same speed it does not necessarily mean that they will take the same amount of time to process a particular job.

Job shop problems (J_m).

In this problem setting with m machines there is a set of n jobs that need to be processed using a number of the machines for a certain amount of time. Each job has its own

predetermined route through the machines so a job can use some machines more than once and may not use some machines at all. For instance, in a problem consisting of m=10 machines labelled serially $M_1, ..., M_{10}$, a job J_j might have a predetermined route through machines M_1, M_3, M_5 and M_1 exactly in that order. The order in which a job executes must follow the order in the predetermined route.

Flow shop problems (F_m).

Here all m machines are ordered linearly and all the jobs in the job set must follow the same route from the first machine to the last machine in order to be processed.

Open shop problems (O_m).

In an open shop problem of m machines all n jobs in the job set needs to visit each of the machines at least once and the order in which this happens is not relevant.

1.2.4 Input structure and constraints

Apart from the machines a certain structure or constraint can exist in the set of jobs to be processed known as precedence constraints. This could be in the form of an *intree, outtree* or *chain*. Precedence constraints are specified explicitly in the β field or generally written as *prec* and it is always given as a directed acyclic graph where each vertex represents a job. Job i precedes job j if there is a directed arc from i to j meaning that i must be completed before j starts. A *chain* exists when each job has at most one predecessor and at most one successor. An *intree* is such that each job has at most one successor while an *outtree* is when each job has at most one predecessor. It also possible to restrict the number of jobs by including the symbol *nbr* in the β field which is the maximum number of jobs to be processed. For flow shop problems only a no-wait constraint (*nwt*) can be specified to denote that jobs are not allowed to wait between two successive machines. Also a restriction can be imposed on the processing time of the jobs using $p_j = p$ in the β field to denote that each job's processing time is p units. Finally, in job shop problems where jobs can have operations, the sub-field n_j is used to restrict the number of operations allowed for each job.

1.3 Problems Studied and related work

1.3.1 Online scheduling of deteriorating jobs on parallel machines

In classical scheduling problems, it is usually of the assumption that the processing of jobs are fixed, however, this is not always the case in some practical situations. There

are various scenarios where the processing time of a job increases of deteriorates as the start time increases or is delayed. An instance of this scenario would be in the continuous casting stage in a steel production process. This requires that the steel is still in molten form in order to be cast into slabs, blooms or billets. Any delays in the prior processes might result in the steel cooling down to unacceptable temperatures which may result in a restart in the whole process, leading to additional time as a consequence. Other examples can be found in fire-fighting, cleaning, maintenance tasks and financial management [53, 74]. In general, any delay in the start time of such task would incur additional effort to complete it at a later time, hence, the reason and motivation behind the study of deteriorating jobs.

The problem of scheduling deteriorating jobs was first introduced by Gupta and Gupta [41], and Brown and Yechiali [17]. Although both works studied makespan minimization on a single machine, the processing time of a job is given as a monotone linear function of its start time in [17] while in [41], it is given as a non-linear function. The problem has since attracted huge interests and has been studied in other time-dependent models with other objective functions [5, 24, 34].

Linear deterioration. The problem of scheduling jobs with *linear deterioration* has been studied in great details as a result of its simplicity while capturing the vital properties of practical situations. The processing time of a job is expressed as a monotone linear function of its start time. To be precise, the processing time p_j of a job J_j is defined as $p_j = a_j + b_j s_j$, where $a_j > 0$ is the *normal* or *basic* processing time, $b_j > 0$ is the deteriorating rate and $s_j > 0$ is the start time. As the start time of a job gets larger, the processing time also gets larger and therefore, the processing time of a job is dependent on the schedule. Furthermore, linear deterioration is said to be *simple* if $a_j = 0$, that is, $p_j = b_j s_j$.

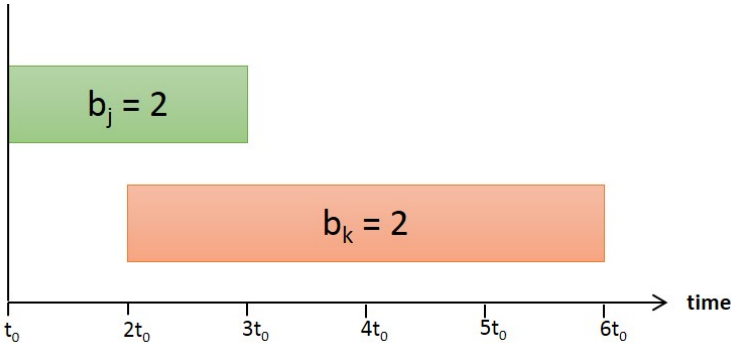


FIGURE 1.1: An illustration of linear deterioration.

Consider the illustration in Figure 1.1 that shows two jobs, J_j and J_k, with identical deteriorating rates of 2 scheduled at different times. Assume that $t_0 = 1$ and J_j starts at t_0, then the completion time is given by $t_0 + b_j t_0 = 1 + 2(1) = 3$. On the other hand, J_k

is started later at time $2t_0$, therefore, the completion time is given by $2t_0 + b_k 2t_0$ which evaluates to 6.

As the number of jobs grows, the start time of jobs gets larger and the processing time of infinitely many jobs is no longer affected by the normal processing time but only by the deteriorating rate [73, 74]. In addition, Gupta and Gupta [41] observed that in the case of linear deterioration, it is optimal to process jobs in ascending order of $\frac{a_j}{b_j}$. From this example, we can see that even though both jobs have identical deteriorating rates, their processing times differ significantly and this demonstrates how the schedule of a deteriorating job can greatly affect its processing time.

Scheduling jobs with linear deterioration has been studied in the contexts of both single machine [74, 80] and parallel machines [33, 49, 51, 75, 92]. In [80], Ng et al presents a study of the problem on a single machine where preemption is allowed and each job is associated with a release time. They show that minimizing the maximum job completion cost is polynomial time solvable whether or not precedence constraints are imposed on the jobs. They also show that minimizing the total weighted completion time is NP-hard even with only two distinct release times. On parallel machines, Garey and Johnson [33] showed that the problem is NP-hard for two machines and strongly NP-hard for arbitrary number of machines as a result of the complexity of the corresponding problems with fixed processing time. A FPTAS is proposed by Kang and Ng [49]. For the case of simple linear deterioration, the problem has been independently shown to be NP-hard [51, 75] and a FPTAS is proposed by Ren and Kang in [92].

Other time-dependent models. In the definition of linear deterioration, $p_j = a_j + b_j s_j$, b_j is assumed to be non-negative and non-decreasing. However, in a different model, b_j is assumed to be a non-positive and non-increasing factor. In other words, the processing time of each job decreases as its start time increases. This model was initiated in [43] and represents a real-world problem in the military domain where the objective is to eliminate an aerial threat target. In this case the execution time decreases as the target gets closer.

Another time-dependent model involves scheduling with procrastination. In [15] the scheduler is assumed to execute a job with increasing speed as the deadline approaches. This models a number of real-world scenarios, most especially human behaviour, where one would postpone execution of an arduous task close to the deadline in order to spend as little time as possible. Another example is the addition of more resources or people to a project in order to meet the project's deadline. For this problem, an optimal offline algorithm was presented for case where a feasible schedule or solution exists. They also present results in the online case showing a $\Theta(1)$-competitive online algorithm for

maximum interval stretch, that is, the time allowed for the procrastinator to finish a job beyond its due date.

Online models. Most of the results discussed above assume that the all jobs are available at the same time so an algorithm has full knowledge (that is, a_j and b_j) of the all jobs in advance. However, the reality is that jobs may also have arbitrary release times. Pruhs et al [91] formalized two online models, namely, *online-list* and *online-time* models. In the online-list model, jobs are released only one at a time so an algorithm must schedule the released job before further subsequent jobs can be released. Once a job is scheduled by the algorithm it cannot be modified in the future. One the other hand, in the online-time model, an arbitrary release time is associated with each job so the information relating to each individual job is only available after it has been released.

Graham [36] proposed a List Scheduling (LS) algorithm for online parallel machine scheduling with fixed processing time. Online algorithms for linear deteriorating jobs with release times have been studied in [23, 60]. In [60], the application of deteriorating jobs with release times to steel production is discussed. Cheng and Ding [23] showed that the problem with release times on a single machine is strongly NP-hard for the case of identical normal processing time a or deteriorating rate b.

To the best of our knowledge, the only work on online scheduling of deteriorating jobs that we are aware of is by Cheng and Sun [22]. In this thesis, we present results in both contexts of online-list and online-time models for simple linear deteriorating jobs on parallel machines.

1.3.2 Energy-efficient scheduling of precedence-constrained jobs on parallel machines

In Section 1.3.1, we introduce a scheduling problem with no constraints present in the structure of the jobs, in other words, each job can be treated as a completely independent entity. However, there are practical situations where the execution of an operation or collection of operations can depend on some factors related to the problem domain. Some of these factors include storage, transportation, resource availability and maintenance, and process constraints which is related to a particular order in which operations must be fulfilled. A simple example can be found in queueing systems which are applicable to financial institutions, health care and service industries, where requests are fulfilled in the order of their arrivals so that earlier ones finish first. Other examples include automated processes in industrial assembly and fabrication systems in which some operations cannot be carried out without some prerequisite operation, for example, spray-painting process of a car in a vehicle assembly plant cannot begin without first assembling all vehicle parts.

These kind of restrictions in the scheduling process gave rise to the research and study of scheduling subject to constraints, a class of which consists of *precedence constraints*.

The concept of precedence constraints was introduced to capture the essence of scheduling where operations are not totally independent of each other. In simple terms, precedence constraints specify that a job cannot be started unless all the jobs preceding it have been completed. The precedence constraints between jobs is usually depicted with the use of a directed acyclic graph. Figure 1.2 illustrates the different types of precedence constraints.

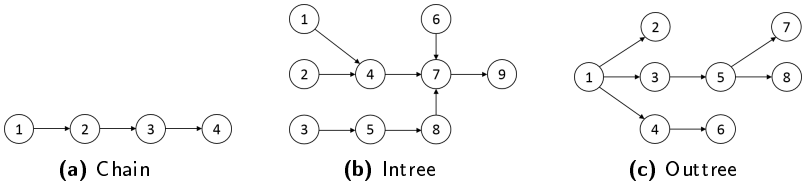


FIGURE 1.2: Examples of job precedence constraints.

The study of scheduling with precedence constraints began as early as 1970s with notable theoretical studies including [58, 61, 99]. Lenstra and Rinnooy Kan [61] were able to show that most of the relatively simple, classic scheduling problems become NP-complete when precedence constraints are added to these problems.

In this thesis, we present a study of the scheduling of jobs with chain precedence constraint on parallel machines with the objective of minimizing energy. Our motivation is two-fold. First we present an algorithm for solving the problem, and secondly, our goal is to use our algorithm as a case study in our study involving heterogeneous and parallel computing. The study of energy-aware scheduling on parallel machines or chip multiprocessors have received numerous attention in research, however, we are concerned with problems involving non-independent tasks. Some closely related works in this aspect include [14, 97]. Here, tasks are represented as directed acyclic graphs (DAG) where nodes represent the task while the edges represent the precedence between tasks. The cost associated with processing time and energy as well as communication time and energy are also given as input to the problem. We present a problem using similar model to the works mentioned but our focus in a task set with chain precedence constraint and each task can have a different deadline. We present both theoretical results and empirical results for this problem.

1.3.3 Energy-efficient flow time scheduling

Single processor scheduling. The theoretical study of dynamic speed scaling to reduce energy was initiated by Yao et al. [111] and has received a considerable amount

of attention [16, 26, 88]. Yao et al. considered the *infinite speed model* where a processor can dynamically change its speed to a value between zero and infinity. They studied the problem of online scheduling of jobs with arbitrary sizes and deadlines, and their objective is to produce a schedule such that all jobs complete within their deadline while minimizing the energy incurred by the schedule. The study has been extended to take into consideration the fact that in realistic situations processors can only adjust speeds within a finite speed spectrum with often pre-defined voltage levels. Here, processor voltage is determined from a finite spectrum of discrete voltage levels which in turn determines the speed levels the processor is capable of running at. Earlier approaches to the problem involve first computing the continuous version of the problem and then adjusting the speeds to the fit into the discrete voltage levels for the final schedule [47, 56]. Yao and Li [65] further extended the study and presented results where the computation of the continuous schedule is skipped entirely.

On the other hand, the problem of minimizing total flow time plus energy compacted into a single objective function defined as the sum of flow time and energy was proposed by Albers and Fujiwara [4]. They considered jobs of unit size and propose a speed scaling algorithm that changes the speed of the processor based on the number of *active jobs* (AJC), which refers to jobs that have arrived but are yet to be completed. They, however, studied a batched version of the algorithm where newly released jobs are queued until all jobs in the current batch are completed and showed that it is $8.3e(1+\Phi)^{\alpha}$-competitive for minimizing total flow time plus energy, where $\Phi = \frac{(1+\sqrt{5})}{2}$ is the *golden ratio*. Further improvement on their work has been presented by Bansal et al. [11, 12].

Multiple processor scheduling. The problem of scheduling on multiple processors with fixed speed and without energy considerations has been widely studied [8, 9, 19, 20, 62]. The use of multiple processors can be seen in practice in most modern computer systems and devices where the most common configuration consists of identical processing cores, for instance, processors found in home desktop machines and most recently, mobile phones and tablets. For the problem involving multiple processors, jobs still arrive in a sequential fashion and cannot be executed by more than one processor in parallel. Online algorithms such as *shortest remaining processing time* (SRPT) and *immediate dispatch* (IMD) [8] that are $\Theta(log\ P)$-competitive have been proposed for the migratory and non-migratory model respectively, where P is the ratio of the maximum job size to the minimum job size [8, 9, 62]. Furthermore, IMD has been shown to be $O(1+\frac{1}{\epsilon})$-competitive when using processors that are $(1+\epsilon)$ times faster. In the case where migration is allowed, SRPT has been to achieve a competitive ratio of 1 or even smaller when using sufficiently fast processors [71, 87].

Subsequently, Bunde [18] initiated the study of multiple processor scheduling that takes both flow time and energy into account and presented a study of the offline approximation for jobs with unit size. On the other hand, Lam et al. [57] presented the first online, non-migratory algorithm for minimizing total flow time plus energy for jobs of arbitrary size. Conserving energy with multiple processors typically involves distributing jobs as evenly as possible among the available processors in order to avoid running any processor at higher than required speed, therefore, it is only natural to consider techniques similar to *round-robin*. This is demonstrated by Lam et al [57] in the *classified round robin* (CRR) algorithm, which classifies jobs according to their sizes and allocates jobs of the same class to the processors in a round robin fashion. They showed that CRR can be $\Theta(log\ P)$-competitive for the problem of minimizing total flow time plus energy.

In this thesis, we present an empirical study that focuses on the analysis of the speed-scaling heuristic based on the number of active jobs and investigate the possibility of designing a simpler heuristic that is capable of achieving the sort of performance close to that of the speed-scaling heuristic. In addition, our investigation includes several speed selection, job selection and multi-processor allocation heuristics.

1.3.4 Parallel and heterogeneous computing with graphics processors

Over the years, chip manufacturers have greatly concentrated on improving single-thread performance of the central processing unit (CPU), which provides the bulk of the computing power in enterprise and home computer systems. However, the future of computing centres around parallelism and heterogeneous computing. As it is the current trend, the design and development of future microprocessors will continue to focus adding more computing cores instead of simply focusing on single-thread performance or frequency.

The Cell broadband engine. One of the earlier trends in heterogeneous computing involves the use of the Cell broadband engine [69]. The Cell processor was developed by Sony, Toshiba and IBM and it provides the processing power for the Sony Playstation 3 game console. The heterogeneous nature and multicore architecture quickly attracted attention in the science and research communities. A number of the initial works such as [10, 21, 54, 55, 108] focused on using the Cell processor to accelerate scientific computations including 1D/2D Fast Fourier Transforms, and several linear algebra techniques including QR factorization, Cholesky factorization, dense matrix operations and sparse matrix-vector operations. These results show that using the Cell processor for these applications resulted in significant performance increase compared to the conventional CPUs available during the same period.

GPU computing. Another kind of multiprocessor that quickly gained popularity among the scientific community is the graphics processing unit (GPU). The GPU is a multiprocessor primarily designed to accelerate compute tasks involved in graphics rendering. It is specifically designed for high throughput computation rather than low latency computation as is seen with the CPU. It is also optimized to handle compute tasks that require high levels of parallelism as is associated with rendering of pixels and other graphics-related tasks. Due to these reasons, the GPU is characterized by a massively parallel architecture with hundreds of processing elements. In addition, the GPU is readily available and the desktop versions can be very affordable. These factors make it very appealing to the scientific community, hence, the significant amount of attention it has received over the years. The use of graphics processors for accelerating applications is commonly referred to as *GPU computing* or *general purpose computing on GPUs* (GPGPU). Owens et al. provides a comprehensive survey and introduction in [85, 86].

GPU computing has seen applications in several fields. In Computer Science, there has been several work in literature that study the implementation of several algorithms on the GPU and evaluate their performance. Problems in graph theory including graph cuts, traversal and layout have already been studied in [31, 42, 72, 107]. Furthermore, sorting algorithms including merge sort, radix sort, quick sort and bitonic sort, just to mention a few, have been analysed on the GPU [37, 39, 94, 96, 100].

Another field of study where GPU computing has received a lot of attention is Bioinformatics. Some of the applications used in Bioinformatics are typically characterized as mainly requiring high throughput computation. Therefore, it is no surprise that the graphics processor has been adopted to accelerate the performance of such applications. GPU computing is applied to applications used for sequencing, alignment and database searches and some the works include implementation of the very popular Smith-Waterman algorithm [101] on the GPU [67], and tools such as [66, 98, 113, 114]. Majority of the works provided by the authors relating to GPU computing are accompanied with empirical studies demonstrating the advantages that the GPU offers over conventional computing with CPUs.

In this thesis, we are not only interested in presenting new implementations of algorithms on the GPU. We will be presenting a study detailing our experiences in developing applications on the GPU. We will study several factors that can easily be overlooked when designing applications for GPGPU which are key in order to achieve desired levels of performance on the GPU. In order to do this, we will select a few algorithms as representative case studies. Some of which are already known algorithms but have not been studied in the context of GPU computing, at least to the best of our knowledge, and others will be a novel approach.

1.4 Contribution of thesis

In Chapter 2 of this thesis, we present a theoretical study of the problem of scheduling linear deteriorating jobs on parallel machines with the objective of minimizing makespan. This is a joint work with Sheng Yu, Prudence Wong and Yinfeng Xu, and is published in the *Theory and Applications of Models of Computation* [112]. We present three main results in the online-time model, where jobs are associated with arbitrary release times.

Our first result concerns the List Scheduling (LS) on arbitrary number of parallel machines in which we show that the competitive ratio of LS is at least $(1 + b_{max})$. We present details of the adversary as well as mathematical proof for the adversary used to obtain this competitive ratio. The second result concerns the scheduling of simple linear deteriorating jobs on arbitrary number of machines. We show that no deterministic algorithm is better than $(1 + b_{max})^{1-\frac{1}{m}}$-competitive and that this also holds for the online-list model. Finally we extend our adversary to show that in the case of two-machine scheduling of jobs with simple linear deteriorating rates, no deterministic online algorithm is better than $(1 + b_{max})$-competitive.

In Chapter 3, we present a study of a problem concerned with energy-aware scheduling of n tasks with precedence constraints on m parallel machines. The type of precedence constraints we focus on is the chain precedence constraint, where each task can have at most one predecessor and at most one successor. A task is also characterized by a strict deadline that must be met. The parallel machines are assumed to be unrelated and are connected by an underlying communication network. The execution of a task on a machine costs time and energy and for each machine-job pair, the cost is given by a machine-job matrix. Similarly, communication across the network costs time and energy too given in a machine-machine cost matrix. We assume that the communication links can be asymmetric, that is, the cost depends on the direction between a pair of machines. We present an optimal, pseudo polynomial algorithm using a dynamic programming approach with a running time of $O(nm^2d_{max})$, where d_{max} is the largest deadline. In addition, we implement this algorithm in our empirical studies involving multi-core processors and graphics processors. The aim is to provide a representative case for serial, monadic dynamic programming formulations for analysis on the GPU.

In Chapter 4, we present a study of the problem of scheduling to minimizing total flow time plus energy. Results of our work was presented at the *11th Workshop on Models and Algorithms for Planning and Scheduling Problems, 2013* [83]. Our contribution to the problem is a comprehensive empirical study that aims to complement results obtained from theoretical studies. We implement and investigate several job selection, speed function and processor allocation heuristics. We start by considering SRPT and *shortest*

job first (SJF) job selection heuristics for both single processor and multi-processor setting. We demonstrate that SRPT is indeed better than SJF although the difference in performance is very small.

For single processor simulations, we investigate the effectiveness of dynamic speed scaling by comparing a speed-scaling heuristic with several fixed speed heuristics where the heuristics have no prior knowledge of the job set. We demonstrate that in such case, the speed scaling heuristic performs better than the fixed speed heuristics. However, we also designed a fixed speed heuristic that is capable of selecting a speed based on some prior knowledge of the job set and we show that, if we allow a fixed speed function to have some prior knowledge of the job set, we can achieve good results that are close to the results attainable with speed scaling. The fixed speed heuristic is also meant to serve as an alternative that is very simple and easy to implement compared to the speed-scaling heuristic. We then evaluate how the performance of the fixed-speed heuristic compares to the speed-scaling heuristic. We also demonstrate that when jobs arrive sparingly, the performance gain or effectiveness of speed scaling degrades because there is less pressure on the scheduler in terms of work.

Furthermore, we extend our simulations to multi-processor setting and evaluate the performance of several processor allocation heuristics including round robin. We demonstrate that, in general, using multiple processors can help save time and energy and this is regardless of the processor allocation heuristic in question. However, when job distribution is sparse, we show that there is little or no benefit from adding more processors.

In Chapter 5 of this thesis, we present a brief introduction to general purpose computing on graphics processing units (GPGPU). We introduce description of well-known graphics processor hardware and architecture from Advanced Micro Devices (AMD) and NVIDIA Corporation. We also cover topics such as stream computing and parallel computing frameworks. This chapter is meant to describe concepts related to GPGPU and serve as crash course on the subject in order to assist the reader with the understanding of the work presented in Chapter 6.

In Chapter 6, we present comprehensive and detailed empirical studies concerning parallel and heterogeneous computing with graphics processing units (GPU). Most of the work presented in literature related to GPU computing are merely concerned with speedup achieved in comparison to the best sequential implementation of a known algorithm. We are interested in the several factors that are involved in the development of applications for GPU computing including thread grouping and scheduling, memory management techniques and kernel optimizations related to the GPU hardware. We select a total of 4 applications as our test cases.

Our first application is a dynamic programming application mentioned earlier in this chapter for scheduling jobs with chain precedence constraint on parallel machines to minimize energy. This algorithm is meant to represent a class of dynamic programming formulations known as *serial monadic* formulation. It is a simple case of dynamic programming where the computation for each level only depends on the level directly preceding it (serial) and does not contain any recursive term in its definition (monadic). The second application is a novel implementation of the GapsMis algorithm [3, 13] for GPGPU. It is also based on dynamic programming, however, it can be described as *serial polyadic*. It is a practical tool for sequence alignment with the aim of allowing variable and bounded number of gaps.

The third application is a novel implementation of the Velocity-Verlet algorithm [104] on the GPU. It is a well-known algorithm used in n-body simulations and it is used for the numerical integration of Newton's laws of motion. We chose this algorithm to represent a class of n-body algorithms and because of its practical application in particle and molecular simulations. The last of our application is a tool for graph layout and visualization based on the well-known Fruchterman-Reingold algorithm [32]. It is very similar to the n-body algorithms and finds practical applications in tools developed for visualizing graphs and network structures.

We implement sequential, multi-threaded and GPU kernel versions for each of these applications. We evaluate the performance based on several metrics which we define in order to capture several aspects of the performance the applications including, for instance, execution times of both GPU kernel and whole application, throughput tailored to individual application and also estimation of power and energy consumption for each application. We hope to highlight the challenges involved in developing heterogeneous applications and provide some insight using our case studies as the results presented in this chapter can also be extended to similar algorithms with similar characteristics.

Chapter 2

Online Scheduling of Linear Deteriorating Jobs on Parallel Machines

2.1 Introduction

In this chapter, we extend the study of simple linear deteriorating jobs in the *online-list* model to the *online-time* model. Recall that in the online-list model, jobs are released one at a time and an algorithm must schedule a released job before the next job is released. However, in the online-time model, each job can be characterized by a release time. In the online-list model, with an arbitrary number, m, of parallel machines, List Scheduling has been shown to be $(1+b_{\max})^{1-\frac{1}{m}}$-competitive, where $b_{\max}$ is the largest deteriorating rate. We extend this study to the online-time model, showing that for the case of two machines no deterministic online algorithm is better than $(1+b_{\max})$-competitive. This result implies that the problem becomes more difficult in the online-time model in comparison with the online-list model. In addition, we show that List scheduling is $(1+b_{\max})^{2(1-\frac{1}{m})}$, hence, it is optimal for the case where $m=2$.

In the organization of this chapter, we begin with a formal problem definition and description of notations in Section 2.2. In Section 2.3, we present the new lower bound results in the online-time model and details of the adversary employed are explained comprehensively. Finally in Section 2.4, we present our concluding remarks.

2.2 Preliminaries

2.2.1 Problem definition

Consider a set of n jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ to be scheduled on m machines labelled $\{M_1, M_2, \ldots, M_m\}$. Each job J_j is associated with a *release time*, r_j, and a *processing time*, p_j. We assume that the jobs are indexed non-decreasing order of their release times such that $r_j \leq r_{j+1}$. The *start time* of job J_j, denoted by s_j, refers to the time at which job J_j starts being executed on a processor. The processing time p_j of a job J_j depends on its start time s_j and this implies that p_j differs with different schedules.

We focus on *linear deterioration* where a job J_j is characterized by a *normal processing time*, $a \geq 0$, and a deteriorating rate, $b_j > 0$, such that $p_j = a_j + b_j s_j$. Consider a case where all jobs in the job set have identical normal processing times, we refer to this case as *simple linear deterioration*. We denote by $b_{\max}$ the largest deteriorating rate among all jobs in the job set.

Our concentration is on an online formulation where jobs arrive in an online manner which means that jobs are not available at the beginning and information related to a particular job is only known on arrival. Furthermore, an algorithm constructs a schedule that shows, for each $J_j \in \mathcal{J}$, the machine on which J_j is to be executed. Let us consider a schedule S, the completion time of each job $J_j \in \mathcal{J}$ in S is denoted by $c_j(S)$. For any machine M_k, where $1 \leq k \leq m$, the group of jobs assigned to M_k by the schedule S is denoted by $\mathcal{J}^k(S)$. This can also be written as $\mathcal{J}^k$ for simplicity when no ambiguity exists.

We denote by $C^k_{\max}(S)$ the *makespan* of machine M_k, which is the largest completion time among the jobs in $\mathcal{J}^k(S)$. The makespan of the schedule S, denoted by $C_{\max}(S)$, is the largest makespan among all machines, hence,

$$C^k_{\max}(S) = \max_{j \in \mathcal{J}^k(S)} \{c_j(S)\}$$

and

$$C_{\max}(S) = \max_{1 \leq k \leq m} \{C^k_{\max}(S)\}$$

The objective of the problem is to minimize the makespan of the constructed schedule. Furthermore, we describe the following terms,

List Scheduling (LS). This comprises of a list into which jobs are inserted in non-decreasing order of arrival. The next available job in the list is assigned to whichever machine becomes idle.

OPT. This denotes the optimal offline algorithm (and its schedule).

Let us consider the following example that helps to illustrate the working principle behind LS along with a counterpart optimal schedule.

Example 2.1. *Consider a set of five jobs labelled $J_1, J_2, \ldots, J_5$ according to their release times, illustrated in Figure 2.1, to be scheduled on two machines, M_1 and M_2. The illustrations only show jobs scaled according to the deteriorating rates and not according their processing times so jobs J_1 to J_4 all have the same deteriorating rate while J_5 has the largest deteriorating rate.*

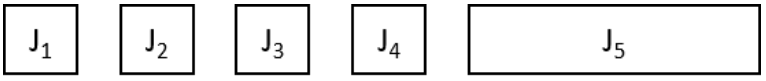


FIGURE 2.1: An illustration of jobs based on the deteriorating rates.

For these jobs, the resulting schedules constructed by LS *and* OPT *are illustrated in Figure 2.2. Without loss of generality, we assume that* LS *starts with machine M_1 and continues subsequent assignments accordingly. On the other hand, the schedule illustrated by* OPT *gives an optimal assignment for the same set of jobs. Note that the illustrations in Figure 2.2 do not depict the processing times of the jobs rather they merely depict job assignments on each machine.*

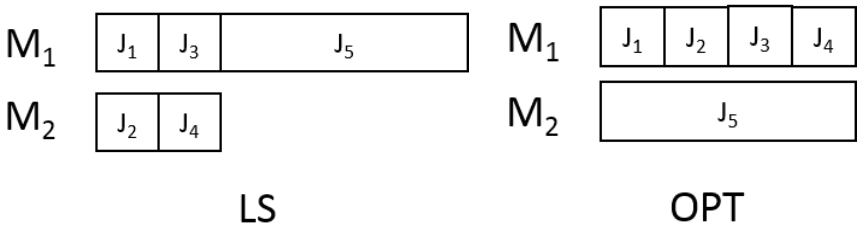


FIGURE 2.2: An illustration of schedules constructed by LS and OPT for the job set shown in Figure 2.1.

2.2.2 Property of simple linear deterioration

Let us consider the following example that shows the assignment of a sequence of jobs on a single machine. All jobs are assumed to have the same deteriorating rate. For full details and proof for the properties relating to simple linear deterioration the reader is referred to Property 1 in [112]. Example 2.2 aims to provide some intuition into these properties.

Example 2.2. *Consider the sequence of n jobs labelled $J_1, J_2, \ldots, J_n$ to be scheduled on a single machine. As in the previous example jobs are only scaled according to their deteriorating rates. Figure 2.3 illustrates an assignment of these jobs on a machine. It shows the start times and completion times of these jobs. Note that we assume $t_0 > 0$.*

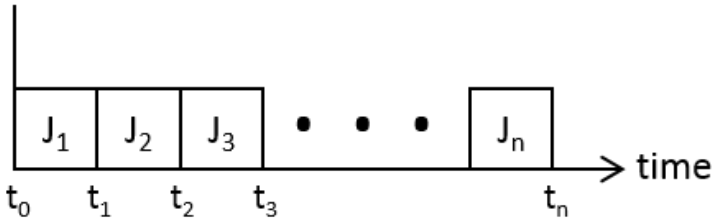


FIGURE 2.3: An illustration of n jobs assigned to one machine.

Given this assignment, we can compute the completion times, labelled $t_1, t_2, \ldots, t_n$, as follows,

$$t_1 = t_0 + b_1 s_1 = t_0 + b_1 t_0$$
$$t_1 = t_0(1 + b_1) \tag{2.1}$$

Equation 2.1 above computes the completion time t_1 for job J_1. Since the start time of job J_3 depends on the completion time of job J_2 which in turn depends on the completion time of job J_1, we can compute t_2 and t_3 as follows,

$$t_2 = t_0(1 + b_1)(1 + b_2) \tag{2.2}$$
$$t_3 = t_0(1 + b_1)(1 + b_2)(1 + b_3) \tag{2.3}$$

Hence, for n jobs, we can generalize the computation of the completion time, t_n, for the n-th job as follows,

$$t_n = t_{n-1} + b_n t_{n-1}$$
$$t_n = t_0(1 + b_1)(1 + b_2)(1 + b_3) \cdots (1 + b_n) \tag{2.4}$$

2.3 New lower bounds in online-time model

In this section, we present our results on the new lower bounds obtained for the online-time model for simple linear deterioration. In Section 2.3.1, we describe the proof of the lower bound for LS as well as the adversary employed in obtaining this lower bound. Then Section 2.3.2, firstly, we extend this adversary to show how to obtain a lower bound on any deterministic algorithm in online-list model. Then we adopt the adversary to obtain

a lower bound on any deterministic algorithm in online-time model for a two-machine case.

2.3.1 List Scheduling on m parallel machines

We begin with the following Lemma.

Lemma 2.1. *Consider simple linear deteriorating jobs with arbitrary release times. The competitive ratio of* LS *is at least* $(1 + b_{\max})$.

Proof. We introduce an adversary that works in stages. Each stage, k, consists of jobs with either of two distinctive deteriorating rates and jobs are released at a particular time that marks the beginning of the stage.

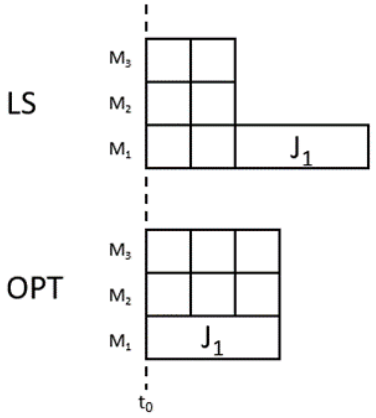


FIGURE 2.4: Stage 1 of adversary: The deteriorating rate, b_1, of job J_1 satisfies $1+b_1 = (1+b)^3$ where b is the deteriorating rate of each of the smaller jobs depicted in the illustration. Jobs are released at time t_0 and scaled according to deteriorating rates only.

Stage $k = 1$. Figure 2.4 illustrates the assignment of jobs on three machines in stage 1 for both LS and OPT. The adversary starts by releasing $m(m-1)$ jobs, each with deteriorating rate b, at time t_0. LS will distribute these jobs evenly across all machines so that it achieves a completion time of $t_0(1+b)^{m-1}$ across all machines. The adversary then proceeds with releasing a single job J_1 with deteriorating rate b_1, such that $1+b_1 = (1+b)^m$. As a result, LS will incur a completion time of $t_0(1+b)^{m-1}(1+b_1)$ on which ever machine it chooses to assign job J_1 to. However, OPT can assign job J_1 on one of the machines while the other jobs are assigned to the remaining machines. Consequently, all machines in OPT will have the same completion time equal to $t_0(1+b_1)$, meanwhile, LS is still executing job J_1 past this time. Furthermore, one can observe that at any given point in time all machines in OPT are busy while for LS, some machines can be idle.

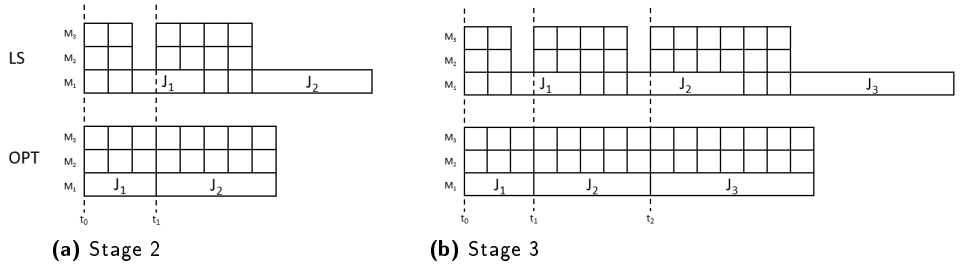


FIGURE 2.5: Stages 2 and 3 of adversary: (A) In stage 2, jobs are released at time $t_1 = t_0(1 + b_1)$ and as a result, LS cannot schedule them earlier on M_1 and M_2. This means machines M_1 and M_2 are idle until time t_1. (B) In stage 3 new jobs start arriving at time $t_2 = t_1(1 + b_2)$ and the trend continues as with the previous stages.

Stage $k \geq 2$. Figure 2.5 illustrates stages 2 and 3 of the adversary following from stage 1 shown in Figure 2.4. For each stage $k \geq 2$, let us define the term p_k such that,

$$p_k = k(m-1) + 1 \tag{2.5}$$

and the deteriorating rate b_k for job J_k such that,

$$1 + b_k = (1 + b)^{p_k} \tag{2.6}$$

At the start of each stage $k \geq 2$, LS is still processing job J_{k-1} on one of the machines until time $t_{k-1}(1+b)^{(k-1)(m-1)}$. Meanwhile, the rest of the machines in LS and OPT are idle from time t_{k-1}. The adversary starts the stage by releasing $(k-1)(m-1)^2+m(m-1)$ jobs with deteriorating rate b at time t_{k-1}. Without loss of generality, let us assume that LS has assigned job J_{k-1} on machine M_1. Consequently, LS will assign $(k-1)(m-1)$ jobs to each machine $M_2, M_3 \cdots, M_m$ and the remaining jobs are assigned to all machines $M_1, M_2 \cdots, M_m$ so that each machine gets $(m-1)$ more jobs. Then, the adversary releases a job J_k with deteriorating rate b_k which LS can schedule on any machine, say M_1, so that the completion time of machine M_1 is given by the following expression,

$$t_{k-1}(1 + b)^{(k-1)(m-1)}(1 + b)^{m-1}(1 + b_k) = t_k(1 + b)^{k(m-1)}$$

while machines $M_2, M_3 \cdots, M_m$ are idle from time

$$t_{k-1}(1 + b)^{(k-1)(m-1)}(1 + b)^{m-1} = t_{k-1}(1 + b)^{p_k - 1} < t_k$$

However, OPT can assign J_k on one of the machines and the remaining jobs on the rest of the machines such that the completion time t_k is the same across all machines, where,

$$t_k = t_{k-1}(1 + b)^{(k-1)(m-1)+m} = t_{k-1}(1 + b)^{p_k}$$

Finally, the ratio of LS to OPT can be expressed as follows,

$$\frac{C_{\max}(\text{LS})}{C_{\max}(\text{OPT})} = (1+b)^{k(m-1)} = (1+b)^{\frac{k(m-1)}{k(m-1)+1}}$$

This ratio can be arbitrarily close to $(1 + b_{\max})$ if the number of stages is arbitrarily large. The lemma then follows.

□

2.3.2 Lower bounds for deterministic online scheduling

We now consider deterministic algorithms for scheduling simple linear deteriorating jobs. Firstly, we show that for an arbitrary number of machines, m, we can obtain a lower bound on any deterministic algorithm by adopting and using the first stage of the adversary described in Section 2.3.1. For this reason we present the following lemma.

Lemma 2.2. *Consider simple linear deteriorating jobs. No deterministic algorithm is better than* $(1 + b_{\max})^{1-\frac{1}{m}}$*- competitive. This also holds for online-list model.*

Proof. Let us denote by ALG any *reasonable* and deterministic online algorithm. We say that ALG is reasonable if it does not allow unnecessary idle times in its schedule. Consider the first stage of the adversary described in the proof of Lemma 2.1. The adversary begins by releasing $m(m-1)$ jobs, each with a deteriorating rate b. ALG could assign m or more jobs to one of the machines and if this is the case, the makespan is at least $t_0(1+b)^m$. On the other hand, OPT can distribute the jobs evenly across all machines so that each machine is assigned $m-1$ jobs which results in a makespan of $t_0(1+b)^{m-1}$. Consequently, the competitive ratio is at least $1+b > (1+b)^{1-\frac{1}{m}}$, therefore, the lemma follows since $b_{\max} = b$ in this case.

However, if ALG assigns exactly $m-1$ jobs to each machine, the adversary will release a job with deteriorating rate b_1 such that $1 + b_1 = (1+b)^m$. As a result, the makespan of ALG will be $t_0(1+b)^{m-1}(1+b_1)$ while that of OPT is $t_0(1+b_1)$. This leads to the following competitive ratio,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = \frac{t_0(1+b)^{m-1}(1+b_1)}{t_0(1+b_1)} = (1+b)^{m-1} = (1+b_1)^{1-\frac{1}{m}}$$

Hence, the lemma follows.

□

We then extend the adversary to a case of two machines in the online-time model and show that no deterministic online algorithm is better than $(1 + b_{\max})$-competitive. The challenge here is that when the jobs with deteriorating rate b are released, ALG does not necessarily distribute them evenly between the two machines. Therefore, we adapt the adversary such that before job J_k with deteriorating rate b_k is released in stage k, more jobs with deteriorating rate b is released in order to equalize the completion time on both machines. We describe this process in the following theorem.

Theorem 2.3. *Consider two-machine scheduling of jobs with arbitrary release times and simple linear deteriorating rates. No deterministic online algorithm is better than* $(1 + b_{\max})$*-competitive.*

Proof. Consider any reasonable, deterministic online algorithm ALG. For each stage k, define b_k such that $1 + b_k = (1 + b)^{k+1}$.

Stage 1. The adversary starts by releasing two jobs of deteriorating rate b at time $t_0 > 0$. If ALG assigns both jobs to the same machine then we obtain the desired competitive ratio as given below,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = \frac{t_0(1+b)^2}{t_0(1+b)} \qquad \text{where } b_{\max} = b$$

Alternatively, in the case where ALG assigns a job to each machine the adversary then releases a job J_1 with deteriorating rate b_1 at time t_0, so that $(1 + b_1) = (1 + b)^2$. Without loss of generality, let us assume that ALG assigns J_1 to machine M_1. As a result, $C_{\max}(\text{ALG}) = t_0(1 + b)(1 + b_1)$, given by machine M_1 while the completion time of M_2 is $t_0(1 + b)$. However, OPT can assign J_1 to M_1, for instance, and remaining two jobs to M_2 which will result in a uniform completion time of $t_0(1+b_1)$ on both machines.

Stage 2. At the start of stage 2, three jobs of deteriorating rate b are released at time $t_0(1 + b_1)$. As illustrated in Figure 2.6, there are three representative cases to consider.

Case (a). ALG assigns all three jobs to machine M_1 and as a result, $C_{\max}(\text{ALG}) = t_0(1 + b_1)(1 + b)^4$. Meanwhile, OPT can assign two jobs to M_1 and the other remaining job to M_2 so that $C_{\max}(\text{OPT}) = (1 + b_1)(1 + b)^2$. Then,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = (1 + b)^2 = (1 + b_1) = (1 + b_{\max})$$

Case (b). In this case ALG assigns one of the jobs to M_1 while the other is assigned to M_2 so that the makespan of both machines is $t_0(1 + b_1)(1 + b)^2$. This is also equivalent to the makespan of OPT.

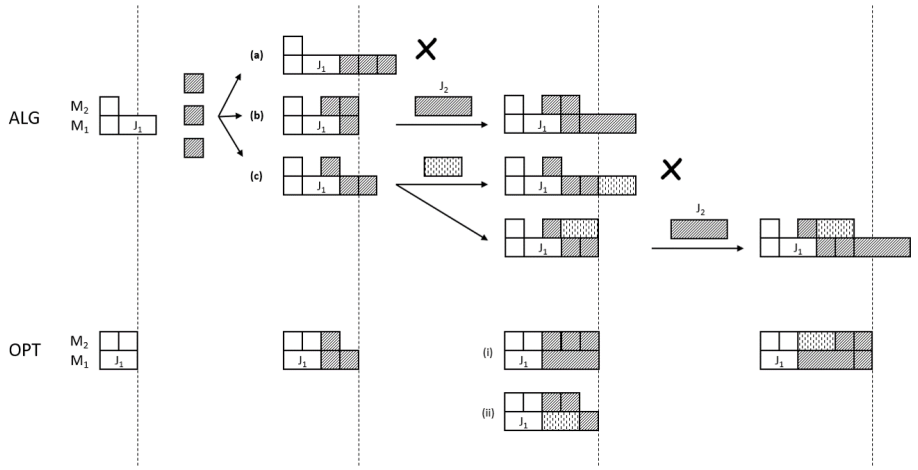


FIGURE 2.6: Illustration of the 3 representative cases, labelled (a), (b) and (c), in stage 2 of the general lower bound.

Case (c). ALG assigns two jobs to M_1 and one job to M_2 so that the resulting makespan of both machines are $t_0(1+b_1)(1+b)^3$ and $t_0(1+b_1)(1+b)$ respectively. The adversary then releases a job, J', with deteriorating rate b_1. After the adversary has released J', OPT can assign these jobs properly such that $C_{\max}(\text{OPT}) = t_0(1+b_1)(1+b)^3$, as illustrated by diagram (ii) in Figure 2.6. If ALG assigns J' to M_1 then the completion time of M_1 becomes $t_0(1+b_1)^2(1+b)^3$ and the ratio becomes $1+b_1 = 1+b_{\max}$. Otherwise, ALG assigns J' to M_2 resulting in a completion time of $t_0(1+b_1)(1+b)^3$ on both machines.

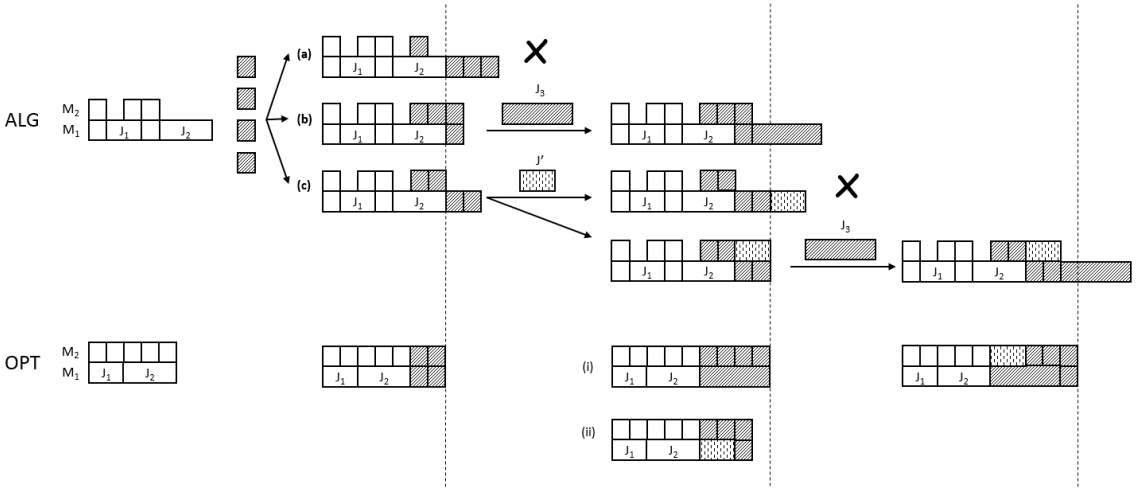


FIGURE 2.7: Example showing Stage 3 of the general lower bound.

When we consider Cases **(b)** and **(c)**, we observe that the situation becomes very similar to the proof of LS in Lemma 2.1, such that the completion time is the same for both machines in ALG. In such scenario, the adversary finalizes the stage by releasing a job with deteriorating rate b_2 resulting in a competitive ratio of $(1+b)^2$. Figure 2.7 shows an instance for Stage 3.

Stage k**.** Let us denote by t_{k-1} the makespan of OPT at the beginning of Stage k, where the exact value of t_{k-1} depends on ALG. At the beginning of Stage k, ALG is still processing job J_{k-1} of deteriorating rate b_{k-1} on machine M_1 while M_2 is idle during this period, such that $C_{\max}(\text{ALG}) = t_{k-1}(1+b)^{k-1}$. We will show that this invariant holds at the end of Stage k. At time t_{k-1}, the adversary first releases $k+1$ jobs of deteriorating b. For simplicity, we consider an odd-valued k and assume $k = 2h+1$. The other case where k is an even value is similar and so we skip the details. We then consider three familiar cases.

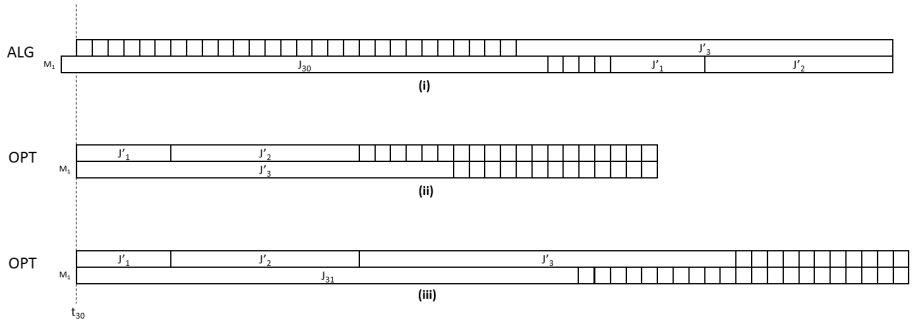


FIGURE 2.8: Illustration of the general lower bound for Stage 31 where $k = 31$ and $h = 15$. (i) At t_{30} ALG is still processing J_{30} from Stage 30 on M_1. (ii) OPT has completed all jobs released before t_{30} including J_{30}. (iii) OPT schedule for Stage 31. Note that OPT can maintain the same makespan on both machines.

Case (1). In this case, ALG assigns at least $h+2$ jobs to M_1 which results in a makespan of at least $t_{2h}(1+b)^{2h+h+2}$. Meanwhile, OPT assigns h jobs to each machine so that the makespan of OPT is $t_{2h}(1+b)^{h}$. Then, the following competitive ratio holds and we are done,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} \geq (1+b)^{2h+2} = (1+b_{2h+1}) = (1+b_{\max})$$

Case (2). In this case, ALG assigns one job to M_1 and $2h+1$ jobs to M_2 so that the completion time of both machines is equal to $t_{2h}(1+b)^{2h+1}$. At this point we have a situation similar to Lemma 2.1. The adversary then releases a job J_{2h+1} of deteriorating rate b_{2h+1}, where $1+b_{2h+1} = (1+b)^{2h+2}$. OPT can assign J_{2h+1} to M_1 and the other jobs to M_2 so that $C_{\max}(\text{OPT}) = t_{2h}(1+b)^{2h+2}$. Then we have the following ratio,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = \frac{t_{2h}(1+b)^{2h+1+2h+2}}{t_{2h}(1+b)^{2h+2}} = (1+b)^{2h+1}$$

The invariant holds for Stage $2h+1$ so the adversary can proceed to the next stage.

Case (3). In this case, ALG assigns x jobs to M_1 and $2h+2-x$ jobs to M_2, where $1 < x < h+2$. Consequently, the completion time of M_1 and M_2 is equal to

$t_{2h}(1+b)^{2h+x}$ and $t_{2h}(1+b)^{2h+2-x}$, respectively. Then, the adversary releases extra jobs with the aim of equalizing the completion time of both machines. In the first attempt, the adversary releases a job J' of deteriorating rate b', such that, $(1+b') = \frac{t_{2h}(1+b)^{2h+x}}{t_{2h}(1+b)^{2h+2-x}} = (1+b)^{2(x-1)}$.

To be precise, the adversary releases jobs $J'_1, J'_2, \cdots$ with deteriorating rates $b'_1, b'_2, \cdots$ such that $1+b'_i = (1+b)^{2i(x-1)}$ until the ALG is forced to assign the first of such jobs to M_2. This will be the case because ALG cannot keep assigning these jobs to M_1 otherwise the makespan becomes to large. More precisely, i is defined such that $2^{i-1} < h \leq 2^i(x-1)$. If ALG assigns the jobs $J'_1, J'_2, \ldots, J'_i$ to M_1, this will result in a makespan of $t_{2h}(1+b)^{2h+x+2(x-1)+2^2(x-1)+\cdots+2^i(x-1)} = t_{2h}(1+b)^{2h+(2^{i+1}-1)(x-1)+1}$. However, OPT can assign J'_i to M_1 and $J'_1, J'_2, \ldots, J'_{i-1}$ to M_2 while the rest jobs are distributed evenly between both machines resulting in the same completion times. As a result, OPT will have a makespan equal to $t_{2h}(1+b)^{h+(x-1)+2(x-1)+\cdots+2^{i-1}(x-1)} = t_{2h}(1+b)^{h+(2^i-1)(x-1)}$. Hence, by the definition of i,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = (1+b)^{h+2^i(x-1)+1} \geq (1+b)^{2h+1} = 1+b_{\max}$$

In essence, we can assume that there exists a job J'_j, where $1 \leq j < i$, such that ALG assigns J'_j to M_2 and jobs $J'_1, \ldots, J'_{j-1}$ to M_1. This results in equivalent completion times on both machines such that the completion time of M_1 and M_2 is $t_{2h}(1+b)^{2h+2-x+2^j(x-1)}$ and $t_{2h}(1+b)^{2h+x+2(x-1)+\cdots+2^{j-1}(x-1)}$, respectively. On the other hand, OPT can assign J_{2h+1} to M_1 and jobs $J'_1 \cdots, J'_j$ to M_2 while the remaining jobs are distributed evenly between both machines to achieve the same completion time on both machines. Consequently, OPT obtains a makespan equal to $t_{2h}(1+b)^{2h+2+2(x-1)+\cdots+2^j(x-1)+2h+2} = t_{2h}(1+b)^{2h+2+(2^j-1)(x-1)}$. OPT is able to achieve such a schedule because $2(x-1)+\cdots+2^j(x-1) \leq (2^j-2)(x-1)+2h$, where the inequality is as a result of the definition of j leading to $2^{j-1}(x-1) \leq h$. In summary, the ratio after Stage k can be expressed as follows,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = (1+b)^{2h+1} = (1+b)^k$$

The variant holds for Stage k. Figure 2.8 illustrates an example showing Stage 31 where $x = 4$.

In both Cases **2** and **3**, we have the following competitive ratio which can approach $1+b_{\max}$ by having arbitrarily large k,

$$\frac{C_{\max}(\text{ALG})}{C_{\max}(\text{OPT})} = (1+b)^k = (1+b_{\max})^{\frac{k}{k+1}}$$

□

2.4 Conclusion

In this chapter, we study the problem of online scheduling of jobs with linear deteriorating rate on parallel machines. Precisely, we consider jobs with simple linear deterioration, i.e., $p_j = b_j s_j$. We also focus on the online-time model where a release time is associated with each job. For this case, we show that List Scheduling (LS) is $(1 + b_{\max})^{2(1-\frac{1}{m})}$-competitive, where $b_{\max}$ is the largest deteriorating rate among all jobs. We also show that for an arbitrary number of machines, m, no deterministic online algorithm is better than $(1 + b_{\max})^{1-\frac{1}{m}}$-competitive. Furthermore, we also show that on two machines, no deterministic online algorithm is better than $(1 + b_{\max})$-competitive.

We conjecture that it is possible to extend the adversary used for the two-machine case to m machines. An immediate open question that arises is whether the gap between the upper and lower bounds can be closed.

Chapter 3

Energy-Efficient Scheduling of Jobs with Precedence Constraints

3.1 Introduction

In this chapter, we present a study of the problem of scheduling jobs with precedence constraints on parallel machines. The type of precedence constraint we will focus on is the *chain* precedence constraint. The jobs in a given chain are characterized by *deadline* and a job is not allowed to be executed past its deadline. The machines are considered to be *unrelated* and may also be viewed as, for instance, processing components or chips in a system-on-chip architecture with an underlying network substructure for inter-chip communication. The time and energy required by a processor in order to execute any particular job is defined in lookup tables. Likewise, time and energy required for communication between a given pair of machines is also defined in lookup tables. The objective of a solution is to compute a feasible such that all jobs meet their deadline constraint while minimizing the total amount of energy incurred by the schedule.

We present a dynamic programming solution to the problem that yields an algorithm that runs in pseudo-polynomial time. We also present the proof to show that the algorithm will always compute an optimal schedule where one exists. The problem presented in this chapter is also intended to be used as a case study in Chapter 6.

The rest of this chapter is organized as follows. In Section 3.2 we present a formal definition of the problem and description of model. Then in Section 3.3.1 we present the dynamic programming approach, as well as a description of the algorithm and the proof in Section 3.3.2. Finally, we conclude in Section 3.4.

3.2 Preliminaries

3.2.1 Problem definition

Consider a set of m unrelated, parallel machines given by $\mathcal{M} = \{M_1, \ldots, M_m\}$ and a set of n jobs given by $\mathcal{J} = \{J_1, \ldots, J_n\}$. The set $\mathcal{J}$ is characterized by a chain precedence constraint such that, given a pair of jobs $J_j, J_{j+1} \in \mathcal{J}$, job J_j must be completed before the processing of J_{j+1} can begin. Each job J_j is also characterized by a strict deadline, denoted by d_j, that refers to the time by which the job must be completed. Any job can be scheduled on any machine.

The time required by a particular machine to execute a given job is given by a table referred to as *processing time matrix*, denoted by $\mathcal{PT}[1 \ldots m, 1 \ldots n]$. Thus, the time required by a machine M_i to process a job J_j is given by the value $\mathcal{PT}[i, j]$. The execution of a job on a particular machine also incurs some amount of energy defined in the *processing energy matrix*, $\mathcal{PE}[1 \ldots m, 1 \ldots n]$.

When two consecutive jobs are executed on different machines, say M_i and M_k, we assume that some time is required for synchronization between the two machines during which data is transmitted between both machines. The amount of time required perform this synchronization phase is given by the *communication time matrix*, denoted by $\mathcal{CT}[1 \ldots m, 1 \ldots m]$, such that the value $\mathcal{CT}[k, i]$ gives the amount of time required to migrate from machine M_k to M_i. In addition, we also assume that the communication link also consumes some amount of energy defined in the *communication energy matrix*, denoted by $\mathcal{CE}[1 \ldots m, 1 \ldots m]$. We assume that the communication links are asymmetric, that is, $\mathcal{CT}[k, i]$ or $\mathcal{CE}[k, i]$ may or may not be equal to $\mathcal{CT}[i, k]$ or $\mathcal{CE}[i, k]$, respectively. Note that the synchronization between two machines must be completed before the execution of the next job can start. On the other hand, there are no costs in terms of time and energy incurred when two consecutive jobs in the chain are executed on the same machine.

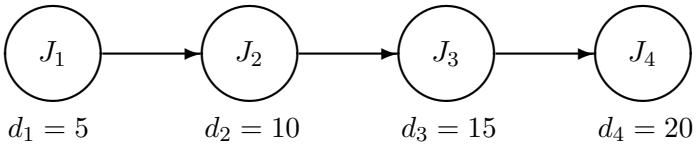
The objective is to construct a *feasible* schedule that minimizes the total amount of energy required to execute all jobs in the chain. A feasible schedule shows, for each job, the machine it is to be executed on as well as the start time, s_j, and completion time, c_j, such that $c_j \leq d_j$ for all $J_j \in \mathcal{J}$. This problem is an offline problem so all jobs in the chain are available from the beginning.

3.3 Discussion

3.3.1 A dynamic programming solution

In this section, we present a dynamic programming solution to the problem. Let us start with the following example.

Example 3.1. *Consider the following jobs shown in the illustration below to be scheduled on 2 machines.*



The processing time matrix and processing energy matrix are shown in Table 3.1a and Table 3.1b, respectively, while the communication time matrix and the communication energy matrix are shown in Table 3.1c and Table 3.1d, respectively.

	J_1	J_2	J_3	J_4
M_1	5	3	5	4
M_2	4	2	7	5

(a) Processing time matrix

	J_1	J_2	J_3	J_4
M_1	5	4	2	6
M_2	3	7	3	3

(b) Processing energy matrix

	M_1	M_2
M_1	0	1
M_2	2	0

(c) Communication time matrix

	M_1	M_2
M_1	0	2
M_2	1	0

(d) Communication energy matrix

TABLE 3.1: Lookup tables showing processing and communication costs with respect to time and energy.

In order to construct a feasible schedule, our dynamic programming formulation must first compute for each job J_j a table, denoted by $\mathcal{S}[1 \dots m, 1 \dots n, 1 \dots d_j]$, so that the value $\mathcal{S}[i, j, t]$ is the least amount of energy required to execute all jobs in the chain up to J_j such that J_j completes at time t on machine M_i, where $t \leq d_j$. We assume that $\mathcal{S}[i, j, t] = \infty$ when a job J_j cannot be completed at time t on machine M_i. Furthermore, if for all machines a job cannot be scheduled to satisfy its deadline constraint, then the algorithm terminates and reports that a schedule cannot be computed. The dynamic programming is comprised of two parts.

The first part deals with the first job in the chain, J_1. The recurrence given in Equation (3.1) describes how the solution for the first job in the chain is computed.

$$\mathcal{S}[i,1,t] = \begin{cases} \mathcal{PE}[i,1] & \text{if } \mathcal{PT}[i,1] \leq d_1 \text{ and } t \in \{\mathcal{PT}[i,1], \ldots, d_1\} \\ \infty & \text{otherwise} \end{cases} \tag{3.1}$$

When we consider the solution for the first job, we can easily observe that the feasible entries for $\mathcal{S}[i,j,t]$ are equivalent to the amount of energy required to execute J_1 on any machine M_i, given by $\mathcal{PE}[i,1]$. Table 3.2 gives the solution to the table for J_1 following our Example 3.1.

	t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
J_1	M_1	∞	∞	∞	∞	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	M_2	∞	∞	∞	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

TABLE 3.2: Dynamic programming table for job J_1 in the problem described in Example 3.1.

After we have determined all energy values for J_1 up to the time d_1, we simply initialize the remaining entries from $t = 6, \ldots, 20$ to the value at time d_1. From the table, we can see that it is not possible to complete J_1 by $t = 4$ on M_1 and by $t = 3$ on M_2 as indicated by the ∞ values. Therefore, possible feasible solutions for J_1 is to complete it by time $t = 4$ or $t = 5$ and both incurs 3 units of energy.

The other part of the dynamic programming deals with the remaining jobs, $j > 1$, after a feasible solution has been computed for J_1. This is given by the recurrence in Equation (3.2).

$$\mathcal{S}[i,j,t] = \begin{cases} \min\limits_{1 \leq k \leq m} \{\mathcal{S}[k,j-1,t_k] + \mathcal{CE}[k,i]\} + \mathcal{PE}[i,j] & \text{if } t \leq d_j \\ \infty & \text{otherwise} \end{cases} \tag{3.2}$$

where $j > 1$ and $t_k = t - \mathcal{PT}[i,j] - \mathcal{CT}[k,i]$

The idea is that at each point in time on each machine, we try to extend the previously obtained solution for job J_{j-1} while taking into account the amount of energy that will be required for migration between machines. Following from Equation (3.2), Table 3.3 gives the solution to the dynamic programming table for all jobs.

Hence, from Table 3.3, a feasible schedule that gives a solution to the problem in Example 3.1 is described below with a total energy consumption of 15 units.

$$J_1 \to M_2, \qquad J_2 \to M_1, \qquad J_3 \to M_1, \qquad J_4 \to M_2$$

	t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
J_1	M_1	∞	∞	∞	∞	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	M_2	∞	∞	∞	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	9	8	8	8	8	8	8	8	8	8	8	8	8
	M_2	∞	∞	∞	∞	∞	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	11	10	10	10	10	10	10	10
	M_2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	13	13	13	13	13	13	13	13
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	17	16	16	16
	M_2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	16	16	15

TABLE 3.3: Solution for the dynamic programming table for the problem described in Example 3.1.

3.3.2 Algorithm DPS

The algorithm DPS computes the table $\mathcal{S}$ and takes as input $\mathcal{J}$, $\mathcal{M}$, $\mathcal{CE}$, $\mathcal{CT}$, $\mathcal{PE}$ and $\mathcal{PT}$. In the design of DPS we first begin with the following observation. Consider the solution given in Table 3.3. For each job, it is not absolutely necessary to store values in the table for any time beyond the deadline of that particular job. In essence, we can discard these values and only maintain the required values for each job which are the ones leading up to the deadline of that job. Table 3.4 is the same solution as Table 3.3 except for the fact that redundant values have been truncated.

	t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
J_1	M_1	∞	∞	∞	∞	5															
	M_2	∞	∞	∞	3	3															
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	9	8	8										
	M_2	∞	∞	∞	∞	∞	10	10	10	10	10										
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	11	10	10					
	M_2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	13	13	13					
J_2	M_1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	17	16	16	16
	M_2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	16	16	15

TABLE 3.4: Solution for the dynamic programming table for the problem described in Example 3.1 without the redundant values.

As a result we must re-define t_k from the recurrence given in Equation (3.2) as follows,

$$t_k = \min\{t - \mathcal{PT}[i,j] - \mathcal{CT}[k,i], d_{j-1}\}$$

This will enable us take into account the fact that values beyond the deadline of a preceding job are undefined. Hence, we restrict the lookup time index, t_k, to the value

of deadline of the preceding job. Algorithm 1 describes the pseudo code for the DPS algorithm.

Algorithm 1 DPS

```
1:  procedure DPS(n, m, CE, CT, PE, PT)
    {For the first job only}
2:      for i ← 1 to m do
3:          for t ← 1 to d_1 do                          ▷ d_1 is the deadline of job j
4:              tidx ← t − PT[i, 1]
5:              if tidx ≥ 1 then
6:                  S[i, 1, t] ← PE[i, 1]
7:              else
8:                  S[i, 1, t] ← ∞
9:      for j ← 2 to n do                                ▷ For the remaining jobs
10:         for i ← 1 to m do
11:             for t ← 1 to d_j do
12:                 minEnergy ← ∞
13:                 for k ← 1 to m do                    ▷ Check all migration costs
14:                     tidx ← min{t − PT[i, j] − CT[k, i], d_{j−1}}
15:                     if tidx ≥ 1 then
16:                         tidx ← min{tidx, d_{j−1}}
17:                         if S[k, j − 1, tidx] ≠ ∞ then
18:                             e ← PT[i, j] + CE[k, i] + S[k, j − 1, tidx]
19:                             minEnergy ← min{e, minEnergy}
20:                 if minEnergy ≠ ∞ then
21:                     S[i, j, t] ← minEnergy           ▷ job j can finish at t on machine i
22:                 else
23:                     S[i, j, t] ← ∞
        return S[1 . . . m, 1 . . . n, 1 . . . d_j]
```

3.3.2.1 Proof of correctness

Let us begin with the following property.

Property 1. Consider the solution $\mathcal{S}$ for a given job J_j on a given machine M_i. $\mathcal{S}[i, j, t]$ is non-increasing for increasing values of t, where $1 \leq t \leq d_j$.

Proof. It is easy to observe that this property holds for J_1 because the value at each point in time on a particular machine is equal to the value given by $\mathcal{PE}[i, j]$ provided that $\mathcal{PT}[i, j] \leq d_1$. Even when a feasible schedule does not exist for J_1, all values will be ∞ and so the property still holds.

Then, in order to compute a solution for the next job, at each point in time on a given machine, we compute $\mathcal{S}[i, j, t]$, for $j > 1$ and $t \leq d_j$, based on all previously obtained values for job J_{j-1}. Since $\mathcal{S}[k, 1, t_k - 1] \leq \mathcal{S}[k, 1, t_k]$ for a given machine M_k, then it

implies that $\mathcal{S}[k,1,t_k-1]+\mathcal{CE}[k,i]+\mathcal{PE}[i,j] \leq \mathcal{S}[k,1,t_k]+\mathcal{CE}[k,i]+\mathcal{PE}[i,j]$ for $\mathcal{S}[i,j,t]$. Hence the property follows. □

An optimal substructure. Consider a chain given by the job set $\mathcal{J} = \{J_1, \ldots, J_n\}$ and the set of machines given by $\mathcal{M} = \{M_1, \ldots, M_m\}$. Let us denote by $B(i,j)$, where $1 \leq j \leq n$, a candidate solution obtained if job J_j is assigned to machine M_i, where $1 \leq i \leq m$. In order to compute a feasible completion time for a job on any given machine, we need to consider all the points in time leading up to the deadline of the job. For instance, consider a point in time, t, on some machine M_i. In order to determine whether it is feasible to complete a job, J_j, at time t, we use the following expression,

$$t_k = t - \mathcal{PT}[i,j] - \mathcal{CT}[k,i]$$

where t_k is the completion time of job J_{j-1} on machine M_k for all $k \in \{1, \ldots, m\}$. As a result it is possible to have more than one feasible solution on a single machine in terms of the possible completion times for a job on that machine. However, from Property 1 we only need to consider the point in time where $t = d_j$.

We denote by $\mathcal{B}_j$ the set of candidate solutions for a given job J_j so that using Property 1, we can enumerate all candidate solutions for J_j on all machines as,

$$\mathcal{B}_j = \{B(1,j), B(2,j), \ldots, B(m,j)\}$$

So for the first job, J_1, in the chain, $\mathcal{B}_1$ can be equally written as follows,

$$\mathcal{B}_1 = \{\mathcal{PE}[1,1], \mathcal{PE}[2,1], \ldots, \mathcal{PE}[m,1]\}$$

Let us denote the optimal solution for job J_1 in the chain by $\mathcal{OPT}(1)$. This implies that the smallest amount of energy required to execute J_1 is given by,

$$\mathcal{OPT}(1) = \min_{1 \leq k \leq m} \{B(k,1) \in \mathcal{B}_1\} \tag{3.3}$$

For remaining jobs, assume that each $B \in \mathcal{B}_j$ is optimal for completing J_j on a particular machine, where $j > 1$. For instance, $B(1,j)$ is optimal for the case where J_j completes on M_1, then we compute $B(1,j)$ as follows,

$$B(1,j) = \min_{1 \leq k \leq m} \{B(k,j-1) + \mathcal{CE}[k,1]\} + \mathcal{PE}[1,j] \tag{3.4}$$

Here we see that in order to compute $B(1,j)$, we have to consider the solutions in the set $\mathcal{B}_{j-1}$ in order to be able to determine which $B \in \mathcal{B}_{j-1}$ will result in the smallest total cost when we include the amount of energy required to process J_j on M_1. Hence

it follows that given $\mathcal{B}_j$, where $j = n$ is the last job in the chain, the optimal solution that computes the smallest amount of energy required to complete all jobs in $\mathcal{J}$ can be expressed as follows,

$$\mathcal{OPT}(j) = \min_{1 \leq k \leq m} \{B(k, j) \in \mathcal{B}_j\}$$

This suggests that in order to compute an optimal solution for a sub-problem including J_j, the optimal solution to the sub-problem consisting of jobs $J_1, \ldots, J_{j-1}$ must first be computed.

Theorem 3.1. *Algorithm* `DPS` *correctly computes* $\mathcal{OPT}(n)$ *for* $\mathcal{J} = \{J_1, \ldots, J_n\}$ *in time* $O(nm^2 d_{max})$*, where* d_{max} *is the deadline of the last job in the chain.*

Proof. Suppose that $j = 1$, then the value of $\mathcal{S}[i, 1, t]$ for increasing values of t is simply given by $\mathcal{PE}[i, 1]$ which is the amount of energy required to process J_1 on a given machine M_i, where $t \leq d_1$. Otherwise ∞ is assigned to indicate that there is no feasible solution for J_1 machine M_i at time t. The implication is that if a feasible solution exists on any machine then $\mathcal{OPT}(1)$ is given by whichever machines requires the least amount of energy to process J_1. Now consider some job J_j, where $j > 1$, and assume that by induction $\texttt{DPS}(j-1)$ correctly computes $\mathcal{OPT}(j-1)$. By the induction hypothesis, we know that $\texttt{DPS}(j) = \mathcal{OPT}(j)$ and that $\texttt{DPS}(j-1) = \mathcal{OPT}(j-1)$. Hence, from Equation (3.4), we have,

$$\begin{aligned} \mathcal{OPT}(j) &= \min_{1 \leq k \leq m} \{B(k, j) \in \texttt{DPS}(j)\} \\ &= \texttt{DPS}(j) \end{aligned}$$

So for each job, on each machine, in order to compute the value $\mathcal{S}[i, j, t]$, we must compare the costs required for migrating from some other machine k where $k \neq i$. As a result, this will require a running time of $O(nm^2 d_{max})$, hence, the theorem follows. □

3.4 Conclusion and future work

In this chapter, we have presented a study of the problem of scheduling jobs with chain precedence constraints on unrelated, parallel machines with the objective of minimizing energy consumed by the schedule. We assume that a communication network connects all machines together and that communication between each pair of machines is asymmetric. The job set is given as a single chain with each job characterized by a strict deadline that must be met. The time and energy required to execute a job on a particular machine is

given in lookup tables. Likewise, the time and energy required to communicate over the network links are also given in lookup tables.

The problem presented in this chapter is meant to serve as a case study as well which we will be discussing in Chapter 6. It serves as a type of dynamic programming algorithm with the characteristic of a structure that allows us to develop a data-parallel algorithm for graphics processors.

A future direction to extend the problem will be to consider other possible types of precedence constraints such as a job set consisting of multiple chains or tree-like precedence constraints. This will closely model task graphs of parallel applications in practice.

Chapter 4

Energy-Efficient Flow Time Scheduling

4.1 Introduction

In this chapter, we present an empirical study of the problem of job scheduling to minimize flow time plus energy in both single-processor and multi-processor settings. We implement and investigate several heuristics used for various aspects of the scheduling process such as processor speed selection, job selection and job allocation heuristics for the case of multi-processors. The motivation behind our study comes from the design constraints associated with ubiquitous computing, especially portable systems such as notebooks and mobile phones or tablets, where battery life is directly related to the energy efficiency of the underlying system. Furthermore, it is expected that these systems do not compromise on performance and quality of service while operating within acceptable levels of energy consumption. Consequently, we incorporate flow time as a measure of quality of service in addition to the objective of minimizing energy consumption. The additional objective of maintaining desired levels of performance is orthogonal to the objective of saving energy.

Modern computer chips are capable of delivering huge amounts of processing power per square inch brought about by technological advances in chip design and fabrication processes. As a result, the power envelope for these parts is pushed up thereby raising the energy demands for computer systems. A direct implication is that, apart from the cost of running these systems, one must factor in the cost of cooling from small scale, embedded in the systems to large scale as the case of data centres and server stores. Due to these reasons, energy conservation has become a critical design feature in modern computer systems and several technologies and techniques have been developed

to better utilize and conserve energy as much as possible. For instance, features such as AMD PowerNow!™ and Intel SpeedStep® allow the operating system to dynamically adjust core frequency and voltage thereby altering the speed of the processor in order to conserve power [1, 45]. This concept of adjusting the speed of the processor to meet computing demands is often known as *throttling* or *dynamic speed scaling*.

The closest work to the study presented in this chapter is that presented in [7], where a number of job selection policies and speed-scaling algorithms were analyzed empirically. The goal of their empirical studies is to analyze these speed-scaling algorithms coupled with the job selection policies in order to demonstrate that their real-world performance can be improved by using knowledge of the input job data. Furthermore, they were also able to demonstrate that different algorithms work better on certain types of input data and as a result, the input data should be taken into account when choosing a speed-scaling algorithm. One of the speed-scaling algorithm they studied was AJC (Active Job Count) [4] and SRPT (Smallest Remaining Processing Time) was also one of the job selection policies. Our empirical studies also includes these two algorithms but for diferent reasons.

In this chapter, we compare AJC with several fixed speed heuristics, including a *semi-clairvoyant* fixed speed function that we designed. We describe this heuristic as semi-clairvoyant since it requires an approximate knowledge of the characteristics of a given instance of jobs. In contrast, a *clairvoyant* algorithm would require exact knowledge, for example, the exact arrival time of each job, while a *non-clairvoyant* algorithm has no knowledge of the jobs. The purpose of this comparison with the semi-clairvoyant fixed speed function is to attempt to demonstrate that it is possible to design a simple fixed speed function that performs close to AJC in objective of minimizing total flow time plus energy. The investigation of a simpler alternative is due to the fact that AJC can be quite computationally intensive to implement in practice and given the arbitrary nature of the speed spectrum, it could be a challenge to support such a capability in hardware. Our studies also uses AJC in multi-processor scheduling that attempts to demonstrate that having more processors can be very cost-effective in minimizing total flow time plus energy.

The highlights of our results in this chapter, within the context of minimizing total flow time plus energy, include,

- Empirically, we are able to confirm the theoretical result that SRPT is a better job selection policy in comparison to SJF.
- As a speed-scaling algorithm, AJC is very effective and performs better than a fixed-speed heuristic.

- Given some prior knowledge about a job sequence, it is possible to design a much simpler fixed-speed heuristic that can perform close to AJC. In other words, without some insight about the job sequence, a fixed-speed heuristic cannot perform better than a speed-scaling algorithm.
- We demonstrate that with multiple processors and speed-scaling, we can achieve significantly better performance over a single processor. Furthermore, it is also interesting to note that we observed a trend where the performance benefit from multiple processors can only be noticed beyond a certain number of processors, regardless of the nature of the job sequence.

The rest of this chapter is organized as follows; in Section 4.2, we present a formal definition of the problem then we present the heuristics we designed, implemented and evaluated, in Section 4.3. Finally, in Section 4.4, we describe the setup for the simulations and present our observations and results.

4.2 Problem Definition

Consider a set of job instances $\{I_{\langle a,p\rangle_1}, \ldots, I_{\langle a,p\rangle_n}\}$ where each job instance, $I_{\langle a,p\rangle_i}$, is characterized by the pair of parameters, a and p, which denote the average *inter-arrival time* and average *job size* respectively. The average inter-arrival time parameter of a job instance is the average time interval between the arrival times of successive jobs in the set while the size of the job refers to the amount of work to be done or processor cycles required in order to process the job. Both parameters are defined as positive integer values that serve as input to a discrete probability distribution function which in turn generates the arrival times and size of jobs in a job set and we always assume that $p > 0$.

We assume the infinite speed model where a processor is capable of changing its speed to any value between 1 and ∞. The energy E consumed by a processor running at speed s is given as s^α per unit time, where $\alpha \geq 2$ [16, 76], and the amount of work completed by the processor is s units of work. The flow time F_i for a job J_i is the amount of time that elapsed between the arrival time of the job and its completion time. Therefore, the total flow time for a given job set can be expressed as $F = \sum_i F_i$. The goal is to produce a schedule that shows, for each job, the time intervals a particular job is being processed and at what speed. In the case of multiple processors, the particular processor the job is being executed. Pre-emption is allowed for a job executing on a particular processor which implies that a job can be suspended and resumed at a later time on the same processor, hence, job migration between processors is not allowed.

In the experiments, we consider job arrivals with Poisson arrival patterns where the inter-arrival time follow an exponential distribution. For the job sizes we consider uniform distribution only.

The schedule aims to minimize the cost G which is the total flow time plus energy incurred for a given job set expressed as $G = F + E$. In the case of multiple processors it is simply a summation of the cost incurred by each processor.

4.3 Heuristics

In this section we discuss the heuristics we implemented and evaluated in our simulations and as mentioned earlier in this chapter, they are grouped into job selection, speed selection and processor allocation heuristics. Note that the names for the heuristics we designed and implemented are prefixed with an asterik (*).

4.3.1 Job selection strategies

The purpose of a job selection heuristic is to prioritize job execution order by determining which job is selected for execution based on some rules. We consider SRPT and SJF which are very common job selection heuristics.

Shortest Remaining Processing Time (SRPT). The selection criteria for this heuristic is based on the amount of work left to be processed for each job in the job queue. A job is selected for execution if it has the least amount of work yet to be processed. In the case where two or more jobs have the same priority, in terms of remaining amount of work, ties can be broken arbitrarily.

Shortest Job First (SJF). This job selection strategy gives execution priority to jobs in the queue based on the total amount of work that characterizes each of them. Priority is given to the job with the smallest amount of work and in case two or more jobs have the same priority, ties can be broken arbitrarily.

4.3.2 Speed functions

Speed Scaling based on Active Job Count ($\mathbf{S_{AJC}}$). A job is said to be active if it has arrived and added to the processing queue but yet to be completed. The AJC heuristic relies on the number of active jobs in order to determine the speed of the processor. The speed determined by S_{AJC} is computed as $n^{\frac{1}{\alpha}}$, where n is the number of active jobs and

$\alpha = 3$. Note that this heuristic does not have any prior knowledge of the job set with regards to average inter-arrival time or average job size parameters.

Fixed speed function (*$\mathbf{S_F}$). This refers to a heuristic that is oblivious to the input job set and it will always use a fixed speed value of 1.

Semi-clairvoyant fixed speed function (*$\mathbf{S_D}$). This fixed speed heuristic is based on the distribution parameters, a and p, of the job set and therefore has some prior knowledge about the job set in order to determine a speed. The speed is determined as $(\beta \cdot \frac{a}{p})^{\frac{1}{\alpha}}$, where $\beta = 5$ is a constant determined experimentally.

Fixed Speed based on Mean AJC Speed (*$\mathbf{AJC_{AVG}}$). We present a fixed speed heuristic that is based on the average speed obtained from the different speed values obtained by executing $\mathrm{S_{AJC}}$ on the same job set. In order to compute the average speed, we take into account the length of the interval(s) a particular speed value was active, in other words, we determine a weighted average of the speed values obtained from $\mathrm{S_{AJC}}$. For each speed value, $s(t) > 0$, active for the duration of time denoted by the interval I, the average speed s is given by the following expression;

$$s = \frac{\int_0^I s(t)\mathrm{d}t}{I}$$

Fixed Speed based on Maximum AJC Speed (*$\mathbf{AJC_{MAX}}$). Similar to *$\mathrm{AJC_{AVG}}$, this fixed speed heuristic depends on a prior execution of $\mathrm{S_{AJC}}$ on the job set. The speed is determined by selecting the maximum speed recorded by the execution of $\mathrm{S_{AJC}}$.

4.3.3 Processor allocation strategies

In our simulations, we implement and evaluate four processor allocation heuristics and they are described as follows.

RoundRobin. This processor allocation heuristic attempts to distribute jobs as evenly as possible across all available processors. Therefore, priority is given to the processor with the least number of jobs allocated to it already.

***MinCost**. In this heuristic for processor allocation, a job that is ready to be allocated is dispatched to the processor that yields the least total cost if that particular job was assigned to it.

***MinSize**. Each processor keeps track of the jobs allocated to it including the total size of all the jobs. This heuristic gives priority to the processor with the least total job size.

***MinActiveCount**. This heuristic utilizes the number of active jobs after a particular job is allocated to a processor. Priority is then given to the processor with the smallest number of active jobs.

4.4 Simulations Conducted and Results

4.4.1 Preliminaries

4.4.1.1 Overview of the Simulator Software Program

The simulation software program consists of a scheduler as well as a data generator that can be used to generate the job instances. Figure 4.1 shows the different parts of the simulator software.

JobGenerator and Job classes. The `JobGenerator` class is responsible for the generation of the job sequences used as input data for the simulations. The method to be used for generating either the job sizes or the job arrival times can be specified as input. The `PoissonProcess()` method of the `JobGenerator` class implements the Knuth's algorithm for generating Poisson random events [25]. In our experiments, the size of each job in a given job sequence is chosen uniformly at random within $[1, p]$, and the size of each job is an integer value. On the other hand, the arrival time of each job is obtained through a Poisson process with rate p, which is the average inter-arrival time parameter.

ProcessorObject and Scheduler classes. In the simulator, a `ProcessorObject` represents a processor. Each processor manages its own job queues, one each for jobs being processed and jobs waiting to be processed. A processor is also aware of which speed function to use and this is set as a property of the `ProcessorObject`. During execution, a processor keeps track of statistics including total energy consumed, total flow time, maximum speed, average speed and number of jobs. These are reported to the `Scheduler` at the end of the simulation. The `Scheduler` manages the creation of processors and the scheduling tasks including allocating jobs to processors, and setting any processor property that are not meant to change during the simulation. The scheduler is designed to run in single or multi-processor mode and, in addition, is easily configured to choose from a combination of job selection, speed selection and processor allocation heuristics The `SchedulerApp` is the main controller of the simulation program that coordinates the execution process, from input parameters to execution to processing results and writing to file.

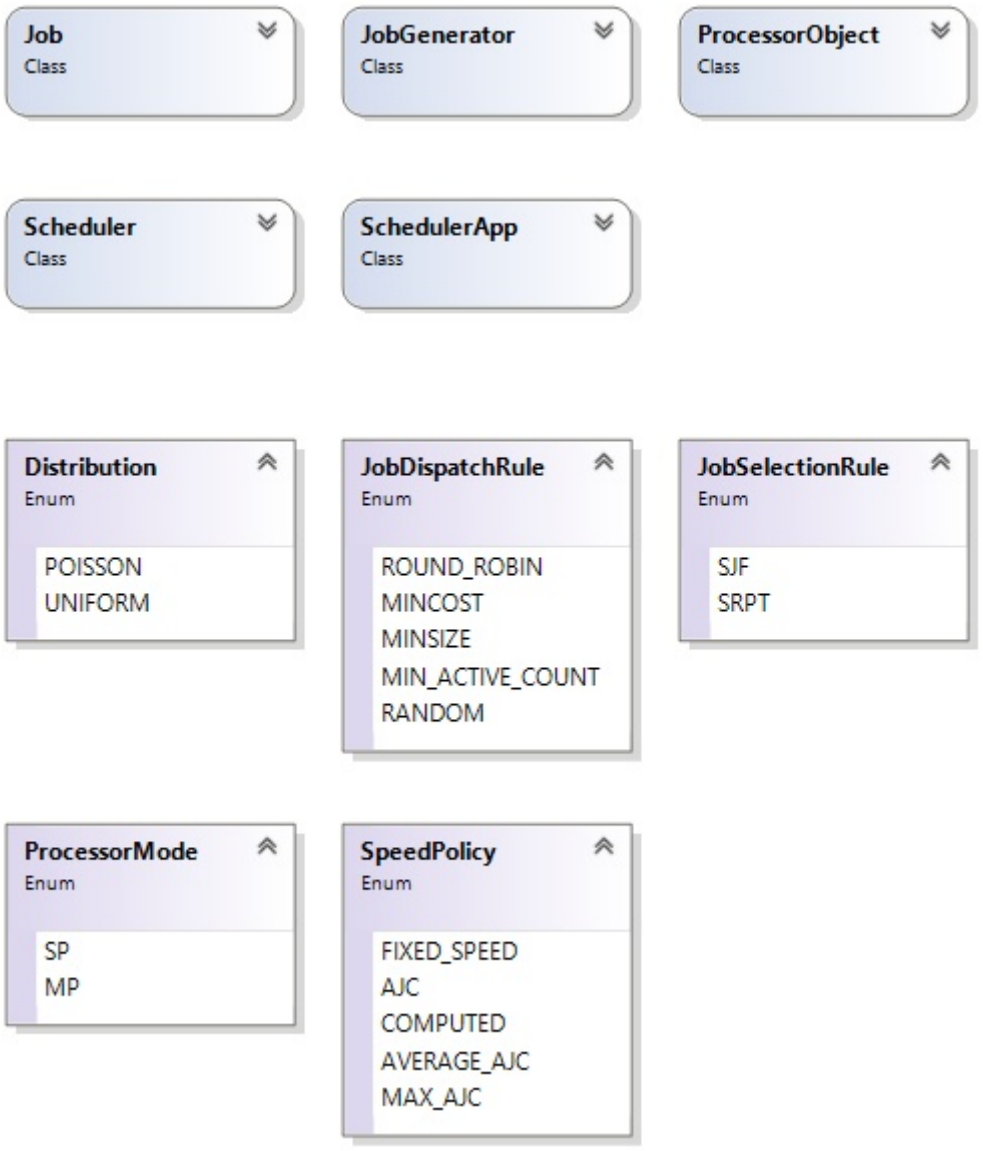


FIGURE 4.1: Class diagram of the simulator software program.

4.4.1.2 Simulation methodology

The scheduler and all the heuristics described in Section 4.3 are implemented using the C++ programming language. The simulation program was run on a machine with an AMD FX-8350 CPU with a clock frequency of 4.0 GHz and 16 GB of system memory.

The average inter-arrival time and average job size for each job set instance is given by the set of values, $\mathcal{S} = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$. Each job set contains a total of 2000 jobs and for each pair of parameters $\langle a, p \rangle$, where $a \in \{2^0, 2^1 \ldots, 2^9\}$ and for each value of a, $p \in \{2^0, 2^1, \ldots, 2^9\}$, we generate 10 instances and the cost for a particular $\langle a, p \rangle$ configuration is obtained by taking the average over the 10 instances. The job instances are generated once and re-used for all the simulations presented in

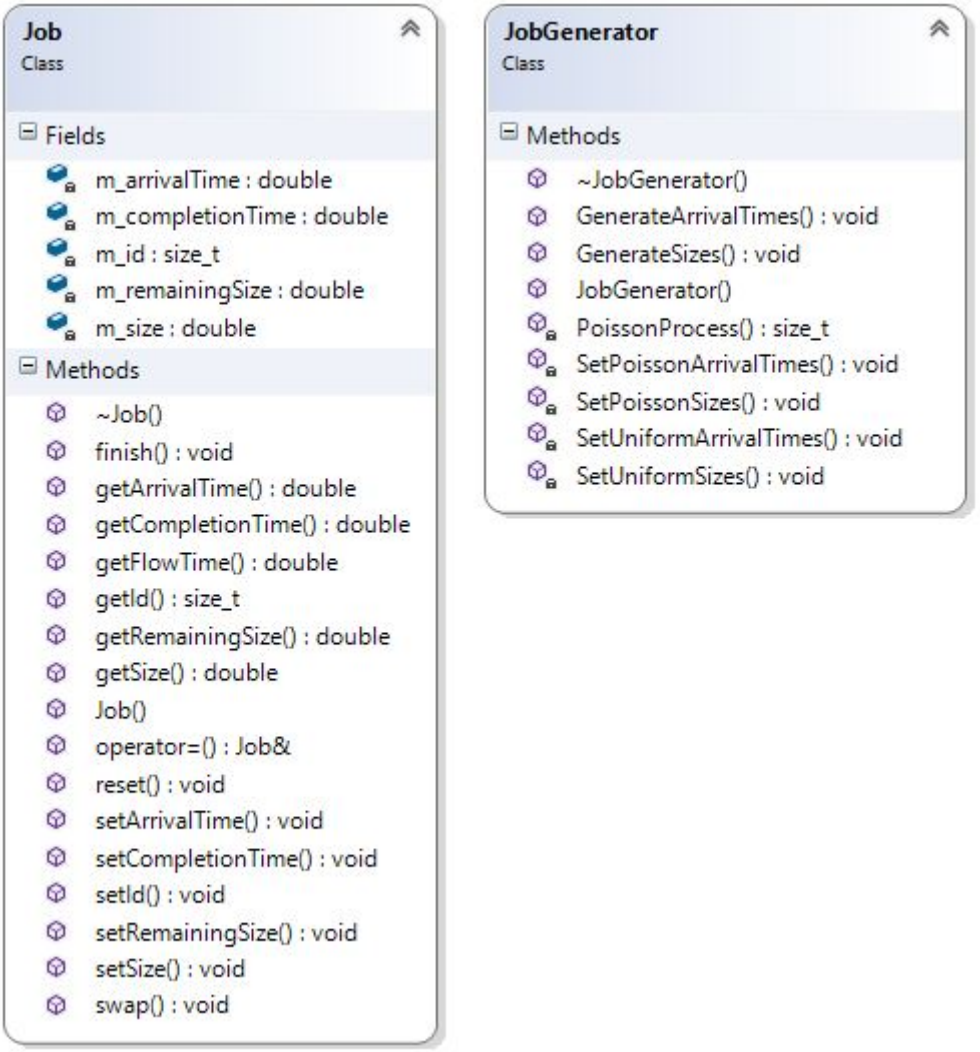


FIGURE 4.2: Details of the Job and JobGenerator classes.

this chapter. For the case of multiple processors, the number of processors used in the simulations are 2, 4, 8 and 16 processors. For all simulations the value of α used in determining the energy consumption of the schedule is set to 3 [16, 76].

4.4.2 Results on job selection strategies

In this simulation we compare the performance of SRPT and SJF as the job selection strategies in a single processor setting as well as with multi-processor setup. Although we already know that SRPT is an optimal job selection strategy we expect the performance of SJF to be close in terms of minimizing the cost incurred by their respective schedules. We present results for the performance of SJF and SRPT on both single and multi-processor setting.

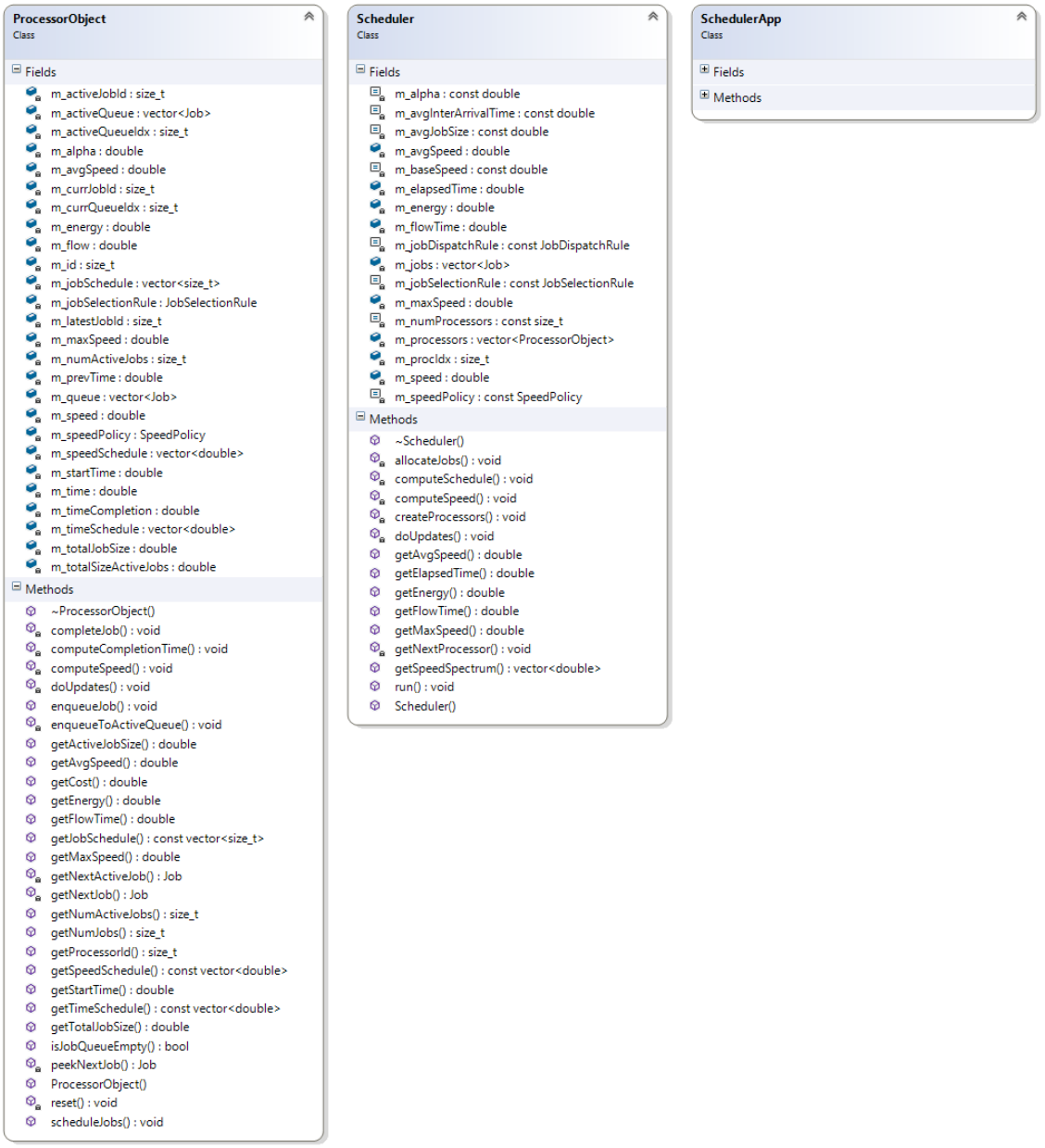
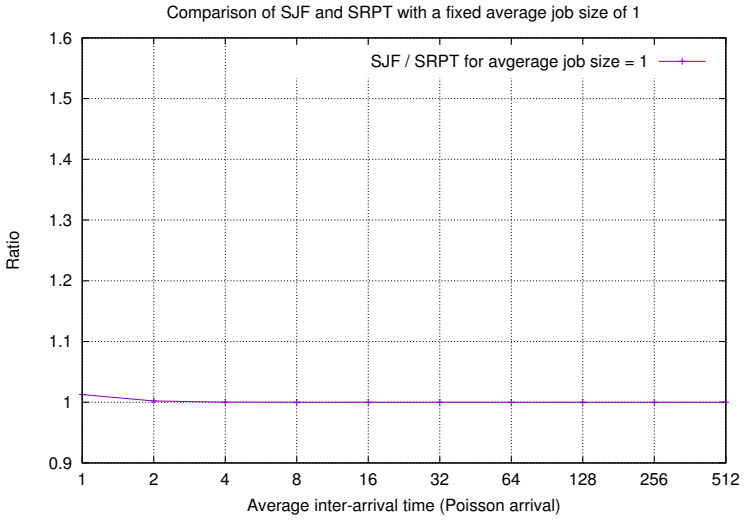


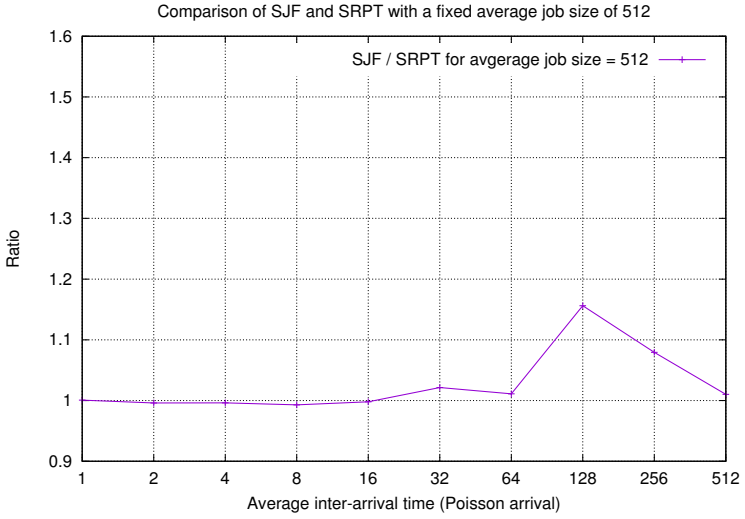
FIGURE 4.3: Details of the scheduler part of the simulator.

Single processor simulation. Figures 4.4 and 4.5 show the performance ratio of SJF to SRPT on a single processor for various samples categorized based on average job size and average inter-arrival time of jobs in the job set, respectively. We observe that when the job distribution is either very dense or very sparse, for instance in Figures 4.5(a) and 4.5(c) respectively, the performance of SJF is very similar to SRPT. However, we notice that when the average inter-arrival time is similar to the average job size, SRPT performs better than SJF in all the cases with a performance ratio of up to 1.45 as seen in Figure 4.4(c). In general, we can confirm that although the performance of SJF and SRPT are quite similar, SRPT is the better job selection strategy for minimizing total flow time plus energy on a single processor.



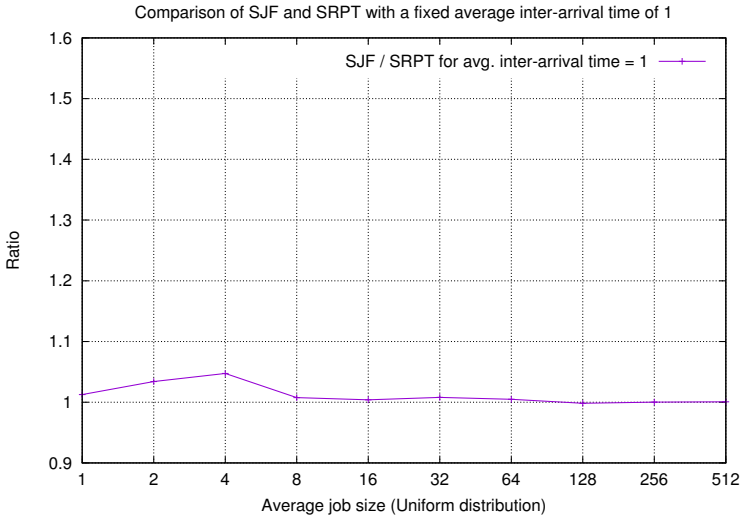
(a) Performance ratio for average job size of 1 with varying inter-arrival time.

(b) Performance ratio for average job size of 16 with varying inter-arrival time.

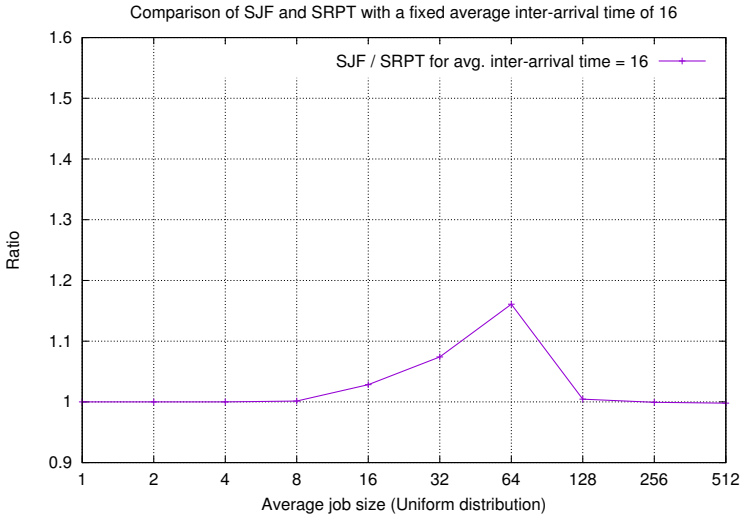


(c) Performance ratio for average job size of 512 with varying inter-arrival time.

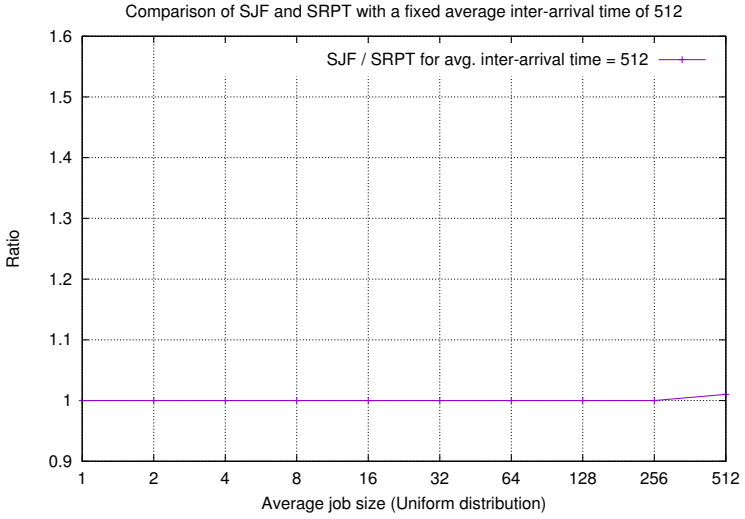
FIGURE 4.4: Measurement shows the ratio of total flow time plus energy for SJF vs AJC on a single processor. Results are grouped according to average job size.



(a) Performance ratio for average inter-arrival time of 1 with varying job size.



(b) Performance ratio for average inter-arrival time of 16 with varying job size.



(c) Performance ratio for average inter-arrival time of 512 with varying job size.

FIGURE 4.5: Measurement shows the ratio of total flow time plus energy for SJF vs AJC on a single processor. Results are grouped according to average inter-arrival time.

Multi-processor simulation. We also evaluate the performance of SJF and SRPT as job selection strategies for multiple processors. Figure 4.6, categorized according to the average job size, shows the performance ratios for the case of four processors using all four processor allocation strategies. Figure 4.7 shows the results from the perspective of average inter-arrival time. We observe that when the job sizes are very small and when job distribution is very sparse, as shown in Figures 4.6(a) and 4.7(c), again the performance of SJF and SRPT are near identical. In other cases, however, especially when the average inter-arrival time and average job size parameters are close, we notice a clear performance difference where SRPT can be up to 1.2 times better than SJF. This can be observed in Figure 4.6(c).

Following the observations and results outlined above, we can conclude that SRPT is better than SJF as a job selection strategy for both single and multiple processors. Hence, the rest of the results published in this chapter will make use of SRPT as the job selection policy.

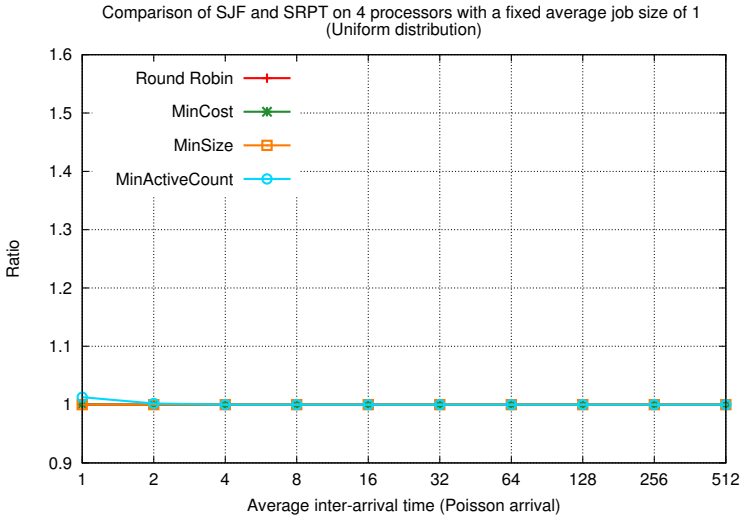
4.4.3 Results on speed functions

4.4.3.1 Effectiveness of speed scaling

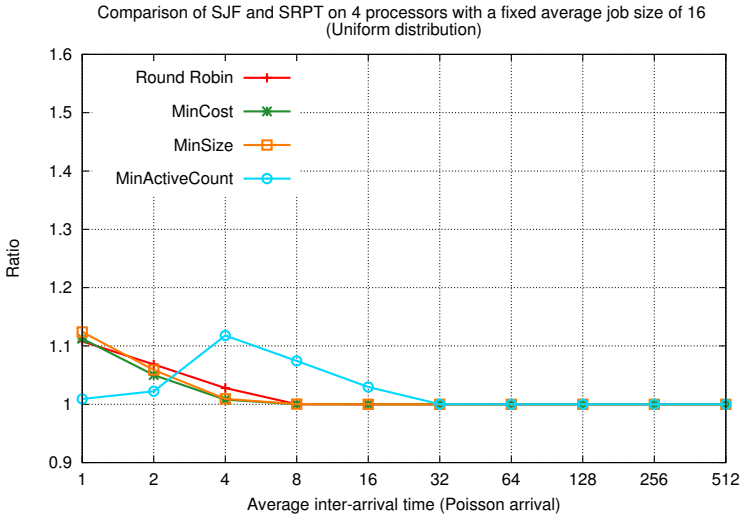
In this simulation we would like to evaluate the performance benefits of speed scaling over using a single, fixed speed of 1. We know that a processor running at a low speed will conserve more energy, however, it will incur a higher cost in terms of the time spent executing the jobs or total flow time. Therefore it will be interesting to investigate how much the variation between the energy saved and flow time affects the total cost when running at a fixed speed. We evaluate the AJC heuristic and a fixed speed heuristic that uses a fixed speed of 1, denoted by $^*S_F$, on a single processor. Results are presented as the performance ratio of S_{AJC} to $^*S_F$.

Figures 4.8 and 4.9 show the performance ratio of $^*S_F$ to S_{AJC} for various job input samples. Firstly, we observe that in Figure 4.8(a) where amount of work in the job set instances is very small and in Figure 4.9(c) where job distribution is very sparse, both speed functions are almost identical in performance and this should not come as a surprise. The reason is because in such cases, the number of active jobs is mostly 1 throughout the duration of a scheduling process, hence, the speed selected by AJC will be 1 or very close to 1 which is the same speed as the fixed speed heuristic running at a speed of 1.

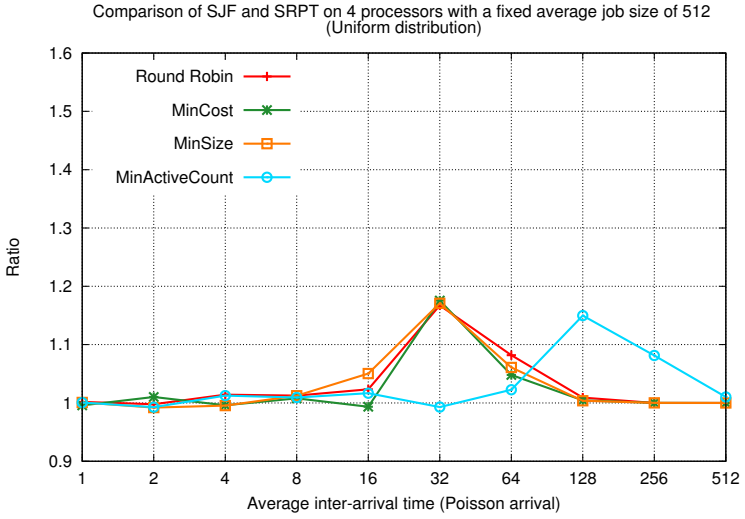
On the other hand, in cases where there is a considerable amount of work to be done either in terms of job arrival density or size of the jobs, we notice significant differences



(a) Performance ratio for average job size of 1 with varying inter-arrival time.

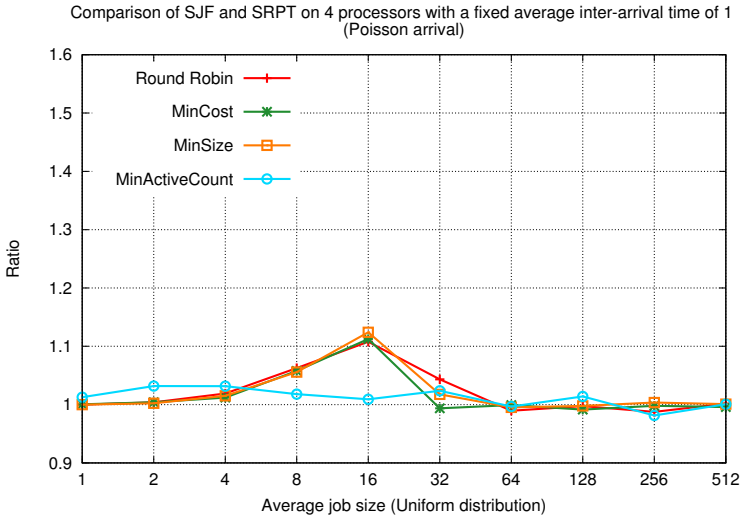


(b) Performance ratio for average job size of 16 with varying inter-arrival time.

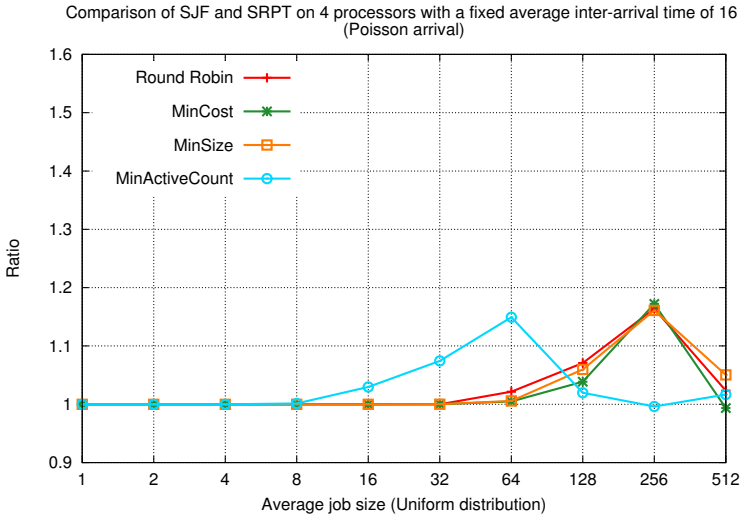


(c) Performance ratio for average job size of 512 with varying inter-arrival time.

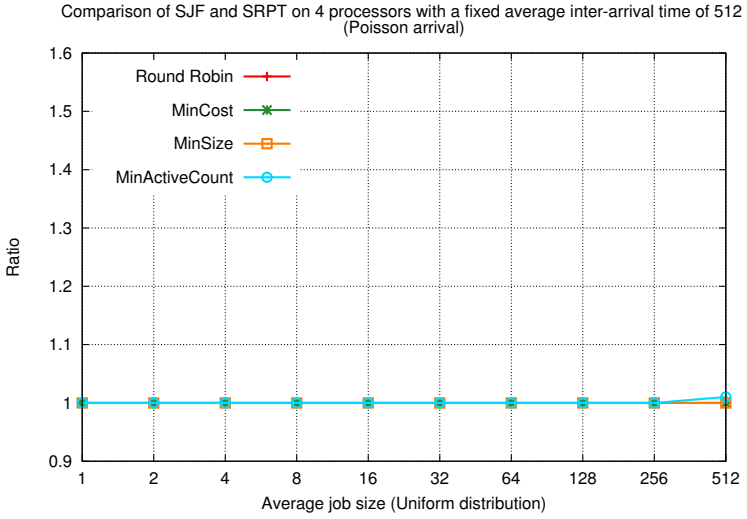
FIGURE 4.6: Measurement shows the ratio of total flow time plus energy for SJF vs AJC on 4 processors. Results are grouped according to average job size.



(a) Performance ratio for average inter-arrival time of 1 with varying job size.

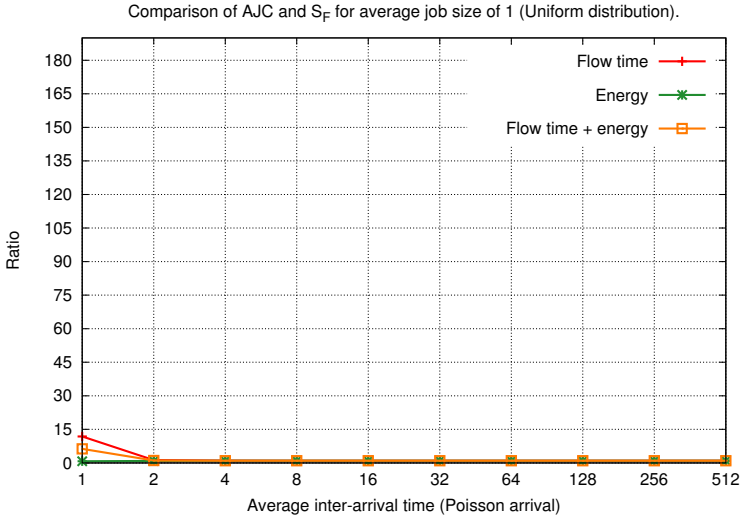


(b) Performance ratio for average inter-arrival time of 16 with varying job size.

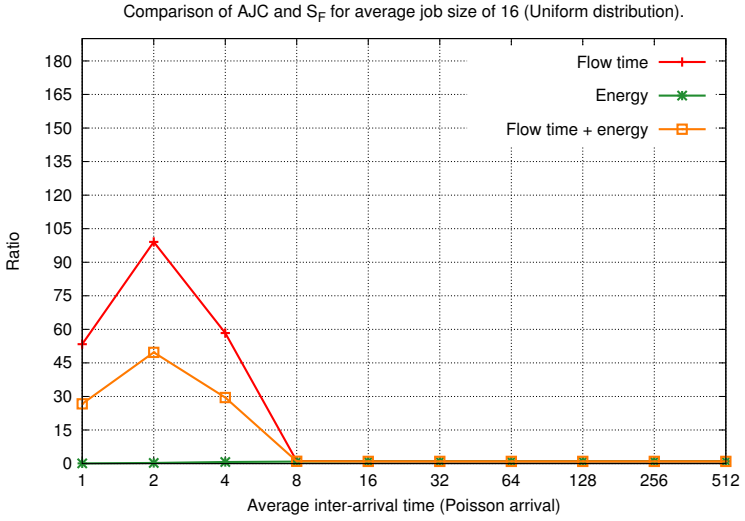


(c) Performance ratio for average inter-arrival time of 512 with varying job size.

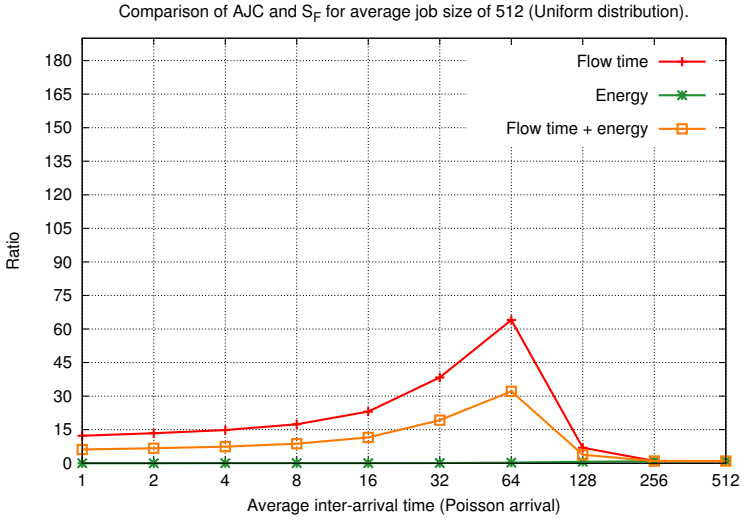
FIGURE 4.7: Measurement shows the ratio of total flow time plus energy for SJF vs AJC on 4 processors. Results are grouped according to average inter-arrival time.



(a) Performance ratio for average job size of 1 with varying inter-arrival time.

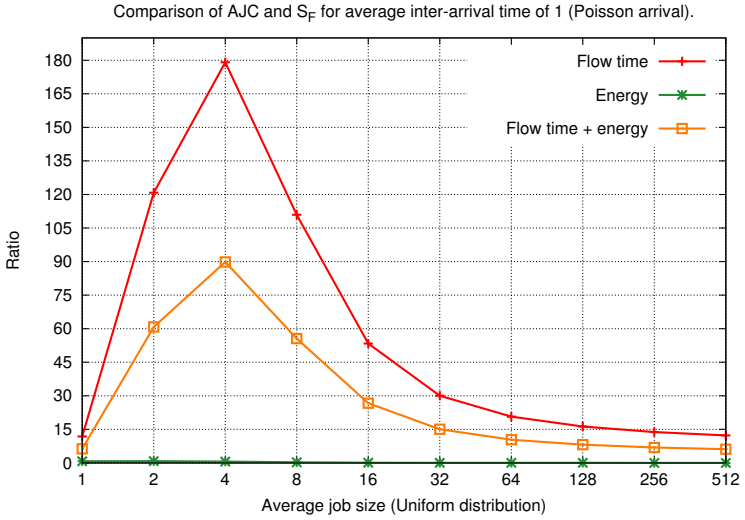


(b) Performance ratio for average job size of 16 with varying inter-arrival time.

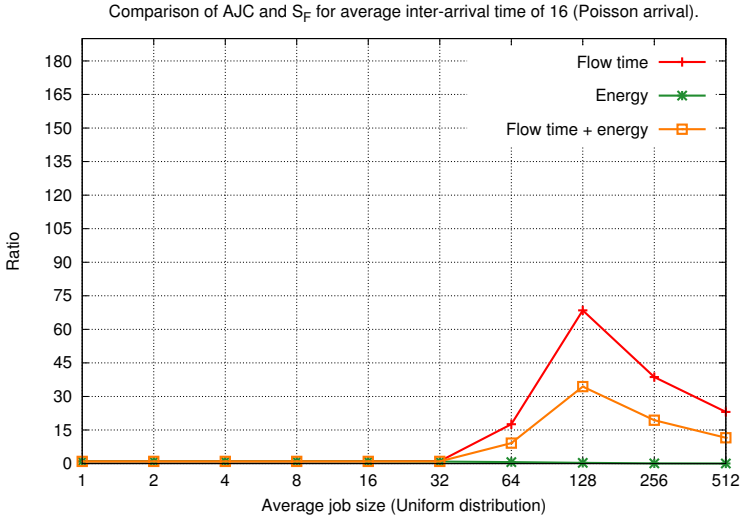


(c) Performance ratio for average job size of 512 with varying inter-arrival time.

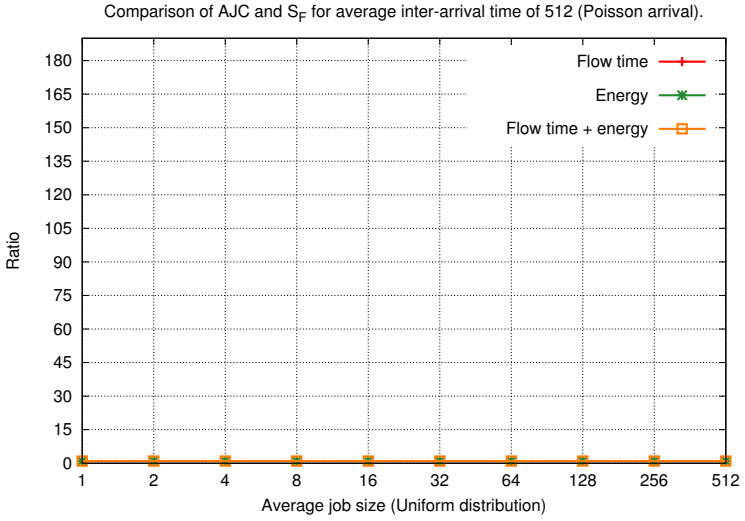
FIGURE 4.8: *Effectiveness of speed scaling*: Measurement shows the ratio of total flow time plus energy for a fixed speed heuristic using a speed of 1 against AJC on a single processor. Results are grouped according to average job size. *Note*: ratio is always at least 1.



(a) Performance ratio for average inter-arrival time of 1 with varying job size.



(b) Performance ratio for average inter-arrival time of 16 with varying job size.



(c) Performance ratio for average inter-arrival time of 512 with varying job size.

FIGURE 4.9: *Effectiveness of speed scaling*: Measurement shows the ratio of total flow time plus energy for a fixed speed heuristic using a speed of 1 against AJC on a single processor. Results are grouped according to average inter-arrival time. *Note*: ratio is always at least 1.

in performance. The fixed speed heuristic performs significantly worse in terms of flow time but performs much better with regards to saving energy. This is due to the fact that running a processor at a low speed will help save a lot of energy but this will result in longer job processing times. In fact, when we consider the performance ratio in terms of the overall cost of total flow time plus energy, we observe a huge gulf in performance with S_{AJC} performing up to 92.7 times better than $*S_F$ (Figure 4.9(a)). This confirms that speed scaling is important in providing a balance between saving energy and not compromising on the quality of service.

Another important observation is that, despite the fact that both heuristics make use of SRPT, the order of the jobs in the final schedule will not necessarily be identical. This is because $*S_F$ will spend more time executing a particular job as a result of the its low speed thereby causing more jobs to accumulate in the queue. As a result, there is a higher chance for another with a higher priority to arrive in the queue. This is particularly common occurrence in cases where job arrival is dense and job sizes are reasonably large.

4.4.3.2 Speed scaling vs. semi-clairvoyant fixed speed function

In the previous simulation discussed in Section 4.4.3.1, we demonstrated that a speed function that has no prior knowledge about the job set and running at a fixed speed does not perform well when compared to a speed scaling heuristic like AJC. In this simulation we compare the performance of AJC and that of a heuristic which has some prior knowledge about the job set, precisely, only the $\langle a, p \rangle$ parameters are used by the heuristic to determine a fixed speed. The aim of this simulation is to evaluate how well a fixed speed heuristic with some knowledge about the job set can perform when compared to a speed scaling heuristic like AJC.

The first observation from the results shown in Figures 4.10 and 4.11 is that when we allow the fixed speed heuristic to have some prior knowledge about the job set, it can perform considerably better especially for cases with dense job arrival times.

On the other hand, given the simplicity of $*S_D$ and limited knowledge about the job sets, we can also observe that the resulting speed can be too low in some cases, like in Figures 4.10(a) and 4.11(c), or too high as seen in Figure 4.10(c). As expected, when the speed is too high jobs are completed faster at the expense of incurring high energy costs. However, when we consider the performance ratio of $*S_D$ to S_{AJC} in general, we can see that $*S_D$ achieves a performance close to that of S_{AJC} in contrast to $*S_F$.

In conclusion, as seen from the performance of $*S_D$, we are able to show that a speed function that has some information about the job set performs better than one without

any information. However, when we examine the performance of S_{AJC}, we can assert that some form of speed scaling is required in order to achieve even better performance in terms of minimizing flow time plus energy.

4.4.3.3 Effectiveness of AJC speed spectrum

In the preceding sections, we have demonstrated the effectiveness of speed scaling in minimizing flow time and energy incurred by a schedule. The AJC speed scaling heuristic is characterized by a spectrum of speed values, of which the range of speeds achieved greatly depends on the distribution of jobs and their sizes. In this simulation, our aim is to investigate how the variations in an AJC speed spectrum compare to AJC itself if a fixed speed value was selected from this speed spectrum. To be precise, we compare the performance of AJC to a fixed speed heuristic. However, unlike the previous fixed speed heuristics discussed earlier in the chapter, this fixed speed heuristic depends on a prior execution of AJC in order to obtain two fixed speed values in the form of its average speed, $*AJC_{AVG}$, and maximum speed, $*AJC_{MAX}$.

The results shown in Figures 4.12 and 4.13 are presented in the form of a ratio of the fixed speed heuristic to AJC in terms of their respective total costs. The first observation that we easily make is, given the fact that there is no form of speed scaling involved, the fixed speed heuristic still performs reasonably well when compared to previous results involving fixed speed heuristics. It seems to indicate that at each point in time, AJC is able to determine a speed that is very good for the situation. This gives further credence to the claim that a heuristic based on speed scaling will yield better performance over a fixed speed heuristic in the problem of minimizing flow time plus energy.

On the other hand, there is not much difference in performance between using the average speed and maximum speed from AJC. When we focus on $*AJC_{MAX}$ in particular, the performance is not far off from that of S_{AJC}. Since we already know that running at high speeds will help save time at the expense of higher energy costs, the $*AJC_{MAX}$ is still low enough to achieve reasonable performance across the variations and distributions in the job inputs.

4.4.4 Results on processor allocation strategies

In this section we present the results from the simulations involving processor allocation strategies in a multi-processor environment. We aim to evaluate how multi-processors can contribute to minimizing flow time plus energy while utilizing several processor allocation strategies as listed in Section 4.3.3. For these simulations, we make use of AJC as the

speed function and SRPT as the job selection strategy. We simulate systems with 2, 4, 8 and 16 processors independently and for each simulation, the cost is presented as the summation of costs across all processors.

4.4.4.1 RoundRobin

The first of the multi-processor allocation strategy to be considered is RoundRobin. Recall that this processor allocation strategy aims to evenly distribute jobs across all available processors. Figures 4.14 and 4.15 shows the results of the simulation according to average job size and average inter-arrival time, respectively.

We can observe that when job density is relatively low, up to 4 units of work per unit time, there is no difference in performance between single processor and multiple processors. However, above this threshold, we begin to observe the benefits of adding multiple processors over a single processor. This benefit ranges from 3 times, as seen in Figure 4.15(b), up to around 46 times with 16 processors, as shown in Figure 4.15(a). This performance boost increases as the job density increases.

4.4.4.2 *MinActiveCount

This processor allocation strategy is similar to RoundRobin, however, priority is based on the number of active jobs. Figure 4.16 shows the results with respect to average job size while Figure 4.17 is with respect to average inter-arrival time.

The performance of this processor allocation strategy is quite similar to RoundRobin. Again, the performance benefit from using multiple processors over a single processor can only be observed when the ratio of average job size to average inter-arrival time is larger than 4. In this case, the performance gain is up to 22 times with 16 processors as shown in Figures 4.17(a) and 4.16(c). This is considerably less than what we observe with RoundRobin.

We can observe that when job density is relatively low, up to 4 units of work per unit time, there is no difference in performance between single processor and multiple processors. However, above this threshold, we begin to observe the benefits of adding multiple processors over a single processor. This benefit ranges from 3 times, as seen in Figure 4.17(b), up to around 21 times, as shown in Figure 4.17(a). This performance boost increases as the job density increases.

4.4.4.3 *MinCost

Among the four processor allocation strategies, *MinCost can be considered the most computationally intensive. This is because priority is given to the processor that will yield the least amount of cost, meaning, that the job queue for each processor needs to be processed before the final allocation is made. Nevertheless, the results for *MinCost are also straight-forward and a distinct observation, similar to *MinActiveCount. Figures 4.16 and 4.17 show some results with respect to average job size and average inter-arrival time, respectively.

The same trend observed in *MinActiveCount continues here with *MinCost too. When ratio of unit work to time is 4 or less, single processor and multi-processor performance is quite the same. However, above this threshold, we observe an even larger performance boost, compared to *MinActiveCount, with using multiple processors. The performance boost is up to 42 times, as shown in Figure 4.19(a). This is around double the performance boost observed with *MinActiveCount.

4.4.4.4 *MinSize

This processor allocation strategy gives priority to the processor with the least amount of work. Figures 4.16 and 4.17 show some simulation results with respect to average job size and average inter-arrival time, respectively.

As with the results from other processor allocation strategies discussed so far, the same trend continues. That is, the benefit of using multiple processors over a single processor only begin to manifest when the ratio of average job size to average inter-arrival time is greater than 4. However, in Figures 4.20(c) and 4.21(a), *MinSize shows a performance boost of up to 49 times when using 16 processors over a single processor. This is the best performance ratio we have observed so far.

4.4.5 Conclusion

In this chapter, we presented an empirical study of the problem of minimizing flow time plus energy on single and multiple processors. We implemented and evaluated several strategies for job selection, speed selection and processor allocation.

Job selection strategies. For job selection strategies, we considered SJF and SRPT. Our simulations confirm that, although both are close in performance, SRPT is better than SJF in both single and multiple processor configurations. This result is an emperical

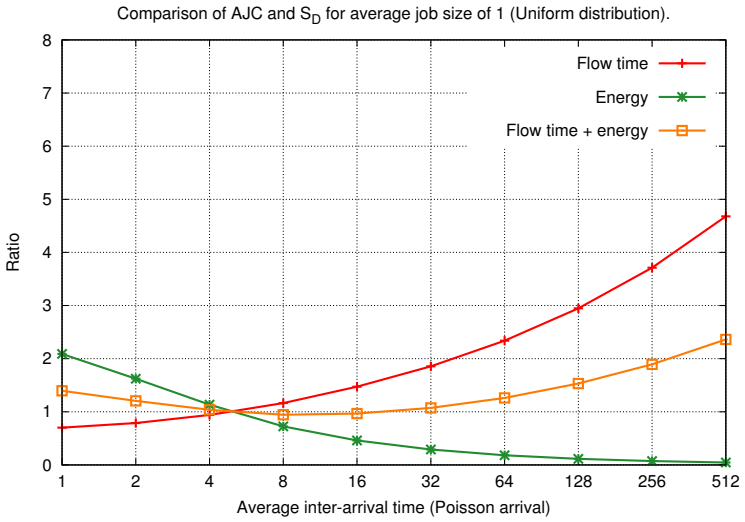
	RoundRobin	*MinActiveCount	*MinCost	*MinSize
2 processors	3.4	3.08	3.39	3.39
4 processors	9.87	7.11	10.83	10.25
8 processors	21.62	12.77	24.76	22.58
16 processors	46.05	21.53	42.11	49.87

TABLE 4.1: Summary of the best performance ratios for all processor allocation strategies.

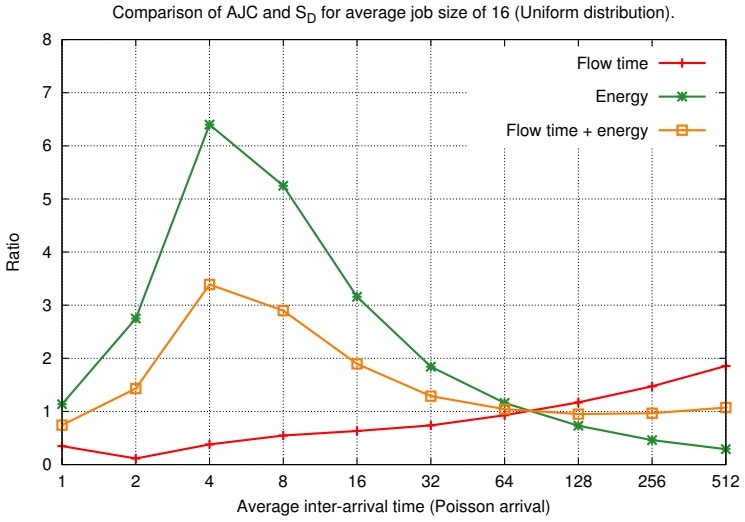
confirmation of result that is already known in literature which presents SRPT as an optimal job selection strategy [90].

Speed functions. In this round of experiments we considered a speed-scaling heuristic, AJC, and several other fixed speed heuristics including a semi-clairvoyant fixed speed heuristic, $*S_D$. Our results from the simulations conducted with AJC and $*S_D$ show that, if some knowledge of the incoming jobs is known, it is possible to design a much simpler fixed-speed heuristic that can perform quite close to AJC. In practice, AJC might be computationally intensive to implement and a processor has to be able to support a wide speed spectrum with the capability to select an arbitrary speed within this spectrum. Furthermore, our results also show that some form of speed scaling is required in order to minimize the cost of total flow time plus energy. This was clearly demonstrated by the performance of AJC in our simulations when compared to other fixed speed heuristics.

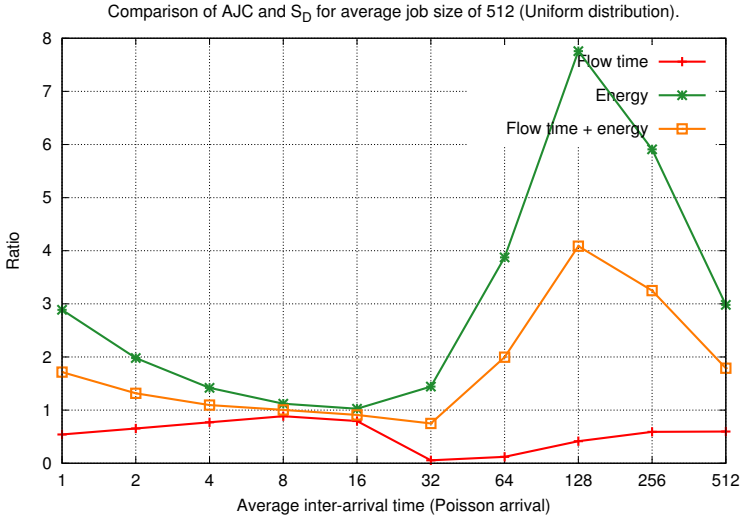
Processor allocation strategies. Our simulations on multi-processor allocation strategies included the comparison of RoundRobin, *MinActiveCount, *MinCost and *MinSize. The performance of multiple processor configuration was compared to a single processor configuration with respect to minimizing total flow time plus energy. Our results clearly show that adding more processors can help to minimize total flow time plus energy, especially when job density is quite high. An interesting observation, however, for all four processor allocation heuristics is that we only begin to observe the benfits of multiple processors when the ratio of average job size to average inter-arrival time is larger than 4. Another interesting observation is that the simplest heuristics, RoundRobin and *MinSize, performed better than the other two, more sophisticated heuristics. A summary of results is shown in Table 4.1.



(a) Performance ratio for average job size of 1 with varying inter-arrival time.

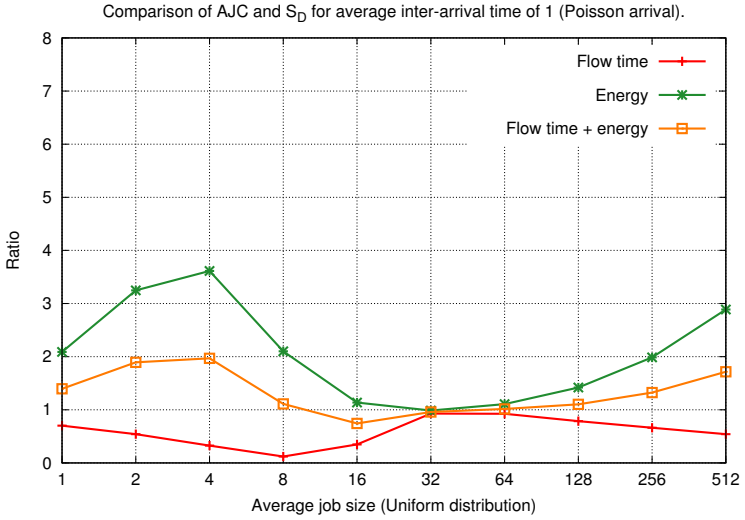


(b) Performance ratio for average job size of 16 with varying inter-arrival time.

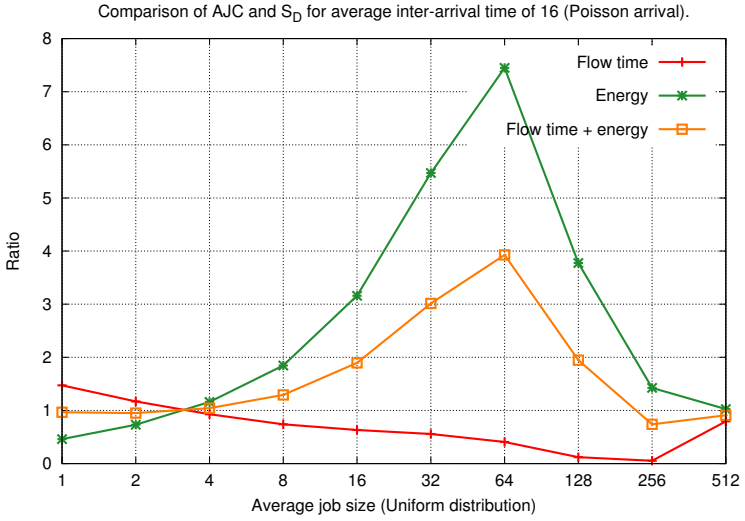


(c) Performance ratio for average job size of 512 with varying inter-arrival time.

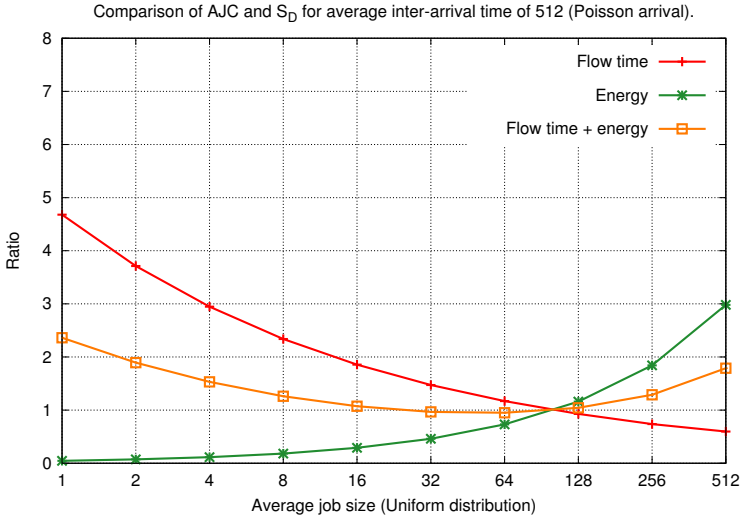
FIGURE 4.10: *Speed scaling vs. semi-clairvoyant fixed speed function*: Measurement shows the ratio of total flow time plus energy between AJC and a fixed speed function that has some information about the job set. Results are grouped according to average job size.



(a) Performance ratio for average inter-arrival time of 1 with varying job size.

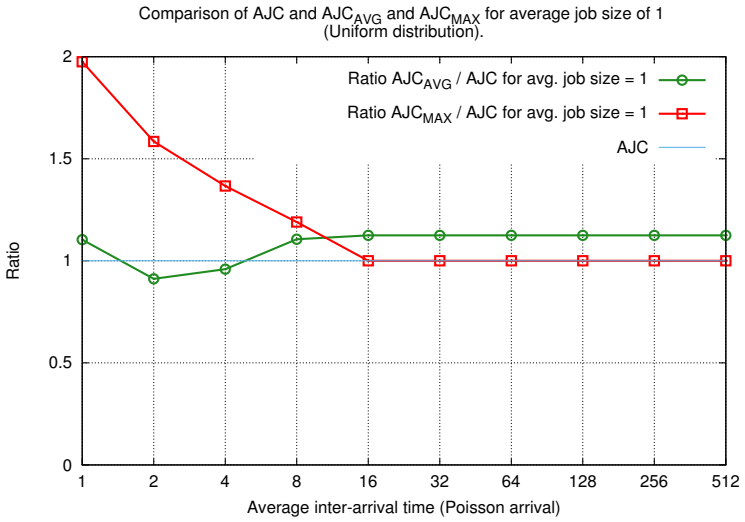


(b) Performance ratio for average inter-arrival time of 16 with varying job size.

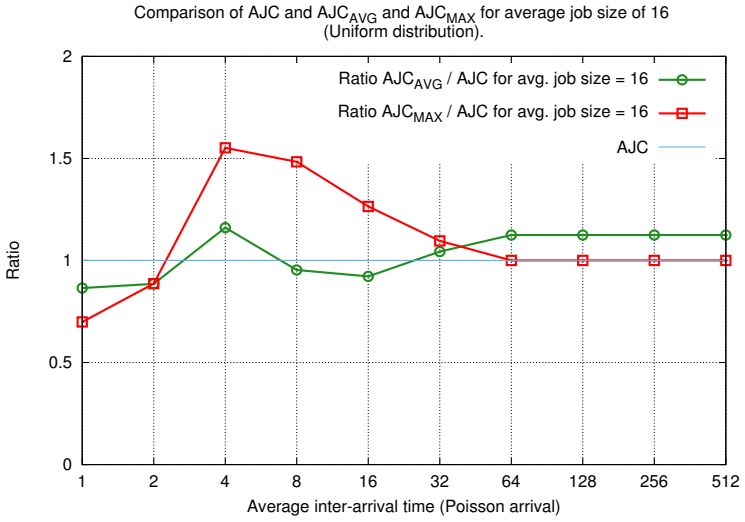


(c) Performance ratio for average inter-arrival time of 512 with varying job size.

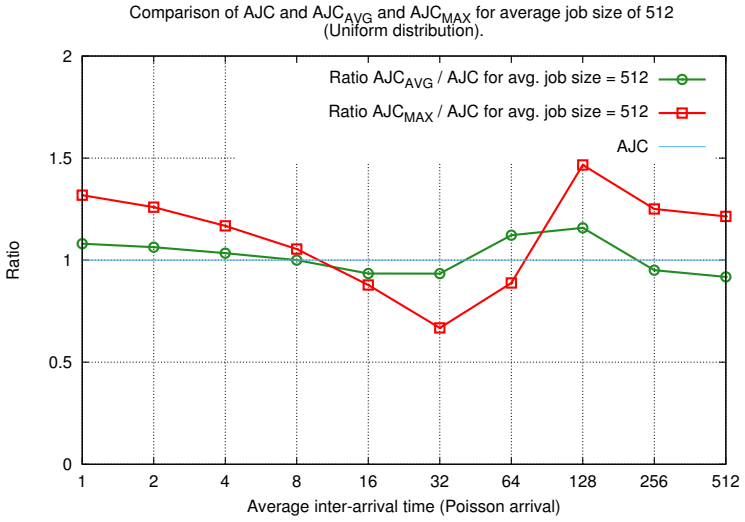
FIGURE 4.11: *Speed scaling vs. semi-clairvoyant fixed speed function*: Measurement shows the ratio of total flow time plus energy between AJC and a fixed speed function that has some information about the job set. Results are grouped according to average inter-arrival time.



(a) Performance ratio for average job size of 1 with varying inter-arrival time.

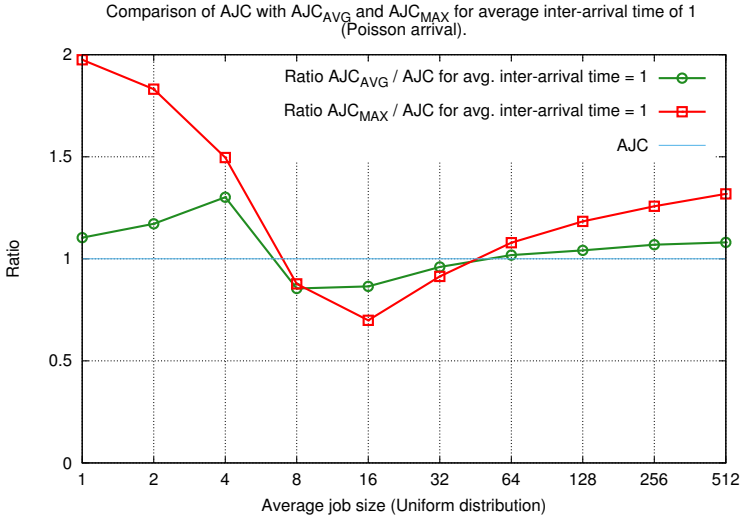


(b) Performance ratio for average job size of 16 with varying inter-arrival time.

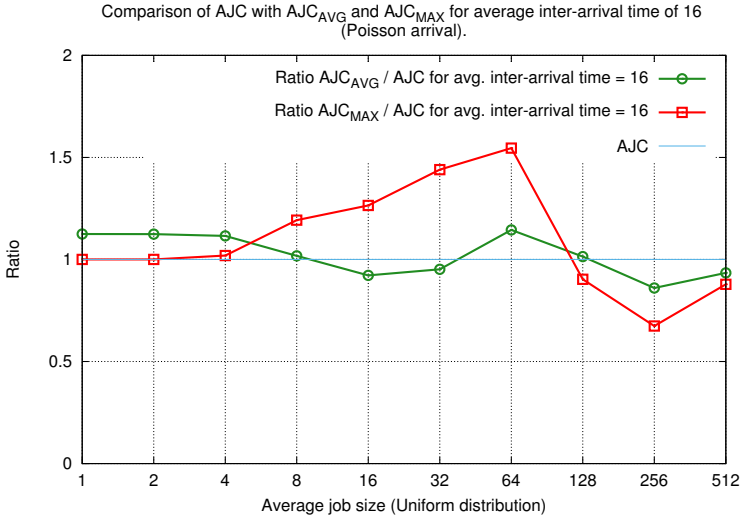


(c) Performance ratio for average job size of 512 with varying inter-arrival time.

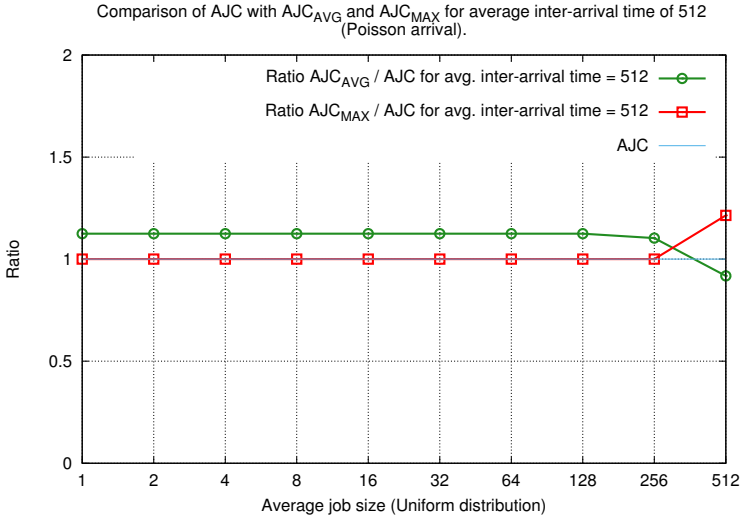
FIGURE 4.12: *Effectiveness of AJC speed spectrum*: Comparison of AJC to a fixed speed function that uses, as fixed speed values, the average and maximum speeds obtained from a prior AJC run. Results show the performance ratio of the total flow time plus energy of fixed speed functions vs. AJC.



(a) Performance ratio for average inter-arrival time of 1 with varying job size.

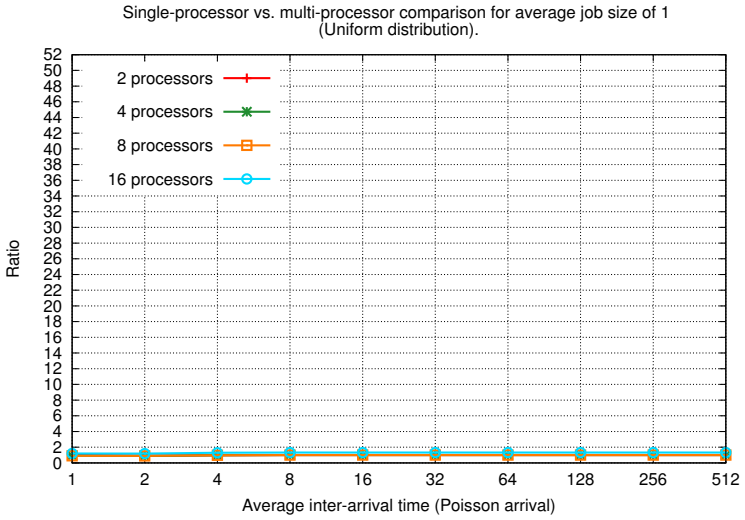


(b) Performance ratio for average inter-arrival time of 16 with varying job size.

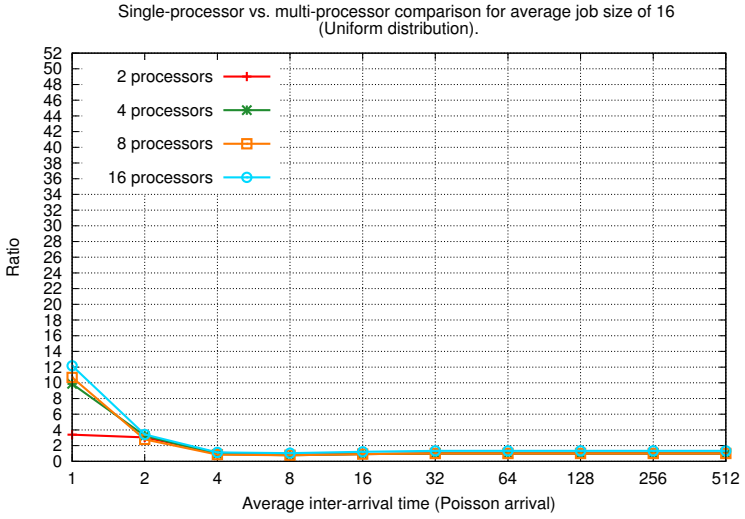


(c) Performance ratio for average inter-arrival time of 512 with varying job size.

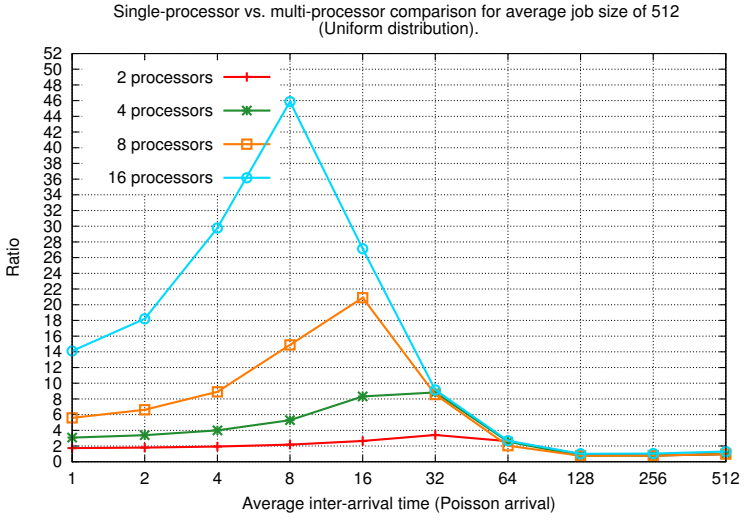
FIGURE 4.13: *Effectiveness of AJC speed spectrum*: Comparison of AJC to a fixed speed function that uses, as fixed speed values, the average and maximum speeds obtained from a prior AJC run. Results show the performance ratio of the total flow time plus energy of fixed speed functions vs. AJC.



(a) Average job size of 1 with varying inter-arrival time

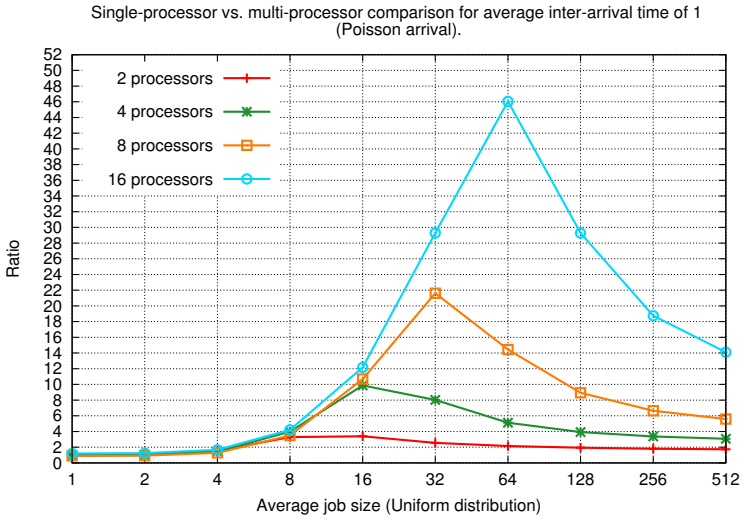


(b) Average job size of 16 with varying inter-arrival time

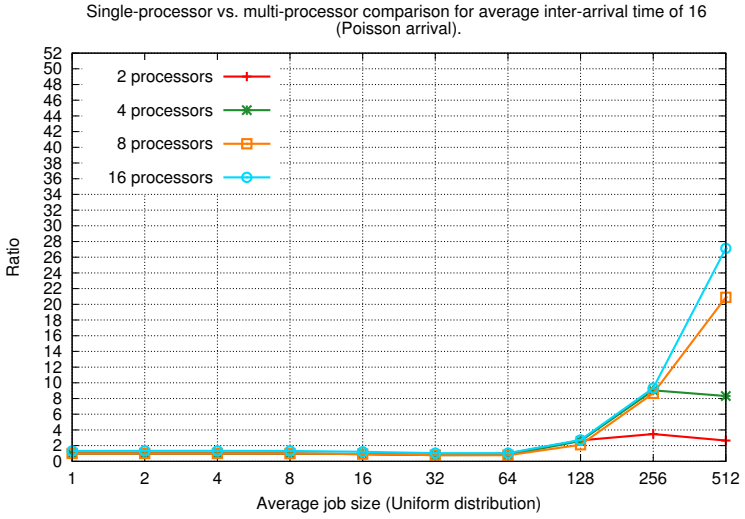


(c) Average job size of 512 with varying inter-arrival time

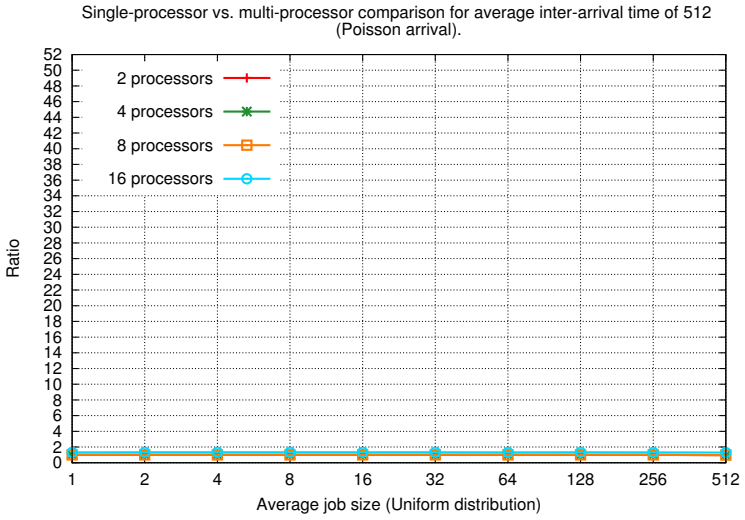
FIGURE 4.14: Results for ROUNDROBIN in terms of average job size comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average inter-arrival time of 1 with varying job size

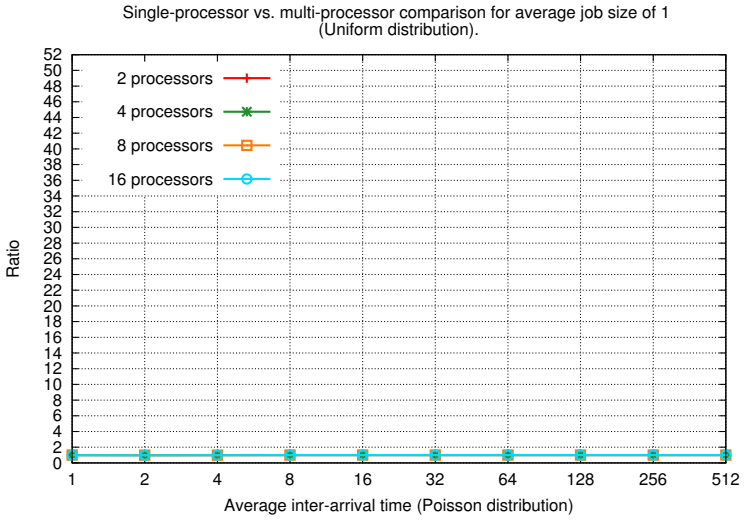


(b) Average inter-arrival time of 16 with varying job size

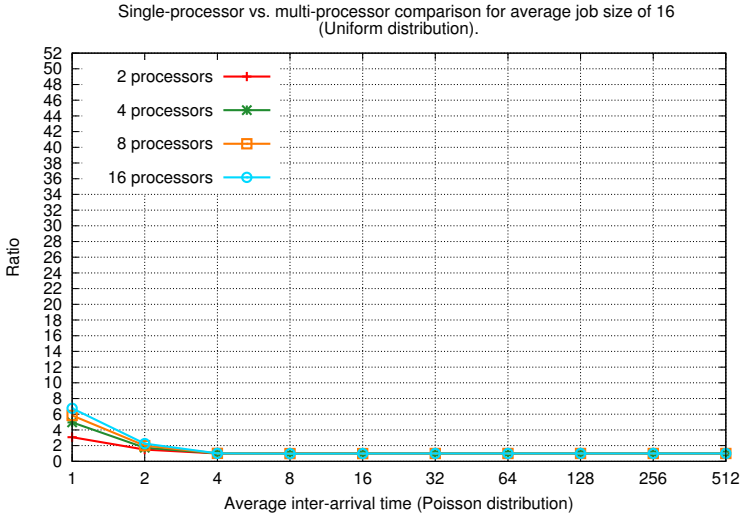


(c) Average inter-arrival time of 512 with varying job size

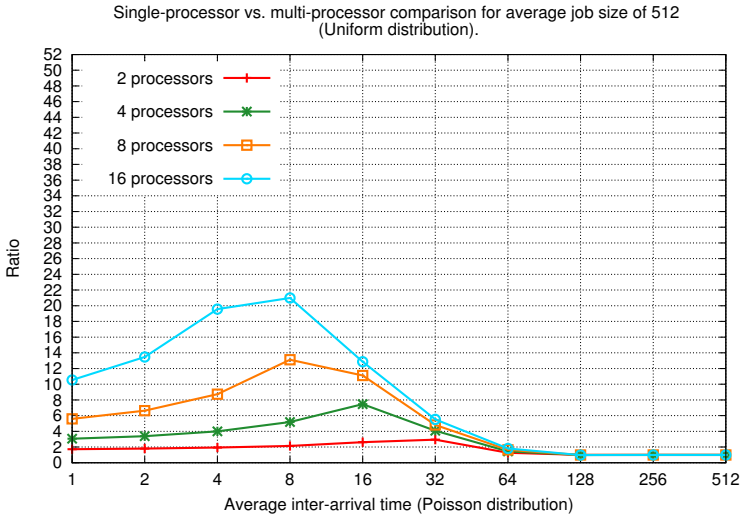
FIGURE 4.15: Results for ROUNDROBIN in terms of average inter-arrival time comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average job size of 1 with varying inter-arrival time

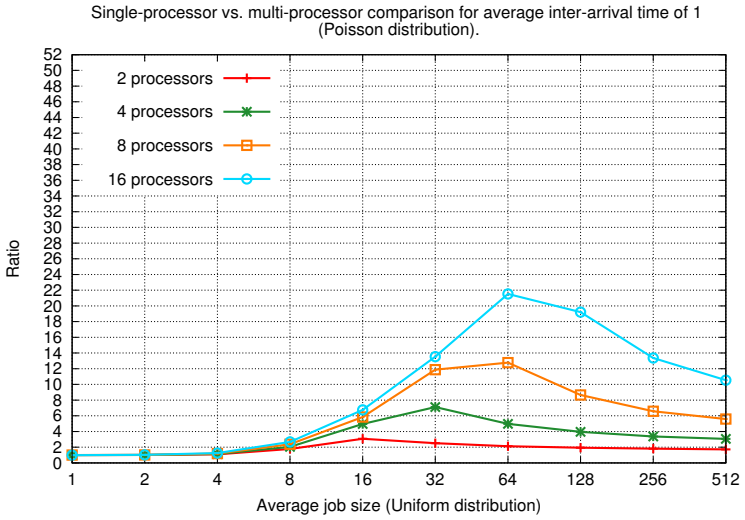


(b) Average job size of 16 with varying inter-arrival time

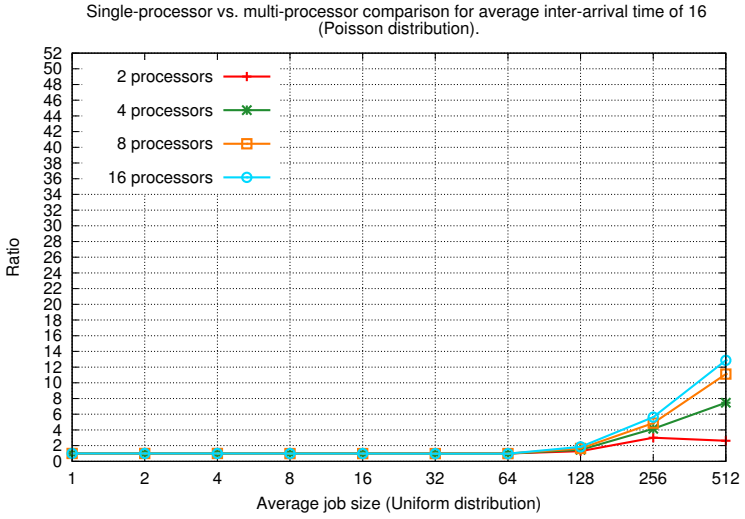


(c) Average job size of 512 with varying inter-arrival time

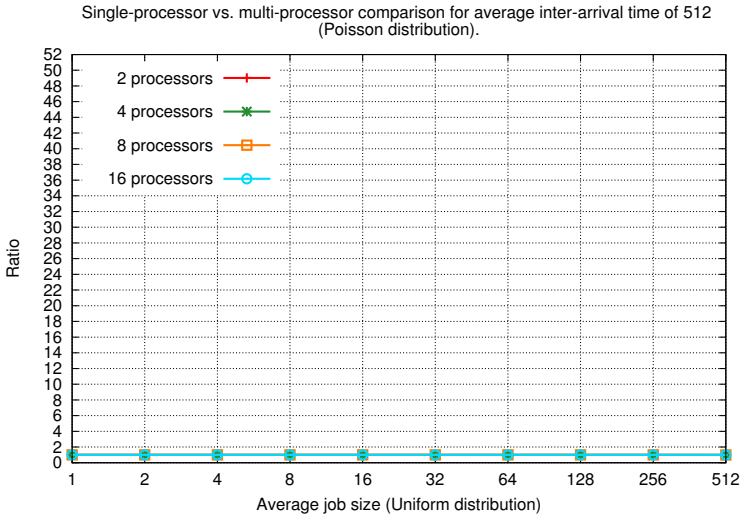
FIGURE 4.16: Results for *MINACTIVECOUNT in terms of average job size comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average inter-arrival time of 1 with varying job size

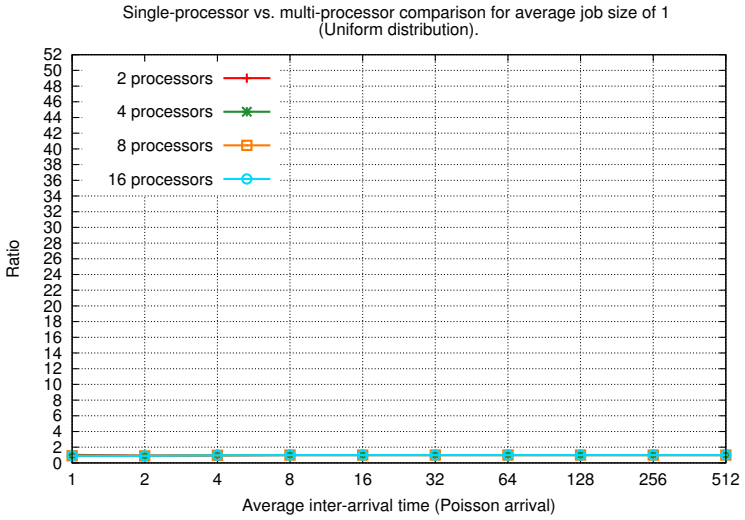


(b) Average inter-arrival time of 16 with varying job size

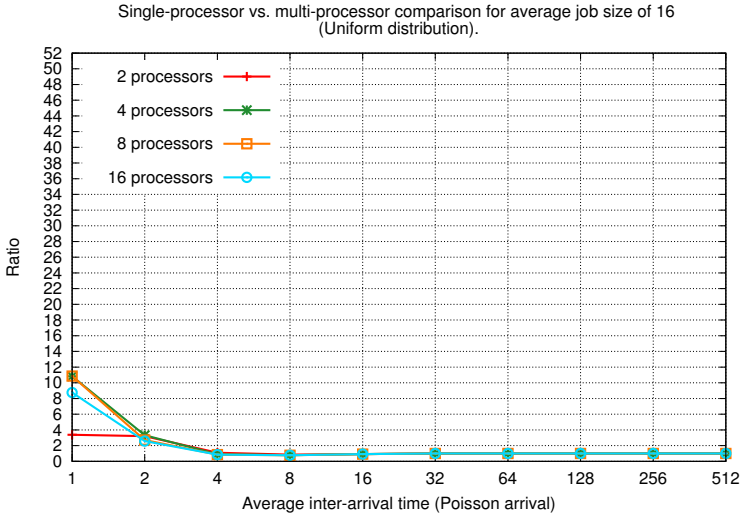


(c) Average inter-arrival time of 512 with varying job size

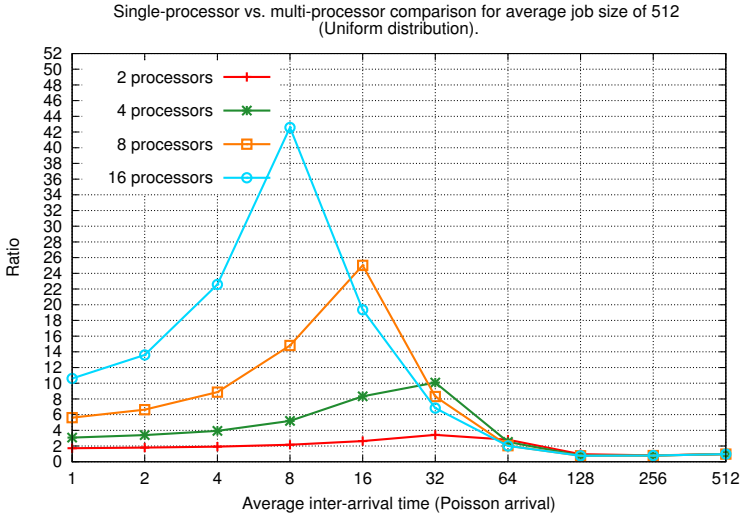
FIGURE 4.17: Results for *MINACTIVECOUNT in terms of average inter-arrival time comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average job size of 1 with varying inter-arrival time

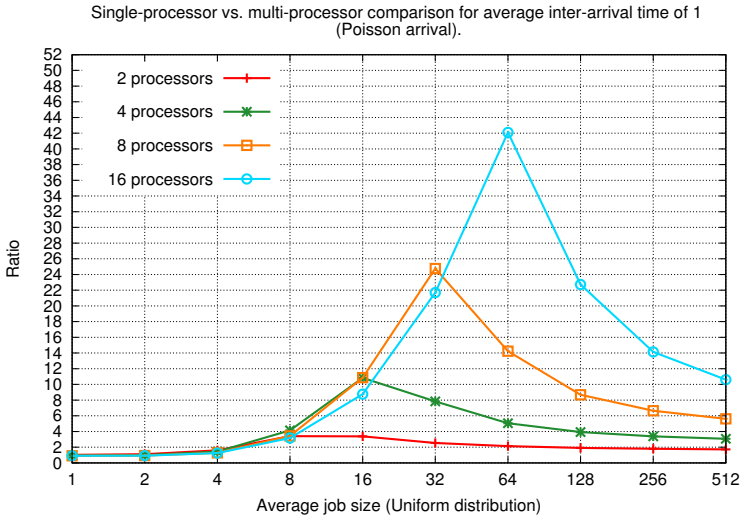


(b) Average job size of 16 with varying inter-arrival time

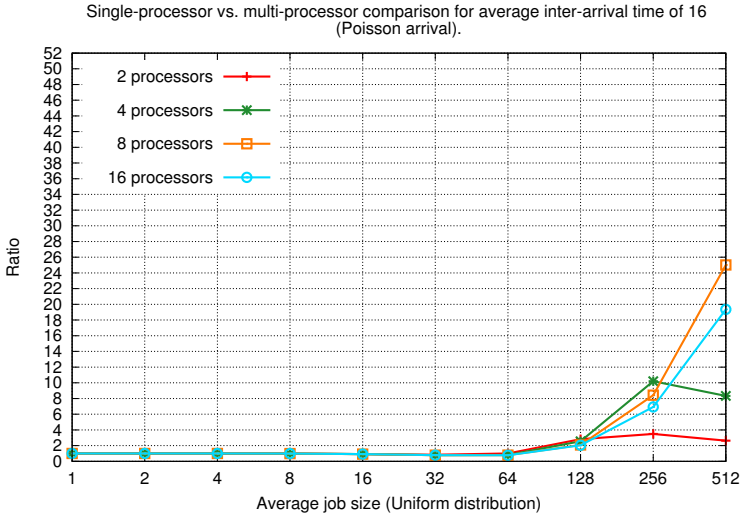


(c) Average job size of 512 with varying inter-arrival time

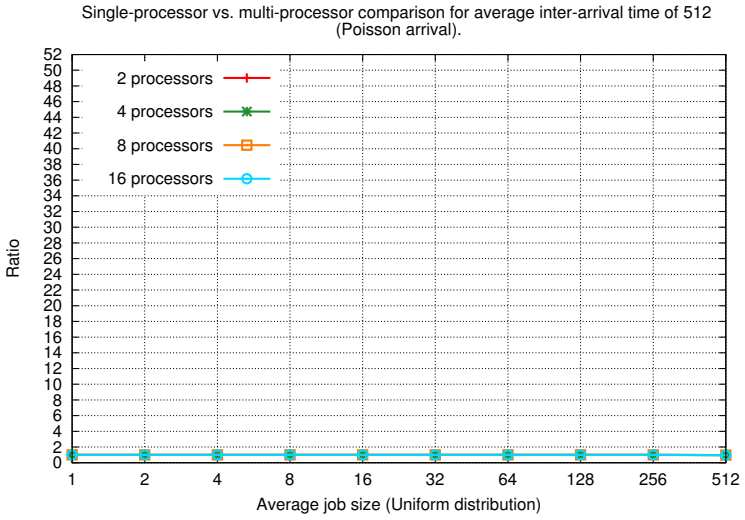
FIGURE 4.18: Results for *MINCOST in terms of average job size comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average inter-arrival time of 1 with varying job size

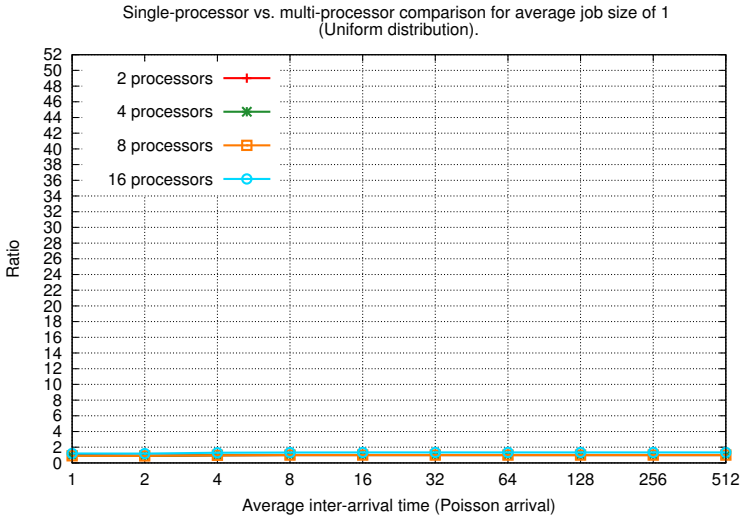


(b) Average inter-arrival time of 16 with varying job size

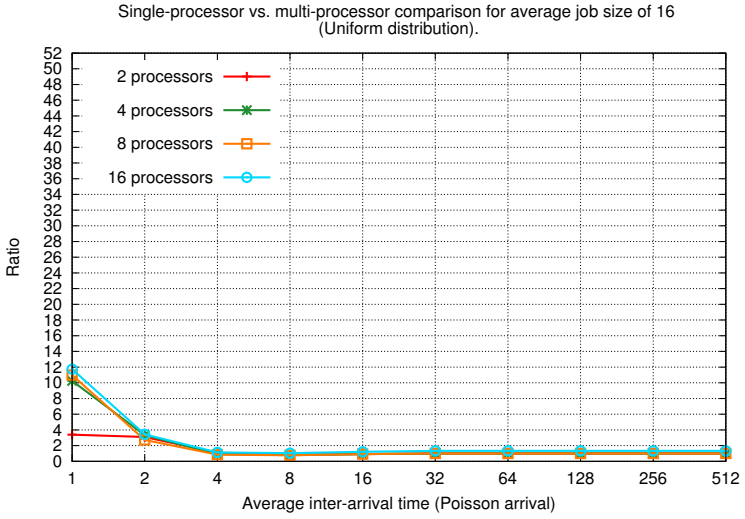


(c) Average inter-arrival time of 512 with varying job size

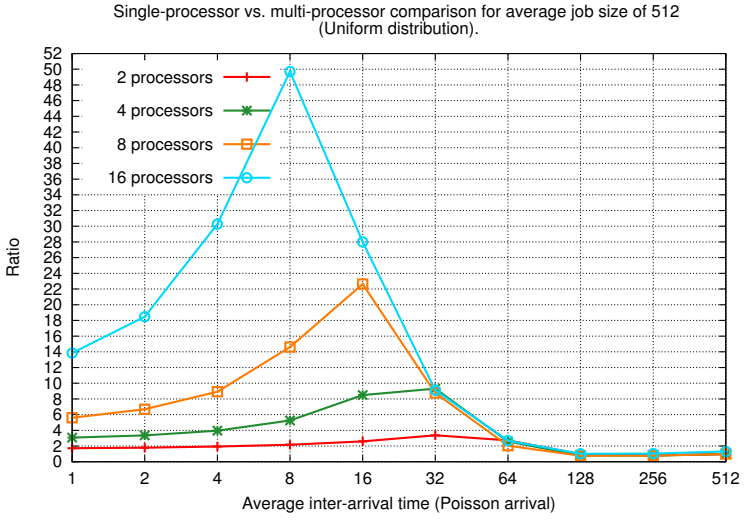
FIGURE 4.19: Results for *MINCOST in terms of average inter-arrival time comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average job size of 1 with varying inter-arrival time

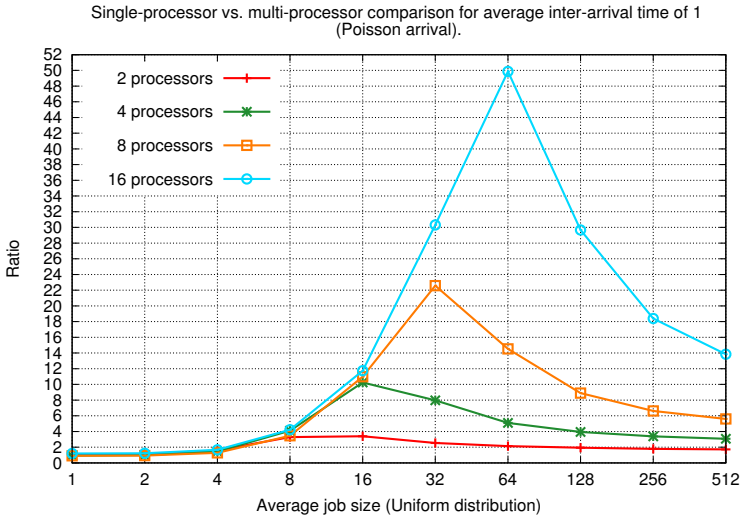


(b) Average job size of 16 with varying inter-arrival time

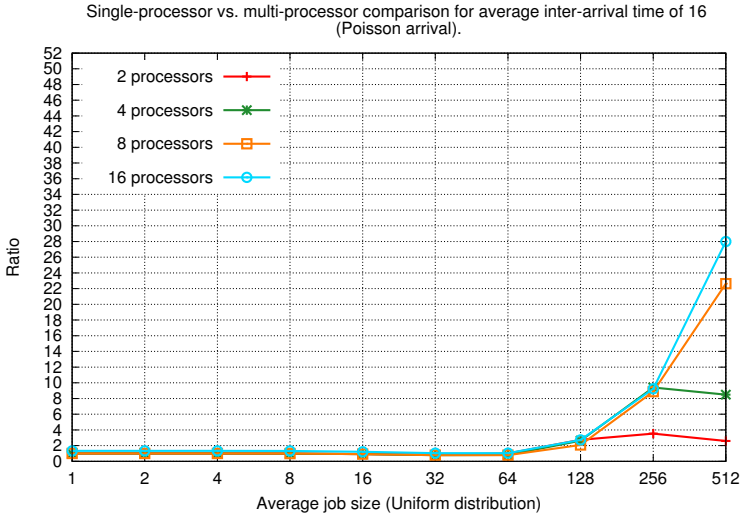


(c) Average job size of 512 with varying inter-arrival time

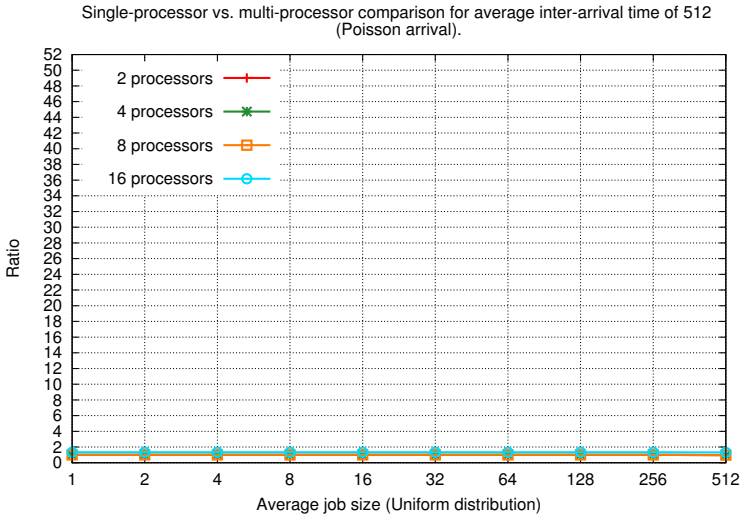
FIGURE 4.20: Results for *MINSIZE in terms of average job size comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.



(a) Average inter-arrival time of 1 with varying job size



(b) Average inter-arrival time of 16 with varying job size



(c) Average inter-arrival time of 512 with varying job size

FIGURE 4.21: Results for *MINSIZE in terms of average inter-arrival time comparing the performance ratio of total flow time plus energy for a single processor vs. multiple processors.

Chapter 5

Background on Parallel Computing with General Purpose GPUs

5.1 Introduction

The modern graphics processing unit (GPU) has evolved to become a more general-purpose, programmable parallel processor in contrast to the earlier generations of specialized, fixed-function processors. However, regardless of the generation, GPUs have always possessed substantial amounts of processing power and computational resources. The difficult aspect has always been a matter of how to leveraging the vast compute resources on the GPU for general-purpose, high performance computing. Earlier approaches relied on writing programs structured according the GPU rendering pipeline and the use of graphics APIs. This means that programs were written in a style very similar to those used in rendering graphics and may not allow for a lot of flexibility and convenience. However, with the introduction of these new GPUs with *unified* (programmable) shaders, programmers now have adequate access to the core computational resources without the details of the fixed-function, rendering pipelines. In the description of the GPU hardware, we will focus on the programmable aspects of the GPU pipeline only because these are the parts we are concerned with when writing general purpose applications that make use of the GPU.

5.2 Comparison of CPU and GPU Hardware Architecture

In this chapter, we begin with a brief introduction to describe the main architectural differences between a CPU and GPU from a hardware point of view. Next, we discuss the

GPU hardware implementation offered by the leading GPU vendors, AMD and NVIDIA. Finally, we describe the concepts and models in the parallel computing framework we use for application development on GPUs. The aim of this chapter is to provide some background knowledge on some of the main working principles and fundamental differences between a CPU and GPU. We shall also discuss some key architectural intricacies associated with the computer system controlled by the CPU.

5.2.1 Memory management in a computer system

In a modern computer application, the actual hardware details of the underlying memory subsystem and medium is abstracted from the user through using what is referred to as *virtual memory*. The virtual memory maps the *virtual addresses* used by a computer program into *physical addresses* on the computer's physical memory or storage medium. This process makes data byte addressable regardless of its location. When a process is launched, the operating system is responsible for allocating a virtual address space for that process as well as the assignment of the physical address space to the virtual address space. This address management is handled by a hardware unit in the CPU often referred to as a *memory management unit* (MMU). One of the functions of the virtual memory is that it enables a process to execute without the need for its code to be resident in the system's main memory. It also relieves the burden of managing a shared memory space with other running processes.

Paging in virtual memory. Another benefit of virtual memory is that it enables a process to be able to utilize more memory than is physically possible through a technique known as *paging*. The virtual address space is divided into fixed-sized blocks referred to as *pages*. Likewise, the physical address space is also divided into fixed-sized blocks called *frames* so that each page of virtual memory can fit into a frame. These virtual pages can either be mapped to any frame in the system's main memory or secondary storage or can also be pending allocation. However, in order to access data contained within a page, the CPU requires that this data already resides in main memory frame. It is the responsibility of the MMU to intervene and provide the appropriate mapping whenever an instruction uses virtual memory. A *page fault* occurs when the data being requested in virtual memory is not yet available in main memory, therefore, the process is halted while the mapping is created after data has been made available in main memory.

Direct Memory Access (DMA). When designing a program that uses discrete GPUs, it is important to know how the operating system's memory management techniques affects the program because data transfer between the GPU device and system's main memory can be a costly operation in such a program. This is where the technique known

as *direct memory access* (DMA) comes into the picture. Any peripheral device, such as a GPU, attached to a computer system will require the help of the CPU in order to access data stored in main memory. DMA offers an efficient mechanism through which peripheral devices can access data in memory without the intervention of the CPU. DMA requires that the data being accessed is already resident in main memory and will not be moved by the operating system before the operation can be initiated. When the operating system is not allowed to move a page, the page is often described as *pinned*. During a DMA operation, a physical address is supplied by a device driver from the CPU to the DMA engine on the GPU. This allows the discrete GPU to perform memory operations while the CPU is free to do other useful work. When the DMA operation is complete the page(s) can then be unmapped from main memory.

5.2.2 Stream processing hardware implementation

Modern GPU architectures from both AMD and NVIDIA are based around the idea of a scalable collection of *streaming multiprocessors* (SMs). These multiprocessors are designed to achieve substantial amounts of parallelism individually with the capability to execute tens to hundreds of threads concurrently. This is made possible because each SM is made up of arrays of execution units commonly referred to as *processing elements* (PEs). Each PE contains a number of *arithmetic and logic units* (ALUs), each capable of executing single-precision floating-point and integer operations. Each PE executes an instance of the program being run on the GPU and can handle both scalar and vector instructions. Despite the fact that a single GPU can hold hundreds to thousands of processing elements, it is still about the same size as a CPU in terms of die size and one might wonder how this is possible?

The architectural design goal of a CPU is to minimize latency, hence, a CPU aims at minimizing the number of execution cycles required to perform a task. Threads executing on a CPU are expected to be quite responsive and in the case of multiple threads executing simultaneously, the CPU is able to manage this by switching very rapidly between threads. The CPU relies heavily on its cache memory and so a vast amount of transistors is devoted to memory and this takes up a considerable amount of die space. In addition, the CPU also devotes a significant amount of transistors on the control unit which is responsible for tasks such as fetching data, decoding instructions, managing execution and storage of results.

On the other hand, the GPU is designed to achieve high rates of throughput and parallelism. In order to accomplish this it becomes rather imperative that a significant amount of transistors is dedicated to execution units, as illustrated in Figure 5.1(b). As a result,

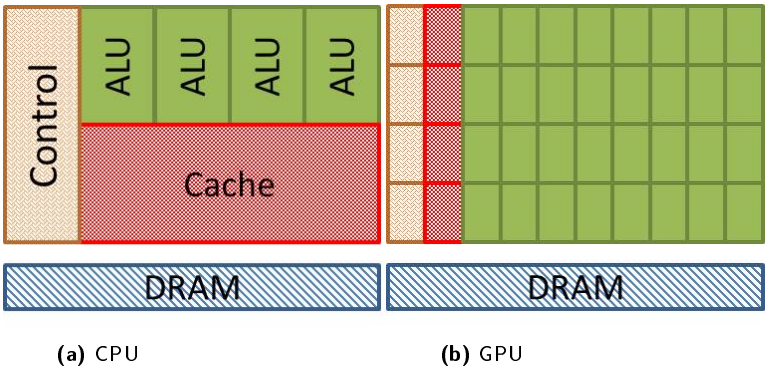


FIGURE 5.1: A fundamental difference between a CPU and a GPU is that the GPU dedicates majority of its transistors to execution units.

GPUs are typically equipped with very small caches (fast, on-chip memory) compared to a CPU and since there is not much execution management involved, in comparison to the CPU which has to deal with things like branch conditions and predictions, control units consume a small amount of die space. Threads executing on a GPU are considered to be *light-weight* when compared to CPU threads and as a result, context-switching between threads on the GPU is extremely fast and this enables the GPU to 'hide latency' by swapping out threads while they perform memory read/write operations. We shall see more details when we discuss scheduling in the next subsection.

Furthermore, the memory subsystem on a GPU is optimized for throughput as well because, given the SIMD nature of execution on the stream processors, memory operations occur in bulk. This means that memory transactions are coordinated in such a way that requests made to consecutive memory addresses are serviced in a single transaction depending on the width of the memory controller. A GPU usually consists of multiple memory controllers each capable of handling separate memory transactions, hence, providing a high level of throughput and memory bandwidth. As a result, the memory subsystem on GPUs usually have peak bandwidths orders of magnitude higher than can be achieved between the CPU and RAM.

5.2.3 Scheduling - threads, warps and wavefronts

In a computer system, an application executing on the CPU usually consists of execution fragments commonly referred to as *threads*. A thread refers to the smallest unit of execution that can be influenced by the scheduler *process* of the operating system. A process is an instance of an application running on a computer system and it contains the program code for that particular application. A process can invoke multiple threads to handle its tasks and these threads can share common system resources such as main

memory or RAM, however, threads from different processes cannot share the same portion of memory. Each thread within a process can execute different paths through the program code of the application and can also access data different memory locations in memory. At any given time a CPU core can only execute a single thread, therefore, a single-core CPU can only execute a single thread or process at a time. However, for the case of Intel CPUs with *Hyper-Threading™* [46, 110] technology, a physical CPU core can be viewed as two virtual or logical cores by the operating system allowing the operating system to schedule more than one task to the virtual cores so that the single physical core can execute up to two threads simultaneously. This feature is made possible because the Hyper-Threaded processor can duplicate parts of the CPU responsible for storing the architectural state only. A typical computer system usually runs a host of applications at the same time which in most cases exceed the number of processors available in the system and given that each application can have one or more processes, each with a number of threads, how does the system cope such a work load?

The CPU is capable of processing multiple threads, in order words, multi-task, by implementing what is known as *time-division multiplexing*. Signals are propagated over a single path using synchronized switches to achieve time-sharing of the available bandwidth and this is the main idea behind *context switching*, which allows the CPU to switch between threads and allowing each one a fraction of the CPU time for execution. During the process of a context switch, the state of the running process is saved so that it can be restored at a later point in time when the process needs to resume execution. A context switch typically involves saving the state of a register, stack, thread or even a process. The process usually happens quite rapidly and is so quick that it gives the user the impression that the processes or applications are being executed simultaneously. This is the basic idea behind how the CPU handles threads and processes in order to respond to user applications on a computer system, however, the mode of operation on the GPU is quite different from a general perspective.

An application executing on the GPU also comprises of threads but in contrast to the CPU, and due to the SIMD (single instruction multiple data) pattern of execution, the smallest unit of execution that flow control can affect is a group of threads instead of a single thread. This group of threads, referred to as a *wavefront* and *warp* for AMD and NVIDIA hardware respectively, usually comprises of a fixed number of threads subject to hardware design. For instance, a wavefront is made up of 64 threads while a warp consists of 32 threads. Any given application executing on the GPU can be made up of one or more thread groups. All threads in execute the same piece of code but the data processed individually by each thread can be different. Threads in the same group are usually scheduled on the same streaming multiprocessor by the GPU scheduler and each thread executes on the individual processing elements. When threads are created

each one is assigned a unique serial number that serves as a unique index for each thread and this unique index can be queried from the piece of code that particular thread is executing, hence, a thread is aware of its own index. Since the scheduler on the GPU does not schedule each thread individually like the CPU does and given the fact that the GPU does not dedicate a lot of transistors to control units, managing branch conditions and predictions in a GPU program becomes more involving compared to a CPU program.

Although all threads execute the same piece of code while working on possibly different data, there are cases where some threads might be required to execute a path of the code different from other threads. The more critical case is a situation where some threads within a wavefront/warp execute a path of the code different from the rest due to some branch condition in the code and we refer to this as *thread divergence*. It may appear as though the threads are carrying out their individual tasks concurrently, however, since the GPU is not optimized for handling branching in program codes, an extra step is incorporated during execution when thread divergence occurs. Firstly, all threads within a warp/wavefront will execute all paths of the code and the unwanted outcomes or results are 'masked' out so that only the valid operations are maintained. Let us consider the example illustrated in Figure 5.2.

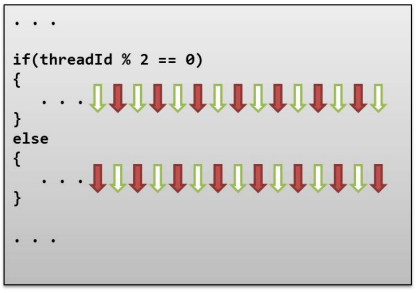


FIGURE 5.2: Thread divergence occurs as a result of threads within a wavefront/warp taking different code paths.

In this example, the program code requires that threads with indices that are even numbers perform a different task from the odd-numbered threads. Regardless of the GPU hardware, this sort of program code will result in thread divergence as half of the wavefront/warp will be executing a different code path from the other half and as a result, the whole group will need to make two passes in order to execute each branch. In the first pass, the threads with even indices indicated by the green (unshaded) arrows are activated to perform their operations while the threads with odd indices shown in red (shaded) are deactivated. In the second pass, the process is inverted so that the threads with odd indices become activated while the remaining threads with even indices are deactivated.

Note that thread divergence occurs when the *branch granularity* of the GPU hardware is not maintained when assigning tasks to threads. Branch granularity is simply the number of threads that must be executed during a branching procedure and this number is typically equivalent to the size of a wavefront/warp [2]. This means that it is still possible to allow some threads to execute a different portion of the program code as long as the branch granularity is not broken. Let us consider the following example illustrated in Figure 5.3.

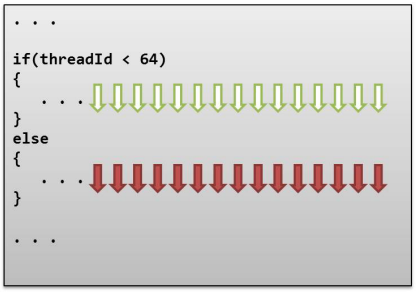


FIGURE 5.3: Thread divergence can be avoided if branch granularity of the GPU hardware is maintained.

Recall that the wavefront size for AMD hardware is 64 threads while the warp size for NVIDIA hardware is 32 threads. In this example, threads numbered 0 to 63 are assigned a different task from the rest of the threads. In this case, regardless of the hardware whether AMD or NVIDIA, we can observe that branch granularity is maintained. For the AMD hardware the first wavefront will execute the first part of the control statement while for NVIDIA hardware, it is the first two warps. Since the branch granularity is maintained both parts of the branch condition can be executed concurrently and the extra operation required to mask out invalid operations will not be required. Hence, it is important that thread divergence is avoided as much as possible when writing program codes for GPUs.

Another important factor that must be taken into account when designing algorithms that leverage GPUs is the arithmetic intensity of the program code, in other words, the ratio of arithmetic operations to memory operations. Accessing the GPU's video memory usually incurs a considerable amount of clock cycles which usually results in stalling during execution process. The GPU is highly optimized for throughput and programs with very high arithmetic intensity will usually perform better compared to a program that is mostly memory bound. However, the GPU scheduler is quite robust in terms of dealing with such cases where a wavefront/warp is stalled due to memory transactions using a technique regarded as *hiding latency*. This involves swapping out wavefronts/warps that have stalled as a result of a memory transaction so that idle ones can continue with their tasks. We illustrate the concept of hiding latency with the example illustrated in Figure 5.4. In this example, four groups of threads are shown

labelled $W0$, $W1$, $W2$ and $W3$, and they are meant to be executing on the same streaming multiprocessor.

FIGURE 5.4: The GPU is able to hide latency by swapping out wavefronts/warps that stall during memory operations.

Let us assume that execution begins at time $t0$. During execution, the first group, $W0$, encounters a memory operation at time $t1$ and stalls as a result while waiting for data to be read/written to memory. While the memory transactions are being handled by the memory controller, the GPU scheduler immediately swaps out group $W0$ for group $W1$ and execution continues. Subsequently, group $W1$ will reach the same point in the code that requires the same memory transactions and the GPU again swaps out group $W1$ for the next group $W2$ at time $t2$ and the process continues as long as there idle wavefronts/warps. In this example, we can easily observe that there isn't enough arithmetic intensity in the program code to hide the latency incurred during memory transactions, however, there is a considerable amount of work to keep the GPU busy with sufficient wavefronts/warps to hide latency. In this case, we say we have achieved maximum *occupancy* for the GPU which refers to the ratio of active wavefronts/warps in flight to the hardware limit for the maximum number of wavefronts/warps that can execute concurrently. When a program code is implemented correctly, it is still very possible to achieve high amounts of throughput from the GPU hardware. One can also improve memory operations and overall performance even further by coordinating the memory access patterns from the threads.

The memory controllers on the GPU can only transfer a given amount of data, measured in bits (or bytes), when processing a memory request and this is usually referred to as the *memory bus width*. The memory bus width is often equivalent in size to the cache line width and a similar setup exists between the CPU and the system main memory. When a request is made to, for instance, fetch data from a particular location in memory, the memory controller will return data equivalent to the size of its bus width per request. In order get the best possible performance out of the memory controllers on the GPU it is imperative that memory access patterns are *coalesced*, which simply means that consecutive threads should be made to access consecutive locations in memory.

The example shown in Figure 5.5 illustrates a non-optimal memory access pattern from threads in a wavefront/warp.

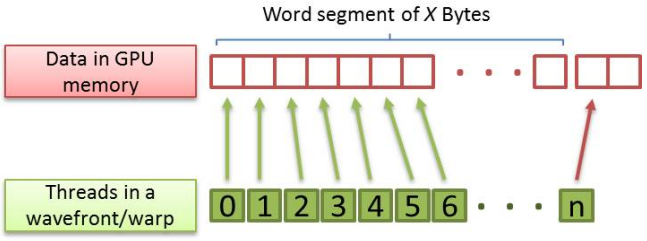


FIGURE 5.5: Non-coalesced memory access patterns can result in poor performance on the GPU hardware.

Here, the last thread, n, is requesting data from a location that is non-contiguous with the rest of the threads in the group. Given that the memory controller will return data equal to the size of the bus width or word segment, the unused data is wasted and the whole group will have to be serviced in two memory transactions. However, if the memory requests are coalesced so that threads access consecutive locations in memory as shown in Figure 5.6, data fetched by the memory controller will be fully utilized by all threads and will only require a single memory transaction thereby avoiding the expensive overheads related to memory operations.

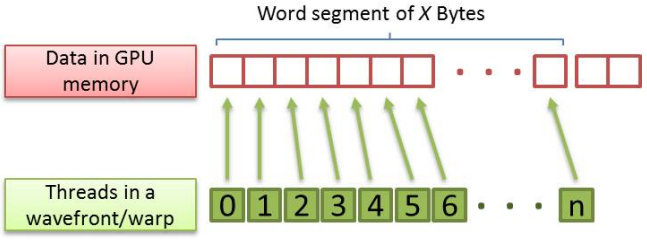


FIGURE 5.6: Coalesced memory access patterns can improve performance on the GPU hardware.

5.3 Vendor-specific SIMD implementations

So far in this chapter we have discussed some aspects of the GPU hardware but from a general perspective. In this section we shall be discussing the hardware implementations on offer from AMD and NVIDIA, the leading GPU vendors. Precisely, we shall discuss the implementation details of their latest hardware architecture and will be focusing on the parts that are directly related to running general purpose applications on the GPU.

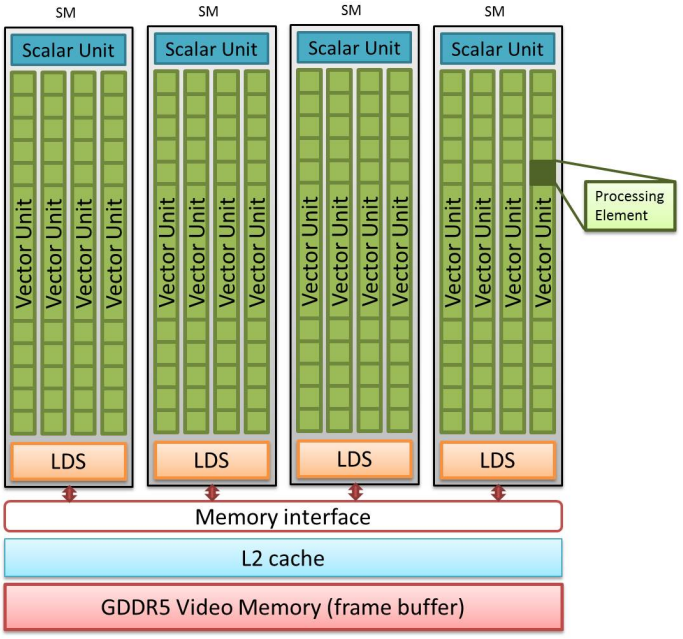


FIGURE 5.7: Generalized block diagram of AMD's GCN architecture.

5.3.1 The Graphics Core Next architecture (AMD)

The Graphics Core Next (GCN) architecture was first introduced by AMD during their Fusion Developer Summit (AFDS) in 2011 [102, 109] and later launched with their Southern Island devices with model names in the form of Radeon™ HD 7xxx. For instance, at the time of launch, their flagship model was the HD 7970 GPU model. The GPU devices based on the GCN architecture are available as discrete peripheral devices or part of their system-on-chip line of products called *accelerated processing units* (APU), which is a combination of a GPU and a CPU on the same die to provide a heterogeneous solution for computing tasks.

At the heart of the GCN architecture are the streaming multiprocessors and, without loss of generality, Figure 5.7 depicts a simple block diagram of a GPU with four streaming multiprocessors. Each multiprocessor comprises of processing elements grouped into what is known as *vector units* and each vector unit is made up of 16 processing elements. There are a total of 4 vector units in each streaming multiprocessor which amounts to a total of 64 processing elements per multiprocessor. At any point in time during the execution of a program code on the GPU, a vector unit is responsible for executing a wavefront, hence, each thread in the wavefront is assigned to a single processing element. However, a wavefront is scheduled quarterly so only 16 threads gets scheduled at a time until eventually all 64 threads in the wavefront are scheduled.

Each processing element consists of its own private memory or registers visible only to the thread it is executing. A form of high-speed, low-latency memory, known as local

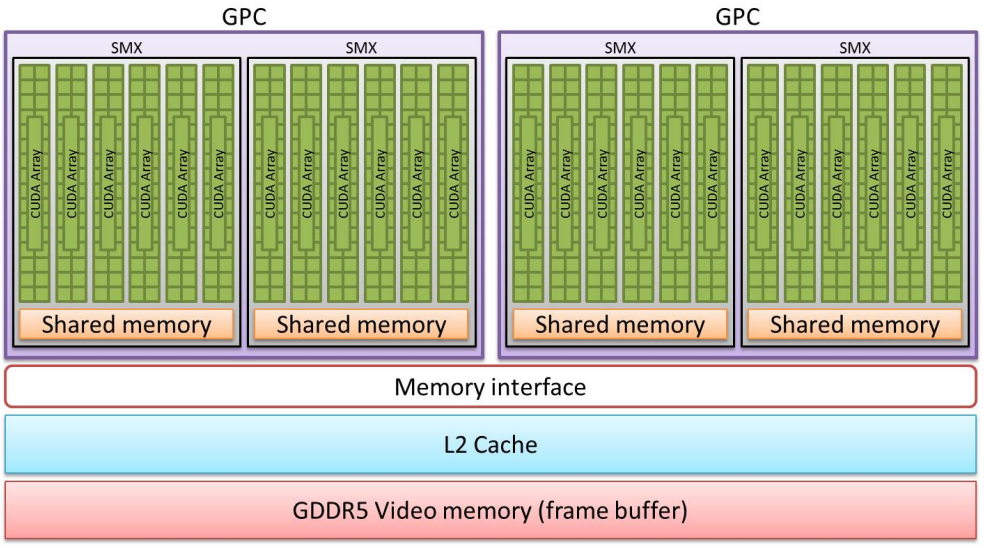


FIGURE 5.8: Generalized block diagram of NVIDIA's Kepler architecture.

data store (LDS), also exists on each multiprocessor and it is visible to all threads in a group executing on the multiprocessor. The LDS provides support for scatter and gather operations and a means for threads to share data with each other. The frame buffer or video memory resides off-chip and provides huge amounts of storage but incurs the highest amount of latency. A robust memory controller provides high-speed and high-bandwidth access for the streaming multiprocessors to the frame buffer with several memory channels. To put this into perspective, let us consider an actual GCN device like the AMD Radeon™ HD 7970 GPU. This GPU features 32 SMs which equates to a total of 2,048 PEs. Each SM has a LDS size of 32 kB and the reference design has a 3 GB GDDR5 frame buffer with a 384-bit wide memory interface.

5.3.2 The Kepler architecture (NVIDIA)

NVIDIA's Kepler architecture is a successor to the previous Fermi architecture and was introduced with the NVIDIA GeForce™ GTX 6xx series. They are also available as discrete GPUs or embedded devices, like their Tegra™ line of mobile GPUs. The Kepler architecture is also based around the streaming multiprocessor, which NVIDIA refers to as SMX [82]. The block diagram in Figure 5.8 illustrates how the compute components are arranged from a general point of view.

The heart of the Kepler architecture for compute lies in the *Graphics Processor Cluster* (GPC) and a single GPU comprises a number of these. The GPC is simply a group of 2 streaming multiprocessors. Each SMX in a GPC is made up of 6 CUDA (*Compute Unified Device Architecture*) arrays and each CUDA array further consists of 32 CUDA cores. A CUDA array is akin to the vector unit in the AMD's GCN architecture and

the CUDA cores are simply the processing elements. This implies that a single SMX consists of 192 CUDA cores or processing elements. The CUDA array is responsible for executing a warp with a single thread being executed by one CUDA core.

A shared memory also exists for each SMX which is only accessible to warps executing on that SMX. The shared memory also supports scatter and gather operations and allows warps on the same SMX to share data. Each CUDA core also has registers which are private to the thread executing on each of them. As usual, a large video memory resides off-chip and a memory interface provides high-speed, high-bandwidth access for the multiprocessors.

As an example, the NVIDIA GeForce™ GTX 680 GPU features 4 GPCs which equates to 8 SMX for a total of 1,536 CUDA cores. Standard video memory configuration is 2 GB GDDR5 memory with a 256-bit wide memory interface.

5.4 GPU Computing Framework

In this section, we shall discuss the parallel computing framework and model around which applications are developed that leverage computing resources of the GPU. Such a framework allows regular computer programs to interact with the GPU and offload compute-intensive tasks where necessary. This thesis is primarily based on the OpenCL parallel computing framework so we will be focusing on it in this section.

5.4.1 The Open Computing Language

The *Open Computing Language* (OpenCL) is a cross-platform API for writing programs intended to take advantage of heterogeneous and multi-core systems. It was initially developed by Apple Inc. who submitted the first proposal to the Khronos Group. This proposal was refined in collaboration with technical teams at AMD, IBM, Qualcomm, Intel and NVIDIA and the first public release after review by members of Khronos Group was on December 8, 2008. The API is built on the C programming language. Existing parallel computing frameworks includes CUDA by NVIDIA for writing programs for its GPUs as well as vendor-specific OpenCL implementations from AMD, Intel and IBM. Language bindings and wrappers also exist for programming languages like C++, C#, Python and Java [30, 48, 89, 105].

OpenCL is built around a hierarchy of models that make up its foundation and these are the platform, execution, memory and programming models.

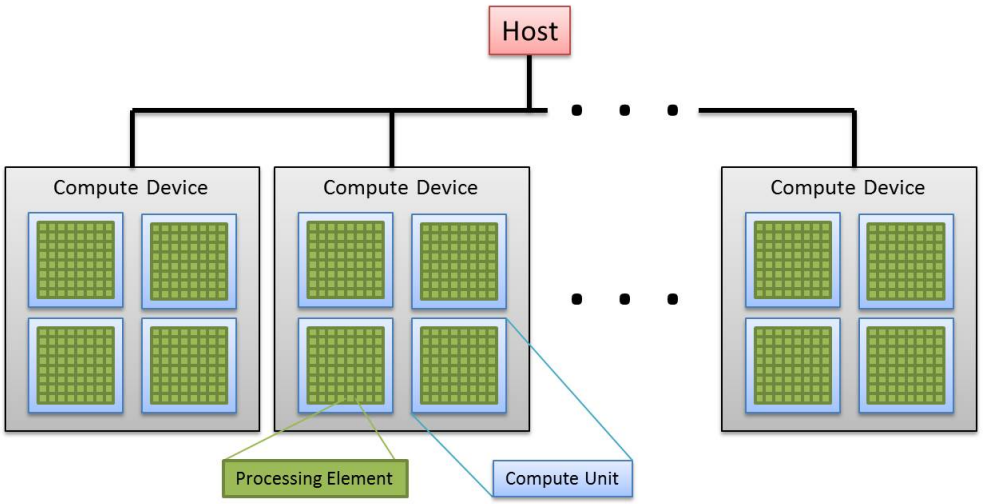


FIGURE 5.9: Block diagram illustrating the major components of the OpenCL platform.

5.4.1.1 Platform model

The platform model provides a definition that encapsulates a hardware system the supports heterogeneous computing with the OpenCL framework. It aims to consolidate different devices including CPUs, GPUs, DSPs and other discrete devices under a single platform. The block diagram in Figure 5.9 illustrates the components of the OpenCL platform.

The OpenCL application is executed by the *host*, which typically refers to the computer system including the CPU, its main memory and secondary storage like hard drives. The host is connected to one or more OpenCL devices and such a device is referred to as a *compute device*. A compute device is simply any device that supports the OpenCL specifications like the modern GPUs. A CPU can also be seen as a compute device, hence, OpenCL can also be used to leverage multi-threading capabilities of multi-core CPUs. A compute device is further divided into one or more *compute units*. For instance, a streaming multiprocessor in a GPU is a compute unit. Furthermore, the compute unit is also divided into one or more processing elements and computations within a compute device happens within the processing elements.

The OpenCL application running on the host can be written in any native programming language such as C, C++, Python and so on. The host application coordinates program execution normally and interacts with the compute device by submitting *commands* that are then interpreted by the compute device usually through software drivers installed with the device.

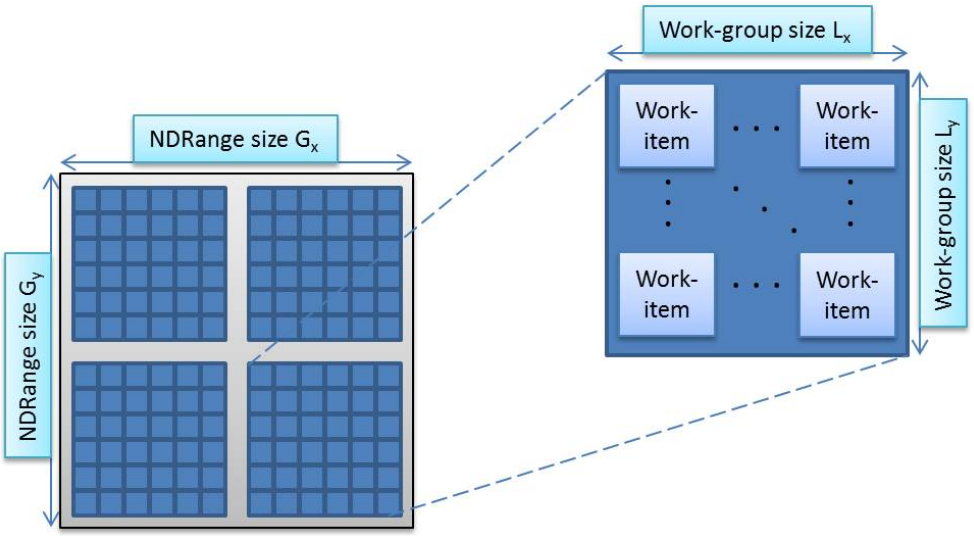


FIGURE 5.10: Decomposition of an OpenCL index space into work-groups and work-items.

5.4.1.2 Execution model

Under the platform model, we mentioned the OpenCL application, or *host program*, that runs on the host. However, the actual program code that executes on the compute device is usually written separate from the host code and it is known as a *kernel*. A just-in-time compilation method is used to compile the kernel to produce the program binary to be executed on the compute device. A kernel is written as a *data-parallel* code, that is, parallelism is achieved through spatial distribution of data across threads running on parallel processors. This idea is part of the core principles behind an OpenCL application, so, how does it work?

To execute a kernel, OpenCL defines an index space called an *NDRange* and this represents an N-dimensional index space where N can be 1, 2 or 3. An instance of a kernel executes at each point in this index space. This kernel instance is known as a *work-item* and each work-item can be uniquely identified in this index space by a global ID. Each work-item executes the same code, however, the path through the code and data operated on might vary among work-items. Furthermore, work-items are organized into *work-groups* which offers a coarse-grained division of the NDRange. Consequently, each work-item can be uniquely identified in the NDRange through its global ID, and within a work-group through a local ID. Work-groups can also be uniquely identified using IDs which means that a work-item can also be uniquely identified by a combination of its local ID and its work-group ID. On the compute device, a work-group is assigned to a single compute unit and each work-item in the work-group executes on a processing element. Figure 5.10 illustrates how the OpenCL NDRange is decomposed into work-items and work-groups.

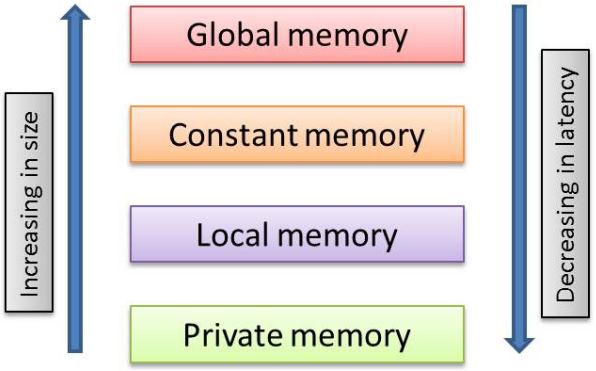


FIGURE 5.11: Illustration of the OpenCL memory model.

5.4.1.3 Memory model

OpenCL defines four different memory regions each of which vary in size, speed of access and accessibility by work-items in the NDRange.

1. **Global memory**. This region of memory is accessible to all work-items in the NDRange and it is usually the largest region. Depending on the compute device capabilities, read/write operations from/to this memory region may be cached. On a GPU, for instance, the global memory maps to the video memory.

2. **Constant memory.** This is a region of memory that remains constant during execution. Data stored in constant memory is initialized by the host and cannot be modified by the kernel.

3. **Local memory.** This region of memory is private to a single work-group and is accessible to all work-items in that work-group. Work-items can allocate and share variables using this memory region. On the GPU this region of memory usually maps to the LDS (AMD)/shared memory (NVIDIA) that resides in each compute unit.

4. **Private memory.** This region of memory is private to a single work-item executing on a processing element. Variables declared in a work-item's private memory cannot be accessed by other work-items.

Memory consistency in OpenCL is relaxed which implies that the state of memory is not guaranteed to be consistent across all work-items in the NDRange. To be precise, the state of the local memory is guaranteed to be consistent for all work-items in a work-group after a *work-group barrier* (synchronization point). However, OpenCL does not guarantee a consistent state for global memory across work-groups.

5.4.1.4 Programming model

Parallelism in an OpenCL program is mainly achieved through data-parallel program code that utilizes the SIMD units in a compute device. However, OpenCL also supports a task-parallel approach where a single instance of a kernel is enqueued without the notion of an index space. This is equivalent to executing a kernel with only one work-item in a work-group on a compute unit. In this approach parallelism is usually achieved at instruction level via vector data types or by enqueuing multiple kernels.

Chapter 6

Parallel Algorithms for Heterogeneous Systems with GPGPUs

6.1 Introduction

In this chapter, we present an extensive study of the development of applications for heterogeneous systems that consists of one or more graphics processing units (GPU) as the main form of co-processor or accelerator. The massively parallel nature of the GPU architecture makes them more than capable candidates for offloading portions of an application that are computationally intensive. In order to fully benefit from the huge compute resources available on these GPU devices, the portion of code that is being targeted on for the GPU device must exhibit some parallel substructure. As a result not all applications can benefit from using a GPU. To be precise, applications that benefit the most from GPU devices are those that can be formulated into a *data-parallel* application where data locality is very important and communication between processing nodes is almost non-existent. This is because GPUs are optimized for high throughput data manipulation as seen in image processing and graphics rendering.

Developing applications that leverage the GPU device as an accelerator is quite different from writing applications for the conventional CPU because there are aspects of the GPU hardware that need to be taken into consideration, as discussed in Chapter 5. As a result, it is difficult and quite challenging to predict how a program will perform on various GPU devices given the number of factors in play. To demonstrate these intricacies, we present four applications that can be grouped into two subcategories consisting of dynamic programming and n-body method based on *Berkeley's 13 dwarfs*

of scientific computing [6, 27]. A dwarf is meant to classify important applications based on their pattern of computation and communication with the aim of providing an optimized and common way of designing parallel formulations. We design and implement these applications from ground up and evaluate their performance on different hardware configurations. In addition, we define a number of performance metrics that help to capture several system parameters as they interact with each specific application. Since the aim of an accelerator is to help improve overall application performance compared to using only the CPU, we also implement single-threaded and multi-threaded (task-parallel) versions of each application so that we can also compare and contrast their performance.

The highlights of our results in this chapter include,

- We present a true data-parallel implementation of GapsMis, a tool for sequence alignment with bounded number of gaps. Further details are described in 6.5.1. The data-parallel implementation presented in this chapter decomposes the problem to achieve higher degree of parallelism on GPUs, compared to what was achieved in the original implementation [3]. We also present detailed and comprehensive analysis of our implementations, such that, when comparing the CPU vs. GPU implementations, we analyze the system as a whole and highlight several factors that could impede or improve performance. The CPU implementations include both single and multi-threaded versions for a thorough comparison.
- We study two dynamic programming formulations, *serial monadic* and *non-serial monadic*, and two *n-body* related algorithms and show that the GPU can achieve better energy efficiency, in addition to speedup, compared to the CPU. We also present performance data showing how these different algorithms can be affected by several factors involved with GPGPU applications, such as, device-host memory communication, multi-threading on GPU and memory usage on the GPU.
- In addition to comparing GPU performance to the CPU, we also present multi-GPU implementations. We show that, although certain algorithms perform particularly well on a single GPU, some do scale well when more GPUs are added to the system. This can further improve speedup and energy efficiency.

This chapter is organized as follows. We begin with a description of the theoretical analysis for parallel algorithms in Section 6.2. Then we present a detailed description of each application and corresponding algorithms in Sections 6.4, 6.5, 6.6 and 6.7. The methods employed in the empirical evaluation process including performance metrics and hardware configurations are discussed in Section 6.8. Our observations and results

are presented in Section 6.9. Final remarks, conclusion and possible future works are discussed in Section 6.10.

6.2 Theoretical analysis of parallel algorithms

Sequential algorithms. The time and space complexity of a sequential algorithm is typically expressed using the *big O notation*. The big O notation for a given algorithm characterizes that algorithm according to its time and space requirements depending on the size of the input. As an implication it can be used to classify algorithms according to their growth rate relative to their input size and provides an upper bound on the computational complexity for an algorithm. For example, consider a naive algorithm that finds a given element in a collection of n elements by checking each individual element in the collection. The time complexity for such an algorithm can be expressed as $O(n)$ because in the worst case, the algorithm needs to perform a comparison test on all n elements. In terms of space complexity, we can say that the algorithm requires constant space, written as $O(1)$. This is because, apart from the space required to store the input, the space required to evaluate the final result does not depend on the size of the input.

Parallel algorithms. In order to theoretically analyse our parallel solutions, we use the following terms described in literature such as [50, 52]. Consider a problem with an input size, n, then

Time complexity $t(n)$ This denotes the running time of the parallel algorithm.

Processor complexity $p(n)$ This is the number of processors used by the parallel algorithm.

Work complexity $w(n)$ This denotes the aggregate amount of work done by all processors involved in a parallel computation. In other words, it is the product $t(n) \cdot p(n)$.

According to [50], a parallel algorithm is considered to be *efficient* if it performs the same amount of work as the sequential algorithm to within a constant factor. Furthermore, a parallel algorithm is considered *optimal* if it is both efficient and $w(n)$ serializes into a sequential algorithm with time complexity $O(T(n))$.

Example 6.1.

Consider a problem X with an input size denoted by n. Assume $\mathcal{A}$ is the best available sequential algorithm that solves X in $O(n^2)$ time. On the other hand, $\mathcal{A}'$ is a parallel counterpart that solves the same problem in $O(n)$ time using n processors. $\mathcal{A}'$ is considered

to be optimal because it is both efficient and work complexity $w(n) = O(T(n)) = O(n^2)$. *On the other hand, another parallel algorithm that solves* X *in* $O(log\ n)$ *time using* n^2 *processors is not considered to be optimal because of its work complexity of* $O(n^2 log\ n)$.

6.3 Naming convention and notations

We introduce a simple naming convention that will be used to distinguish between the different implementations of an algorithm. The following symbols will be prefixed by the name of a given algorithm.

s This will be used to denote the sequential implementation of an algorithm.

t This will be used to denote the task-parallel implementation of an algorithm.

d This will be used to denote the data-parallel implementation of an algorithm.

This convention will be adopted throughout this chapter for all the algorithms discussed herein.

6.4 `DPS`: energy-aware scheduler for precedence-constrained jobs on parallel machines

The `DPS` tool is an application that encompasses the sequential implementation of the dynamic programming scheduler discussed in Chapter 3. In addition it also includes a task-parallel version for multi-core CPUs and data-parallel version for GPUs which is also robust enough to scale automatically to benefit from multiple GPU devices.

The dynamic programming formulation behind `DPS` can be described as *serial monadic*. This means that solving the subproblem at a given level in the algorithm only requires solution to the subproblem at the immediately preceding level. In this case, in order to compute schedule for job j, we only have to refer to the solution for job $j - 1$.

6.4.1 Sequential approach

The pseudo code in Algorithm 2 describes `DPS-`s, a sequential or *single-threaded* implementation of `DPS`. The formulation is a pseudo-polynomial time algorithm with a time complexity of $O(nm^2 \cdot d_{max})$, where n is the number of jobs, m is the number of machines and d_{max} is the largest deadline among all job deadlines. In order to optimize

Algorithm 2 DPS-s: single-threaded version of DPS

```
1:  procedure DPS-s(n, m, CE, CT, PE, PT)
    {For the first job only}
2:      j ← 0                                   ▷ j is set to zero to indicate the first job
3:      isFeasible ← False
4:      for i ← 0 to m do
5:          for t ← 0 to d_j do                 ▷ d_j is the deadline of job j
6:              tidx ← t − PT[i, j]
7:              if tidx ≥ 0 then
8:                  S[i, j, t] ← PE[i, j]
9:                  isFeasible ← True
10:             else
11:                 S[i, j, t] ← 0              ▷ Zero value indicates infeasibility
12:     if isFeasible == False then return S    ▷ Report infeasibility and terminate
13:     for j ← 1 to n do                       ▷ For the remaining jobs
14:         isFeasible ← False
15:         for i ← 0 to m do
16:             for t ← 0 to d_j do
17:                 minEnergy ← ∞
18:                 for k ← 0 to m do           ▷ Check all migration costs
19:                     tidx ← t − PT[i, j] − CT[k, i]
20:                     if tidx ≥ 0 then
21:                         tidx ← min{tidx, d_{j−1}}
22:                         if S[k, j − 1, tidx] > 0 then
23:                             e ← PT[i, j] + CE[k, i] + S[k, j − 1, tidx]
24:                             minEnergy ← min{e, minEnergy}
25:                 if minEnergy ≠ ∞ then
26:                     S[i, j, t] ← minEnergy  ▷ job j can finish at t − 1 on machine i
27:                     isFeasible ← True
28:                 else
29:                     S[i, j, t] ← 0
30:         if isFeasible == False then
31:             break                           ▷ Report infeasibility and terminate
        return S
```

the application, we exploit the fact that for each job we only need to store energy values for the time points $t \in \{0, \ldots, d_j\}$ on each machine. This greatly reduces the memory footprint of the application as well as the running time.

The algorithm consists of two parts. The first part, described by lines 2 to 11, computes all possible assignments for the first job. This is simply a case of checking for the completion time of the first job on each machine and assigning the processing energy value for each corresponding machine. The last part of the algorithm, lines 13 to 29, computes all possible assignments for the remaining jobs. Since the table of possible assignments for each job is truncated at the deadline of each corresponding job, it is important to validate the time value used to lookup entries from preceding jobs. This

is taken care of by limiting the this time value to the deadline of the preceding job, as shown in line 21.

6.4.2 Task-parallel approach

We present a very simple task-parallel formulation, DPS-t, that exploits the *serial monadic* nature of DPS-s in order to further optimize the running time of the application by taking advantage of multi-core processors. When we examine the structure of the algorithm from a very broad perspective, one can easily observe that the algorithm can be subdivided into n levels of computation, where n is the number of jobs. In order to compute energy entries for the j-th level the algorithm only needs to lookup values from the preceding level. This implies that computation for each job can be parallelized. When we consider the pseudo code given in Algorithm 2, there are two possible ways to achieve a task-parallel formulation.

Algorithm 3 DPS-t: multi-threaded version of DPS

```
1:  procedure DPS-t(n, m, CE, CT, PE, PT)
2:      p ← unique thread ID
    {For the first job only}
3:      j ← 0
4:      for i in M_p do        ▷ M_p is the number of machines allocated to CPU thread p
5:          for t ← 0 to d_j do
6:              tidx ← t − PT[i, j]
7:              if tidx ≥ 0 then
8:                  S[i, j, t] ← PE[i, j]
9:              else
10:                 S[i, j, t] ← 0
11:     for j ← 1 to n do                          ▷ For the remaining jobs
12:         for i in M_p do
13:             for t ← 0 to d_j do
14:                 minEnergy ← ∞
15:                 for k ← 0 to m do
16:                     tidx ← t − PT[i, j] − CT[k, i]
17:                     if tidx ≥ 0 then
18:                         tidx ← min{tidx, d_{j−1}}
19:                         if S[k, j − 1, tidx] > 0 then
20:                             e ← PT[i, j] + CE[k, i] + S[k, j − 1, tidx]
21:                             minEnergy ← min{e, minEnergy}
22:                 if minEnergy ≠ ∞ then
23:                     S[i, j, t] ← minEnergy
24:                 else
25:                     S[i, j, t] ← 0
        return S
```

The first can be achieved by sharing the workload in the loop defined on line 15 such that each CPU core takes care of one machine. This is possible because the computation for each machine is independent of the others. Therefore, we can exploit this fact and compute the entries for several machines concurrently for each job. The second possibility exploits the fact that for each machine, the computation of the energy entry at each point in time is independent of other time points. Therefore, we can achieve a similar task-parallel formulation by computing several entries concurrently.

The task-parallel formulation requires only a very slight modification of DPS-s as illustrated in Algorithm 3. In fact, we need only modify lines 15 and 4 of DPS-s in order to come up with a simple task-parallel formulation.

Analysis of DPS-t. Using the methodology described in Section 6.2, let us evaluate the theoretical performance of DPS-t. Firstly, assuming we use a total of λ processors or concurrent CPU threads, where $\lambda \leq m$, then $t(n, \lambda) = O\big(nm \cdot d_{max} \cdot \frac{m}{\lambda}\big)$. Next we consider the work complexity, $w(n) = O(nm \cdot d_{max}) \cdot O\big(\frac{m}{\lambda}\big)$, which evaluates to $O\Big(\frac{nm^2 \cdot d_{max}}{\lambda}\Big)$ where $p(n, \lambda) = \frac{m}{\lambda}$. Finally, we can conclude that DPS-t is an optimal parallel algorithm.

6.4.3 Data-parallel approach

In Section 6.4.2, we mentioned two possible ways of achieving a parallel formulation for DPS-s, namely, via parallelizing the for loops in lines 15 and 16 of Algorithm 2. DPS-t achieved this by performing the computation involved in the for loop at line 15 of Algorithm 2 concurrently across several CPU threads. Although the DPS-t algorithm provides a single *layer* of parallelism, it is possible to achieve an extra layer of parallelism by parallelizing both loops and the GPU device has the capability to meet this requirement. We present DPS-d, a data-parallel formulation for DPS-s that takes advantage of general purpose GPU devices.

Defining the NDRange for DPS-d. An important aspect of any data-parallel implementation on GPU devices is defining the *NDRange* (see Section 5.4.1.2). Recall that at the heart of modern general purpose GPUs lies an array of compute units, each of which can be further decomposed into arrays of processing elements. Each processing element is capable of executing an instance of a kernel. Our data-parallel formulation aims to fully utilize this structure in order to enable us simultaneously parallelize the loops mentioned earlier.

As part of the input parameters, the number of jobs, n, and machines, m, are specified. The first step in the process is to choose an appropriate work-group size, say W. Then

in order to run the iterations in the first for-loop simultaneously we define the size of our NDRange as $W \cdot m$. As a result each machine in the input is mapped directly to a work-group in our NDRange. Within the work-group, each work-item is then mapped to a time index.

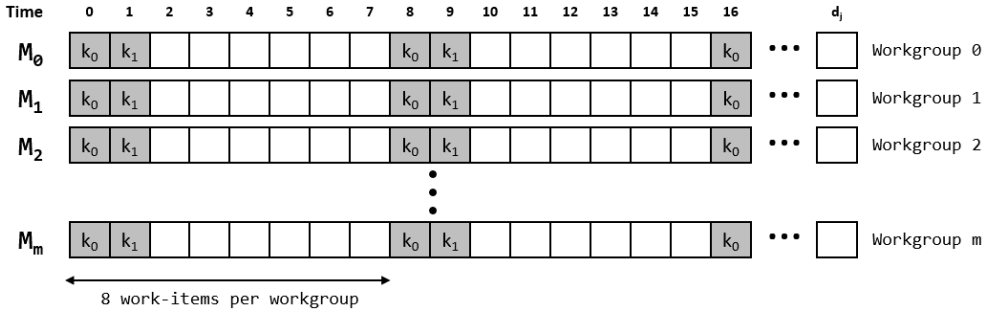


FIGURE 6.1: Illustration of how the NDRange is defined so that work-groups are mapped to machines in the input and a work-item maps to a time index.

This mapping is illustrated in Figure 6.1. In this figure, we assume that there are 8 work-items per work-group. The first work-item is labelled k_0 and it starts by computing the energy entry for time $t = 0$, which corresponds to its local ID of 0, in the first iteration. Then for subsequent iterations, it progresses by computing the energy entry for every W-th time index, in this case $\{t = 8, 16, \ldots\}$. For work-item with local ID 1, it computes for $t = \{1, 9, \ldots\}$. All work-items in the work-group follows this same pattern and continue as long as their current time index is no more than the deadline of the current job, d_j.

Memory management. This problem is very memory intensive in the sense that a large amount of memory is required to store the table $\mathcal{S}$. Therefore, depending on the size of the problem instance, we can quickly and easily use up the entire global memory on the GPU device. However, we can observe that in the structure of the dynamic programming, the computations for job j only depends on job $j - 1$, hence, it is not necessary to store $\mathcal{S}$ on the GPU device. Instead we employ a *double-buffering* technique. Assume that we have two buffers, A and B. Say the table for jobs J_{j-1} and J_j are stored in buffer A and B respectively. Before the computations for job J_{j+1} starts, we rotate the buffers so that the results for J_{j+1} are written to A while utilizing the values in B. This is a constant time operation and it is as easy as swapping kernel arguments.

The lookup tables $\mathcal{CE}$, $\mathcal{CT}$, $\mathcal{PE}$ and $\mathcal{PT}$ are stored in global memory of the GPU device. In order to better utilize the global memory bandwidth, we make use of vector data type so that tables $\mathcal{CE}$ and $\mathcal{CT}$ are merged into a single array and likewise, tables $\mathcal{PE}$ and $\mathcal{PT}$. Figure 6.2 illustrates an example for tables $\mathcal{CE}$ and $\mathcal{CT}$. Since it costs a considerable amount of clock cycles to process memory requests, using vectors helps us reduce the amount of access by two-fold.

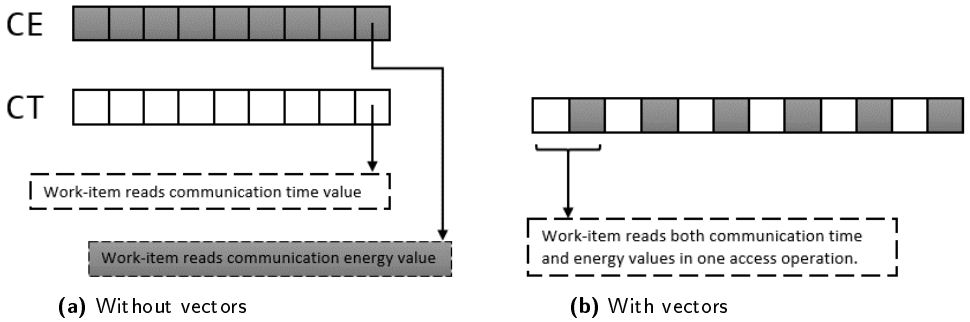


FIGURE 6.2: An example showing the advantage of using vector data type. (a) Without vectors, work-items need 4 memory accesses in order to retrieve values from tables. (b) Using vectors, two read operations are merged into a single read.

Analysis of `DPS-`d. Following the data-parallel formulation given above, we achieve an SIMD kernel with time complexity $O\left(\frac{d_{max}}{W} \cdot m\right)$ per work-item, where W is the size of the work-group. Depending on the GPU hardware limits, let us assume that the maximum number of work-groups that can execute concurrently is λ. Then the time complexity for the GPU kernel is $O\left(\frac{d_{max}}{W} \cdot m \cdot \frac{m}{\lambda}\right)$. Finally, since we execute for each job, we arrive at a time complexity $t(n, W, \lambda)$ of $O\left(n \cdot \frac{d_{max}}{W} \cdot m \cdot \frac{m}{\lambda}\right)$. Now consider a GPU device with sufficient amount of compute units and work-items per work-group. The parallel time complexity can be expressed as follows.

$$\lim_{(W,\lambda)\to(d_{max},m)} O\left(n \cdot \frac{d_{max}}{W} \cdot m \cdot \frac{m}{\lambda}\right) = O(nm)$$

Consequently, if we regard the total number of *"processors"* used by the GPU device to be $W \cdot \lambda$, then the work complexity evaluates to $O\left(nm^2 d_{max}\right)$. Hence, our data-parallel algorithm is also an optimal parallel algorithm.

6.5 `GapsMis`: a tool for sequence alignment with bounded number of gaps

6.5.1 Introduction

`GapsMis` is a tool for performing pairwise global and semi-global sequence alignment that allows for a bounded number of gaps. It is the work of Barton *et al.*[13] as an extension of a previously developed algorithm, `GapMis`, that performs pairwise sequence alignment with a single gap[3, 29]. In addition to the sequential version of the `GapMis` algorithm, the authors also developed parallel versions for CPUs (optimized further with SSE) and GPUs. However, the GPU version of the algorithm is implemented such that parallelism

is achieved through a task-parallel approach on the GPU. Results of their implementations demonstrate that `GapMis` is faster and more accurate compared to EMBOSS `needle` [93], which is a tool that implements the Needleman-Wunsch algorithm [79] for semi-global alignment.

6.5.2 Problem definition

In order to provide a definition of the problem, we need to present some definitions and describe some notations. Consider an alphabet Σ, which is a finite and non-empty set of elements known as *letters*. A finite sequence of letters formed from Σ is known as a *string*. A zero-letter or *empty string*, denoted by ε, is a string that does not contain any letters. The length of a string x, denoted by $|x|$, is the length of the sequence associated with string x. The letter at index i of x is denoted by $x[i]$, for all $1 \leq i \leq |x|$.

Consider a string y such that $y = uxv$. In this case, x is referred to as a substring of y. Furthermore, x is a prefix of y if $u = \varepsilon$. Similarly, x is the suffix of y if $v = \varepsilon$. We say a given pair of letters, (a, b), is an *aligned pair* where $(a, b) \in \Sigma \cup \{\varepsilon\} \times \Sigma \cup \{\varepsilon\} / \{\varepsilon, \varepsilon\}$.

Consider a *gap sequence* or *gap*, which is a finite non-empty maximal sequence of aligned pairs. A gap consists of one or more gap characters, where a gap character is denoted by *. An aligned pair of letters can be further described as consisting of at most a single gap character. Hence, $(a, b) \in \Sigma \cup \{*\} \times \Sigma \cup \{*\} \setminus \{*, *\}$.

In the alignment of two strings x and y, the pair of letters $(x[i], y[i])$ *matches* if $x[i] = y[i]$. A *substitution* is when $x[i]$ substitutes $y[i]$ and $x[i] \neq y[i]$ and $x[i], y[i] \neq *$. The letter $y[i]$ is said to *inserted* if it is not present in x, and $y[i]$ is said to *deleted* if it is present in y.

The quality of the alignment between two strings, x and y, for a pair of letters, $x[i]$, $y[i]$, can be measured using a score function, denoted by $\delta(x[i], y[i])$. The score function defines a value that describes the similarity between the pair of letters, and also including gap character *. Furthermore, the score of the alignment of two strings x and y, denoted by $\delta(x, y)$, is the sum of $\delta(x[i], y[i])$ over all i. Observe that we can simply count the number of matches between x and y if $\delta(x[i], y[i]) = 1$, for $x[i] = y[i]$, and $\delta(x[i], y[i]) = 0$, for $x[i] \neq y[i]$.

A *gap opening penalty* is the score assigned to the insertion of a gap. A *gap extension penalty* is the score assigned to the extension of an existing gap. Hence, the penalty for a gap of length $\ell > 0$ is defined as *gap opening penalty* $+ (\ell - 1) \times$ *gap extension penalty*.

```
G  C  G  A  T  T  C  A
|  |  -  -  -  -
G  C  A  T  C  A
```

FIGURE 6.3: Alignment with no gap.

```
G  C  G  A  T  T  C  A
|  |  |  |  |  -  -
G  C  *  A  T  C  A
```

FIGURE 6.4: Alignment with 1 gap.

```
G  C  G  A  T  T  C  A
|  |  |  |  |  |  |  |
G  C  *  A  *  T  C  A
```

FIGURE 6.5: Alignment with 2 gaps.

Definition 6.1. Given a text x of length n, a pattern y of length $m \leq n$, an integer ℓ, such that $0 \leq \ell \leq k$, the problem is to find a prefix of x, x', such that $\delta(x', y)$ is maximum and the corresponding alignment $z = z_0 g_0 z_1 g_1 \cdots g_{\beta-1} z_\beta$, is such that $\beta \leq \ell$.

Example 6.2.
Consider the text, `GCGATTCA`*, and pattern,* `GCATCA`*. The Figures 6.3, 6.4 and 6.5 show alignment results with 0, 1 and 2 gaps, respectively.*

6.5.3 Sequential `GapsMis` Algorithm

The pseudocode presented in Algorithm 4 describes the `GapMis` algorithm, which computes matrices G_1 and H_1 for the first gap, given strings x and y with lengths n and m respectively. Then the algorithm for `GapsMis` described in Algorithm 5 is applied to further compute the remaining gaps as required.

The dynamic programming formulation for `GapsMis` is *non-serial monadic*. This implies that the solution to the current subproblem depends on both a subset of the solution in the current subproblem, and, the solution to the previously computed subproblem.

6.5.4 Task-parallel approach

In a typical alignment tool, the application is usually able to perform tens of millions of alignment tasks in a single run. Hence, our application is designed with this in mind. In this section we describe a task-parallel implementation for `GapsMis`, which we will label as `GapsMis-`t.

Algorithm 4 `GapMis`: Computes matrices G_1 and H_1

```
    {Initialize matrices G_1 and H_1}
 1: procedure GapMis(x, n, y, m)
 2:     for i ← 0 to n do
 3:         G_1[i, 0] ← i
 4:         H_1[i, 0] ← i
 5:     for j ← 0 to m do
 6:         G_1[0, j] ← j
 7:         H_1[0, j] ← −j
    {Compute matrices G_1 and H_1}
 8:     for i ← 1 to n do
 9:         for j ← 1 to m do
10:             if i < j then
11:                 u ← G_1[i − 1, j − 1] + δ_E(x[i], y[j])
12:                 v ← G_1[i, i] + (j − i)
13:                 G[i, j] ← min{u, v}
14:                 if v < u then
15:                     H_1[i, j] ← i − j
16:                 else
17:                     H_1[i, j] ← 0
18:             if i > j then
19:                 u ← G_1[i − 1, j − 1] + δ_E(x[i], y[j])
20:                 v ← G_1[j, j] + (i − j)
21:                 G[i, j] ← min{u, v}
22:                 if v < u then
23:                     H_1[i, j] ← i − j
24:                 else
25:                     H_1[i, j] ← 0
26:             if i = j then
27:                 G_1[i, j] ← G_1[i − 1, j − 1] + δ_E(x[i], y[j])
28:                 H_1[i, j] ← 0
        return G_1 and H_1
```

Since the application is built to accomplish multiple sequence alignment tasks, it is quite intuitive to realize that this quickly becomes an embarrassingly parallel problem. The reason is that the alignment of each pair of sequences is a completely independent task on its own. Therefore, we can simply formulate a task-parallel algorithm where an alignment task is assigned to each available processing core or CPU thread. The only thing we need to consider is how utilize memory efficiently, because, given the scale of performing millions of alignment tasks, space requirement can easily explode if not well managed. Luckily, the solution is quite trivial.

For instance, consider a case of Q *query* sequences and T *target* sequences. Now suppose that we have to perform a total of $P = Q \cdot T$ alignment tasks. Let the length of the texts and patterns be n and m respectively, and the number of gaps to insert is ℓ. Suppose we

Algorithm 5 `GapsMis`: Computes matrices $G_{2,\cdots,l}$ and $H_{2,\cdots,\ell}$

```
    {Initialize matrices G_{2,...,ℓ} and H_{2,...,ℓ}}
 1: procedure GapsMis(x, n, y, m)
 2:     for s ← 2 to ℓ do
 3:         for i ← 0 to n do
 4:             G_1[i, 0] ← i
 5:             H_1[i, 0] ← i
 6:         for j ← 0 to m do
 7:             G_1[0, j] ← j
 8:             H_1[0, j] ← −j
    {Compute matrices G_{2,...,ℓ} and H_{2,...,ℓ}}
 9:     for s ← 2 to ℓ do
10:         minI[0 ··· m] ← 0
11:         for i ← 1 to n do
12:             minJ ← 0
13:             for j ← 1 to m do
14:                 newMinI ← 0
15:                 if G_{s−1}[i, j] < G_{s−1}[minI[j], j] then
16:                     minI[j] ← i
17:                     newMinI ← 1
18:                 u ← G_{s−1}[minI[j], j] + i − minI[j]
19:                 newMinJ ← 0
20:                 if G_{s−1}[i, j] < G_{s−1}[i, minJ] then
21:                     minJ ← j
22:                     newMinJ ← 1
23:                 v ← G_{s−1}[i, minJ] + j − minJ
24:                 w ← G_s[i − 1, j − 1] + δ_E(x[i], j[j])
25:                 G_s[i, j] ← min{u, v, w}
26:                 if u = min{u, v, w} and newMinI = 1 then
27:                     H_s[i, j] ← i − minI[j]
28:                 else
29:                     H_s[i, j] ← H_{s−1}[i, j]
30:                 if v = min{u, v, w} and newMinJ = 1 then
31:                     H_s[i, j] ← −(j − minJ)
32:                 else
33:                     H_s[i, j] ← H_{s−1}[i, j]
34:                 if w = min{u, v, w} then
35:                     H_s[i, j] ← 0
       return G_{1,...,ℓ} and H_{1,...,ℓ}
```

intend to store the entire matrices for each alignment in memory, then we end up with space complexity $O(2Pnm\ell)$.

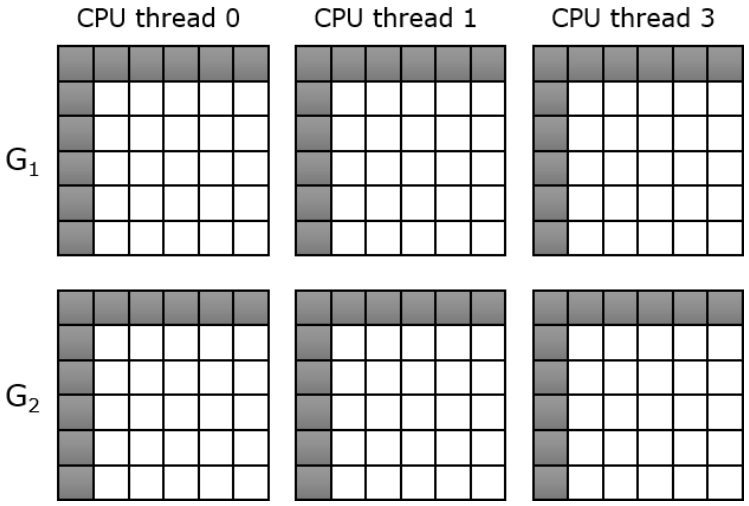


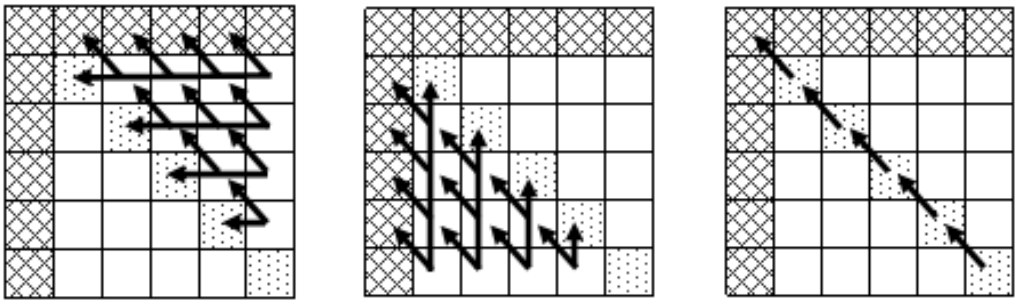
FIGURE 6.6: Block diagram showing the memory requirement for matrix G for each processor in `GapsMis-`t when executing for a 2-gap alignment.

On the other hand, we do not need to store the entire matrix instead, we could re-use allocated memory blocks. Figure 6.6 illustrates an example for $\ell = 2$. Since each alignment task requires just $O(2nm\ell)$ memory in total for matrices G and H, we simply allocate this amount of memory for each processor. The allocated space is re-used in each subsequent alignment task and this greatly reduces the memory footprint to $2\lambda nm\ell$, where λ is the number of processors.

Analysis of `GapsMis-`t. The approach in terms of task-parallelism in `GapsMis-`t is quite different in the sense that, instead of trying to parallelize the actual algorithm, we focus on the running time of the application as a whole. When we have multiple alignment tasks to perform, the tasks are distributed across the available processors. Hence each processor still has a running time $t(n, \lambda) = O(nm\ell)$.

6.5.5 Data-parallel approach

Data dependencies. In order to formulate a data-parallel algorithm for `GapsMis` it is imperative to understand the structure of the data dependency and how much communication is required between processing nodes. When we consider the general structure of `GapsMis`, the data dependency between cells of matrix G can be grouped into three different cases, (a), (b) and (c), for illustration purposes as shown in Figure 6.7. This grouping is directly related to the three cases found within the algorithm for determining value of $G[i, j]$.



(a) Case $i > j$ **(b)** Case $i < j$ **(c)** Case $i = j$

FIGURE 6.7: Illustration of the data dependencies among cells in the three cases within the `GapsMis` algorithm.

The algorithm progresses in a row-major order, however, the values in the first row and first column do not need to be computed but rather pre-initialized. In all three cases computation for a cell depends on data from the row that directly precedes the row in which the cell resides. In addition, the cases illustrated in Figures 6.7(a) and 6.7(b) each depends on an already computed cell on the diagonal. This extra data dependency in the first two cases means that we have a *non-serial monadic* DP formulation. In order to achieve maximum parallelism for each row, we must somehow overcome this data dependency.

In our implementation the solution is quite simple. For case (a), on each row, all work-items compute the value for the cell on the diagonal and this eliminates the need to rely on the work-item located on the diagonal. In addition to eliminating the data-dependency, it also eliminates writing code that will cause thread divergence which is advantageous with regards to performance of the kernel. Before progressing to the next row, the cell value on the diagonal is cached so that it can be re-used for case (b).

The final data dependency to consider is between the ℓ matrices in an ℓ-gap alignment where $\ell > 1$. For example, the current matrix being computed, G_i for $i \leq \ell$, only depends on the matrix that directly precedes it, i.e, G_{i-1}.

Defining the NDRange for `GapsMis`-d and memory management. Considering the scale of alignment tasks that `GapsMis`-d can perform it is not possible to store all the matrices in the GPU memory. Therefore, `GapsMis`-d is designed to be robust enough to handle this situation by executing the alignment tasks in *batches*. The size of a batch is simply the number of pairs of sequences the GPU device should process. The batching process is handled intelligently. The GPU device is first queried for the size of its global memory. Then using this information `GapsMis`-d computes the largest possible batch size such that all data required to execute a batch can fit into the GPU's global memory.

Once we have computed a batch size we can then determine the total number of work-items in our NDRange as $W \cdot P$, where W is the size of the work-group and P is the batch size. Within each work-group, all work-items are working on the same row of the matrix. However, the columns are distributed among the work-items. Figure 6.8 illustrates how alignment tasks are mapped onto the GPU device.

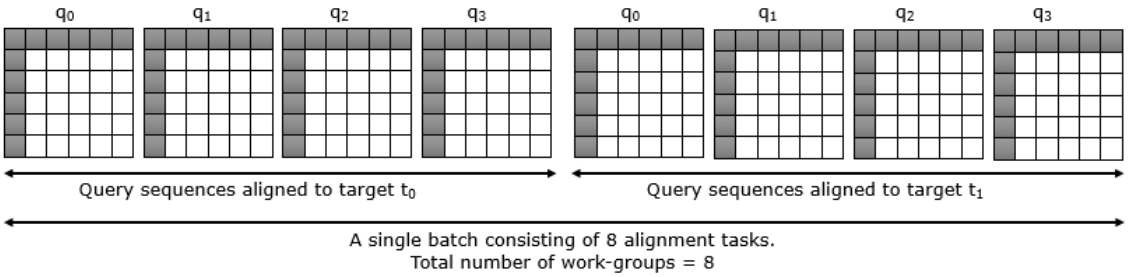


FIGURE 6.8: Illustration of how `GapsMis-`d maps alignment tasks to the GPU device across work-groups.

In addition we can further reduce the memory footprint by storing only the necessary data. Recall that the computation of a matrix depends on the matrix that directly precedes it for each gap. Hence, we do not need to store all the matrices on the GPU device. For this purpose, we make use of the double-buffering technique described earlier in Section 6.4.3.

Analysis of `GapsMis-`d. The GPU device not only enables us to achieve parallelism across the multiple alignment tasks, it also allows use to achieve a layer of parallelism within the algorithm. Since the algorithm cannot be parallelized row-wise, we have shown that is possible to achieve parallelism column-wise. Let us consider a single work-group with a total of W work-items. Each work-item will process $\frac{m}{W}$ columns, hence, the running time of our `GapsMis-`d kernel is $O\big(n \cdot \frac{m}{W} \cdot \ell\big)$. With processor complexity $p(n) = O(W)$ this implies that the work complexity $w(n, W) = O(nm\ell)$, therefore, `GapsMis-`d is an optimal parallel algorithm.

6.6 `Velvet`: Velocity-Verlet integrator

We present an application which we would like to refer to as `Velvet`, an implementation of the *Velocity-Verlet* algorithm [104] that is used for the numerical integration of Newton's equations of motion. It is commonly used in molecular dynamics to simulate motions and for calculating trajectories of interacting particles. This algorithm falls under the category of n-body method according to Berkeley's 13 dwarfs.

The Velocity-Verlet method of integration consists of three distinct computation phases - computation of forces, positions update and velocities update. The most compute-intensive of these three phases is the phase that computes the forces. In this phase, we must evaluate the pair-wise interactions between all particles in order to determine the forces related to that particle. This phase needs to be repeated twice in each iteration, hence, we can identify this phase as a *hot spot* in our application such that speeding up this phase directly improves the performance of the application as a whole.

6.6.1 Sequential approach

The initial step of the `Velvet` integrator considers the random placement of particles in 3-dimensional space. However, in order to reduce numerical inaccuracies, it is important that particles are not placed too close to each other or superimposed. For this purpose we project the initial, random positions of each particle inside *3-ball*. The screenshot shown in Figure 6.9 captures the placement of particles in a 3-ball. The other part of the initialization step is to randomly assign initial velocities to each particle.

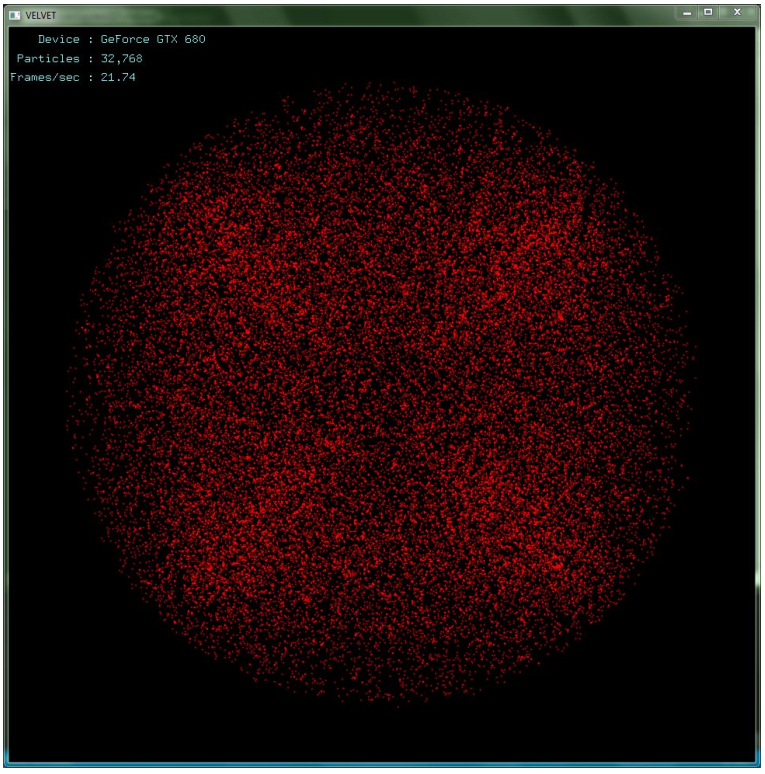


FIGURE 6.9: A screenshot of `Velvet` capturing the starting positions of 32,768 particles projected inside a 3-ball. This sample is running on an NVIDIA GTX 680 GPU.

Once we have finished with the initialization phase the first step in the integration loop is to compute the forces interacting with each particle. These forces are used to update the positions of the particles. Computation of forces requires pair-wise interaction for all

particles, however, this process can be optimized by taking advantage of *Newton's third law of motion*. According to the third law, the force exerted by a body on a second body is equal in magnitude and opposite in direction to the force exerted by the second body on the first body. The implication of this is that we can reduce the number of pair-wise interactions from $O(n^2)$ to $O\left(\frac{n(n-1)}{2}\right)$.

Algorithm 6 `Velvet-`s

1: **procedure** COMPUTE-FORCES(n, pos, $newForces$, $masses$)
2: **for** $p \leftarrow 1$ to $n-1$ **do**
3: **for** $q \leftarrow p+1$ to n **do**
4: $sqrDist \leftarrow \|\vec{pos}[q] - \vec{pos}[p]\|$
5: $gravity \leftarrow (masses[p] \cdot \dfrac{masses[q])}{(\sqrt{sqrDist} \cdot sqrDist)}$
6: $\vec{newForces}[p] \leftarrow \vec{newForces}[p] + gravity \cdot (\vec{pos}[q] - \vec{pos}[p])$
7: $\vec{newForces}[q] \leftarrow \vec{newForces}[q] - gravity \cdot (\vec{pos}[q] - \vec{pos}[p])$

8: **procedure** COMPUTE-POSITIONS(n, δ, pos, $newForces$, $oldForces$, vel, $masses$)
9: **for** $p \leftarrow 1$ to n **do**
10: $acc \leftarrow \delta * 0.5/masses[p]$ ▷ δ is the integration time step
11: $\vec{pos}[p] \leftarrow \vec{pos}[p] + \delta * (\vec{vel}[p] + acc * \vec{newForces}[p])$
12: $\vec{oldForces}[p] \leftarrow \vec{newForces}[p]$

13: **procedure** COMPUTE-VELOCITIES(n, δ, $newForces$, $oldForces$, vel, $masses$)
14: **for** $p \leftarrow 1$ to n **do**
15: $acc \leftarrow \delta * 0.5/masses[p]$
16: $\vec{vel}[p] \leftarrow \vec{vel}[p] + (\vec{newForces}[p] + \vec{oldForces}[p])$

17: **procedure** RUN-INTEGRATOR(n, δ, pos, $newForces$, $oldForces$, vel, $masses$)
18: COMPUTE-FORCES(n, pos, $newForces$, $masses$)
19: COMPUTE-POSITIONS(n, δ, pos, $newForces$, $oldForces$, vel, $masses$)
20: COMPUTE-FORCES(n, pos, $newForces$, $masses$)
21: COMPUTE-VELOCITIES(n, δ, $newForces$, $oldForces$, vel, $masses$)

Algorithm 6 is the pseudo code for `Velvet-`s showing all three procedures for calculating force, positions and velocities. Line 17 shows the order in which the different procedures are called during the integration process. The total running time for computing the velocities and positions is $O(2n)$, which brings the total running time for `Velvet-`s to $O(2(n(n-1)) + 2n) = O(2n^2)$.

6.6.2 Task-parallel approach

When we consider each of the three procedures involved in the integration process, we can easily observe that the Velocity-Verlet algorithm is embarrassingly parallel in nature. However, this is without taking advantage of Newton's third law of motion as doing this

will introduce data dependencies. In that case we simply parallelize the computation of all pair-wise interactions. With this in mind, achieving a task-parallel implementation is relatively straight-forward. The particles are distributed across the available processors during the execution of each procedure. So given, say λ processors, the parallel running $t(n)$ for the COMPUTE-FORCES procedure will be $O\left(\frac{n^2}{\lambda}\right)$. Together with the running time for COMPUTE-POSITIONS and COMPUTE-VELOCITIES procedures, the total parallel running time for an iteration of the RUN-INTEGRATOR procedure is $O\left(\frac{2n^2}{\lambda} + \frac{2n}{\lambda}\right)$.

Analysis of `Velvet-`t. The analysis of `Velvet-`s and `Velvet-`t is interesting because `Velvet-`s takes advantage of an optimization that we cannot apply in the implementation of `Velvet-`t. This optimization does not only reduce the running time but it also reduces the amount of work completed by `Velvet-`s. Recall that $T(n) = O\big(2n^2\big)$. For `Velvet-`t, $p(n) = O(\lambda)$ if λ processors are used, therefore, work complexity for `Velvet-`t is given by $w(n, \lambda) = O\big(2n^2 + 2n\big) > O(T(n))$. So, although we have an efficient parallel algorithm in `Velvet-`t, it is not optimal because $w(n, \lambda) > O(T(n))$.

6.6.3 Data-parallel approach

Defining the NDRange for `Velvet-`d. The data-parallel algorithm is similar to the task-parallel algorithm with respect to the layers of parallelism afforded to us by the algorithm. The data-parallel version does not also include the optimization applied in `Velvet-`s for the COMPUTE-FORCES procedure. However, the GPU device does offer more processors which could in turn increase the amount of speed-up achievable.

In defining the NDRange, we initialize a total of n work-items such that each individual work-item is mapped to an individual particle. In terms of achieving parallelism alone with the NDRange, the size of work-group is irrelevant in this case. The whole computation is split among three kernels, one for each procedure.

Memory management. In application such as this, the simulation process typically involves running the integrator procedure continuously in a loop until either a set number of iterations is reached or the simulator is terminated. The pleasing feature of this algorithm is that, depending on the number of particles being simulated, the memory requirement is relatively small. For `Velvet-`d running on a GPU device, this is good because we need to copy data back from the GPU device memory at the end of each iteration in order to update the simulation on screen with the latest positions. For this reason, we take advantage of the benefits of having pinned memory in order to fully utilize DMA capabilities of GPU-host communications. We only need to pin the memory block for storing positions data in the host since this is the only data required to update the simulation screen.

Analysis of `Velvet-`d. Since each work-item is responsible for computing the forces interacting with a single particle, the time complexity for each work-item is given by $t(n, \lambda) = 2n \cdot \frac{n}{\lambda}$, where $\lambda \leq n$ is the total number of work-items, computing the forces. Computing the positions and velocities are done in $O(1)$ by each work-item. Using λ work-items, then $w(n, \lambda) = O\left(\frac{2n^2+2n}{\lambda}\right)$. Hence, $w(n, \lambda) > O(T(n))$, which means that `Velvet-`d is also not an optimal parallel algorithm.

6.7 `FDGV`: Force-directed graph visualizer

Our fourth and final application, `FDGV`, is a tool that enables the visualization of graphs. It is based on the force-directed algorithm designed by Fruchterman and Reingold [32]. The idea takes inspiration from a physical system where each vertex is considered as a *ring* and edges are considered as *springs*. The rings are made to repel each other, however, a spring connecting a pair of rings will pull both rings together. It is a simple concept yet it is capable of producing decent visualizations of various types of graphs. Figure 6.10 shows a series of screenshots capturing various phases of the visualization process.

6.7.1 Sequential approach

The algorithm consists of three phases - computation of vertex displacements (repulsion), computation of edge displacements (attraction) and computation of vertex positions. Algorithm 7 lists the pseudo code for all phases in `FDGV-`s.

The Repulsion procedure is very similar to the Compute-Forces procedure discussed in Section 6.6, in terms of running time and computation pattern. Due to the sequential mode of calculation we are once again able to apply the Newton's third law of motion in the Repulsion procedure. This allows the sequential algorithm to compute the displacement on a pair of vertices in each iteration and without the need to iterate through all possible pairs. This means that the sequential algorithm can save some time during the repulsion phase.

Given a graph containing V vertices and E edges, the running time of the Repulsion procedure is $O\left(\frac{V(V-1)}{2}\right)$, and for Attraction and Compute-Positions, it is $O(E)$ and $O(V)$ respectively. Therefore, the time complexity of the sequential algorithm is given by $T(n) = O\left(\frac{V(V-1)}{2} + E + V\right)$.

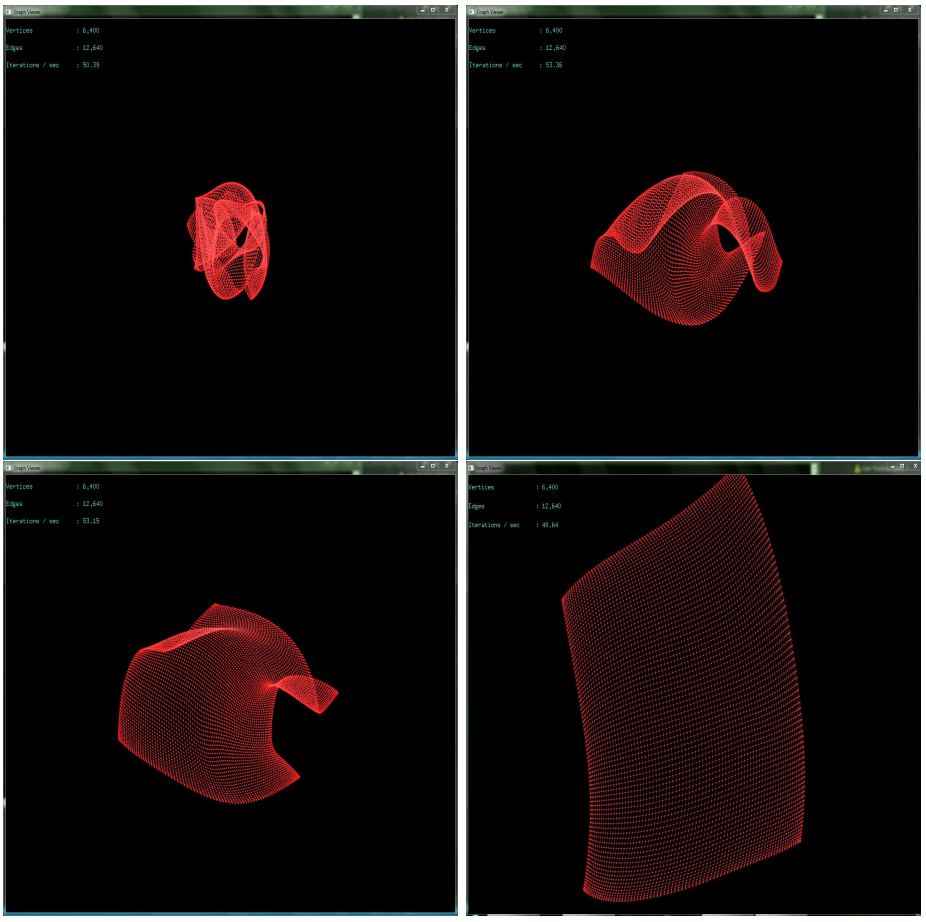


FIGURE 6.10: `FDGV` running a visualization of a graph with a grid-like structure consisting of 6,400 vertices and 12,640 edges.

6.7.2 Task-parallel approach

This algorithm is also embarrassingly parallel so to achieve parallelism in the computation phases one only need to distribute tasks across available processors. However, the optimization used in the REPULSION procedure of `FDGV-`s cannot be applied to its version in `FDGV-`t, hence, all possible pairs must be considered. However, when we consider the REPULSION procedure, we will notice that the positions of the vertices that constitute an edge has to be updated in order to reflect the displacement cause by attraction. The issue is that individual processors can each compute partial displacement values for a single vertex. In the end they will need to update the displacement value for that particular vertex. If all processors attempt to write their values concurrently, then we will end up with inaccurate values. To prevent this issue, known as *data-race*, we must guard the writes to memory such that access is allowed to only one processor at any given time. This can be accomplished via various means and constructs depending on the programming language and threading implementation being used.

Algorithm 7 `FDGV-`s

procedure REPULSION(V, pos, $disp$, k) ▷ k is spring constant
  for $p \leftarrow 1$ to $V - 1$ **do**
    for $q \leftarrow p + 1$ to V **do**
      $\alpha \leftarrow \vec{pos}[p] - \vec{pos}[q]$
      $\vec{disp}[p] \leftarrow \vec{disp}[p] + \frac{\vec{\alpha}}{\|\vec{\alpha}\|} \cdot \frac{k^2}{\|\vec{\alpha}\|}$ ▷ displacement for vertex p
      $\beta \leftarrow \vec{pos}[q] - \vec{pos}[p]$
      $\vec{disp}[q] \leftarrow \vec{disp}[q] + \frac{\vec{\beta}}{\|\vec{\beta}\|} \cdot \frac{k^2}{\|\vec{\beta}\|}$ ▷ displacement for vertex q

procedure ATTRACTION(E, $edges$, pos, $disp$, k)
  for $e \leftarrow 1$ to E **do**
    $\vec{\Delta} \leftarrow \vec{edges}[e].from - \vec{edges}[e].to$ ▷ Distance between vertices linked by e
    $\vec{d} \leftarrow \frac{\vec{\Delta}}{\|\vec{\Delta}\|} \cdot \frac{\|\vec{\Delta}\|^2}{k}$
    $disp[from] \leftarrow disp[from] - \vec{d}$ ▷ $from$ refers to index of source vertex
    $disp[to] \leftarrow disp[to] + \vec{d}$ ▷ to refers to index of destination vertex

procedure COMPUTE-POSITIONS(V, pos, $disp$, s) ▷ s is dampening factor
  for $v \leftarrow 1$ to V **do**
    $\vec{pos}[v] \leftarrow \vec{pos}[v] + \vec{disp}[v] \cdot \frac{s}{\|\vec{disp}[v]\|}$

Analysis of `FDGV-`t**.** For a graph containing V vertices and E edges, the parallel time complexity of `FDGV-`t is given by $t(n, \lambda) = O\left(\frac{V^2 + E + V}{\lambda}\right)$, where $p(n) = O(\lambda)$. The work complexity $w(n, \lambda) = O\big(V^2 + E + V\big)$ implies that `FDGV-`t is not an optimal parallel algorithm.

6.7.3 Data-parallel approach

In order to formulate a data-parallel algorithm, there are two main issues to consider. The first issue involves choosing a data structure to represent a graph in order to maximize performance of our GPU kernels. The second issue is about achieving a true data-parallel formulation for the ATTRACTION procedure.

Choosing a suitable data structure is important because the memory access patterns of the work-items is very crucial to achieving high performance on the GPU device as we discussed in Chapter 5, Section 5.2.3. For this purpose, a graph is stored as two simple arrays in memory, one to represent the vertices and the other for storing information about the edges. The edges are laid linearly in memory such that pairs of vertices that make up the edge appear contiguous. This is to enable us use vector data types in our kernels to access edge information for each work-item thereby making better use

of available memory bandwidth. This is similar to the technique used in Section 6.4.3. Once we have achieved this memory structure we can then be able to tackle the problem of synchronization in our kernel implementation of the ATTRACTION procedure.

In the `FDGV-`t we were able to serialize write accesses for the processors. However, we cannot afford to this in the GPU device given the way the architecture of the hardware because we will incur greater performance degradation. As a result, instead of updating the displacement for each node during the attraction phase, each work-item only computes a partial displacement and stores it in memory. Then during the phase for updating the positions, each work-item will accumulate the partial displacements for the vertex it is assigned to. Consequently, we eliminate the need for communication and synchronization at the expense of extra computation step because each work-item will need to iterate through the list of edges.

Defining the NDRange for `FDGV-`d. The total number of work-items required for computation in each phase of the algorithm varies. The kernels for REPULSION and COMPUTE-POSITIONS procedures require that a work-item is assigned to each vertex while for ATTRACTION procedure, a work-item is assigned to each edge of the graph. In terms of computational requirements the size of the work-group chosen is irrelevant in this case. Therefore, for a graph consisting of V vertices and E edges, V work-items are used for REPULSION and COMPUTE-POSITIONS while E work-items are used for ATTRACTION.

Memory management. The application can be executed with or without visualization and their is a slight difference between these options. Running the application without visualization makes it unnecessary to copy the positions of the vertices from the GPU memory to host memory after each iteration. However, if visualization is enabled we need to optimize the data transfer process between GPU device and host by using pre-pinned memory, similar to our discussion in Section 6.6.3. All other data will remain on the GPU device.

Analysis of `FDGV-`d. When we consider the kernel for COMPUTE-POSITIONS procedure, each work-item needs to iterate through all edges in order to accumulate the displacements due to edge connections for the vertex it is assigned. Using $v \leq V$ work-items, the parallel time complexity for this kernel is $O\left(\frac{V}{v}\right)$. The parallel time complexity for the REPULSION (using v work-items) and ATTRACTION procedures (using $e \leq E$ work-items) is $O\left(V \cdot \frac{V}{v}\right)$ and $O\left(\frac{E}{e}\right)$, respectively. Hence, work complexity $w(n, v, e) = O\left(\frac{V^2}{v} + \frac{E}{e} + \frac{VE}{v}\right)$. This means that the `FDGV-`d is doing more work compared to the other two versions and therefore not optimal. In Section 6.9 we shall investigate how the performance of all three implementations compare to each other given the varying amount of work being performed by each version.

6.8 Preliminary discussion

6.8.1 Evaluation model and performance metrics

In this section we describe the model that governs our empirical investigations. Developing a standard model will enable us make the evaluation process consistent and seamless across the different systems we employ in our experiments.

Our model consists of a *host system* or just *host*. A host refers to a standalone, heterogeneous computer system or machine. The main components of the host include a central processing unit (CPU), one or more compute devices (e.g GPU), *primary storage* (random-access memory or RAM) and *secondary storage* (hard disk). A *host program* executing on the host is controlled by the CPU and it refers to the implementation of a particular algorithm or application being considered. It also comprises of one or more *kernels*. There are two types of kernels, namely, *host kernel* and *compute kernel*. A host kernel is an implementation of the algorithm in question that executes on the CPU, whereas, a compute kernel executes on a compute device. In addition, the host program is also responsible for coordinating execution of compute kernels on the compute devices. The primary storage of the host is used as a *staging area* where input data to the host program resides, as well as the final output data. Based on this model we define the following performance metrics, which are inspired by similar metrics from the *HPEC Challenge* [59].

i **Latency.** This performance metric measures the amount of time that elapses between the start of a kernel execution and completion and is measured in seconds. It does not take into account time required to perform additional tasks such as transferring data from/to primary storage. In the case were there are multiple kernel invocations, the latency is simply the time taken to complete each execution on the compute device.

ii **Effective latency.** This measures the sum of latency incurred executing a kernel and time required for the final output data to become available in the primary storage. In most cases, there is no difference between the latency and effective latency for the host kernel since the CPU reads and writes to primary storage during execution.

iii **Communication latency.** The communication latency is associated with a compute device and it measures the total amount of time required to transfer data from/to primary storage.

iv **Throughput.** This performance metric measures the rate at which work is done during the execution of a kernel. The unit of measurement depends on the type of application. For instance, the throughput for the dynamic programming applications

measures the amount of cell updates per second (CUPS), and for the n-body methods, it measures the amount of floating-point operations per second (FLOPS). It is derived using the latency metric.

v **Effective throughput.** This can be considered as the rate at which work is done by the whole application and it is derived using the effective latency performance metric.

vi **Power.** The power metric measures the amount of electrical power, in Watts, required by a compute device or CPU to execute a kernel. It represents an isolated reading and does not take into account the total power consumption of the host.

vii **Energy.** Expressed in Watt-second, this performance metric measures the amount of power consumed during the execution of a kernel. Since it is isolated to within a CPU or compute device, it is derived using latency metric.

viii **Efficiency.** This performance metric measures the efficiency of a CPU or compute device with respect to the ratio of its throughput to power consumption.

6.8.2 Hardware and software specifications

The hardware setup for our experiments consists of a total of four host systems. The hardware specification for each host is listed in Table 6.1.

	CPU	GPU 1	GPU 2	RAM (GB)	HDD (TB)
AMDAHL	AMD FX-8350 8 cores @ 4.0 GHz	AMD HD 7970 32 CUs (2,048 PEs) @ 925 MHz	AMD HD 7970 32 CUs (2,048 PEs) @ 925 MHz	16	7
KEPLER	Intel i7-3930K 6 cores (12 w/ HT) @ 3.2 GHz	NVIDIA GTX 680 8 CUs (1,536 PEs) @ 1,006 MHz	NVIDIA GTX 680 8 CUs (1,536 PEs) @ 1,006 MHz	32	7
TESLA	AMD A10 5800K APU 4 cores @ 3.8 Ghz	NVIDIA GTX 650 2 CUs (384 PEs) @ 1,058 MHZ	NVIDIA GTX 650 2 CUs (384 PEs) @ 1,058 MHZ	16	6
VOLTA	AMD A10 5800K APU 4 cores @ 3.8 Ghz	AMD HD 7750 8 CUs (512 PEs) @ 820 MHz	AMD HD 7750 8 CUs (512 PEs) @ 800 MHz	16	6

TABLE 6.1: Table listing hardware specifications of all host systems used in the experiments.

The hosts can be classified as high end (`AMDAHL` and `KEPLER`) or mid range (`TESLA` and `VOLTA`). Although there are four CPUs available, we will only be making use of `Intel i7-3930K` CPU as a point of reference for our CPU implementations. This CPU consists of 6 physical cores and 12 logical cores with Intel's *Hyper-Threading Technology* [46] enabled. Also, some experiments might require the measurement of a metric that is only

system dependent, that is, metric depends on some capability of the host rather than the attached compute device. In this case, we will designate `KEPLER` as our reference machine for conducting such experiments. Notice the slight difference in the base frequencies of the GPUs in `VOLTA`. The reason is that the GPUs come from different vendors, in this case GPU 1 is from ASUS and GPU 2 is from MSI. All host systems are running Microsoft Windows 7 Enterprise 64-bit operating system with service pack 1.

The algorithms in our applications are implemented using the C++ programming language. Our task-parallel implementations are realized using the OpenMP API for parallel programming [84], which consists of several compiler directives that can be enabled certain programming languages including C++. The OpenCL API is used for all data-parallel implementations on the GPUs. OpenCL is selected because of its cross-platform support regardless of the hardware vendor. It is written in C, however, we use the C++ wrapper[105] provided by the Khronos Group.

In order to obtain profiling information related to power consumption for our CPU and GPU devices, we use *AIDA64 Extreme* software, version `4.60.3100`, from *FinalWire* [28]. Note that in our experiments, we will not include power and energy profiles for the `NVIDIA GTX 650` and `AMD HD 7750` GPUs. The reason is because these GPUs do not come equipped with hardware sensors that provide such information to a software program.

6.8.3 Input data for experiments

In this section, we outline the process of acquiring the input data used for each application. Our input data consists of both synthetic and real data or a combination of both. In some applications the size of the input data is limited by the capabilities of our hosts. Note that all input data to all the applications are stored in files in secondary storage and re-used on each of the host systems.

`DPS`. The input data for our `DPS` application is wholly synthetic. The number of machines is defined as an integer value from the set $\mathcal{M} = \{16, 32, 64, 128, 256\}$. The number of jobs is also an integer value defined in the set $\mathcal{J} = \{100, 200, 400, 800, 1600, 3200\}$.

The input data for the lookup tables $\mathcal{CE}$, $\mathcal{CT}$, $\mathcal{PE}$ and $\mathcal{PT}$ consist of integers generated uniformly at random based on the intervals shown in Table 6.2.

The $\mathcal{CE}$ and $\mathcal{CT}$ lookup tables are generated once for each $i \in \mathcal{M}$. This means that we have a total of 10 data instances, 5 each for $\mathcal{CE}$ and $\mathcal{CT}$. For example, instance $\mathcal{CE}_{16}$ consists of a lookup table with 16 rows and 16 columns corresponding to communication energy costs among 16 machines, in both directions.

	Minimum Value	Maximum Value
Communication time, $\mathcal{CT}$	1	2
Communication energy, $\mathcal{CE}$	2	10
Processing time, $\mathcal{PT}$	1	4
Processing energy, $\mathcal{PE}$	4	30

TABLE 6.2: Intervals used in data generation for DPS. These intervals are inclusive in the resulting data.

A similar approach is used for $\mathcal{PE}$ and $\mathcal{PT}$. Again, the lookup tables are generated once for each possible combination of $(i \in \mathcal{M}, j \in \mathcal{J})$. Therefore, for instance, $\mathcal{PE}_{16,100}$ is a 16x100 lookup table showing the processing energy costs of 100 jobs on 16 machines. Therefore, we have a total of 60 instances, 30 each for $\mathcal{PE}$ and $\mathcal{PT}$.

The job data instances is simply a list of integers representing the job deadlines. Each deadline is generated such that there is some *reasonable* gap between consecutive deadlines. To achieve this reasonable gap between deadlines, define three constants - *communication frame* (F_c), *processing frame* (F_p) and *window*. *Communication frame* is derived from the maximum possible communication time, in this case, it has a value of 2 (Table 6.2). Similarly, *processing frame* is derived from the maximum processing time plus 1, hence, it has a value of 5. The *window* is the summation of F_c and F_p. With these parameters, the deadline for a given job, J_j for $j > 0$, is generated uniformly at random in $\{I_{min}, I_{min} + window\}$, where $I_{min} = deadline_{j-1} + F_c$. For the first job, J_0, this interval is defined as $\{F_c + 2, window + 2\}$. Finally the deadline of the last job, J_n, is always defined as $deadline_{n-1}$ rounded up to the nearest integer divisible by 256. This is so that we can always have a value divisible by the number of work-items when computing a solution on the GPU since, for NVIDIA GPUs with OpenCL 1.0, the total number of work-items must be divisible by the work-group size.

`GapsMis.` The input data consists of a combination of synthetic (derived from processing real data) and real data obtained from *GenBank* FTP [78], which contains sequence databases in *ASN.1* format. The length of the target sequences are fixed at 250 while the length of the query sequences can be selected from four configurations - 75, 100, 150 and 200. In order to get the desired length for our input sequences, real sequences sequences are sampled and processed. For instance, in order to get an input sequence of 75 characters, a real sequence containing more than 75 characters is chosen uniformly at random from the database and characters are deleted from random positions until we are left with a sequence with 75 characters. Some real input data contain thousands of characters so it is possible to re-use the same sequence to generate multiple synthetic input sequences. The information listed in Table 6.3 shows the details of the exact

databases used. The databases selected, when combined, provides us with enough data to generate our input data.

Name of database file	Number of sequences	Length of longest sequence	Length of shortest sequence
gbbct10.fsa_aa	151,777	16,990	100
gbbct11.fsa_aa	172,113	14,474	100
gbbct24.fsa_aa	164,027	13,362	100

TABLE 6.3: Information for GenBank databases used to generate input sequences for `GapsMis`.

The substitution matrix used is the BLOSUM62 matrix [77] for aligning protein sequences. A *gap open penalty* and *gap extension penalty* of 10.0 and 0.5, respectively, were used for all executions of the experiments. Finally, the experiments were conducted for an alignment that allowed for 2 gaps and then repeated for 3 gaps.

`Velvet.` The input data is a collection of particles. Each particle is characterized by a mass, position and velocity. The position and velocity properties are 3-dimensional vector quantities while the mass is a scalar quantity. The values that make up the velocity vector are generated using a uniform distribution based on minimum and maximum values. For the experiments, we use a minimum value of -10.00 and maximum of 10.00 for the uniform distribution. All values are real numbers.

The starting positions of the particles are obtained using a method for generating uniformly distributed random points within an n-ball [70]. To achieve this, for each particle, we generate a 3-dimensional vector consisting of real numbers in $(0, 1)$. Next we calculate the *radius* for the position vector. Suppose the position is given by the vector $\vec{p} = (x, y, z)$, the radius, r, is computed as $r = \sqrt{x^2 + y^2 + z^2}$. The position on the surface of the n-ball is given by $\frac{1}{r} \cdot \vec{p}$. The final position of a particle within the n-ball is given by $u^{\frac{1}{n}} \cdot \vec{p}$. In our simulations $n = 3$ which gives an ordinary ball.

The problem sizes include ensembles of 2048, 4096, 8192, 16384, 32768 and 65536 particles and the simulation is repeated for each size. For all devices, the integrator is set to run for 10 iterations in total.

`FDGV.` The input data for the graphs consists of a combination of synthetic and real data. The real data graphs are available from *Stanford Network Analysis Platform* (SNAP) [63]. The details of the exact datasets used are listed in Table 6.4

Name of dataset file	ID	Vertices	Edges
p2p-Gnutella08	GNUT 1	6,301	20,777
p2p-Gnutella24	GNUT 2	26,518	65,369

TABLE 6.4: Details of the real graph data obtained from SNAP.

Our synthetic graph data consists of three types of graphs which are complete graphs, grid graphs and trees. The complete graphs and grid graphs were generated by the application while the `igraph`[44] tool was used to generate the tree graphs. The properties of the graphs used as inputs are listed in Table 6.5.

Type of graph	ID	Vertices	Edges
Complete	COMP 1	100	4,950
	COMP 2	200	19,900
	COMP 3	400	79,800
Grid	GRID 1	10,000	19,800
	GRID 2	20,000	39,700
	GRID 3	40,000	79,500
Tree	TREE 1	10,000	9,999
	TREE 2	20,000	19,999
	TREE 3	40,000	39,999

TABLE 6.5: Details of the synthetic graph data generated for `FDGV` application.

The three graph types are chosen to represent three distinct cases with respect to the number of vertices and edges that make up each graph type. The complete graph is characterized as having an edge count that outnumbers the number of vertices in the graph. The number of vertices and edges in a tree graph is almost equal. And for the grid graph, the number of edges is almost twice as much as the number of vertices. The combination of the real and synthetic graph data provides us with a considerable amount of data variety for testing out our `FDGV` application. For our runs the application is set to complete 10 iterations of the graph layout process.

6.8.4 Aims of experiments conducted

Device-Host communication overheads. In designing a data-parallel algorithm that makes use of a GPU device it is quite possible to under-estimate the impact of communication overhead between the host and GPU device. This is particularly important when working with discrete GPUs, that is, a GPU that exists as a peripheral device within the host. All the GPUs used in our experiments are discrete GPUs. A limiting factor with discrete GPUs is the need for explicit memory allocation and transfer between host and device and this is due to the fact that they do not have access to the host

system's virtual memory. Apart from concerns with memory transactions, another factor to consider is the overhead associated with making API calls, for instance, launching a kernel or initiating a copy operation. Sometimes the time taken to make these API calls might dominate the overall running time of the application. These are the kind of issues we need to take into account when designing a heterogeneous application. However, there are cases where the nature of an algorithm makes it difficult to avoid these drawbacks.

For instance, a dynamic programming algorithm is usually associated with high memory requirement in order to compute one or more tables. Since GPUs usually have limited memory capacities, computation is usually divided into portions and data is batched in order to fit into the GPU device. Another example is an algorithm that requires *global synchronization* points during computation in order to synchronize data across all work-items involved. In these two cases we cannot avoid interacting with the host during computation.

In our experiments, we investigate the effect how these kinds of overhead can really impact on the overall performance of an application. Such impact varies greatly between applications depending on the structure of the underlying algorithm. We expect to demonstrate the impact of device-host communication overhead, most especially, in our `GapsMis`.

Importance of work-group size. The size of a work-group in a data-parallel application plays an important role in the performance of the application on the target GPU hardware. Depending on the architecture, a certain work-group size is often generally recommended. However, the performance of an algorithm could vary greatly depending on the size of the work-group chosen. To maximize performance of the GPU device, it must be a multiple of the number of threads the hardware scheduler can manage concurrently (see Section 5.2.3). If a work-group is too small, there might not be enough work-items to schedule in order to hide latency. On the other hand, it is too large, kernel occupancy will degrade as a result of the GPU hardware not being able to reach maximum number of work-items that can execute concurrently.

In our experiments, we investigate how various work-group sizes can affect the performance of our applications. We hope to demonstrate this using our `DPS` and `Velvet` applications.

Effects of using local memory. Depending on the GPU hardware, knowing when to use and when to use local memory can be a challenge because it is difficult to predict exactly how a particular algorithm will perform on a particular GPU. It is often beneficial

to use local memory in cases where there is a high degree of data re-use. Since it is quicker for work-items to access data stored in local memory than global memory, it helps to save some valuable time by avoiding repeated access to global memory. However, a kernel that requires a considerable amount of local memory storage often leads to degraded application performance. This is because the number of work-groups that can be scheduled concurrently depends on the amount of available local memory, and then the amount specified by the hardware limit.

We shall investigate how local memory can affect application performance on a GPU through our `DPS`, `Velvet` and `FDGV` applications. In these applications, it is possible to implement decide whether or not to use local memory so we implement a kernel for each version.

Benefits of pre-pinned memory and DMA. This is particularly related to applications with a reasonably low amount of memory footprint, like `Velvet` and `FDGV`. These are applications are characterized by the frequent need to update some data in host memory in order to accomplish a purpose. In the case of `Velvet` and `FDGV`, the display needs to be refreshed with the latest positions during the simulations. However, in order to maximize the benefits of pre-pinned memory, the host must support DMA so that the CPU is allowed to do other valuable work while data transfers are carried out in the background. We hope to investigate if we can improve the performance of such applications and if so by how much.

Application scaling using multiple GPUs. An application that can scale with the addition of more GPU devices usually stands a good chance to benefit from the potential performance boost. Depending on the structure of underlying algorithm, introducing more GPU devices may introduce additional complexity into the application and this could lead to a counter-productive scenario. For example, the nature of the `Velvet` application will require constant communication between the GPU devices involved and the allocation of additional memory for synchronizing data. These reasons coupled with the complexity introduced into the code means that the `Velvet` will not scale efficiently across multiple GPU devices.

However, we can evaluate the performance of other applications to determine how well they scale by comparing their performance on a single GPU with their performance on dual GPU devices.

Comparison of CPU vs. GPU performance. The aim of this experiment is to evaluate the amount of gain in performance that is achievable when GPUs are used to

accelerate an application. We compare the amount of speed-up, in terms of time, that a GPU can offer for each application as well as the throughput too. In addition, we would also evaluate the energy efficiency of these devices by profiling the power consumption for each device during execution. This is to enable us investigate how efficient a device is and not just focusing on the power consumption figures because a device with a low power rating might not necessarily be efficient in performing a particular task.

6.9 Discussion of experiment results

We would like to state that the results presented in this section do not necessarily depict the overall performance of the GPU devices. This is particularly true for the NVIDIA GPUs which we believe could perform better if NVIDIA's parallel computing platform, known as CUDA (*Compute Unified Device Architecture*), was used since NVIDIA stopped OpenCL support as early as OpenCL version 1.0.

6.9.1 Results on device-host communication overheads

Among the performance metrics discussed earlier in Section 6.8.1, the one that we need to focus on that captures the device-host communication overhead is the effective latency. All our applications are evaluated for this purpose.

`DPS-`d. Since all the data required to compute the dynamic programming table for each can conveniently fit into the GPU memory, we expect that the difference between the latency and effective latency will not be too large. This is primarily due to the relatively small amount of data that is transferred from the GPU device to the host in each iteration. This is, in fact, the case for all our host systems and GPU devices. Figure 6.11(a) and Figure 6.11(b) show the results comparing latency and effective latency for simulation with 1,600 jobs and 3,200 jobs, respectively. As expected, we observe that the effective latency across all GPU devices is up to 18% more than the recorded latency. For instance, for the problem with 256 machines, this translates to roughly 1 second extra for 1,600 jobs and 3 seconds extra for 3,200 jobs. Since we are able to store the entire table for our problem instances in primary storage, we can exclude the time required for back-tracing from the effective latency. In addition, `DPS-`s will need to perform the back-tracing step separately after the whole computation step is complete, hence, the time required for back-tracing is not included in the running time of `DPS-`s.

On the other hand, when we consider performance from throughput point of view, we observe that the effective throughput (considering total running time on host and GPU)

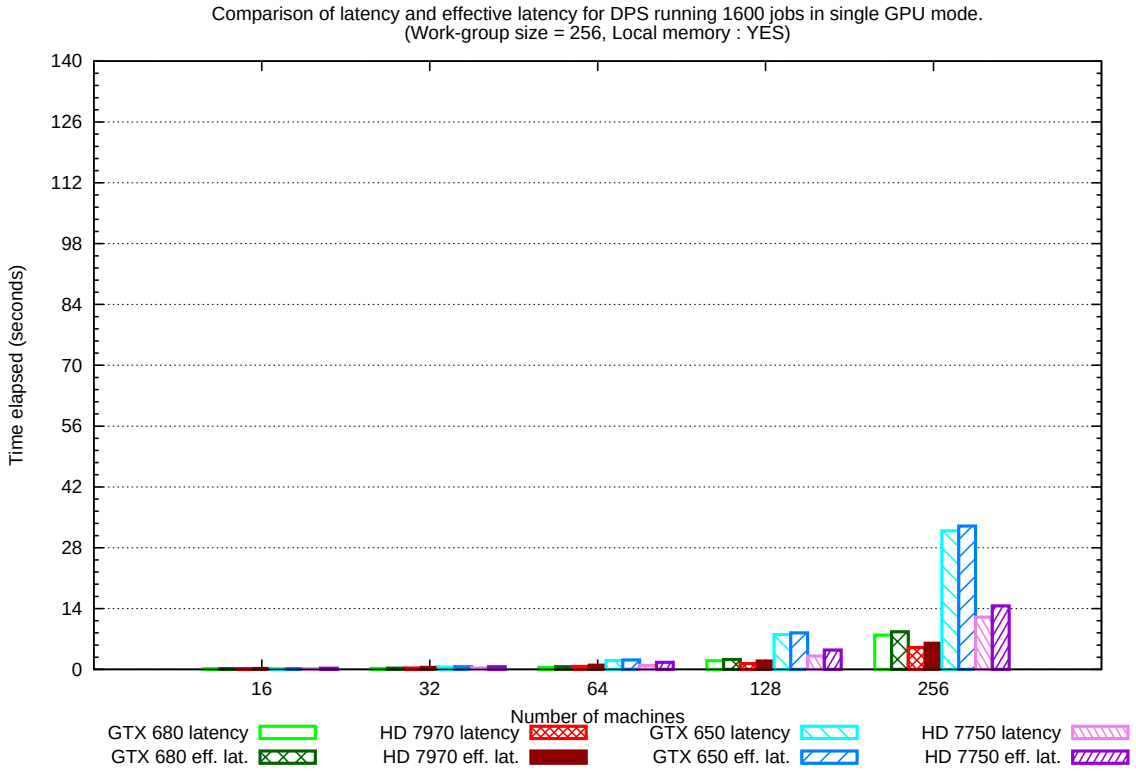
is around 16% lower than the throughput of the compute kernel on the GPU. This translates to a difference of around 25 MCUPS for problem size with 16 machines up to 100 MCUPS for problem size with 256 machines. These results are shown in Figures 6.12(a) and 6.12(b).

We can conclude that, from a practical perspective, `DPS-`d can benefit from using the GPU without suffering significant performance hit that might arise from communication overheads with the host system.

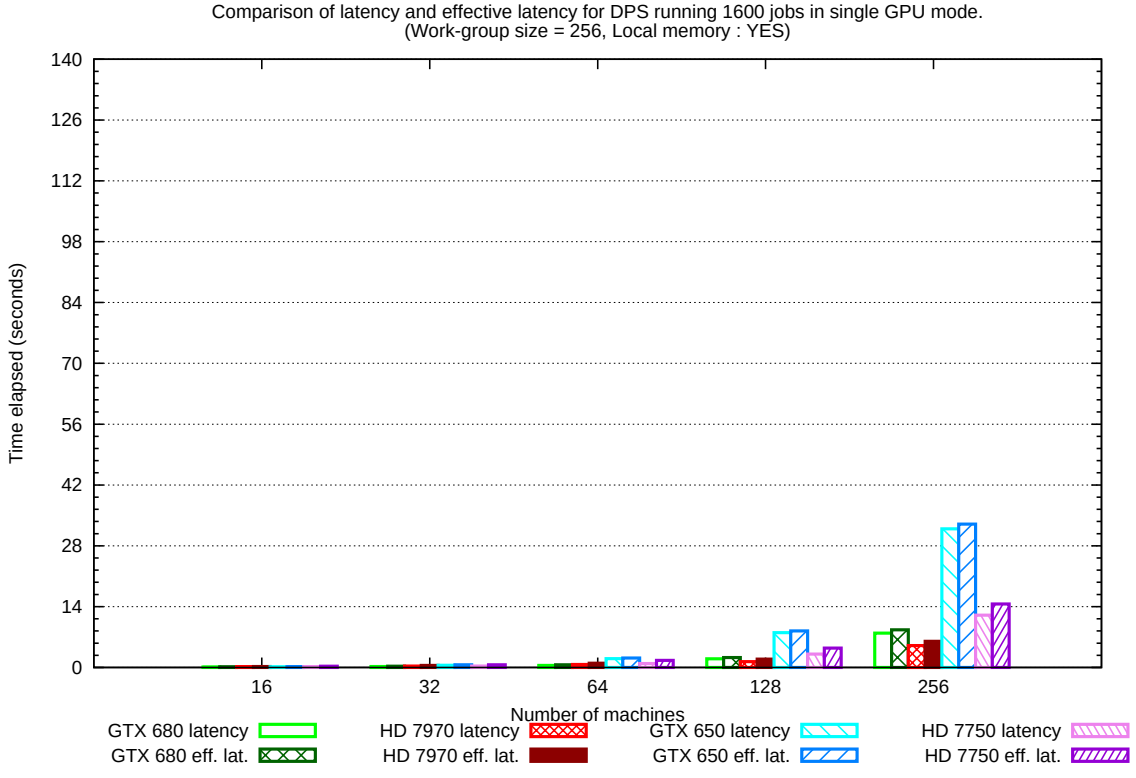
`GapsMis-`d. Using the sequential implementation, `GapsMis-`s, as a point of reference, the effective latency for `GapsMis-`d includes the time required for the CPU to perform the back-tracing step on each batch of tables returned by the GPU device in order to determine the final results. This is because `GapsMis-`s interweaves both computation and back-tracing during execution. As a result of this unavoidable cooperation between the host and compute device, we should expect a significantly large amount of overhead. Figures 6.13 and 6.14 shows the comparison between the latency and effective latency for the execution of `GapsMis-`d allowing 2 and 3 gaps, respectively.

The impact of device-host communication in this algorithm is very significant. For instance, when we look at the results for the high-end AMD HD 7970 GPU for simulation with more than 5 million alignment tasks, only 20% of the total execution time is spent on the GPU compute kernel. To be more precise, considering Figure 6.13, the latency for the AMD HD 7970 GPU is around 407 seconds while the effective latency is around 1614 seconds. A similar trend can also be observed across other GPU devices used in the simulation, that is, only 20% to 30% of the total execution time is spent on the GPU.

This is due to the batching process and back-tracing work that is offloaded to the CPU. This demonstrates the fact that, although some compute intensive algorithms could benefit from using the GPU device, current limitations in hardware architecture could introduce extra complexities that must be taken into account when designing a data-parallel application for the GPU device. In our case, the architectural limitation is due to the fact that we need to perform explicit memory transactions between host and compute device. However, judging from these results, there is room for improvement especially as GPU devices continue to evolve along with current computer system architecture.

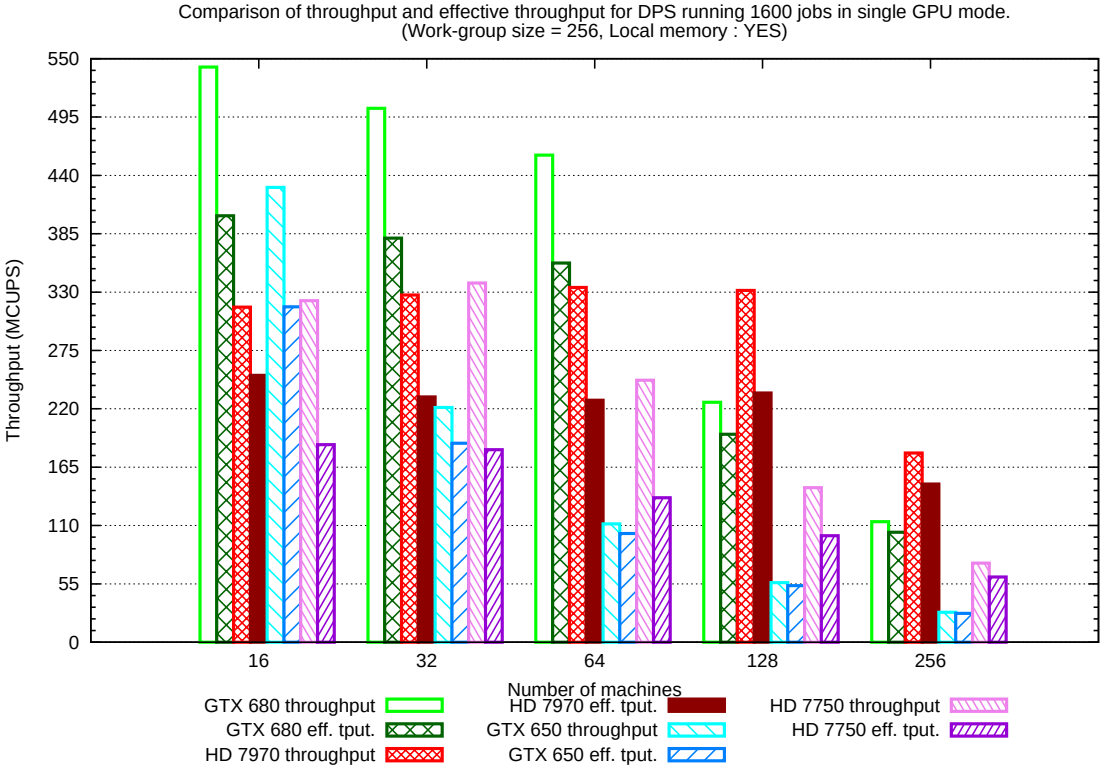


(a) 1600 jobs (Time)

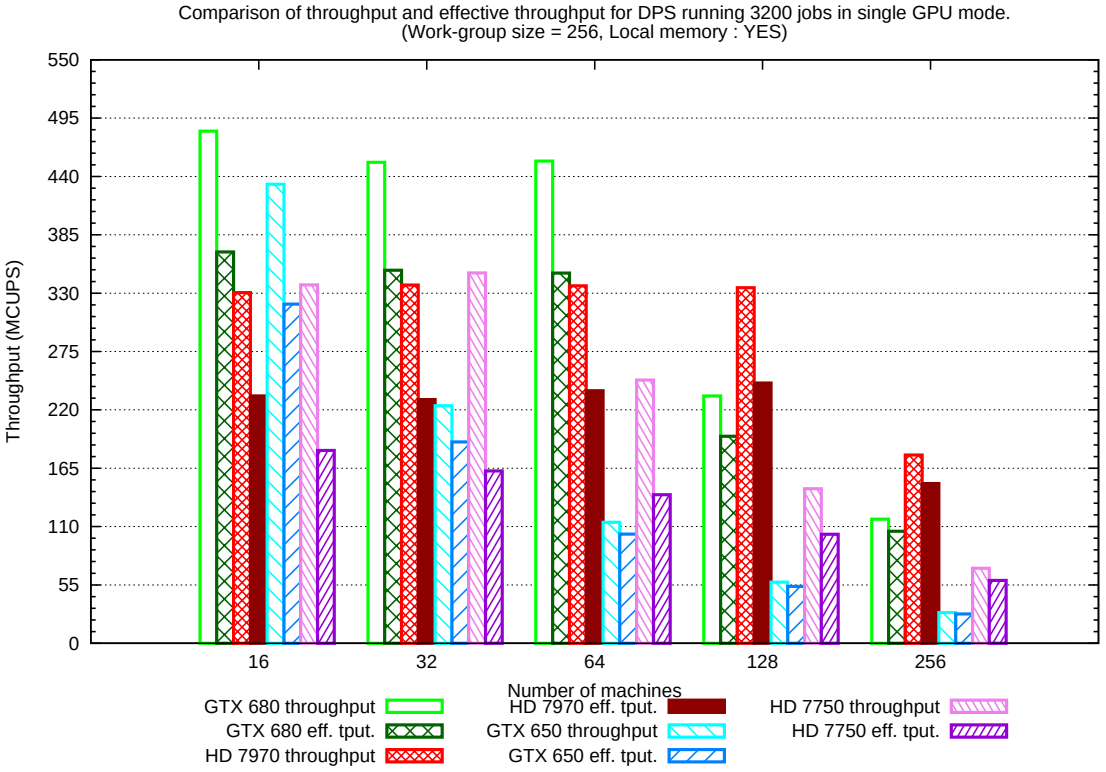


(b) 3200 jobs (Time)

FIGURE 6.11: Comparison of latency vs. effective latency for single GPU performance.

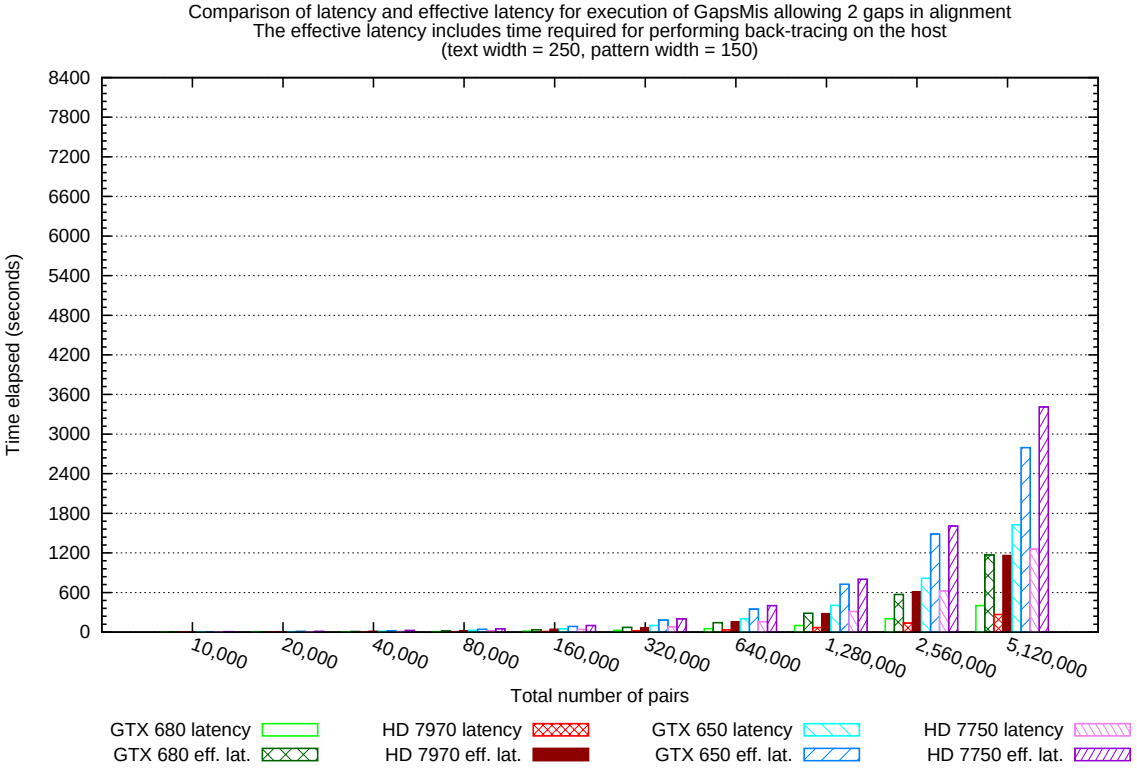


(a) 1600 jobs (Throughput)

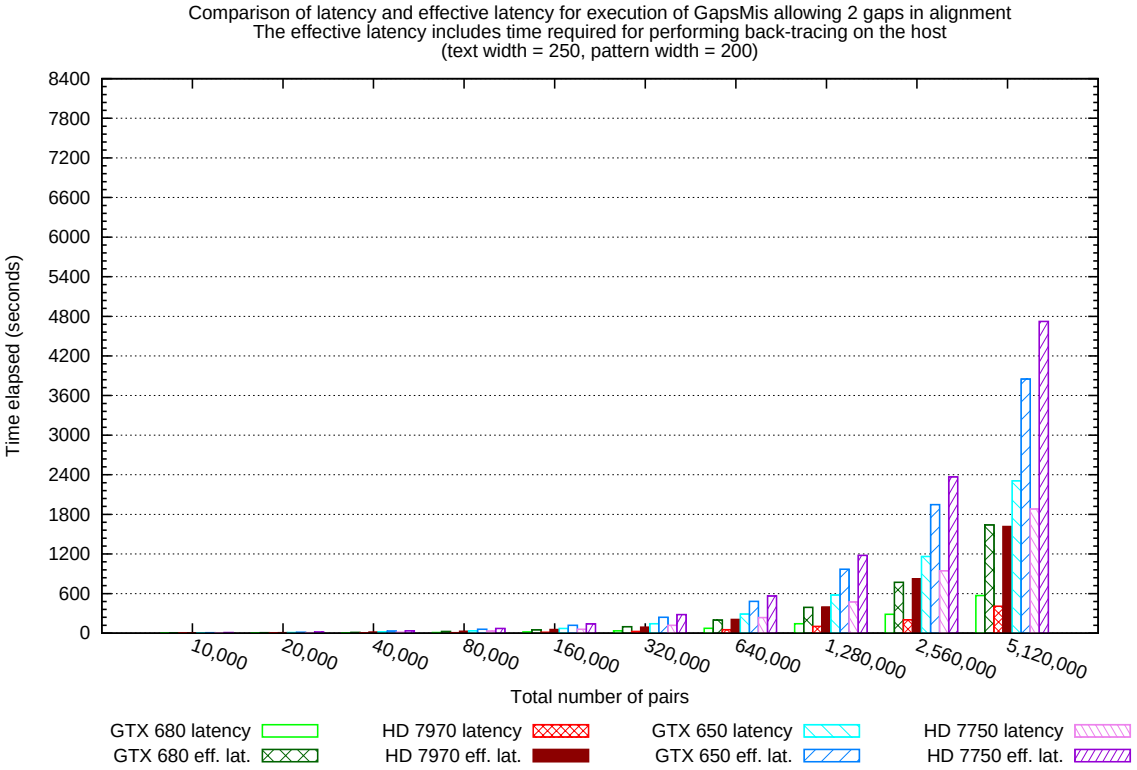


(b) 3200 jobs (Throughput)

FIGURE 6.12: Comparison of throughput vs. effective throughput for single GPU performance.

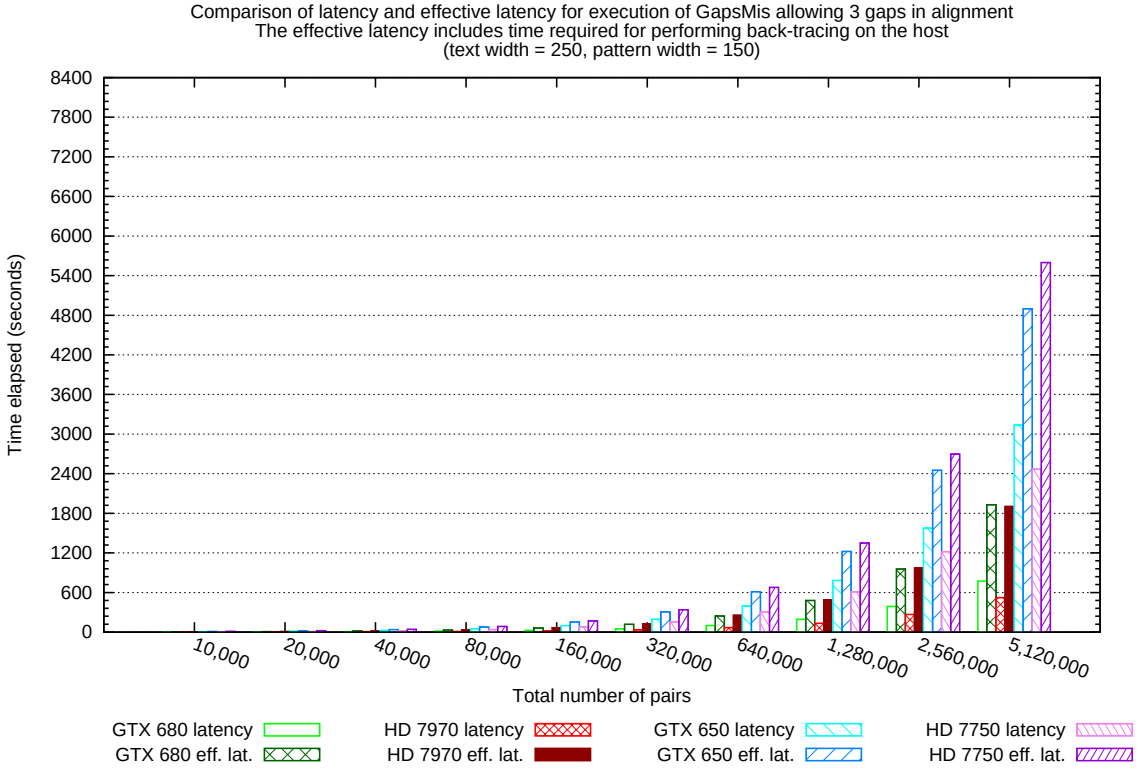


(a) 250×150, 2 gaps

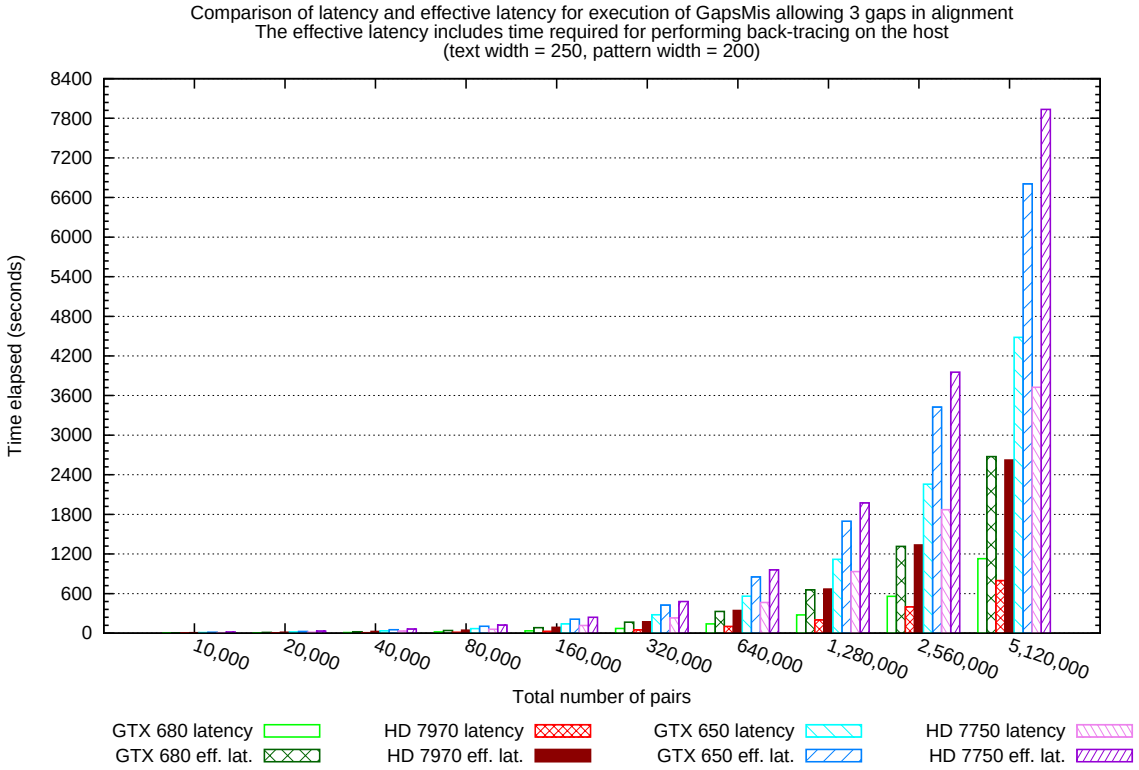


(b) 250×200, 2 gaps

FIGURE 6.13: Comparison of the latency and effective latency for `GapsMis-`d running on a single GPU device performing alignments allowing 2 gaps.



(a) 250×150, 3 gaps



(b) 250×200, 3 gaps

FIGURE 6.14: Comparison of the latency and effective latency for `GapsMis-`d running on a single GPU device performing alignments allowing 3 gaps.

`Velvet-d`. The `Velvet-d` application is well-suited for the GPU device mainly because of its minimal interaction with the host. As a result of this we do not expect any significant overhead in terms of time required to copy the positions of the particles from the GPU device to the host for display purposes. Our expectations are realized by the results shown in Figure 6.15.

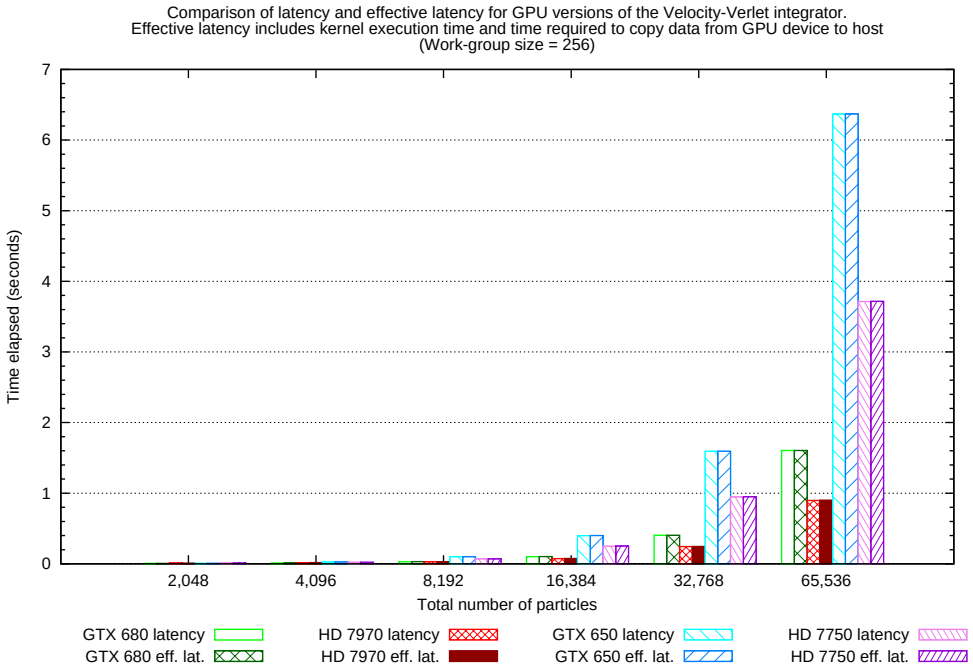
There is almost no difference between the latency and effective latency figures. For instance, the latency on the AMD HD 7970 GPU is around 0.8977 seconds (Figure 6.15(a)) while the effective latency is around 0.9004 seconds. Overall, the maximum difference observed between latency and effective latency across all GPUs is around 0.003 seconds.

Since this application is characterized more for its high compute requirements than its data requirements, there is almost no additional costs in terms of time associated with copying the positions data from the compute device to host after all iterations have been completed. This result demonstrates an almost best case scenario of an application that can leverage the compute resources of a GPU device without incurring excessive amount of overhead, which is particularly the case for applications classed as n-body methods.

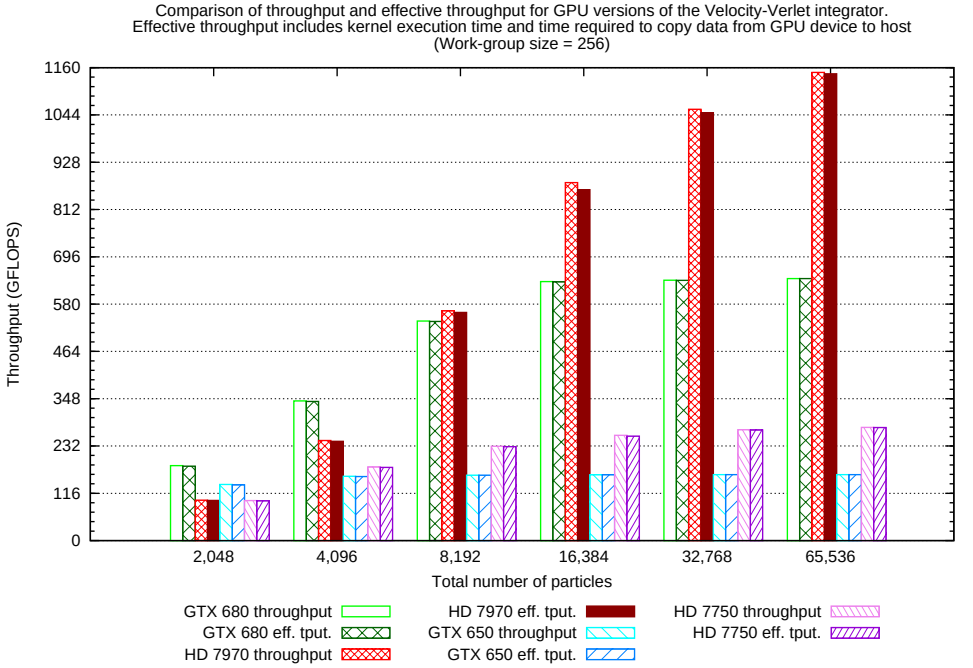
`FDGV-d`. As an n-body method, similar to `Velvet-d`, we do not expect to see too much difference between the latency and effective latency for this application. Recall that the only data we need to copy from the device after all iterations are complete consists of the positions of the vertices in the graph. Figures 6.16 to 6.19 show the results for the four categories of graphs used in our simulation.

If we consider the simulation with the largest input size, grid graph with 40,000 vertices and 79,500 edges (Figure 6.18), effective latency for the AMD HD 7970 GPU is around 0.5332, of which the latency is around 0.5319. Across all GPU devices, the maximum difference observed between effective latency and latency for this same problem size is observed to be around 0.003 seconds. The latency and effective latency are almost the same due to very little overhead in copying position data of vertices back from the GPU. Given the practical nature of `FDGV-d` as a tool that can be integrated into the pipeline of an application that visualizes graphs and other similar networks, we should expect huge performance boosts with data-parallel implementation since it is mainly a compute intensive application.

Conclusion. From the results that we have seen so far in this experiment, we can conclude that, although the performance of the kernel on the compute device is important, the communication overhead between host and compute device must be considered as equally important as this dictates the overall performance of an application. This fact



(a) Latency and Effective latency results.



(b) Throughput and Effective throughput results.

FIGURE 6.15: Results comparing the latency and effective latency of executing `Velvet-d` for all problem sizes (Figure 6.15(a)). Resulting throughput performance is shown in Figure 6.15(b). Here, due to the small data to computation ratio, the communication time between host and compute device is marginal.

is clearly demonstrated by `GapsMis-d` where the communication overhead dominates the overall execution time of the application.

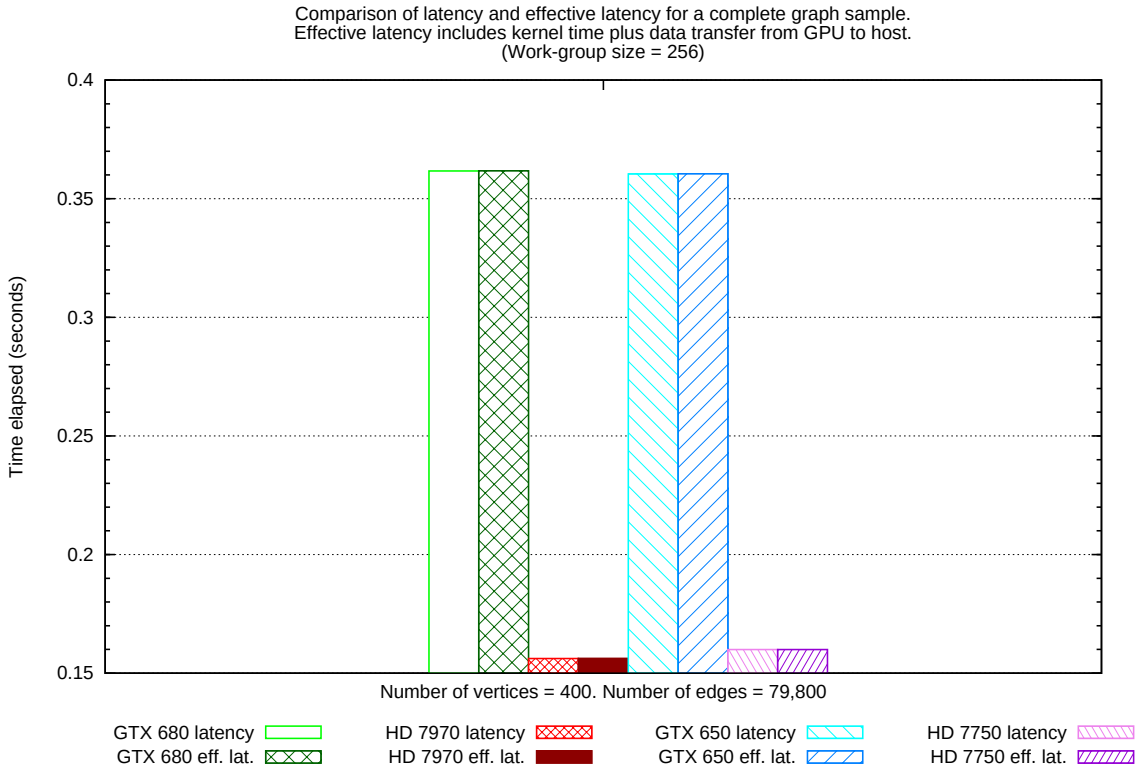


FIGURE 6.16: Comparison of latency vs. effective latency for complete graph (400 vertices, 79,800 edges)

6.9.2 Results on effects of work-group size

In this experiment, we evaluate the performance of `DPS-`d, `Velvet-`d and `FDGV-`d while varying the size of the work-group used in each run. The performance metrics that we are interested in are latency and throughput as these directly measure the performance of the kernel and GPU device. For our NVIDIA GPUs the experiments are run using work-group sizes of 64, 128, 256, 512 and the maximum supported size of 1024, while for our AMD GPUs we use work-group sizes 64, 128 and the maximum supported size of 256.

`DPS-`d. When we consider how the time indices are distributed among the work-items in a work-group, one would expect that the larger the size of the work-group a GPU device can support, the quicker it can complete the iterations. This particular experiment becomes interesting because the NVIDIA GPUs can support a larger work-group size, up to 1024, than the AMD GPUs with a limit of 256 work-items per work-group. However, supporting a larger work-group size does not necessarily guarantee a superior performance. In addition, the amount of data needed by each work-group does not depend on the number of work-items and is therefore constant. This is because for a

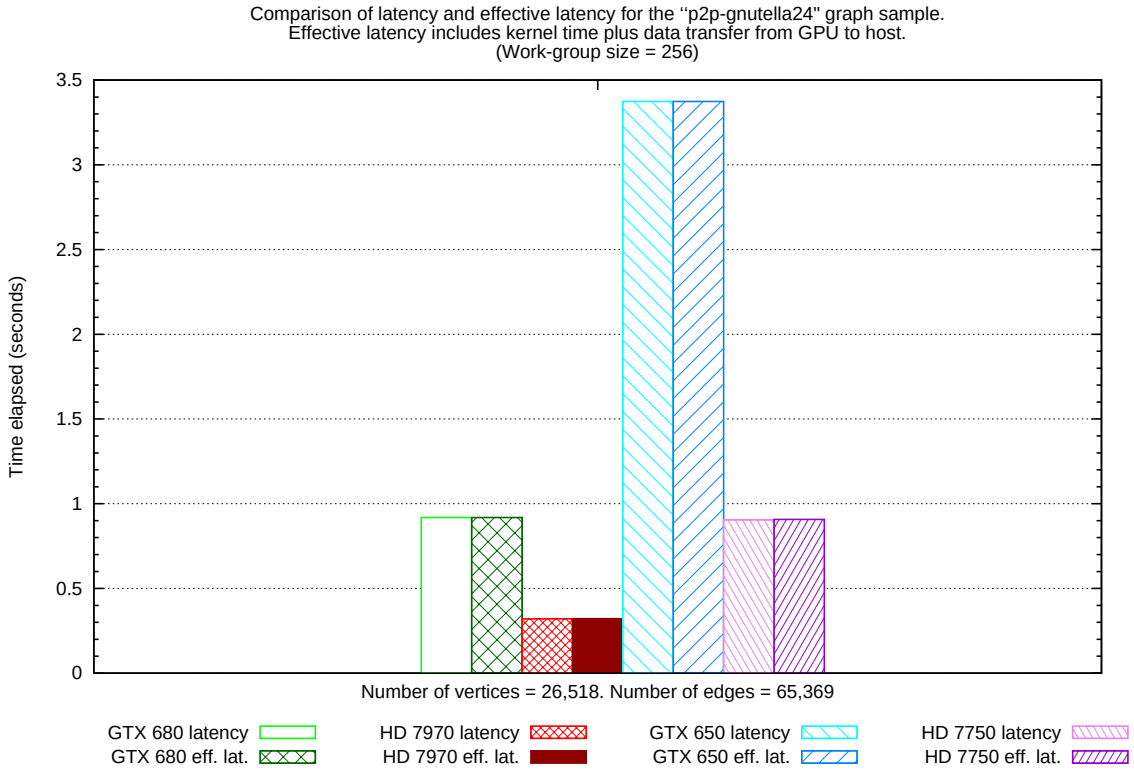


FIGURE 6.17: Comparison of latency vs. effective latency for Gnutella p2p network graph (26,518 vertices, 65,369 edges)

given work-group we only need to cache a row each of communication tables, $\mathcal{CE}$ and $\mathcal{CT}$, related to that particular work-group and the size of this data is the same for all work-groups.

Figures 6.20 to 6.23 show the results obtained using the various supported work-group sizes with respect to how they affect latency.

When we analyse these results we can clearly observe that, regardless of the GPU hardware architecture and problem size, the GPU devices perform significantly worse with a work-group size of 64. For instance, in Figure 6.20, we see the results for 3,200 jobs on the NVIDIA GTX 680 GPU where the latency when using a work-group size of 64 is cut by around 44% compared to using a work-group size of 128. The same observation can be seen on the AMD HD 7970 where the difference is up to 53% (Figure 6.22). This is possibly due to the fact that the number of work-items in a work-group is too small which implies that each work-item will have more work to do. Most importantly, using a work-group size of 64 means that there is not enough work-items to hide latency during execution.

The other consistent observation is that on the AMD GPUs, a work-group size of 256 performs best overall regardless of the problem size and GPU model. In order to have

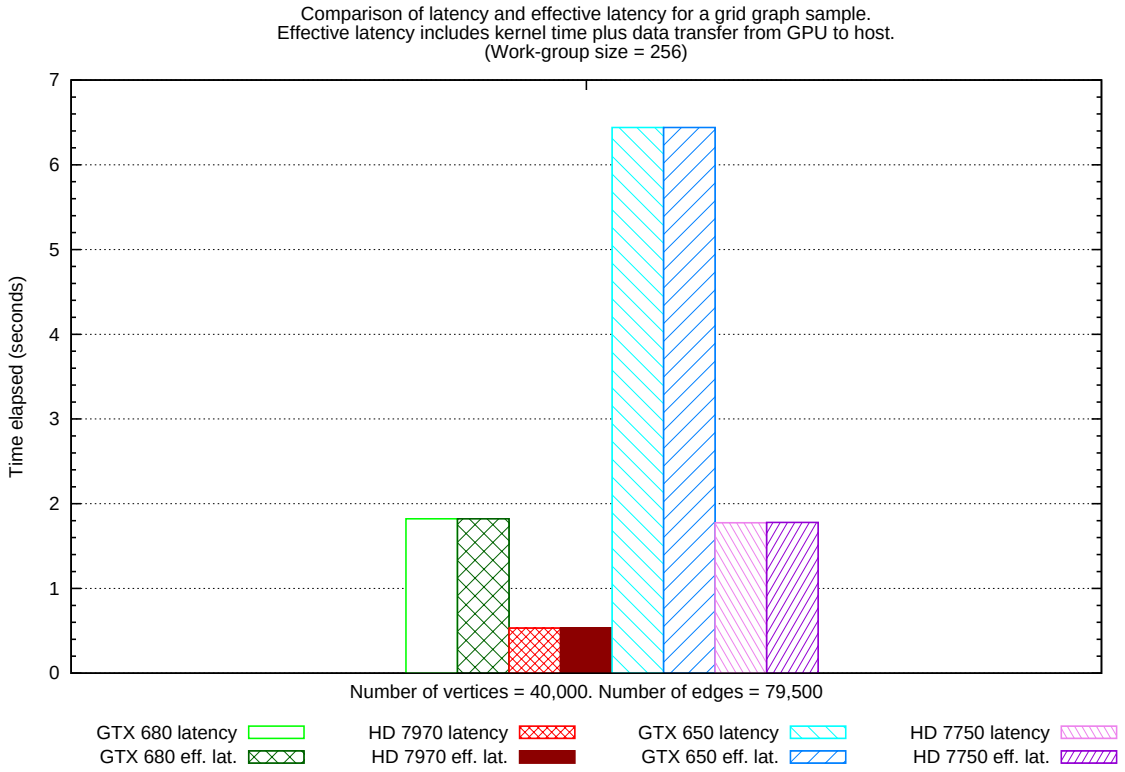


FIGURE 6.18: Comparison of latency vs. effective latency for grid graph (40,000 vertices, 79,500 edges)

a clearer picture so that we can put all the results into perspective, Figures 6.24 to 6.27 show the results based on the throughput of the `DPS-`d kernel. The most significant result involves the small problem sizes, for instance with 16 machines, where the NVIDIA GTX 680 is able to achieve around 1600 MCUPS using a work-group size of 1024 work-items. The NVIDIA GTX 680 is only able to manage around 149 MCUPS when the work-group size is 64. The AMD HD 7970 GPU is able to achieve around 330 MCUPS for the same problem size with a work-group size of 256, which is around 283% improvement from using a work-group size of 64, achieving a throughput of around 86 MCUPS.

However, although both NVIDIA GPUs record the highest amount of throughput for the problem consisting of 16 machines, what is even more interesting is that they achieve this with different work-group sizes. The high-end GTX 680 prefers a work-group size of 1024 while the GTX 650 prefers a work-group size of 256. But as the problem size gets larger, performance from both NVIDIA GPUs begin to converge to a work-group size of 128.

`Velvet-`d. For this application we are mainly concerned with the kernel responsible for computing the forces since this is where majority of the GPU time is spent during each iteration. Unlike the `DPS-`d kernel, the size of the work-group for `Velvet-`d relates to both

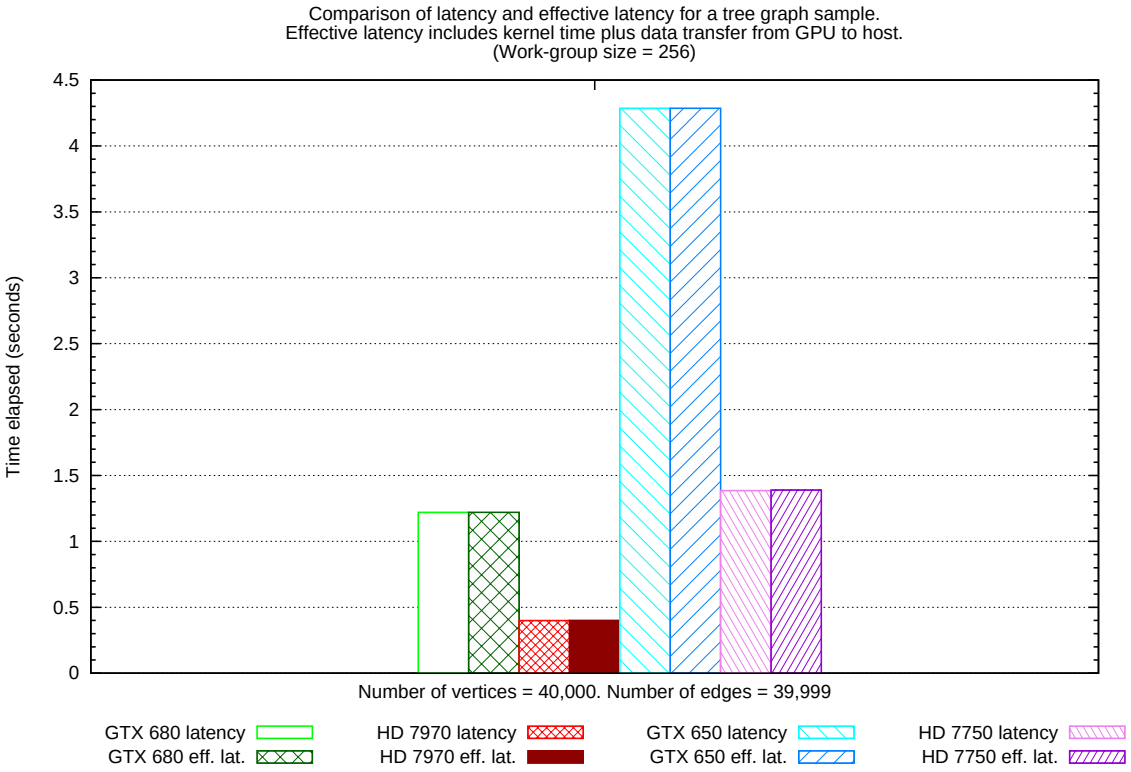


FIGURE 6.19: Comparison of latency vs. effective latency for tree graph (40,000 vertices, 39,999 edges)

work distribution among work-items and the amount of data stored for each work-group by a compute unit. This relationship makes it quite a challenge to predict which factor will take precedence in determining performance for two reasons. Firstly, we know that when the size of data stored for each work-group grows as the work-group size grows, the compute unit fits less and less work-groups which implies that fewer work-groups are scheduled to run concurrently. Lastly, when computation depends on distributing work among work-items in a work-group, we want to have the largest possible work-group in order to finish the work quicker. In addition, these reasons are complicated by the fact that if the work-group size is too small, then performance might suffer.

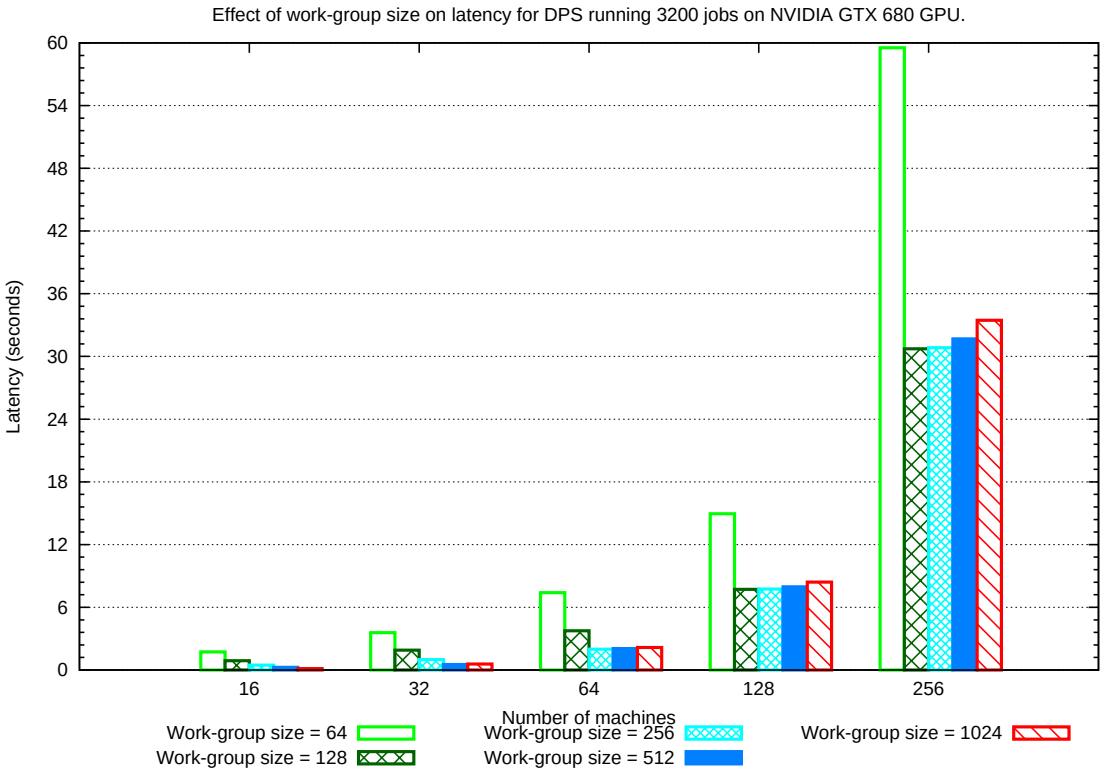


FIGURE 6.20: Latency for DPS-d with 3,200 jobs with varying work-group sizes on NVIDIA GTX 680 GPU.

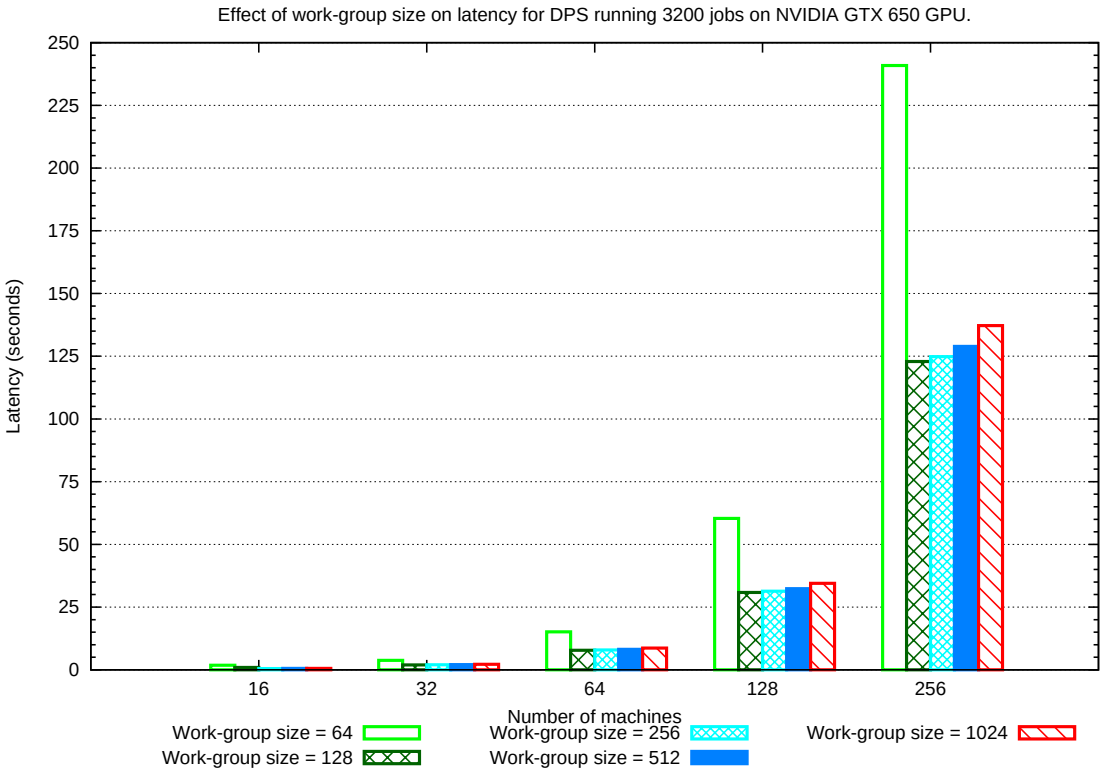


FIGURE 6.21: Latency for DPS-d with 3,200 jobs with varying work-group sizes on NVIDIA GTX 650 GPU.

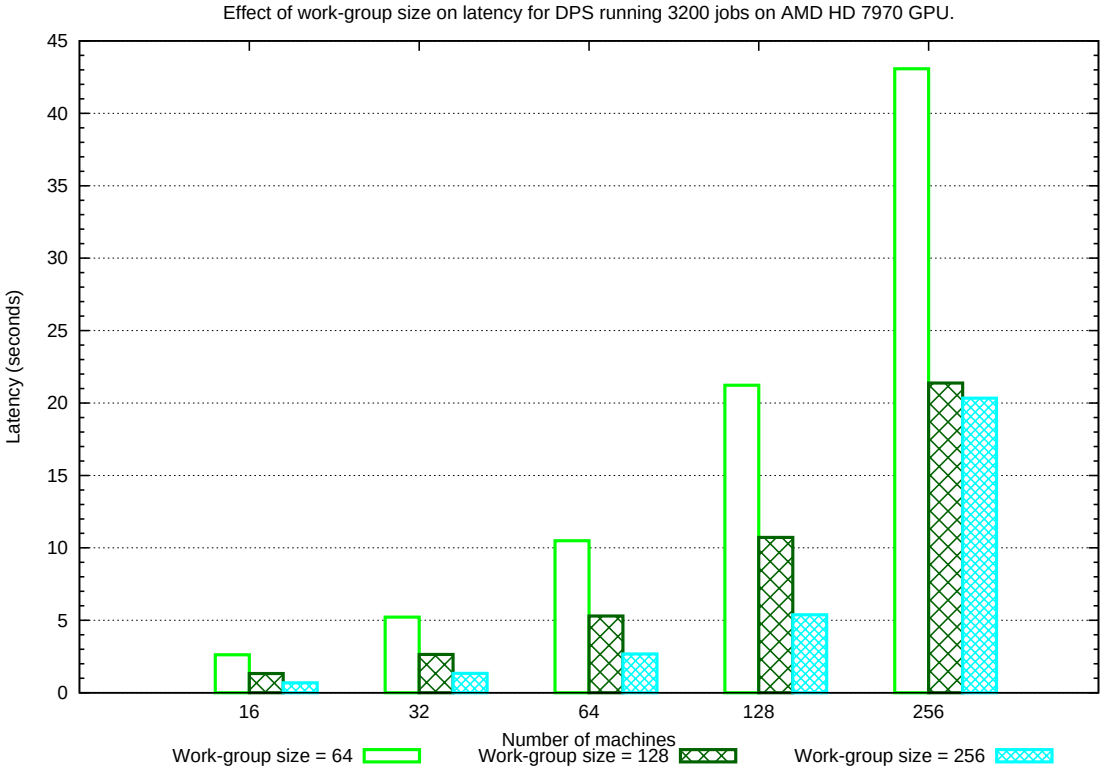


FIGURE 6.22: Latency for DPS-d with 3,200 jobs with varying work-group sizes on AMD HD 7970 GPU.

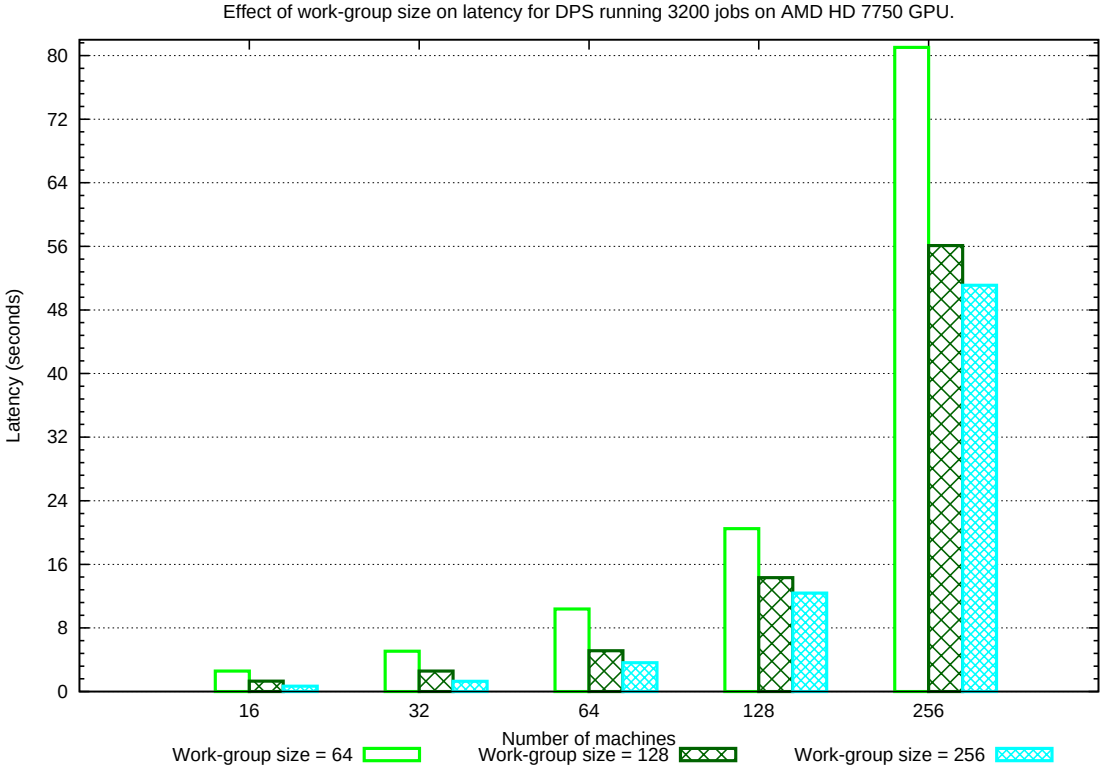


FIGURE 6.23: Latency for DPS-d with 3,200 jobs with varying work-group sizes on AMD HD 7750 GPU.

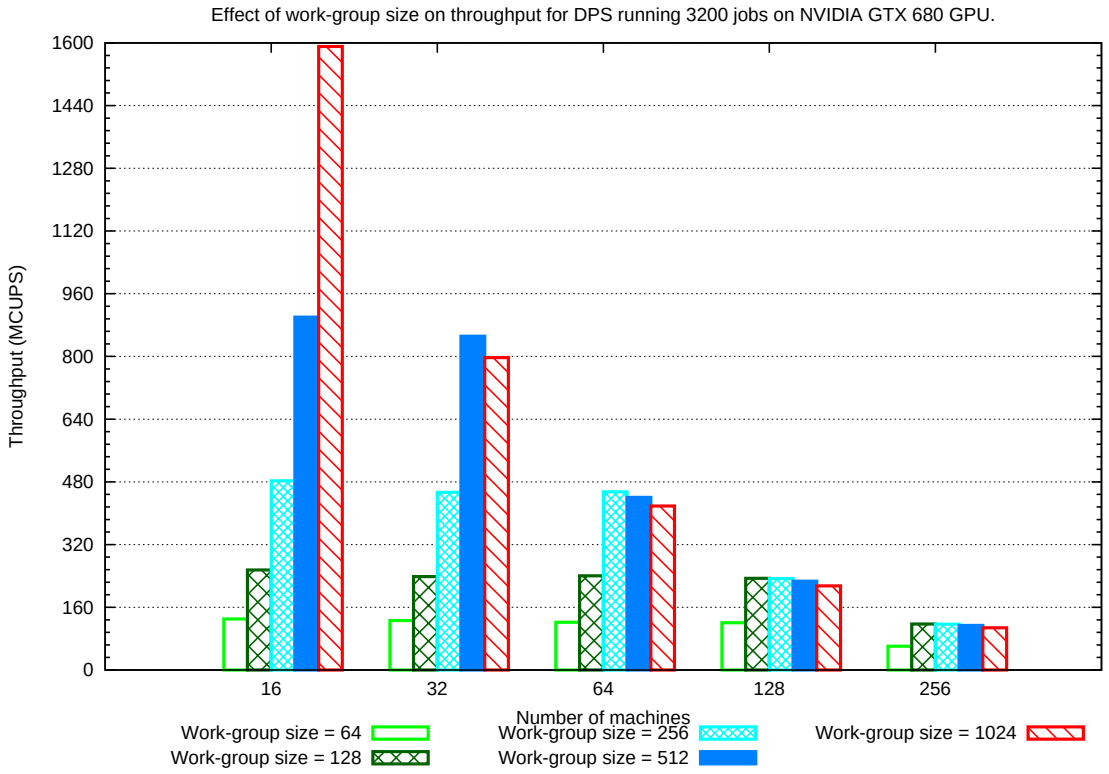


FIGURE 6.24: Throughput for DPS-d with 3,200 jobs with varying work-group sizes on NVIDIA GTX 680 GPU.

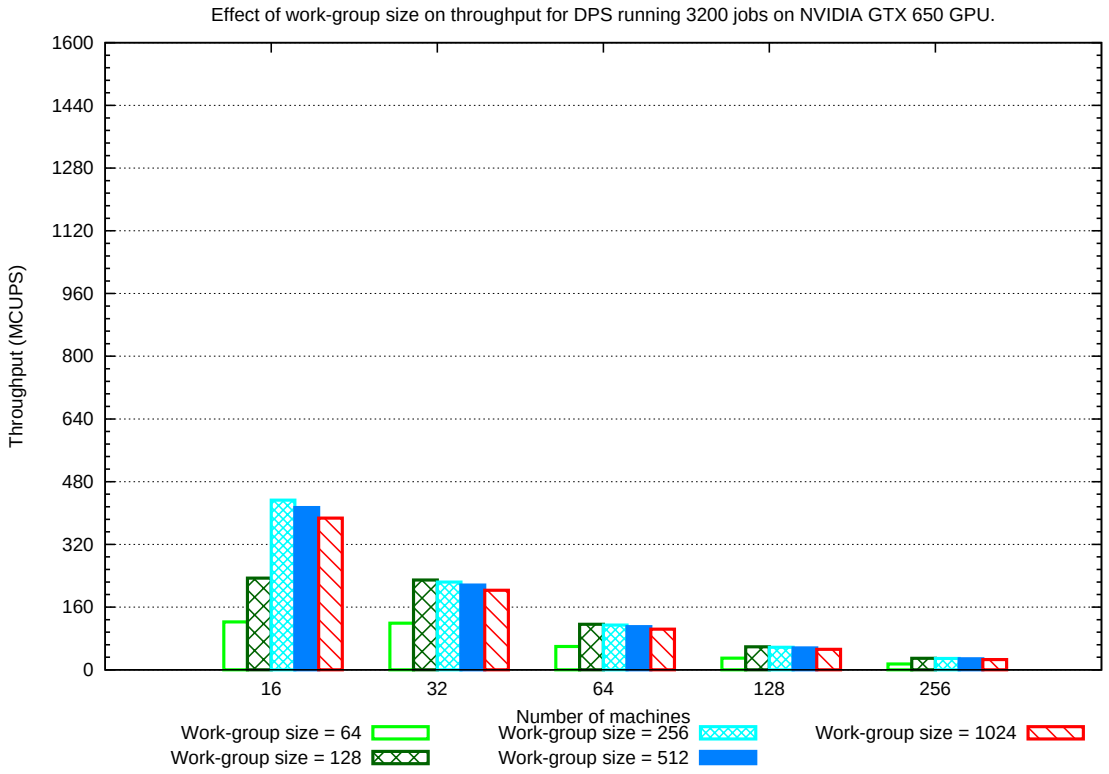


FIGURE 6.25: Throughput for DPS-d with 3,200 jobs with varying work-group sizes on NVIDIA GTX 650 GPU.

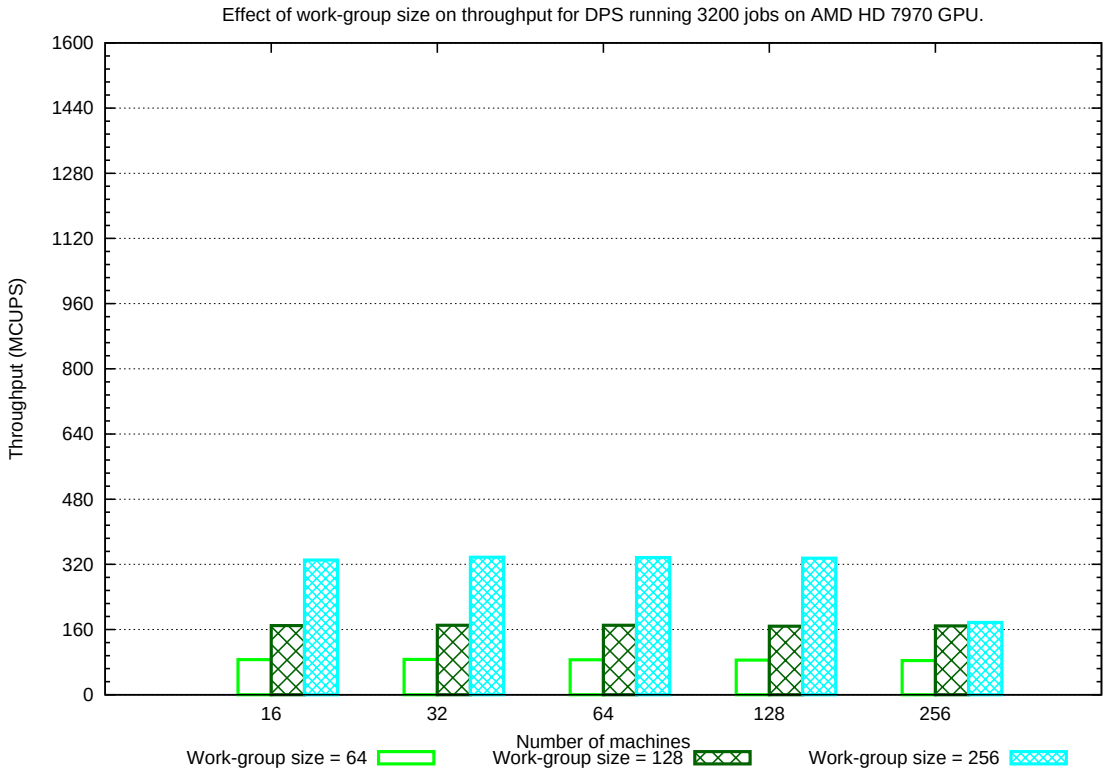


FIGURE 6.26: Throughput for DPS-d with 3,200 jobs with varying work-group sizes on AMD HD 7970 GPU.

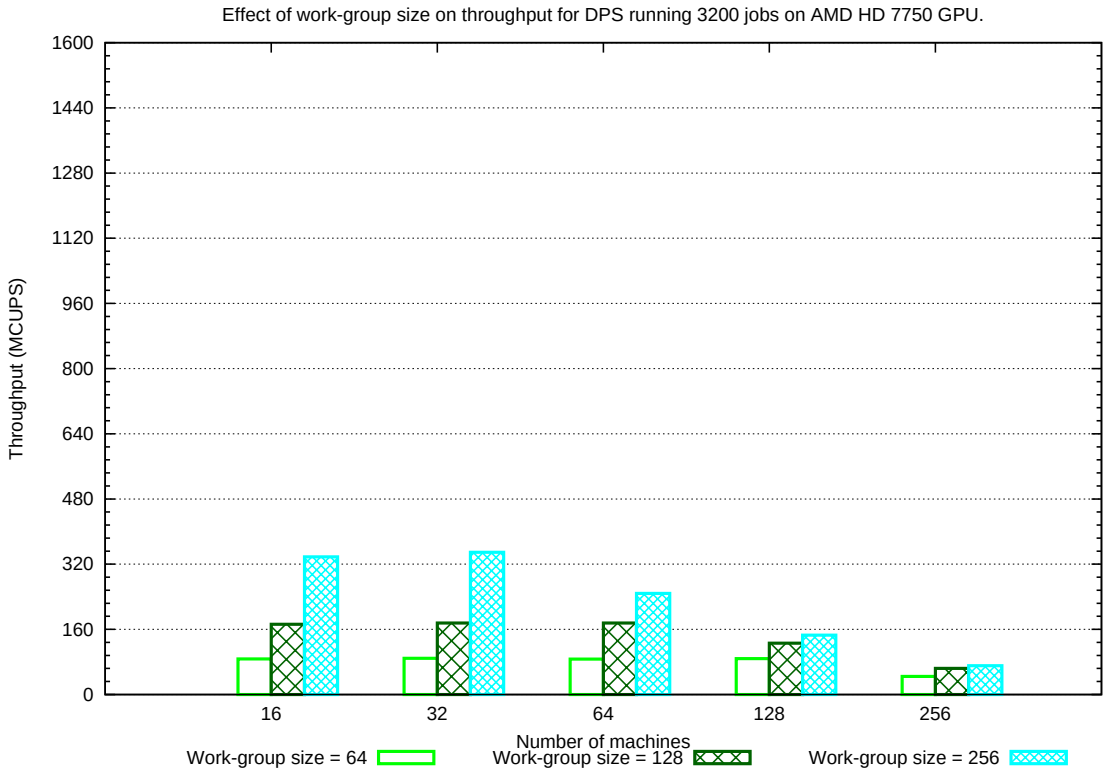


FIGURE 6.27: Throughput for DPS-d with 3,200 jobs with varying work-group sizes on AMD HD 7750 GPU.

Based on these reasons it is difficult to know what to expect from our GPU devices for this application. This is further justified by the results shown in Figures 6.28 to 6.31 for each GPU device.

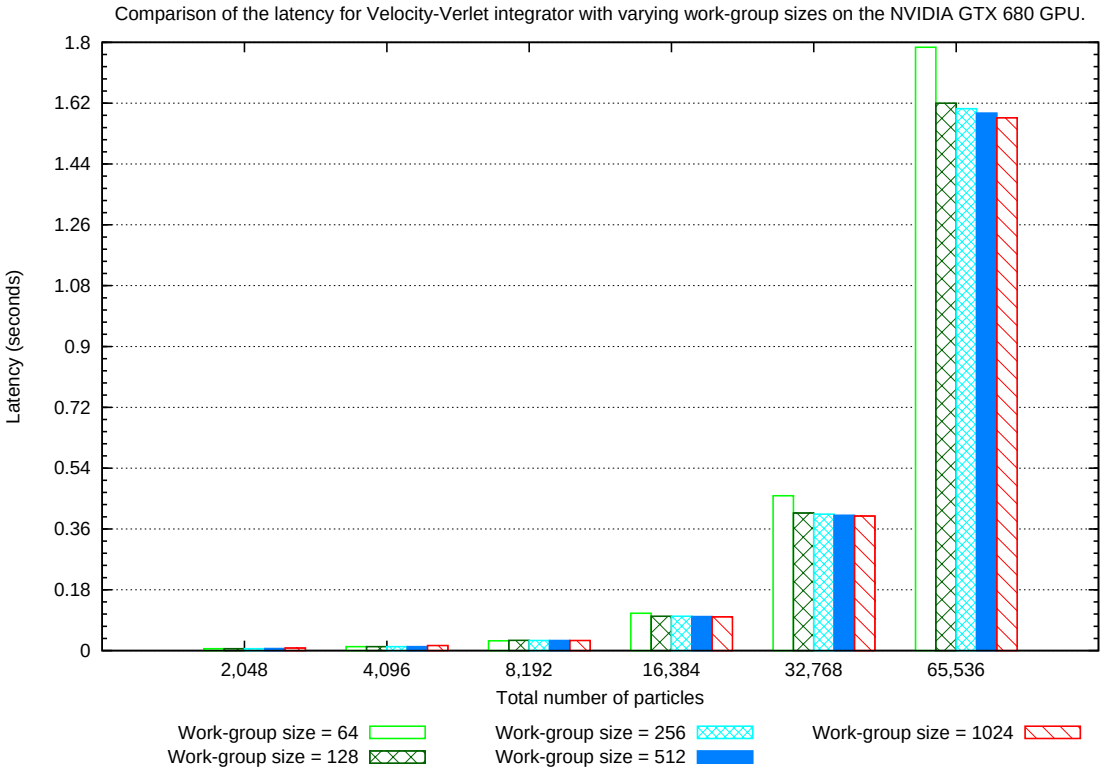


FIGURE 6.28: Latency for `Velvet`-d with varying work-group sizes on NVIDIA GTX 680 GPU.

When we consider the performance of the NVIDIA GPU devices, there is a consistent trend that seems to suggest that a larger work-group is better especially for the larger problem sizes which correlates with having work-items finish computation quicker. In addition, a work-group size of 64 performs worse and this has been a consistent observation so far for our NVIDIA GPUs. For instance, in Figure 6.28 showing the results for the NVIDIA GTX 680 GPU, the latency for a simulation with 65,536 particles descreases by around 15% when the work-group size is changed from 64 to 1024. However, for the AMD GPUs, the results show that a work-group size of 128 performs slightly worse by a margin of between 0.4% and 0.5%. Figures 6.32 to 6.35 shows the throughput performance for all GPU devices.

With respect to throughput performance, it becomes clearer that the NVIDIA GTX 680 prefers smaller work-group sizes for the smaller problem sizes but larger work-group is preferred for the larger problem sizes. As shown in Figure 6.32, for a simulation with 2,048 particles, and work-group size of 64, the NVIDIA GTX 680 GPU is able to reach a throughput of around 181 GFLOPS. But with a work-group size of 1024 it could only

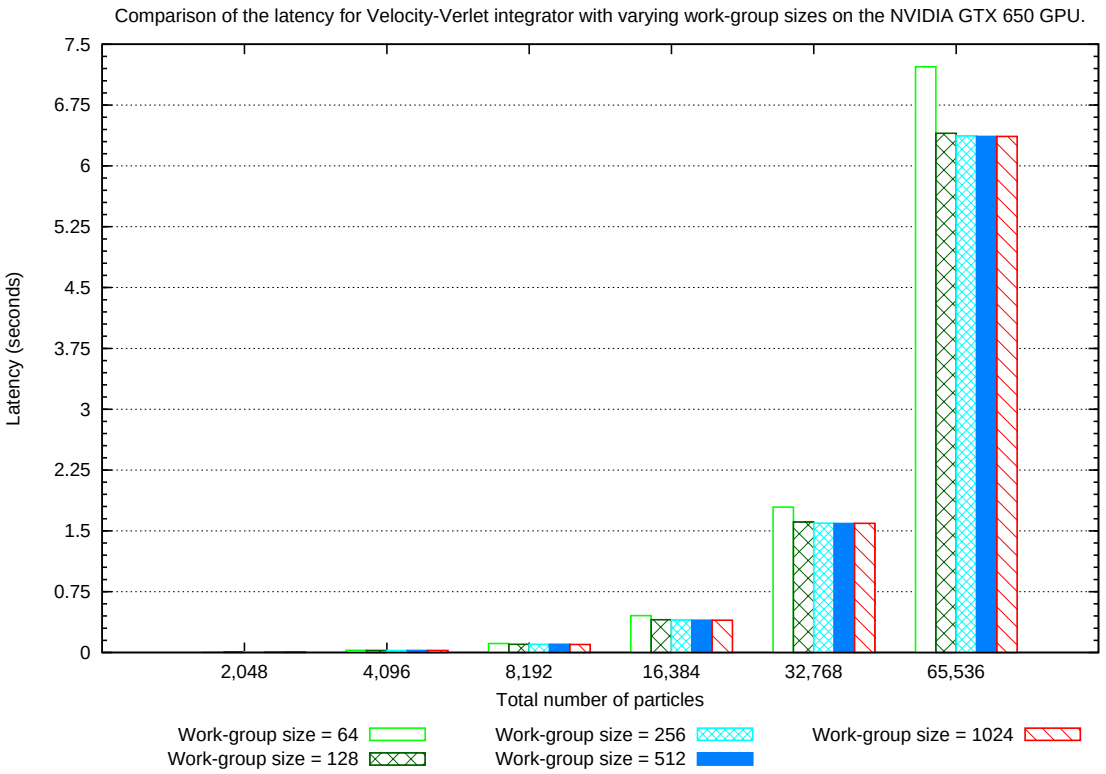


FIGURE 6.29: Latency for `Velvet-`d with varying work-group sizes on NVIDIA GTX 650 GPU.

manage a throughput of around 133 GFLOPS. For large problem size of 65,536 particles, the NVIDIA GTX 680 GPU is able to achieve a throughput of around 653 GFLOPS with work-group size 1024 compared to a throughput of around 577 GFLOPS with a work-group size of 64. On the other hand, the smaller GTX 650 consistently prefers a larger work-group size as seen in Figure 6.33 but the difference between the various work-group sizes is very small. The difference is up to 1.5%. Meanwhile, both AMD GPUs are consistent in preferring work-group sizes of 64 and 256 as opposed to a work-group size of 128 (Figure 6.34 and Figure 6.35). When the work-group is set to 256, the AMD HD 7970 GPU is able to achieve a throughput of around $1,025$ GFLOPS, which is marginally more than it is able to achieve with a work-group size of 64. This margin is less than 1%.

`FDGV-`d. Among the three kernels that make up `FDGV-`d, only the kernel responsible for repulsion of vertices provides the option of having work-items in a work-group iterate through vertex data in global memory or cache them to local memory for re-use. Although this application is very similar to `Velvet-`d, one might expect similar behaviour from the GPU devices. However, unlike `Velvet-`d, each kernel in `FDGV-`d has a different running time and this factor can affect the overall outcome with respect to the latency

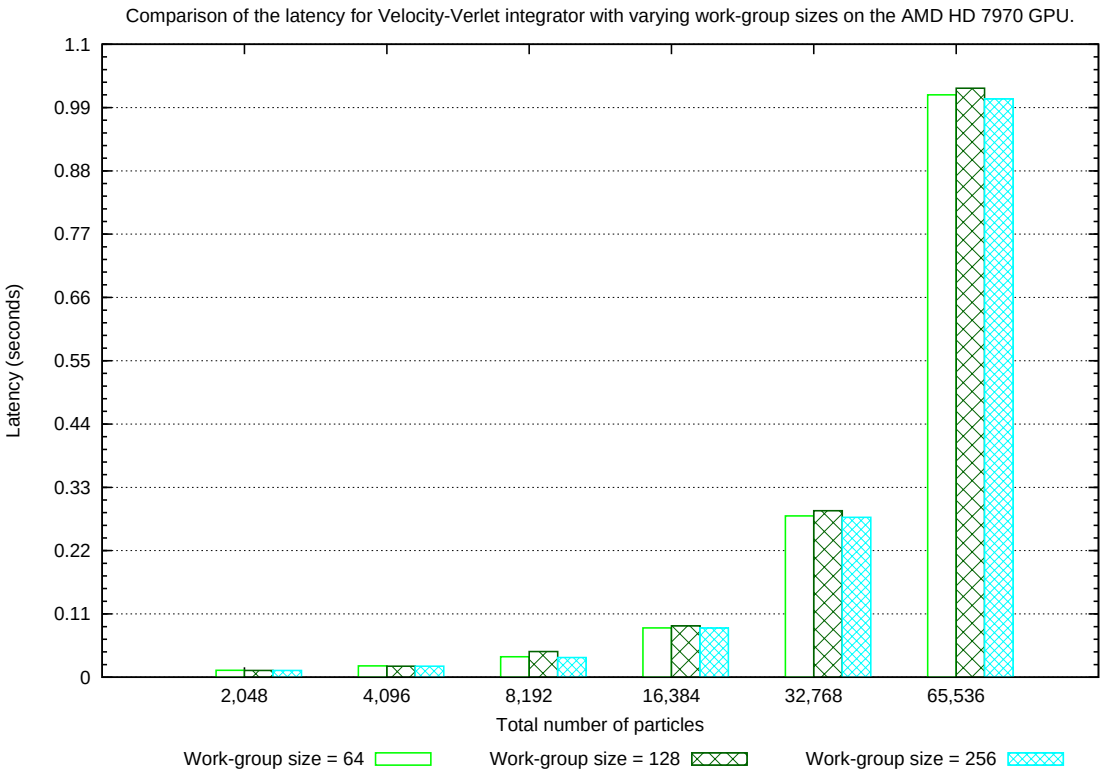


FIGURE 6.30: Latency for `Velvet-`d with varying work-group sizes on AMD HD 7970 GPU.

of all three kernels. This makes it quite a challenge to determine a suitable work-group size and then predict the outcome. However, from our previous results from `Velvet-`d we would expect that using a work-group size of 64 may not produce the best results for our NVIDIA GPUs and, likewise, using a work-group size of 128 on the AMD GPU devices. Figures 6.36 to 6.39 shows the latency performance results for `FDGV-`d and they confirm our expectations.

Both AMD GPUs seem to prefer a work-group size of 64 or 256 work-items over 128 work-items. On the AMD HD 7970, for instance, the latency when using a work-group size of 64 or 256 is around 18% smaller. However, this difference is mostly observed with the grid and tree graphs as no significant difference can be observed with the complete graph and Gnutella real graph data. Meanwhile, Figure 6.36 and Figure 6.37 show slightly different results for the NVIDIA GPUs. The margin of difference is almost the same for the complete graph simulation, and, with work-group size of 64, both GPUs perform worse for other three graph simulations. The NVIDIA GTX 680 performs better, in general, with a work-group size of 512 work-items and worse with work-group size of 64 or 1024. When the work-group size is set to 512, we observe an improvement in latency of around 30%. On the other hand, the GTX 650 only performs worse when work-group size is 64, incurring an additional 50% approximate increase in latency compared to using

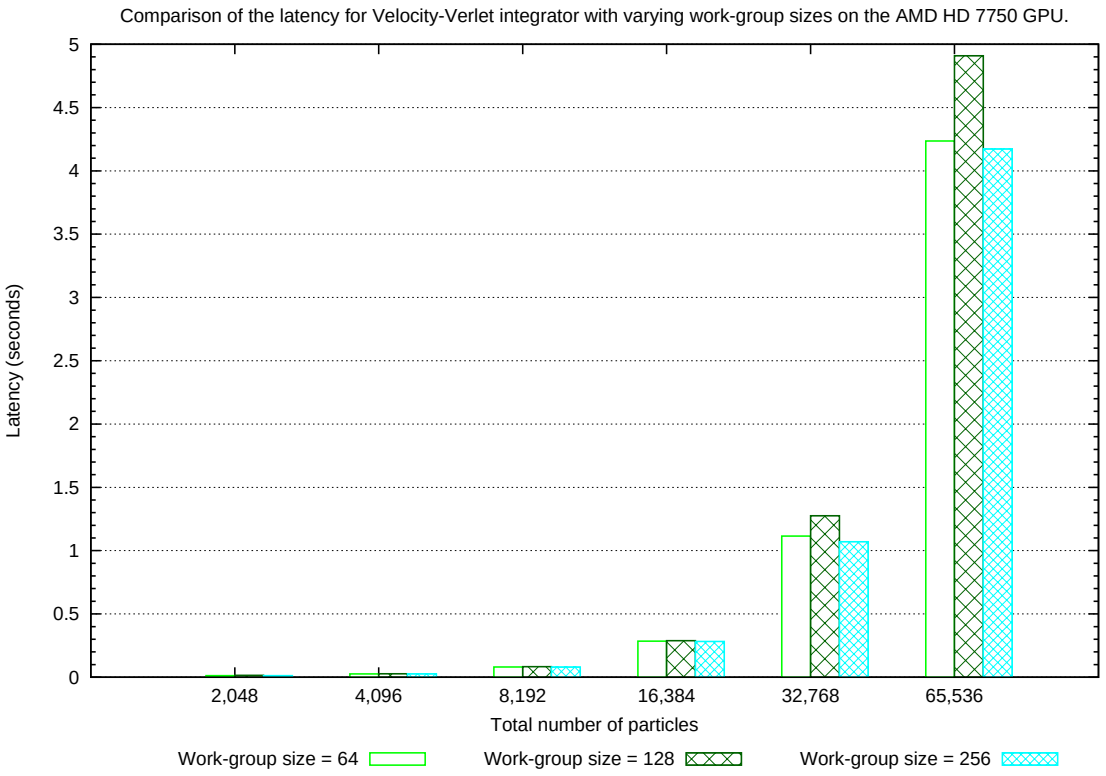


FIGURE 6.31: Latency for `Velvet-`d with varying work-group sizes on AMD HD 7750 GPU.

other work-group sizes. This difference translates to around 3 additional seconds for the grid graph and around 1.7 seconds for the tree graph.

Figures 6.36 to 6.39 put the results into perspective and show how the total throughput of the kernels is affected. The margins observed in latency performance reflects directly in the throughput results as expected.

Conclusion. Finally, we can conclude that choosing the best work-group size for a data-parallel application in order to achieve the best result in all possible application configurations can be quite a challenge. In most cases, there is no definitive work-group size that can guarantee optimal performance for a given GPU device. The results on the NVIDIA GPUs show that even for the same GPU architecture, results can still vary significantly across GPU models belonging to the same GPU family. In addition, the decision on work-group size becomes more difficult when designing a cross-vendor heterogeneous application which seems to suggest that finer and more specific application tuning is necessary if aiming for the best possible performance results.

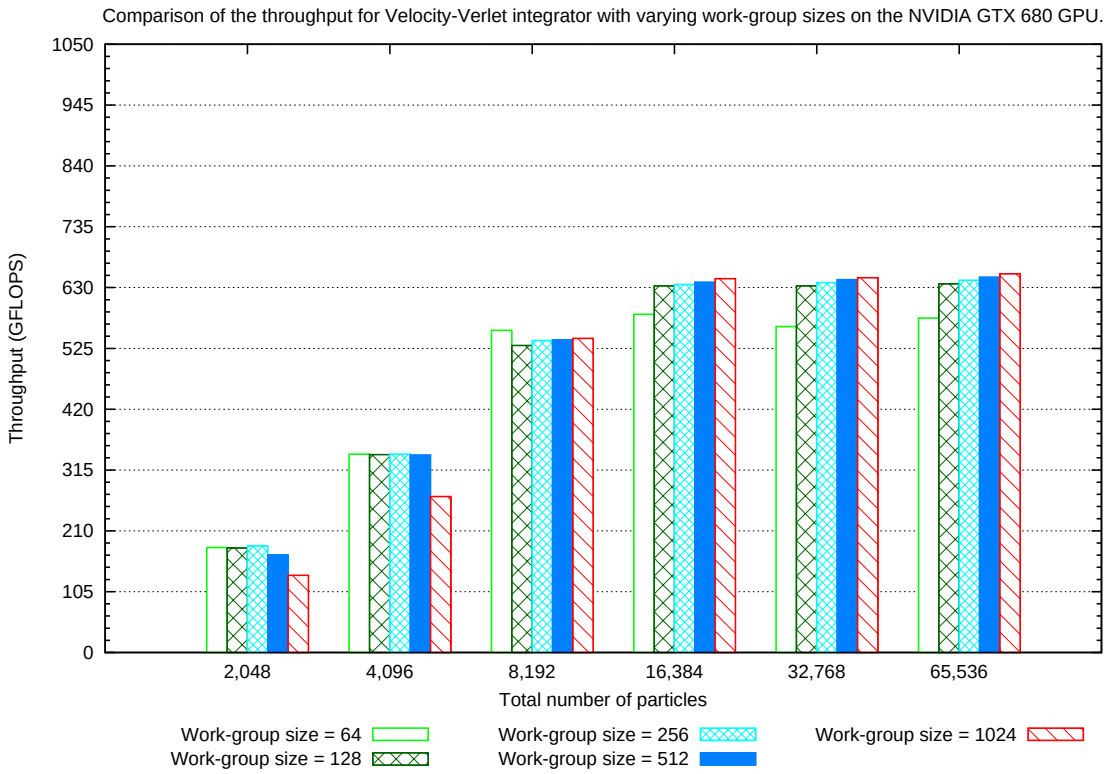


FIGURE 6.32: Results showing the throughput performance of `Velvet-`d as work-group size varies on the NVIDIA GTX 680 GPU.

6.9.3 Results on effects of local memory

This experiment involves `DPS-`d, `Velvet-`d and `FDGV-`d. For the kernels with the possibility for work-items to cooperate and share data by caching data to local memory, we implement a variant where work-items read data directly from global memory instead. Depending on the operation being performed by the work-items, one option might be preferable over the other for best performance.

`DPS-`d. In this application, we have the option of caching partial data from the $\mathcal{CE}$ and $\mathcal{CT}$ tables in the kernel. Since the work-items use values from these tables repeatedly over the course of the computation, we expect that caching data to local memory will help to improve performance across all the GPU devices from both AMD and NVIDIA. Figures 6.44 and 6.45 show the results for latency for problem sizes of 1,600 and 3,200 jobs, respectively.

The results are quite consistent for all GPU devices regardless of the problem size. When data is cached to local memory, we observe a performance benefit of around 25% compared to using data from directly global memory.

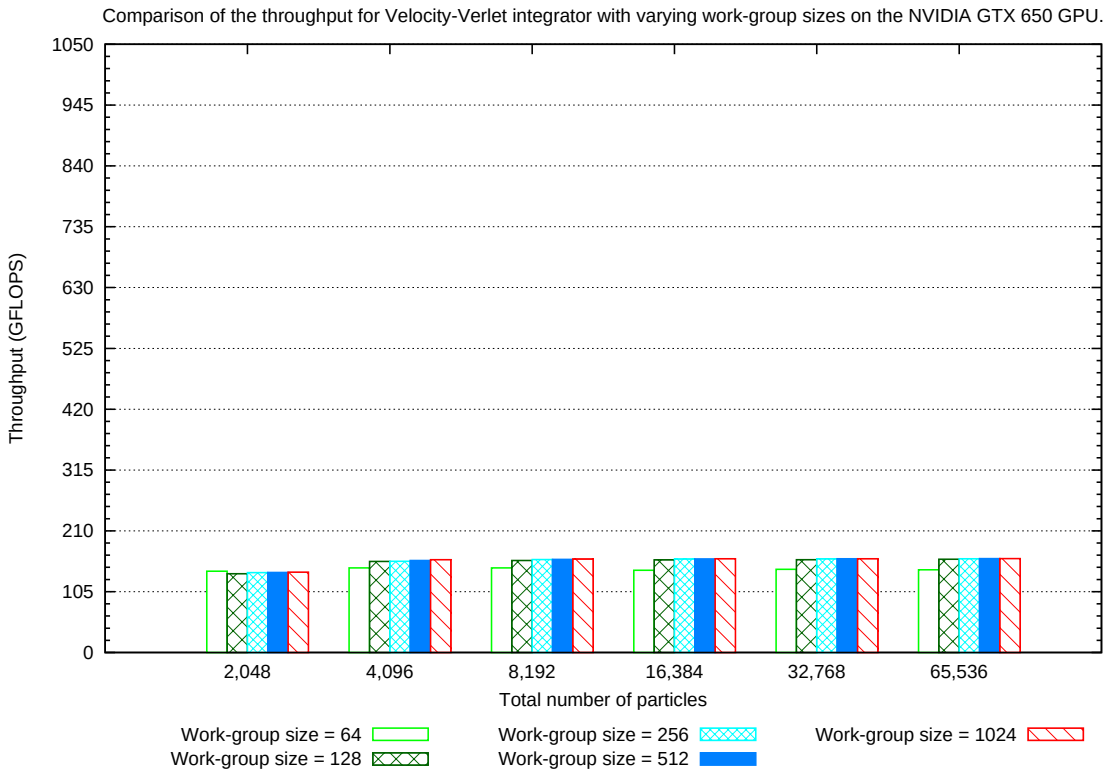


FIGURE 6.33: Results showing the throughput performance of `Velvet-`d as work-group size varies on the NVIDIA GTX 650 GPU.

We can observe that there is a significant difference in performance between caching data to local memory and reading directly from global memory. Accessing global memory is much slower than local memory and depending on the structure of the computation being performed, it is imperative to re-use data where possible by caching to local memory. Figures 6.46 and 6.47 show the throughput performance results. Throughput performance gain for all GPU devices is around 25% when data is cached to local memory.

`Velvet-`d. For the kernel that computes the forces for the particles, we have the option to cache positions data to local memory in order to allow work-items within the same work-group to re-use data. We will expect that caching data to local memory will help to improve the performance of the kernel, similar to what is achieved with `DPS-`d. However, Figures 6.48 and 6.49 show that this is not exactly the case. Note that we have adjusted these results based on the results we obtained regarding work-group sizes in Section 6.9.2 to show the best performing configuration for the GPUs.

These results are very interesting because, although the results for the NVIDIA GPUs meet our expectations, the opposite is exactly the case for the AMD GPUs. For example, in the simulation with 65,536 particles, the NVIDIA GPUs perform around 24% better when data is cached to local memory. On the other hand, the AMD GPUs actually prefer

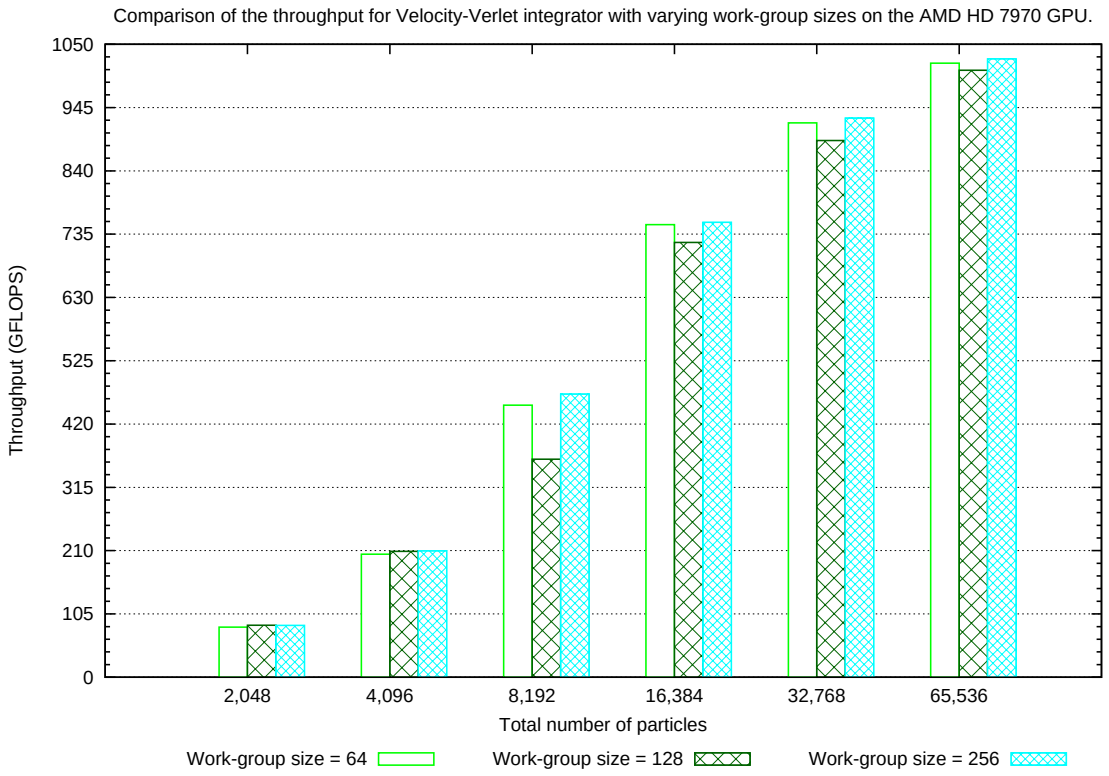


FIGURE 6.34: Results showing the throughput performance of `Velvet-`d as work-group size varies on the AMD HD 7970 GPU.

not to cache data to local memory. In the same simulation, the AMD GPUs perform around 10% better when data is read directly from global memory without caching to local memory.

`FDGV-`d. When we take into account the similarities between `Velvet-`d and `FDGV-`d, in addition to the results obtained in this experiment for `Velvet-`d, it is safe to expect that the same trend will continue for this application. Figures 6.50 and 6.51 confirm that this is in fact the case. We have also adjusted the results so that we use the best performing work-group size for each GPU. In this case, a work-group size of 512 and 256 work-items were used for the NVIDIA and AMD GPUs, respectively.

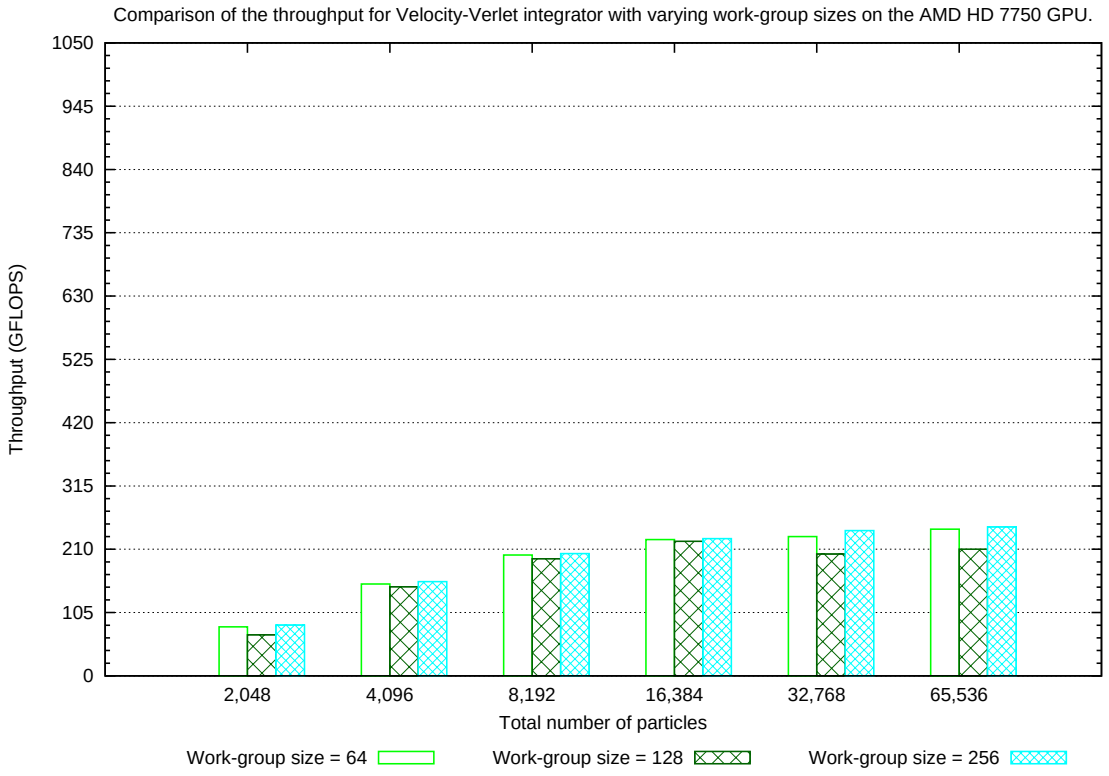


FIGURE 6.35: Results showing the throughput performance of `Velvet-`d as work-group size varies on the AMD HD 7750 GPU.

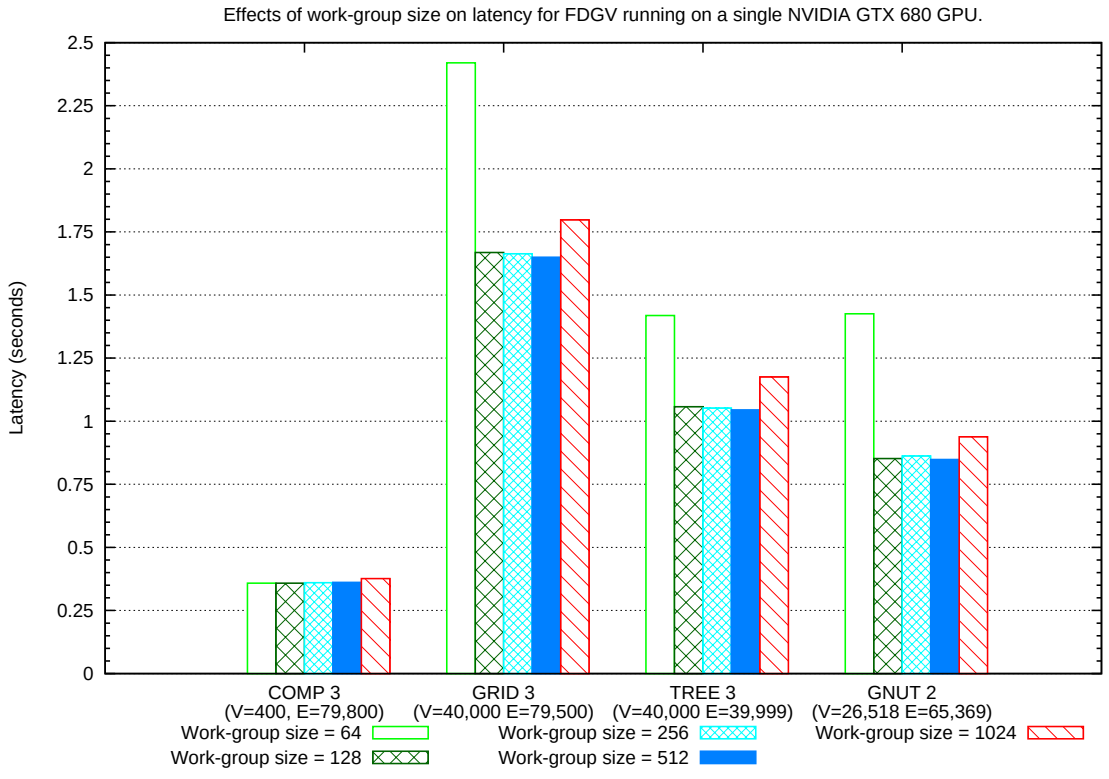


FIGURE 6.36: Results showing latency performance `FDGV-`d as work-group size varies for NVIDIA GTX 680 GPU.

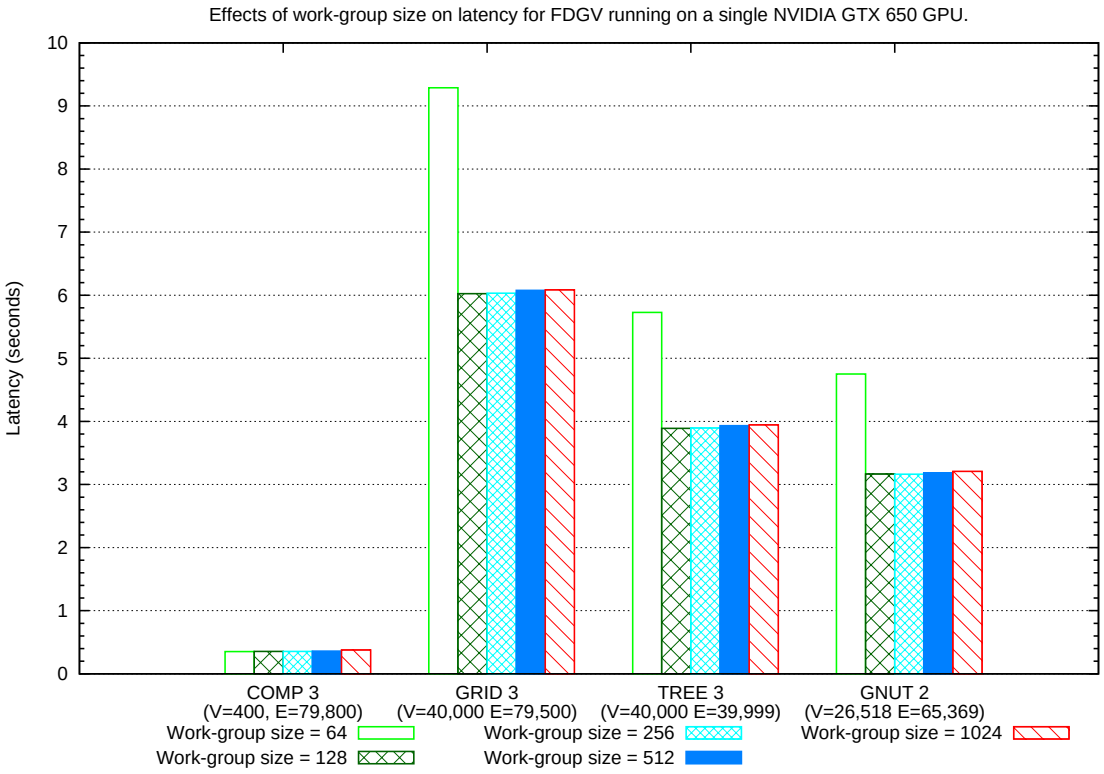


FIGURE 6.37: Results showing latency performance FDGV-d as work-group size varies for NVIDIA GTX 650 GPU.

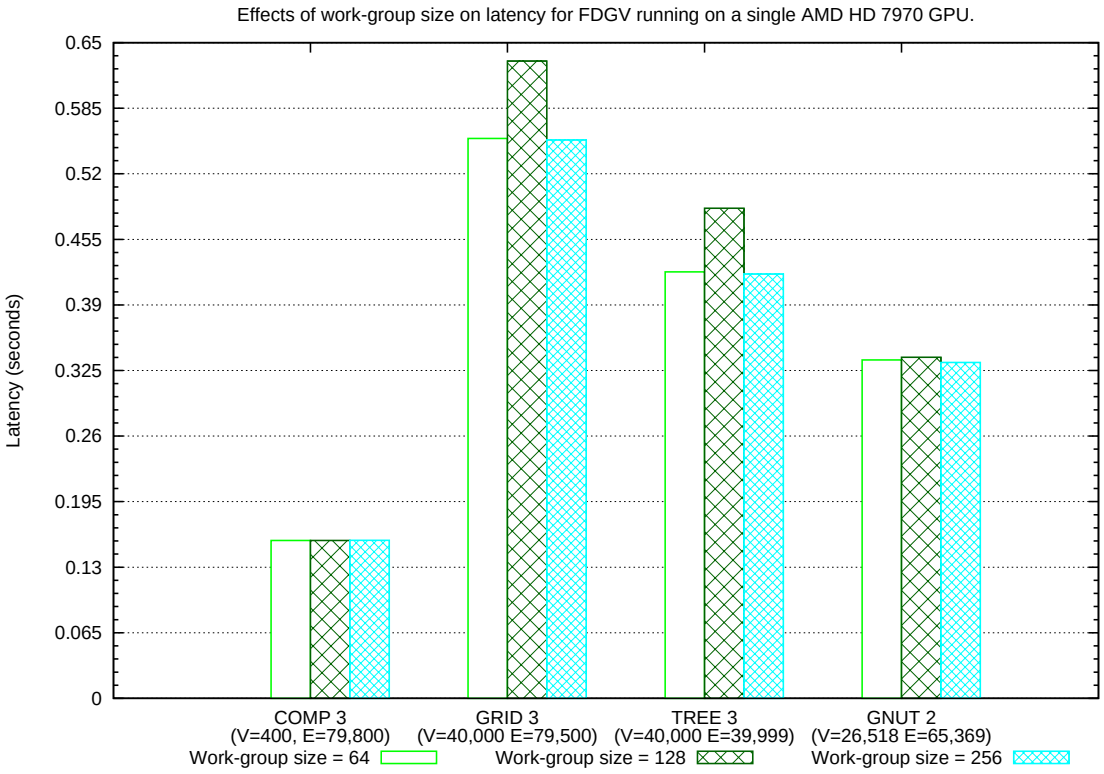


FIGURE 6.38: Results showing latency performance FDGV-d as work-group size varies for AMD HD 7970 GPU.

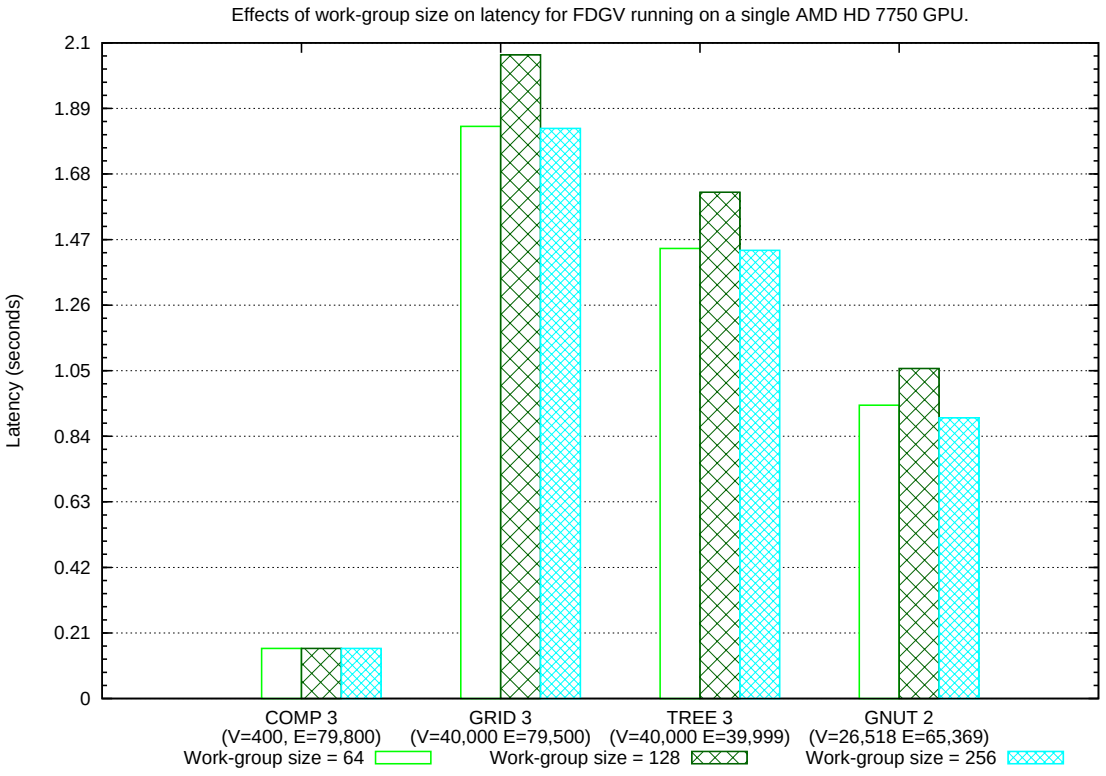


FIGURE 6.39: Results showing latency performance `FDGV-`d as work-group size varies for AMD HD 7750 GPU.

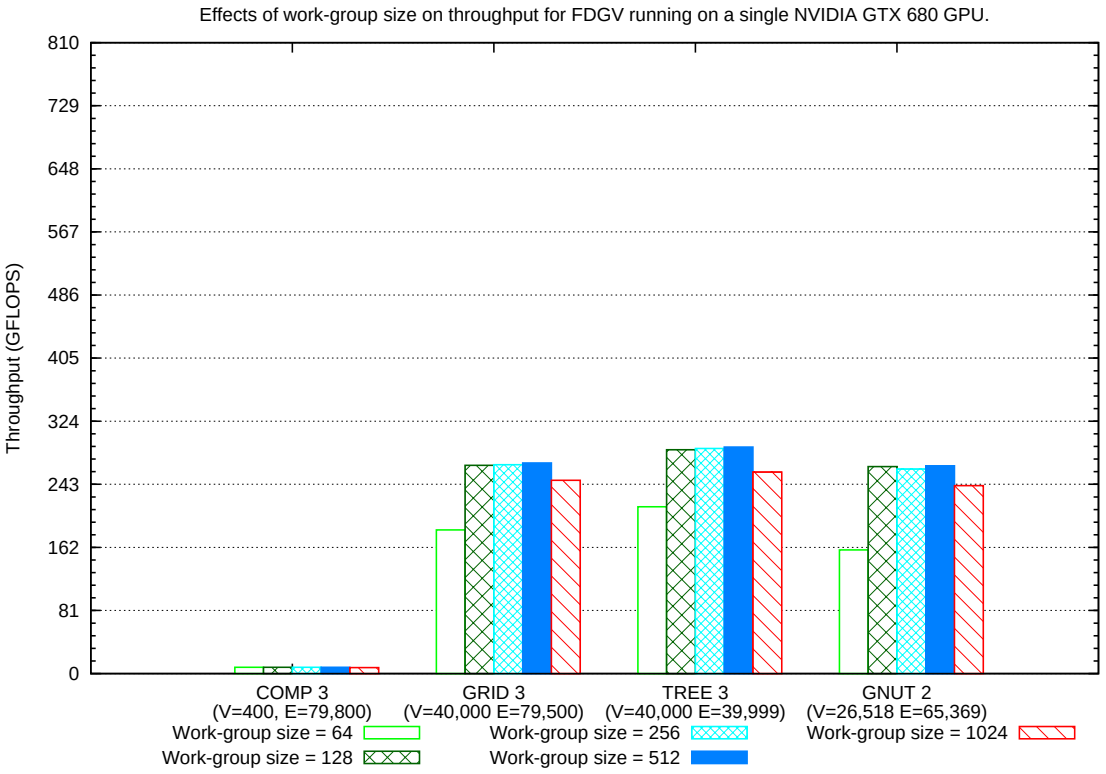


FIGURE 6.40: Results showing throughput performance `FDGV-`d as work-group size varies for NVIDIA GTX 680 GPU.

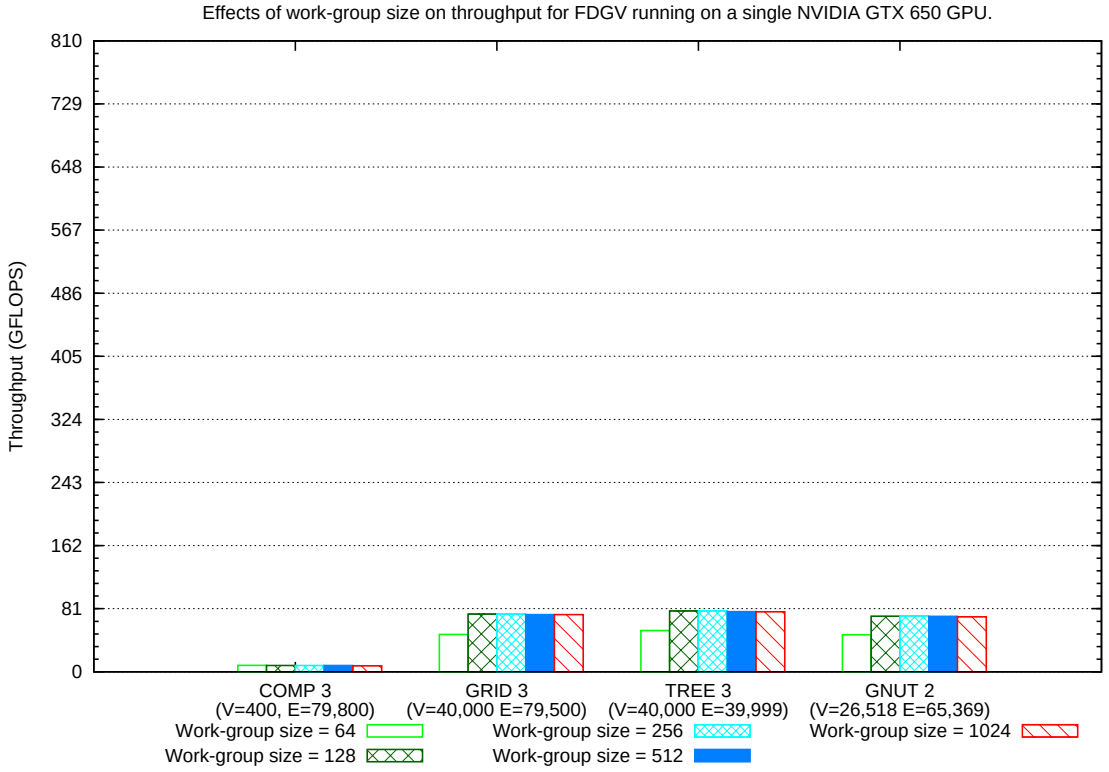


FIGURE 6.41: Results showing throughput performance `FDGV-`d as work-group size varies for NVIDIA GTX 650 GPU.

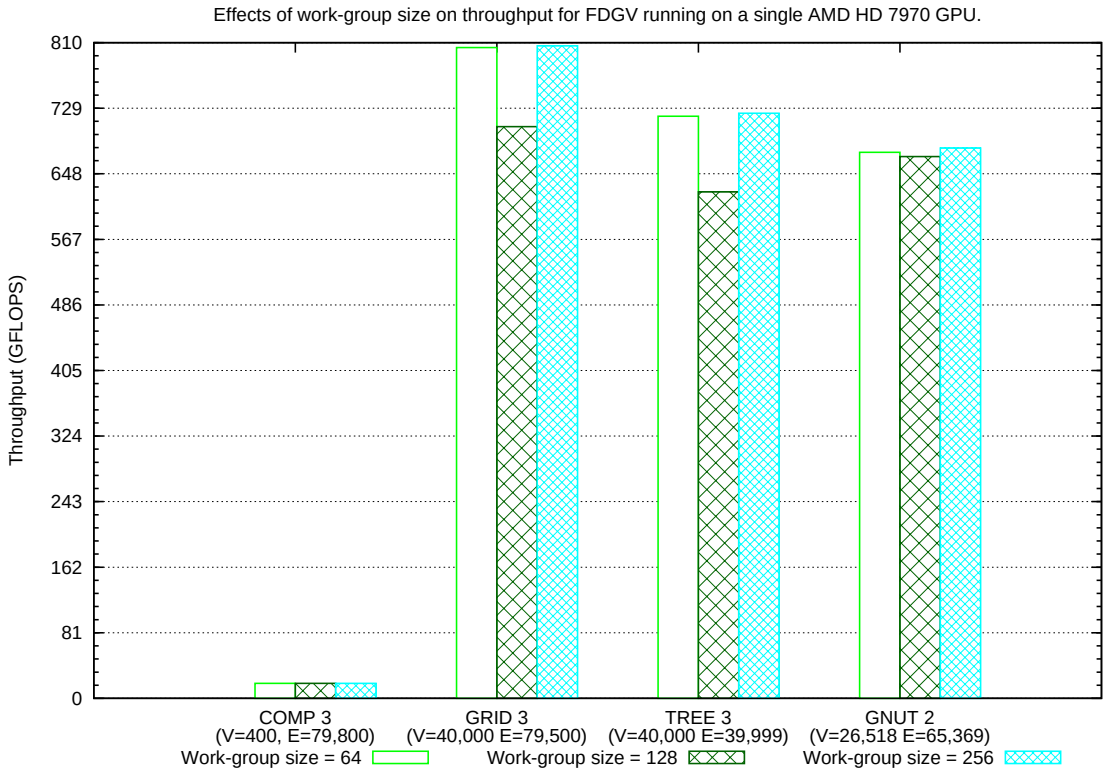


FIGURE 6.42: Results showing throughput performance `FDGV-`d as work-group size varies for AMD HD 7970 GPU.

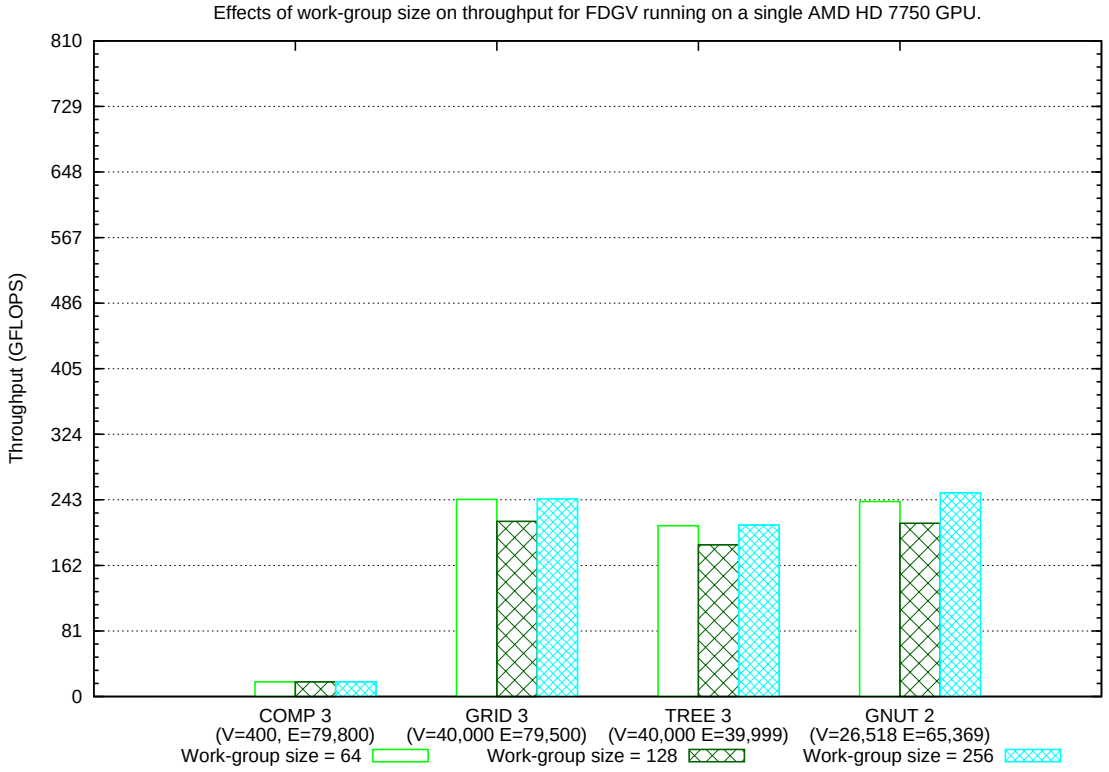


FIGURE 6.43: Results showing throughput performance `FDGV-`d as work-group size varies for AMD HD 7750 GPU.

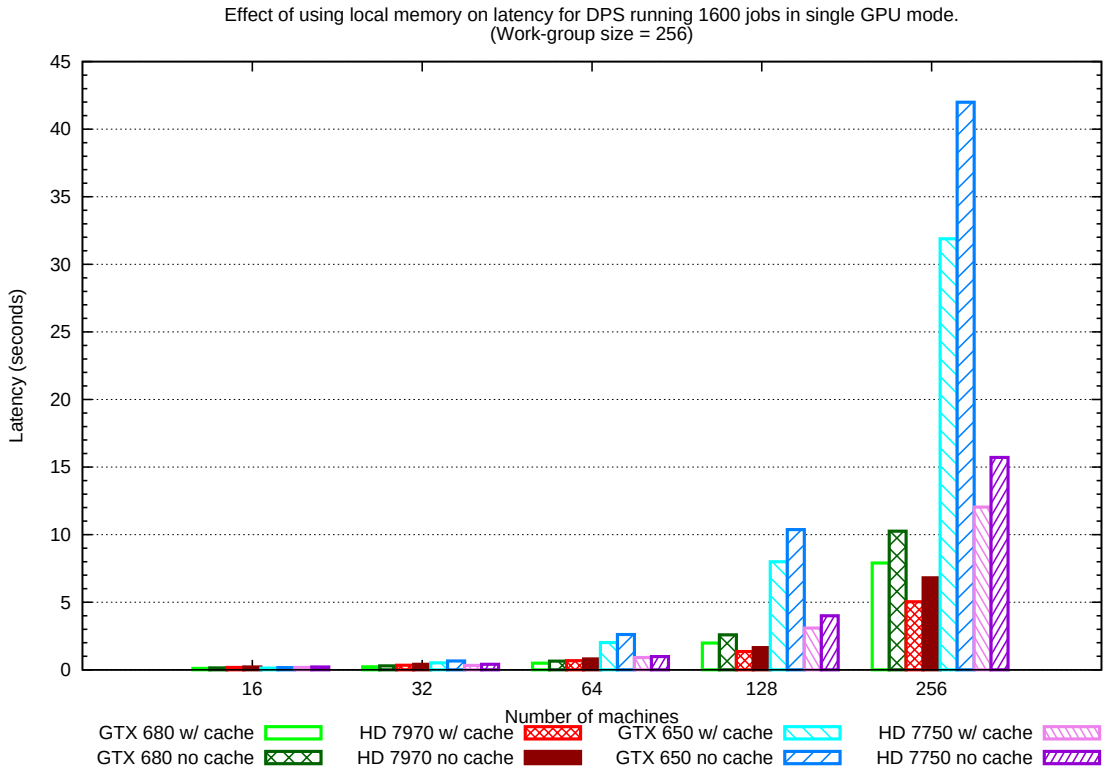


FIGURE 6.44: Comparison of latency for 1,600 jobs with and without using GPU local memory.

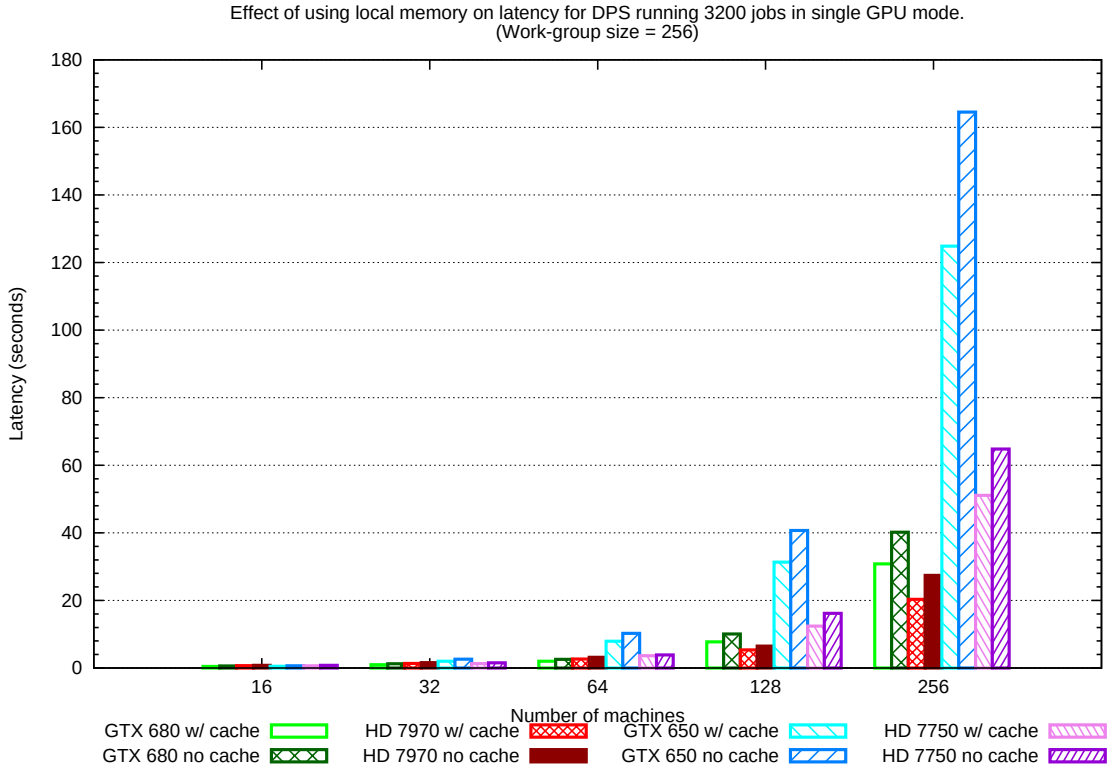


FIGURE 6.45: Comparison of latency for 3,200 jobs with and without using GPU local memory.

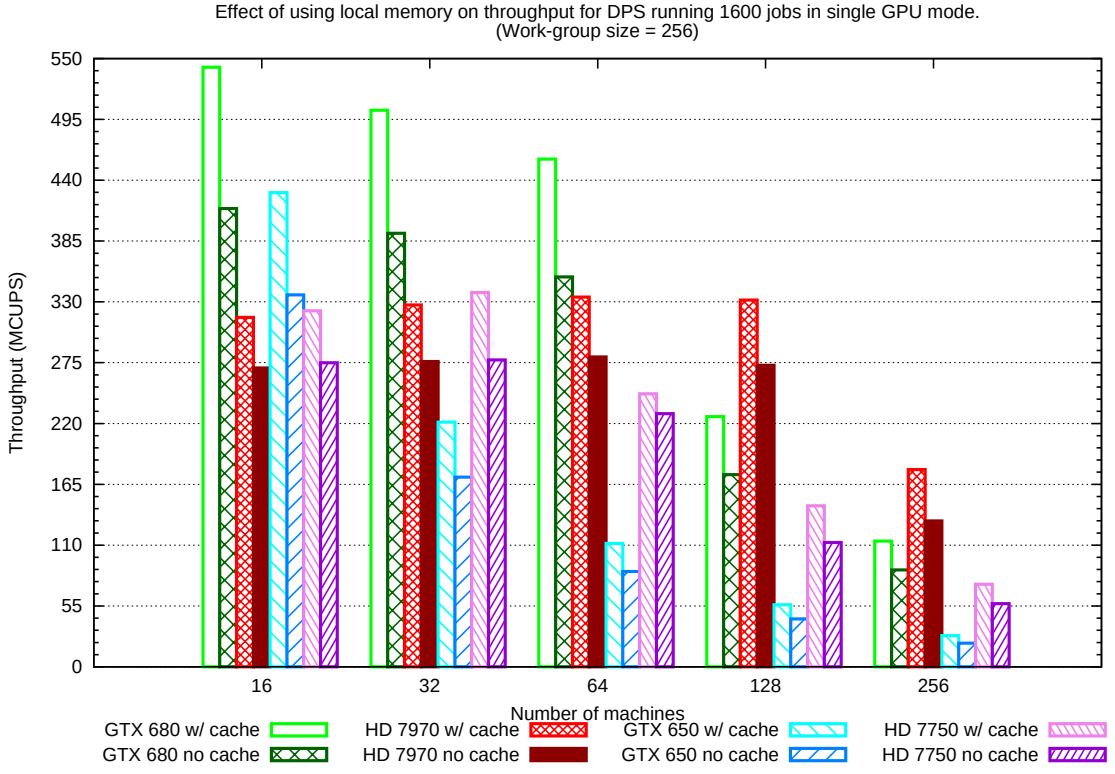


FIGURE 6.46: Comparison of throughput for 1,600 jobs with and without using GPU local memory. Throughput is measured in millions of cell updates per second (MCUPS).

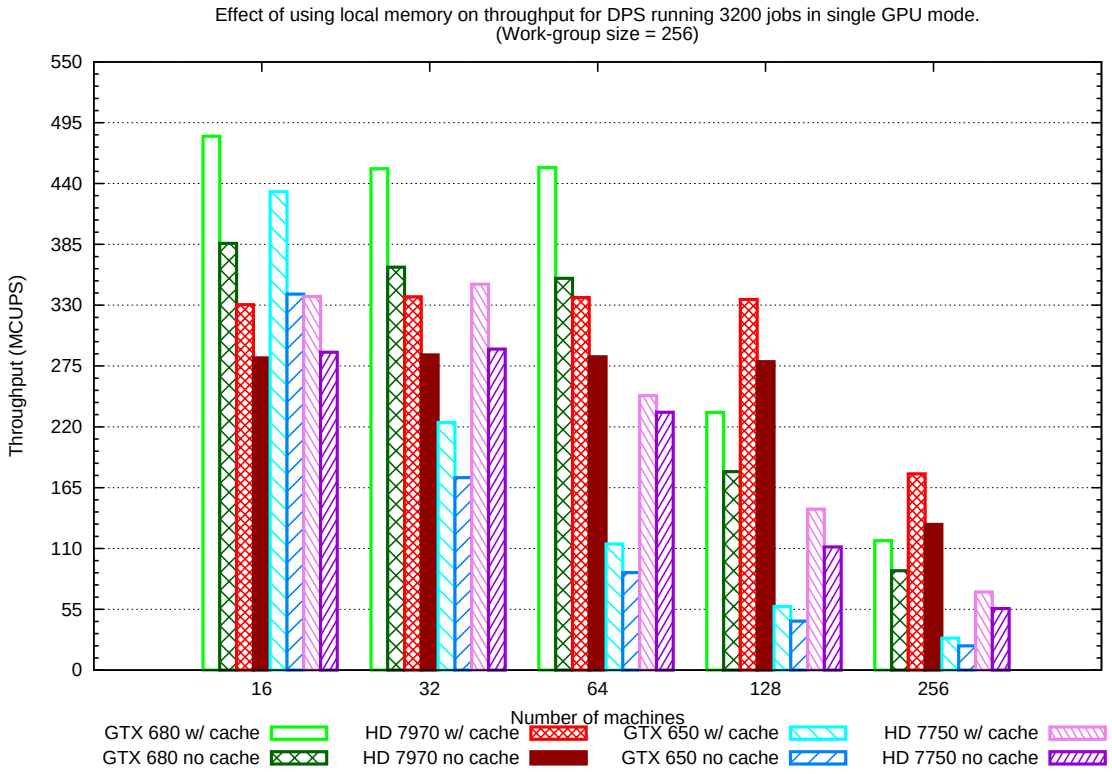


FIGURE 6.47: Comparison of throughput for 3,200 jobs with and without using GPU local memory. Throughput is measured in millions of cell updates per second (MCUPS).

0

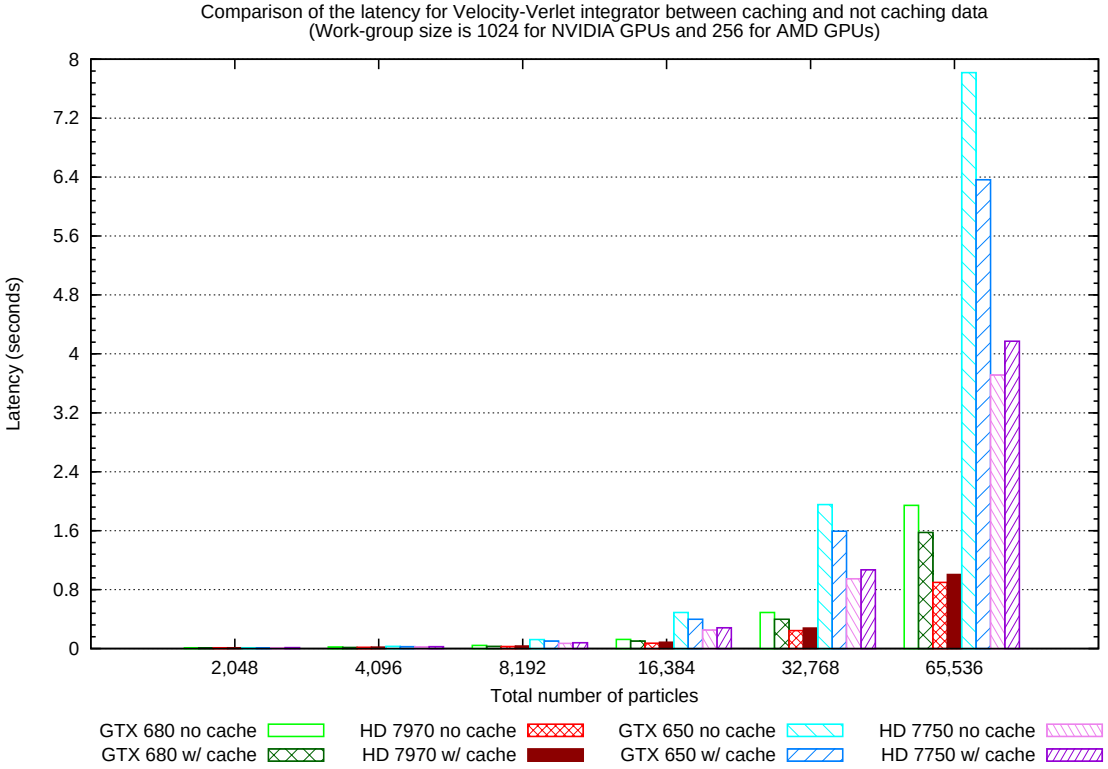


FIGURE 6.48: Results showing the effect of local memory on the latency performance of `Velvet-`*d* for all GPU devices. The work-group sizes in these results are 1024 for NVIDIA GPUs and 256 for AMD GPUs.

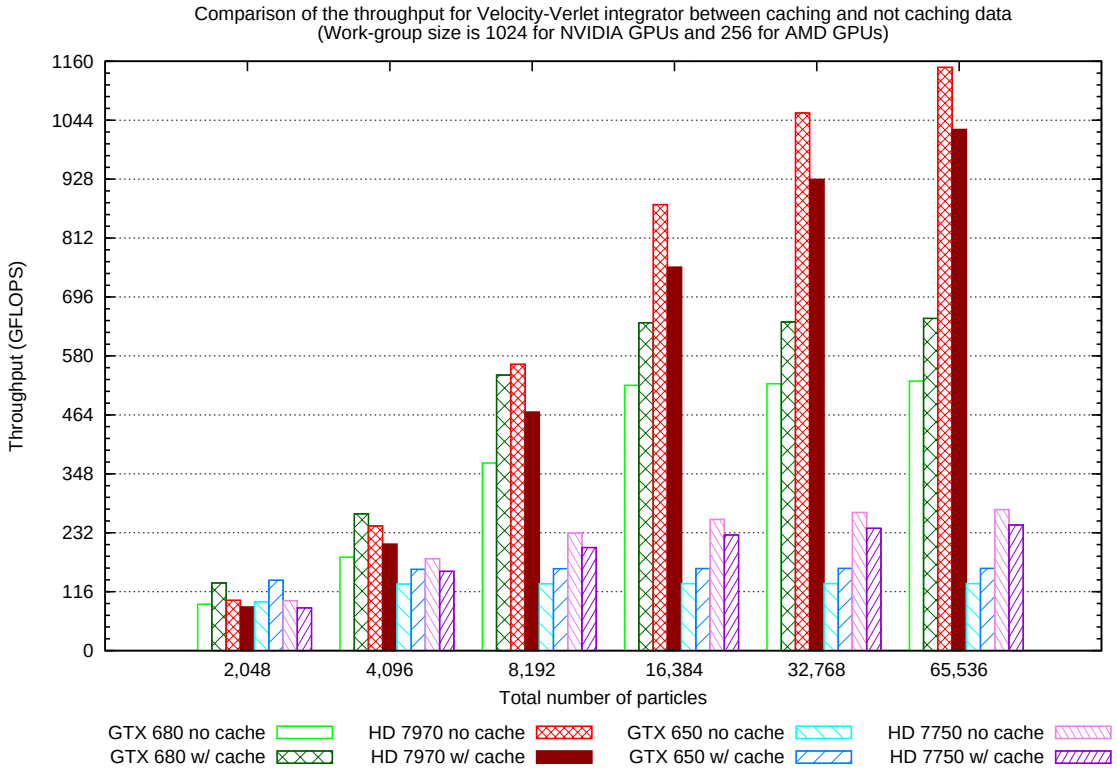


FIGURE 6.49: Results showing the effect of local memory on the throughput performance of `Velvet-`d for all GPU devices. The work-group sizes in these results are 1024 for NVIDIA GPUs and 256 for AMD GPUs. Throughput is measured in billions of floating-point operations per second.

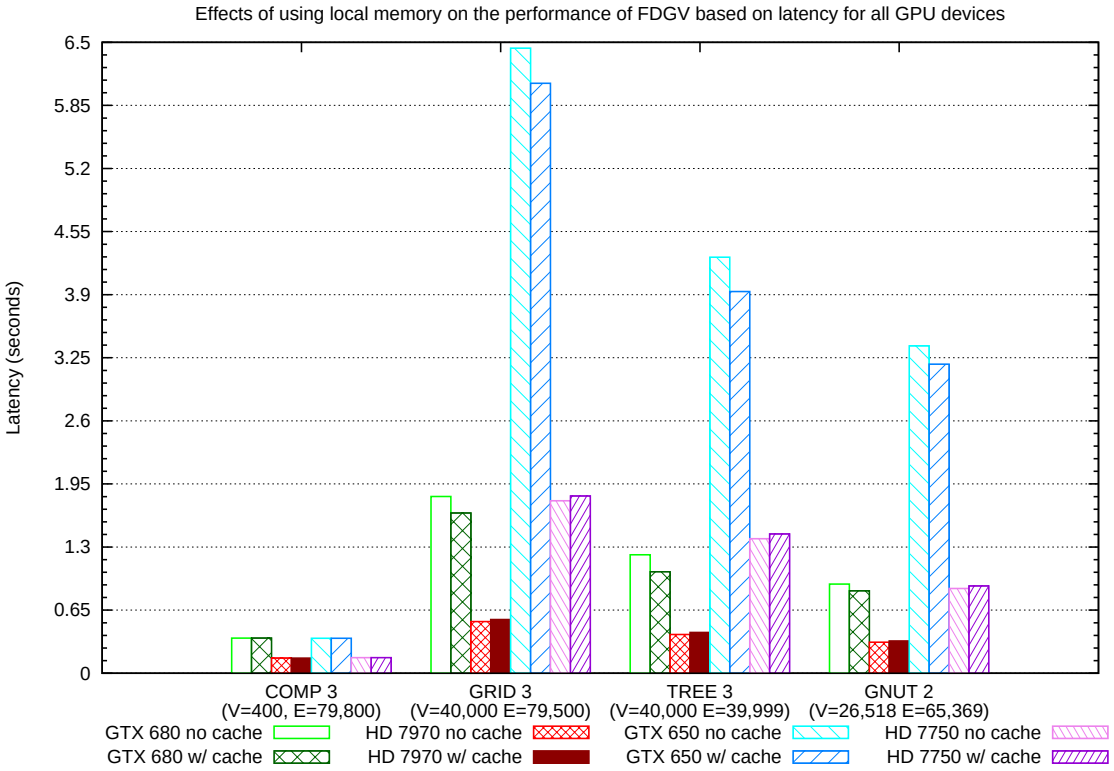


FIGURE 6.50: Results showing the effect of local memory on latency performance of `FDGV-`d for all GPU devices. The work-group sizes in these results are 512 for NVIDIA GPUs and 256 for AMD GPUs.

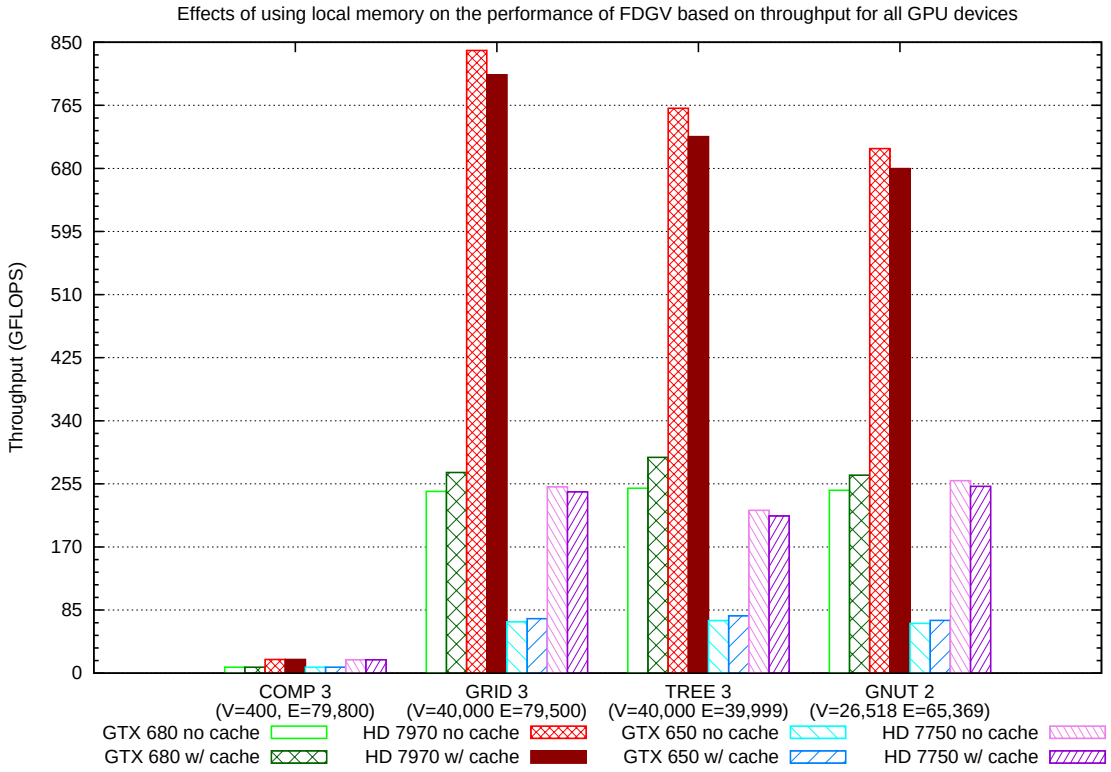


FIGURE 6.51: Results showing the effect of local memory on latency performance of `FDGV-`d for all GPU devices. The work-group sizes in these results are 512 for NVIDIA GPUs and 256 for AMD GPUs. Throughput is measured in billions of floating-point operations per second.

The latency performance for the AMD GPUs show that for each respective GPU, the difference between using local memory or not is almost negligible. However, throughput performance somewhat magnifies the difference and shows that not caching data local memory yields a very marginal gain. This is around 4% difference. The NVIDIA GPUs, on the other hand, perform around 10% better when local memory is used to cache data.

Conclusion In this section, we have seen some interesting results on how an an application can improve its performance on the GPU device by caching data to local memory in cases where there is a high volume of data re-use within the kernel. This is the true for `DPS-`d on all the GPU devices we tested. In addition, we have also seen a case where the same application performs differently across different GPU architectures. The n-body method applications, `Velvet-`d and `FDGV-`d seem to perform better on the NVIDIA hardware with local memory while the AMD hardware performs better without using local memory. This highlights the extent to which architectural differences might affect specific types of applications.

6.9.4 Results on benefits of pre-pinned memory and DMA

In this section, we will be considering the two applications, `Velvet-`d and `FDGV-`d, that have the possibility of benefiting from using pre-pinned memory and DMA. Recall that we can only use pre-pinned memory when the memory block to be pinned is not too large, hence, we are unable to do this with `GapsMis-`d and `DPS-`d. Here, we will be directly measuring the amount of time required to transfer data from the GPU device back to the host, that is, the communication latency, after execution has been completed. Since this experiment does not depend on a feature of the GPU devices, it is conducted using our designated reference machine, `KEPLER`.

Although the data being copied back from the device to host is relatively small, we do expect to observe a smaller transfer when using pre-pinned memory since the overhead associated with pinning memory is eliminated. Figures 6.52 and 6.53 illustrate the impact that DMA can have on performance of `Velvet-`d and `FDGV-`d, respectively, by measuring the time spent during data transfer from host to GPU global memory and vice versa. We observe a significant performance gain when using pinned memory.

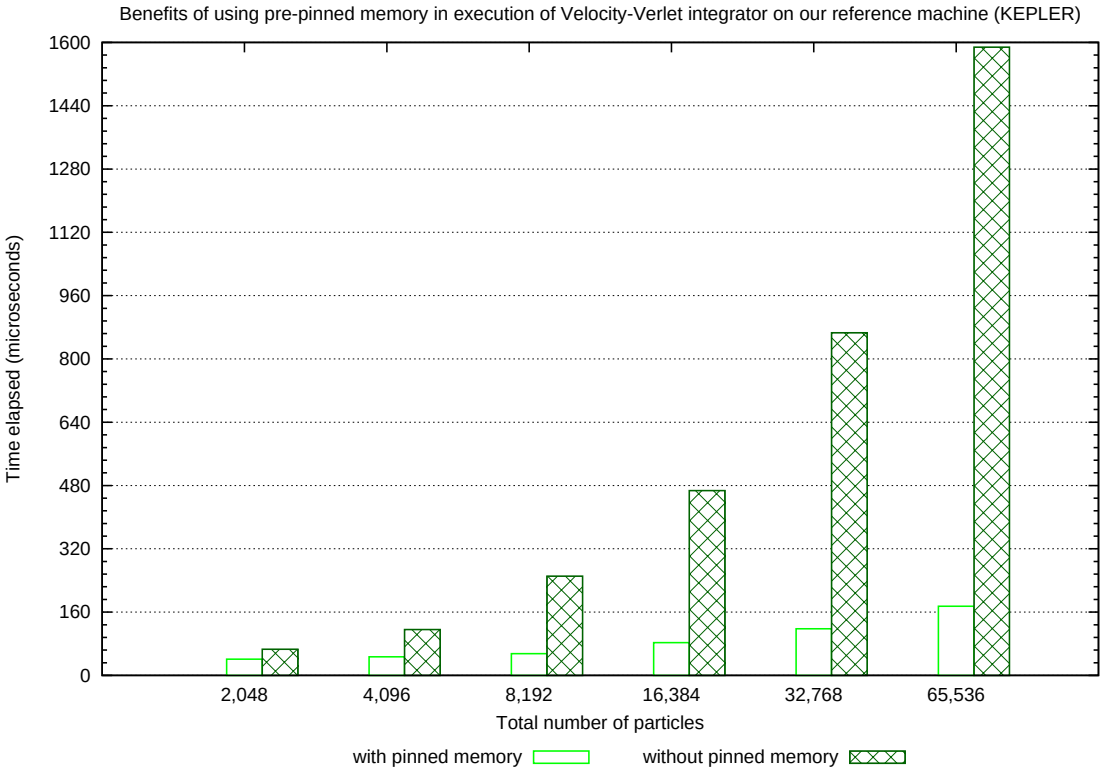


FIGURE 6.52: Results showing the benefits of using pre-pinned memory for `Velvet-`d and `FDGV-`d running on our designated reference machine. Time elapsed is given in microseconds.

Considering the `Velvet-`d application (Figure 6.52), the communication latency can be reduced by up to 88% by using pinned memory. While, for `FDGV-`d (Figure 6.53), this

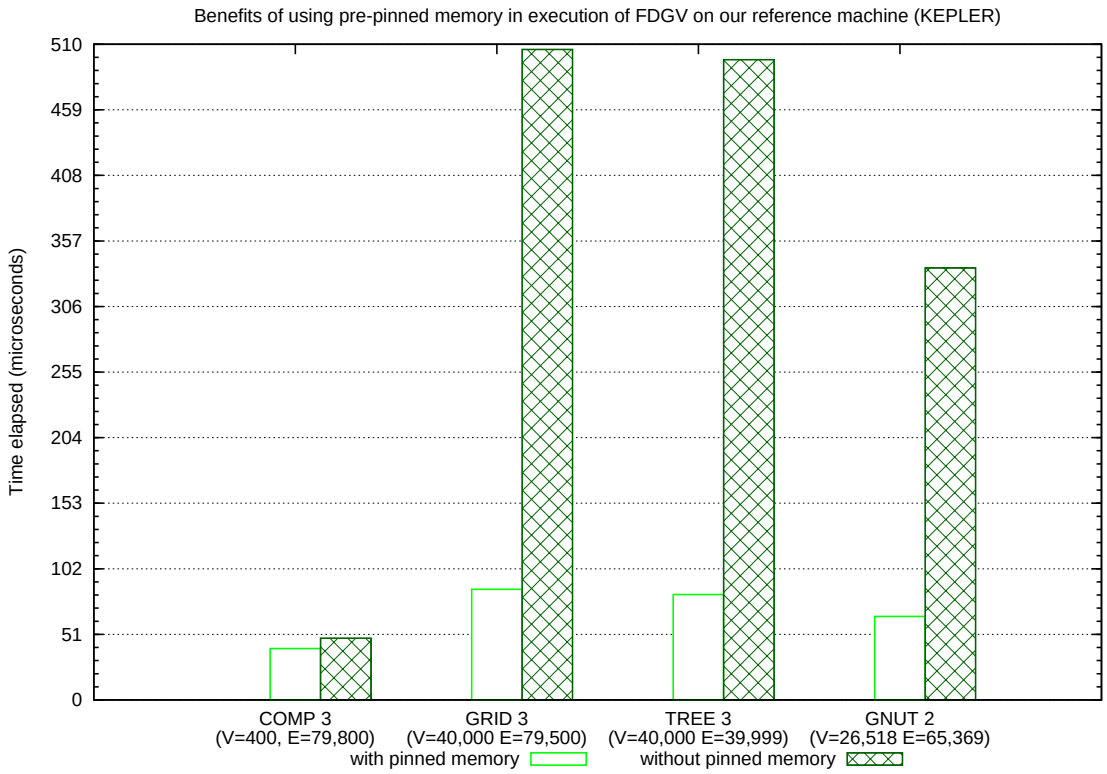


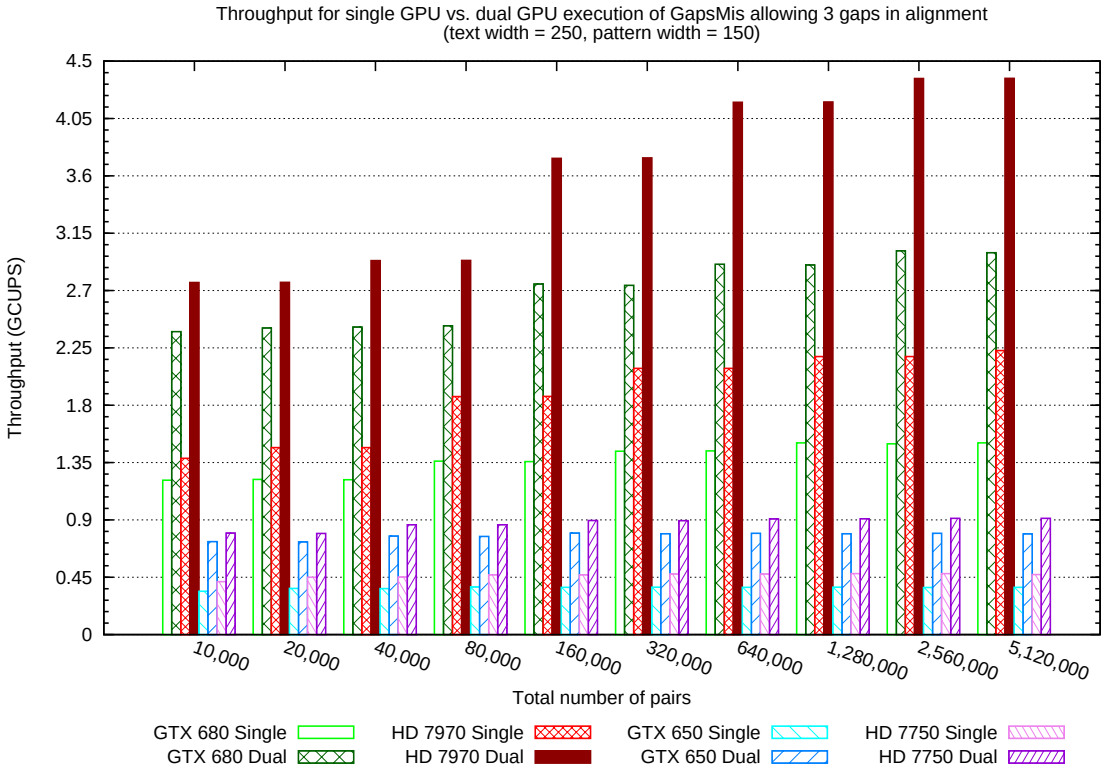
FIGURE 6.53: Results showing the benefits of using pre-pinned memory for `Velvet-`d and `FDGV-`d running on our designated reference machine. Time elapsed is given in microseconds.

performance gain in communication latency is around 83%, especially for the grid and tree graphs. Clearly, we can observe that having a pre-pinned memory on the host prior execution and copying data helps to speed up data transfer.

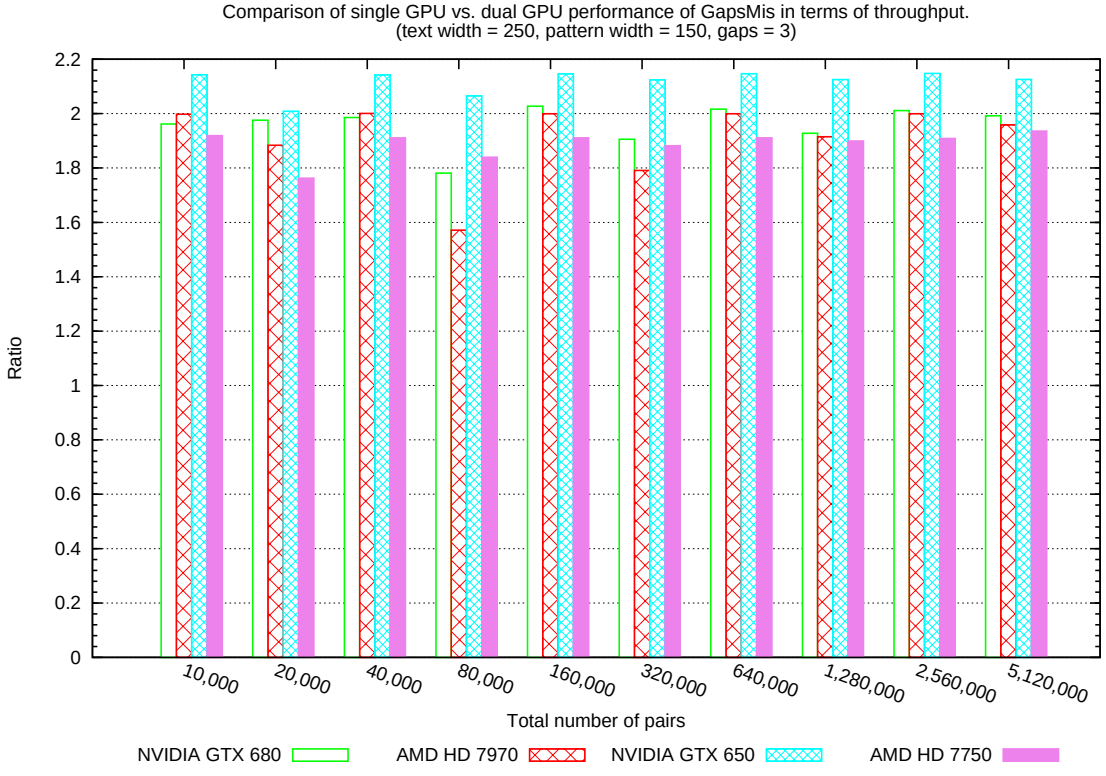
6.9.5 Results on application scaling with multi-GPUs

`GapsMis-`d. This application is designed to be able to, not just take advantage of multiple GPU devices, but can also distribute work evenly based on the global memory capacities of available GPUs. In essence, GPU devices with larger global memory are given more work in the form of a larger batch size than others with a smaller memory capacity. The aim is to have as much work as possible in the GPU devices in order to minimize effective latency incurred during device-host communications. In this case, the total number of pairs to be aligned is distributed among available GPU devices based on this condition.

Figure 6.53 shows the results based on throughput of `GapsMis-`d when using a single GPU compared to the throughput with dual GPUs.



(a) 250×150, 3 gaps (throughput)



(b) 250×150, 3 gaps (ratio)

When we consider Figure 6.54(b) and Figure 6.54(d) that quantify the performance difference between single and dual GPU modes, the first major observation is that `GapsMis-`*d*

scales better on the NVIDIA GPUs. We record a boost in throughput of around 80% to over 100% for the mid-range NVIDIA GTX 650 in dual GPU mode. In general, the results are very impressive and promising as we observe a boost in throughput of around 60% to over 100% when running `GapsMis-`d in dual GPU mode for all GPU devices.

For a tool like `GapsMis-`d with the potential to be a very practical tool among software used for alignment and sequencing, it is good to see that the application can scale properly and perform better when more GPU devices are available to it. This becomes very viable and useful especially when the amount of alignment task is huge.

`DPS-`d. The first layer of parallelism in `DPS-`d is achieved by mapping each machine to a work-group. As we already know by now, the higher the number work-groups that can execute concurrently the quicker the GPU can complete a compute task. Therefore, `DPS-`d takes advantage of multiple GPU devices such that the number of machines in the problem is distributed across the combined work-groups defined on all GPU devices. However, the addition of more GPU devices does not always necessarily translate to a boost in performance due to reasons that relate to the problem instance. We hope to demonstrate this and analyse the reason why this could be the case for `DPS-`d. We present the results comparing the throughput of `DPS-`d in single GPU mode with with the throughput when using dual GPU devices in Figures 6.54 and 6.55.

Let us focus our attention on the high-end GPUs, GTX 680 and HD 7970. We observe that there is not much performance gain in adding a second GPU when the number of machines is below 256 for the HD 7970 and 128 for the GTX 680. For instance, when the number of machines is 16, the NVIDIA GTX 680 GPU achieves 10% more throughput in single GPU configuration compared to the dual GPU configuration. This is not particularly surprising because these two GPU devices have enough compute units to accommodate that many work-groups running concurrently. For this reason, a single GPU still offers the same (or even better) performance than dual GPU devices. Then when the number of machines is large enough we begin to notice a considerable boost in performance of up to 85%, or even above 100%, as is the case with the mid-range GTX 650 GPU, as shown in Figure 6.54(b) and Figure 6.55(b). In general, the overall results demonstrate that when the problem instance is large enough, `DPS-`d can scale well with the addition of more GPU devices.

`FDGV-`d. In order to take advantage of a second GPU, we simply share the total number of vertices and edges between the two GPU devices. As we observed earlier in the analysis of `DPS-`d with regards to the relationship between problem size and GPU device usage, using more than one GPU when the size of the graph is small will only affect performance

in a negative way. This is illustrated in the results shown in Figure 6.56, especially for our complete graph sample.

Figure 6.56(b) shows that our complete graph sample is too small to benefit from an extra GPU, thus demonstrating the fact that adding extra GPU devices will not always yield a boost in performance. From a general point of view, these results show that the GPU devices are capable of handling these graphs when running in a single device mode. This is confirmed when we observe the results for the GPU with the least amount of compute resources, that is, the GTX 650. When two GTX 650 GPU devices are used we gain a performance boost in throughput of up to 100%. The other GPUs gain around 50% to 70% in performance suggesting that the combined compute resources on both GPUs have not been saturated yet.

Conclusion. In this section we investigate how applications can scale with the addition of more GPU devices. The results obtain are consistent for each type of application and the rate at which these applications scale is nearly linear in the number of GPU devices used. In some few cases we even observed performance improvement of over 100% when running with two GPUs. However, the amount of performance boost achievable is still strongly related to the application.

6.9.6 Results on comparison of CPU vs. GPU performance

In Section 6.3 we discussed the implementation details of the applications discussed in this chapter and also presented some theoretical analysis of the parallel implementations to determine how efficient and/or optimal they are. For this section, we present an empirical analysis of the sequential, task-parallel and data-parallel implementations. However, our main focus will be on how our data-parallel implementations compare with the implementations on the CPU. In other words our aim is to evaluate how much performance boost we can achieve for these algorithms if the GPU device is used. Our comparison will be based on execution time and energy efficiency.

In order to obtain an estimate of the amount of power a device consumes during the execution of a particular application, we execute each application repeatedly for a set amount time, enough for the software profiler tool to log the power readings to file. We take this approach because during the experiment phases, some problem instance finish to quickly before a reliable power reading can be recorded. Figure 6.57 shows the power consumption profile for each application.

For the experiments, these figures are used to compute the energy consumption, in *Watt-second*, of a device after execution as a product of power consumed and latency.

6.9.6.1 `GapsMis`

The `GapsMis` tool focuses on performing millions of alignment tasks in a single execution in order to save time. Since the algorithm itself is not particularly compute-intensive when we consider the alignment of a single pair of sequences, this makes it well suited for a task-parallel implementation as well as for a data-parallel implementation. However, the data-parallel implementation does offer that extra layer of parallelism within the computation portion itself.

Although we expect to see significant amounts of speed-up when we compare latency on the GPU device versus latency of the CPU implementations, we expect that overall performance will be very close when we compare effective latency.

As expected, the GPU devices achieve a considerable amount of speed-up on latency over the CPU especially in the case of the sequential implementation and a single GPU device. Figure 6.58(a) shows that the high-end GPU devices are between 10 and 17 times quicker than the `GapsMis-`s with 1 CPU thread. On the other hand, the high-end GPU devices are only around 2.5 to 5 times faster, when paired up, compared to `GapsMis-`t with 12 CPU threads. However, when we compare the overall application performance (Figure 6.58(b) and Figure 6.59(b)), we observe that performance is nearly the same due to the massive amounts of overhead incurred when the backtrack phase is being carried out on the CPU. The preformance gain when using dual high-end GPU devices is only between 20% and 48%. This is the main bottle-neck in this application and only the high-end GPUs offer some speed-up.

Since the computation of the final results, that is backtracking phase, is offloaded to the CPU after the matrices have been computed by the GPU, it is difficult to get an accurate comparison in terms of energy consumption. For that reason we will not include energy results for this application.

6.9.6.2 `DPS`

The `DPS` application can be considered a compute-intensive application when we consider the number of cells to compute in the dynamic programming table. Since we are able to achieve very significant amounts of parallelism in the data-parallel implementation, we expect to achieve huge amounts of speed-up for this application when using the GPU device, even when we consider the overall application performance. Figure 6.57 shows the results comparing CPU running time with running time on GPU and it confirms our expectations.

The fact that the GPU devices perform very well in this application is not particularly surprising because it demonstrates the fact that if a true data-parallel formulation exists for an algorithm, it can benefit greatly from using a GPU device. This fact is demonstrated in Figure 6.60(b) and Figure 6.58(e), where we see that even a single GPU device performs better than the task-parallel implementation by a significant margin. For example, the entry-level AMD HD 7750 GPU achieved a speed-up of around 10 times, in single GPU configuration, over `DPS-`t using 12 CPU threads. In addition, the overall performance of an application will also be improved greatly when the GPU device can perform vast majority of the computation involved.

Since we have now established that the data-parallel implementation gives us massive amounts of performance boost over the sequential and task-parallel implementations, we will now compare how energy-efficient these implementations are. Figure 6.57 shows the efficiency of the implementations in terms of performance-per-watt and energy consumed.

The results show that the GPU devices are indeed very energy efficient and watt-for-watt comparison with the CPU shows that the GPU does considerably more work per watt than the CPU implementations, illustrated in Figure 6.58(d). Here we observe that the GPU devices are able to achieve between 1 and 5 MCUPS per Watt in both single and dual configurations, compared to the CPU which was able to achieve around 1.2 MCUPS per Watt but in the smallest problem size of 16 machines.

This is mainly down to the fact that the GPU devices are very quick in performing computations. Therefore, energy efficiency does not necessarily depend only on the power consumption rating of a device but also how much work a device is able to perform for each unit amount of power.

6.9.6.3 `Velvet`

Being an n-body method, we expect to achieve huge amounts of performance gain because the requirement of the underlying algorithm is very well suited to the massively parallel architecture of the GPU device. Given the high amounts of throughput the GPU can achieve in these types of applications, we expect that the GPU device will be very energy-efficient too. Figure 6.58(a) shows the results for the amount of speed-up achieved in terms of latency, and in Figure 6.58(b), we compare the efficiency of both devices.

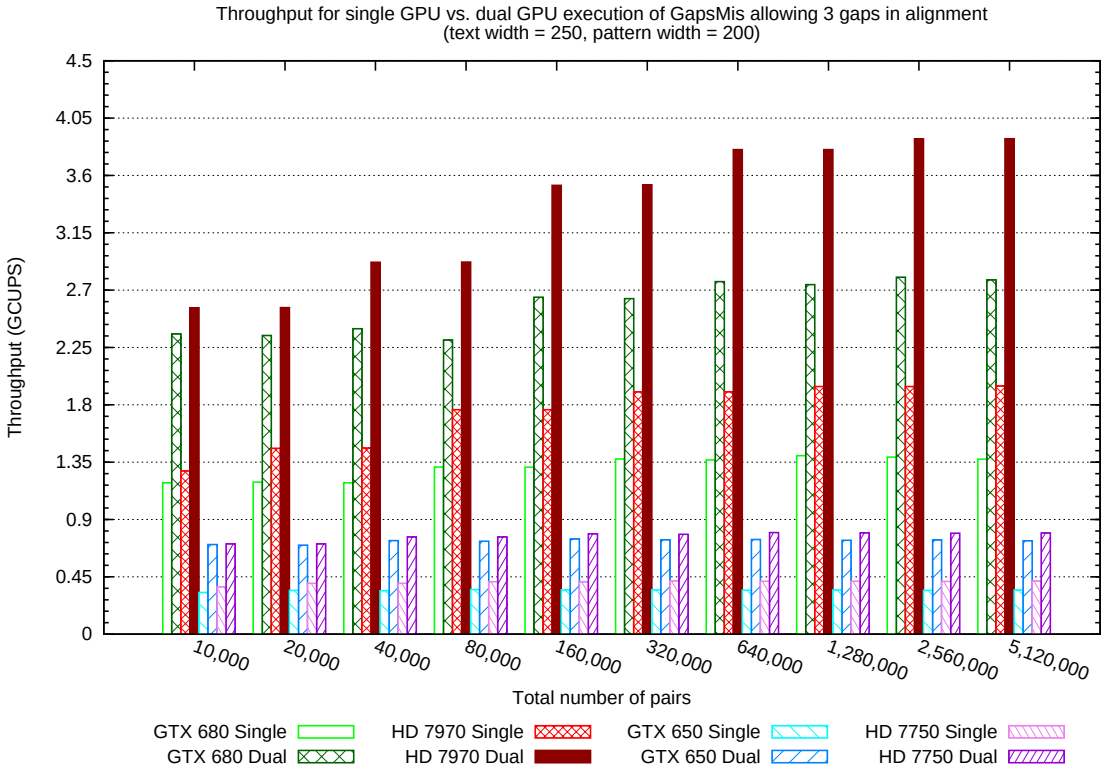
In Figure 6.58(a), we observe that the single GPU configuration achieves massive amounts of speed-up. For example, in the simulation with 65,536 particles, the entry-level AMD HD 7750 GPU is around 40 times quicker than the multi-threaded `Velvet-`t with 12 CPU threads. In the same simulation, the high-end GPUs are around 380 times to 700 times

quicker. These results demonstrate the fact that this application is best run on a device like a GPU because of its demand for high levels of throughput, which is what a GPU architecture is optimized for. We present the energy consumption results for both devices in Figure 6.57. The GPU devices are not just quicker but highly energy-efficient too. We observe that the GPU devices are able to reach between 1 GFLOPS per Watt and 10 GFLOPS per Watt. However, the CPU is only able to peak at around 1.1 GFLOPS per Watt.

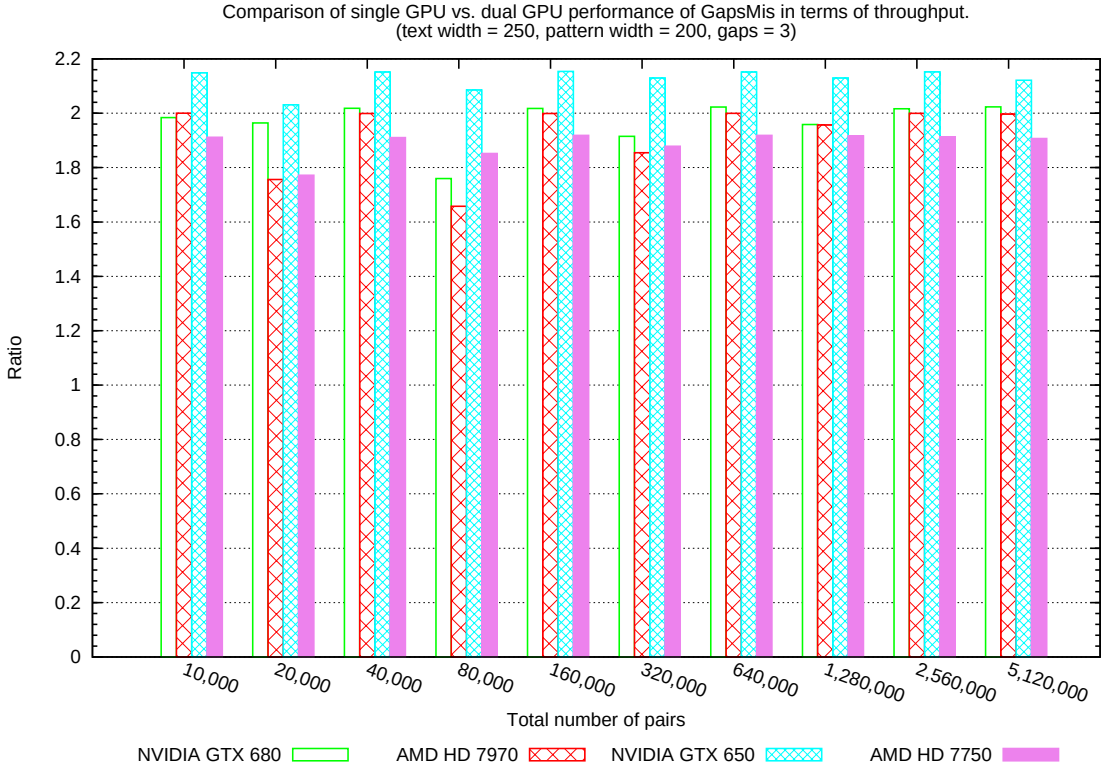
6.9.6.4 `FDGV`

For this application, we also expect to achieve very significant amount of performance improvement with the GPU devices over the CPU implementations. However, the amount of performance gain that can be achieved is expected to vary with the size of the graph and even the type of graph. For instance, considering our previous evaluation of `Velvet` in Section 6.9.6.3, we expect that execution of graphs with a considerably large amount of vertices will perform a lot better on the GPU than it will on the CPU. On the other hand, a graph with very small amount of vertices but has a large amount of edges might result in very similar performance on both CPU and GPU devices. This is because the work-items in the GPU implementation require an extra step to iterate through the edge list in order to accumulate all the partial edge displacements computed during the attraction phase. In Figure 6.57, we present the results comparing the ratio of the execution time of CPU implementations with the latency and effective latency of the data-parallel implementation.

As we expected, the results show that there is not a lot of difference between the performance of the CPU and the GPU device for our complete graph sample, even when comparing with the sequential implementation. This CPU is quick enough to iterate through the edges of this graph. However, when we compare performance on other graph types then the GPU device pulls away significantly because of their high vertex count. For instance, in Figure 6.58(b), we observe that the single GPU device is around 94 times quicker than `FDGV-`t in the grid graph simulation, and, around 126 times and 69 times quicker in the tree and Gnutella network simulations, respectively. Once again this demonstrates how application of this type is highly suited to a GPU device. When we compare the overall application performance when using the GPU device, using effective latency performance metric, we still achieve very similar performance gain because all computation is done on the GPU and the amount of data transferred after execution is very small. Figure 6.57 shows the results for energy consumption and efficiency.

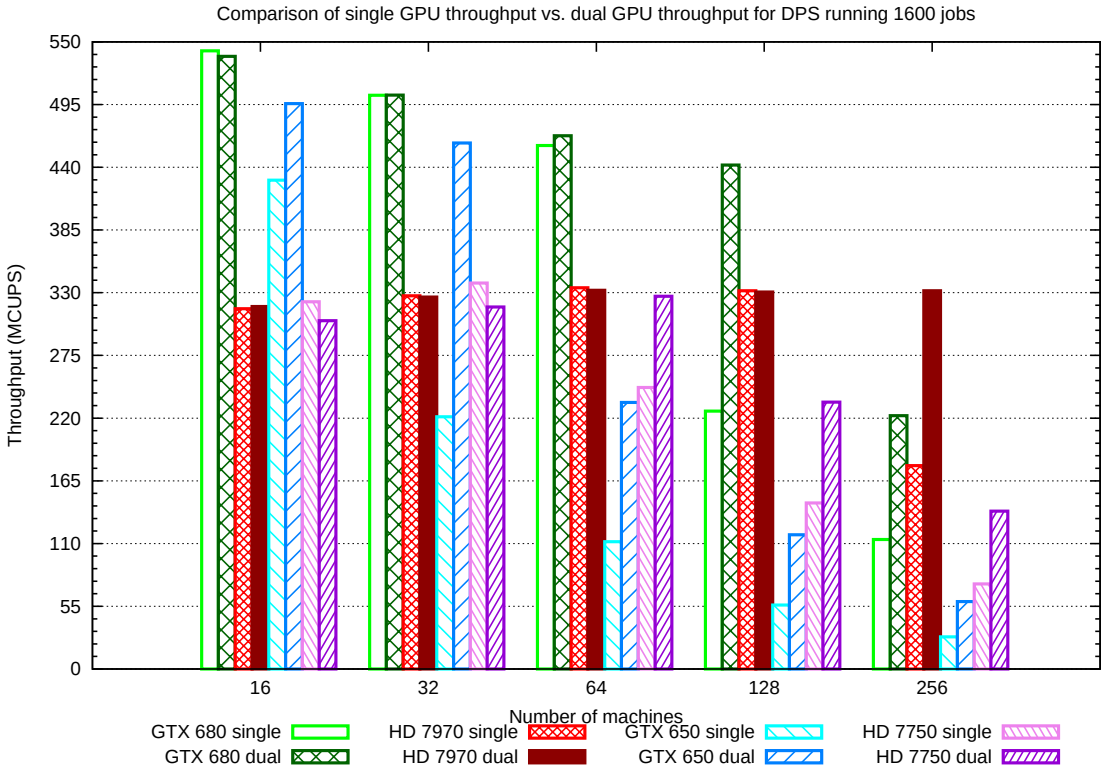


(c) 250×200, 3 gaps (throughput)

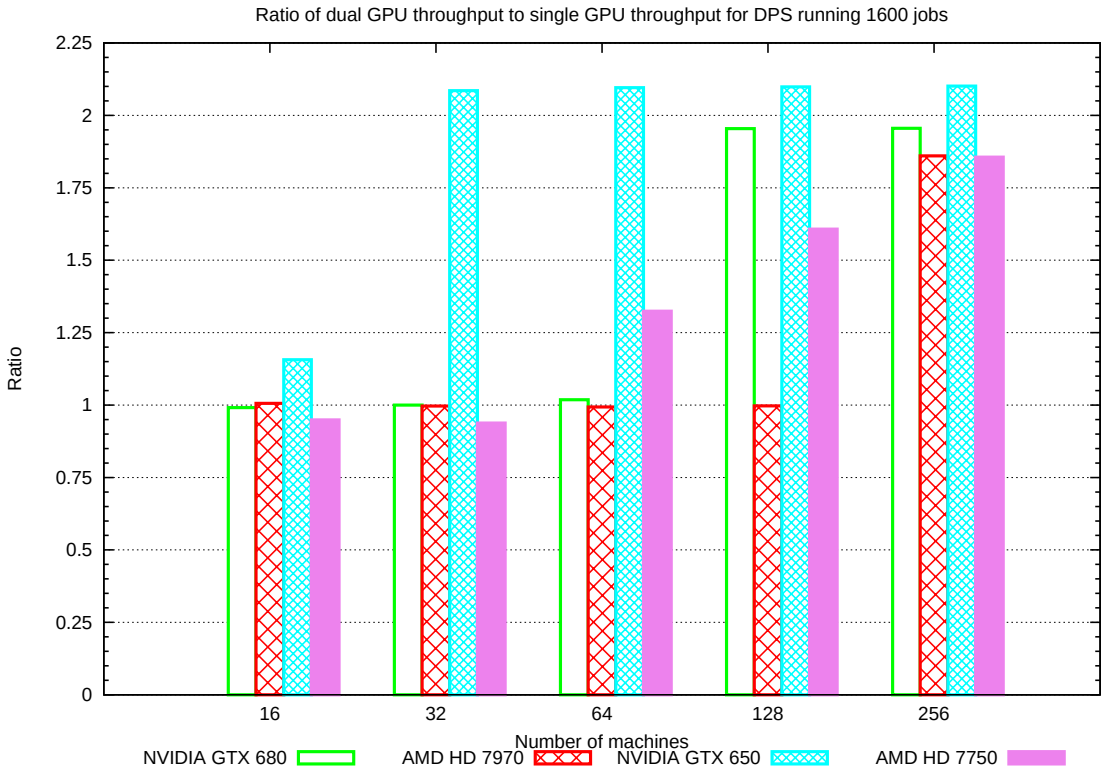


(d) 250×200, 3 gaps (ratio)

FIGURE 6.53: Comparison of how `GapsMis-`*d* scales with the addition of a second GPU device. Results shown here are for an alignment that allows 3 gaps. Throughput is measured in billions of cell updates per second (GCUPS)

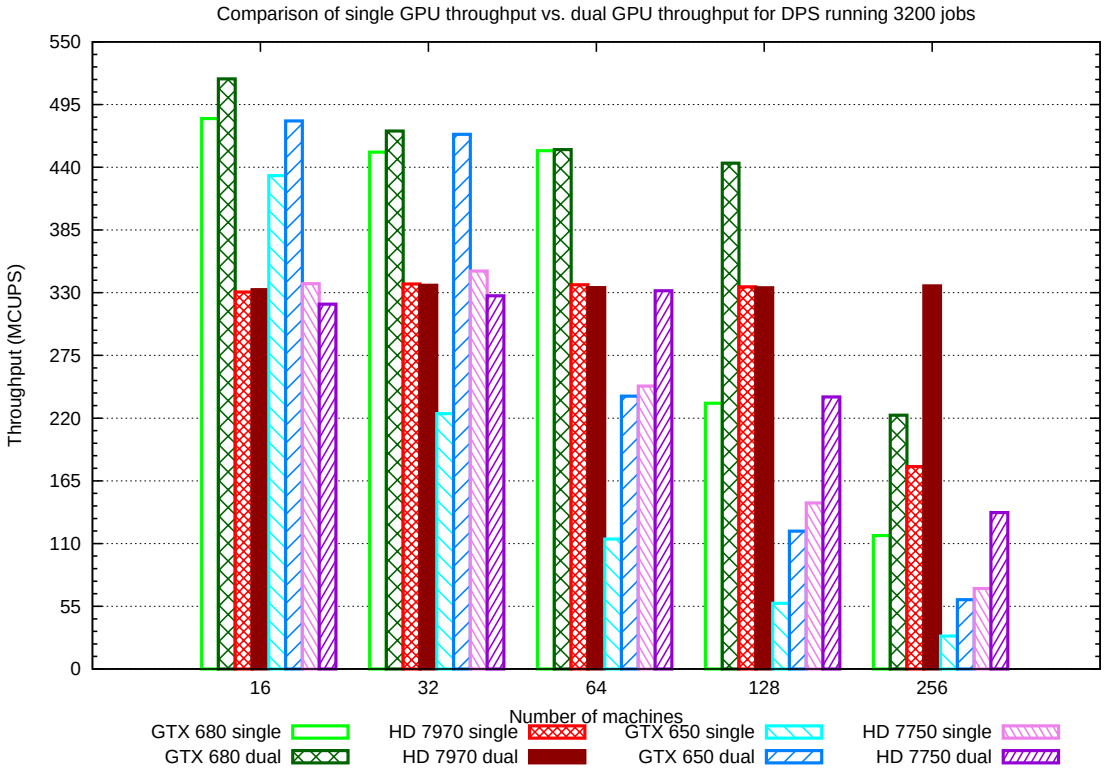


(a) 1600 jobs (throughput)

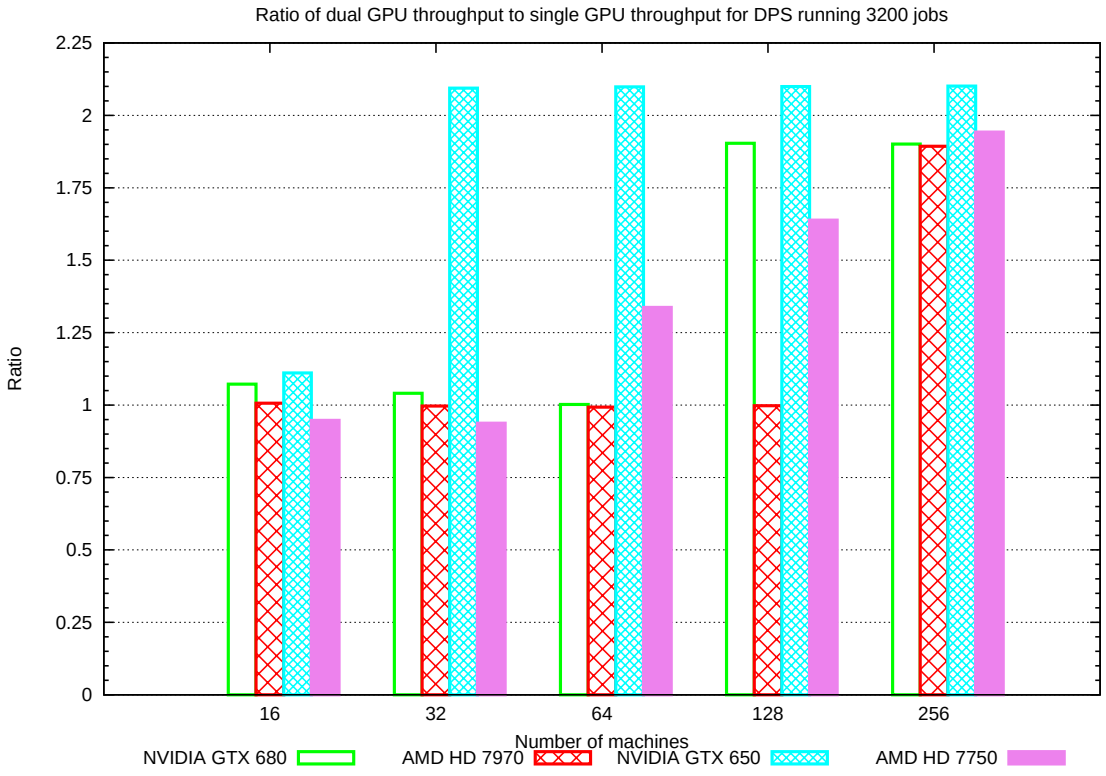


(b) 1600 jobs (ratio)

FIGURE 6.54: Throughput performance comparison of how DPS-*d* scales with the addition of a second GPU device for simulation with 1,600 jobs. The work-group size used for these results is 256 for all GPU devices.

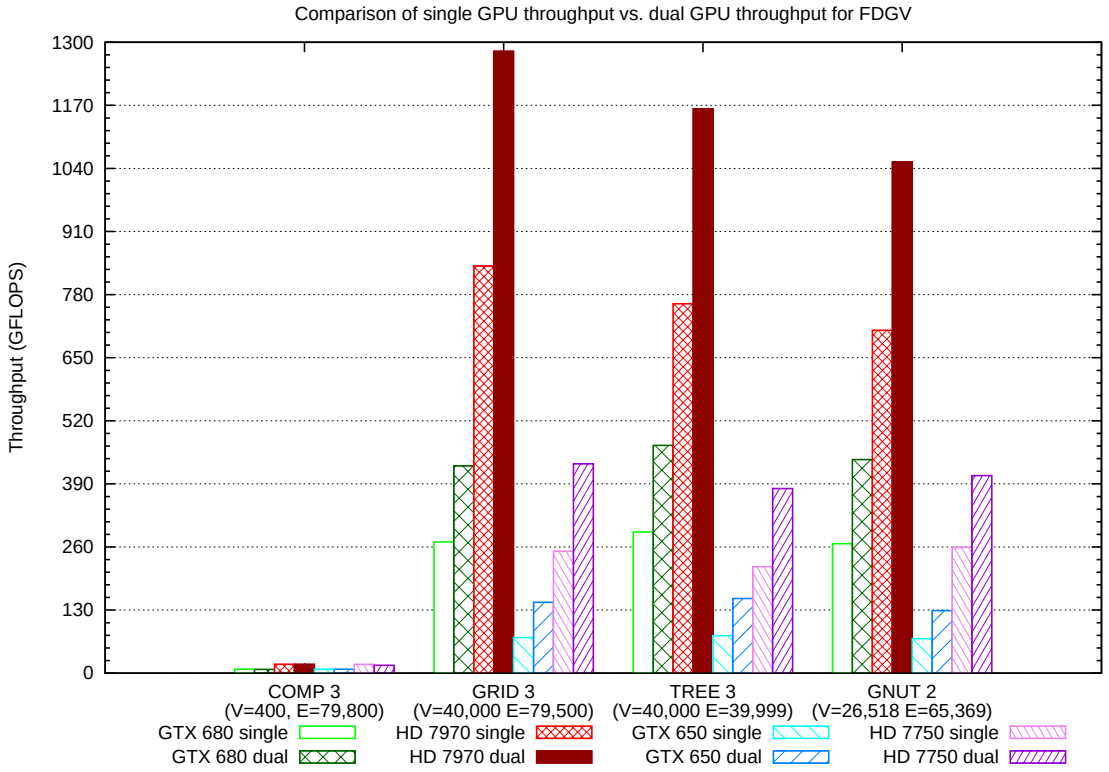


(a) 3200 jobs (throughput)

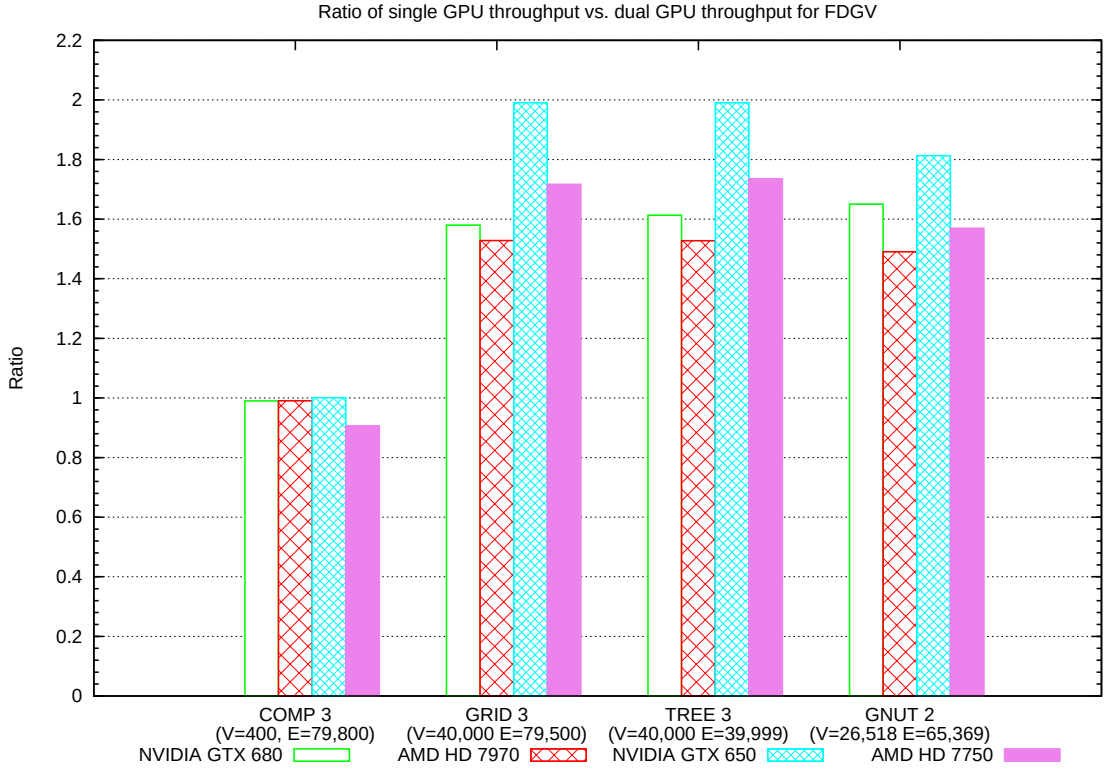


(b) 3200 jobs (ratio)

FIGURE 6.55: Throughput performance comparison of how DPS-*d* scales with the addition of a second GPU device for simulation with 3,200 jobs. The work-group size used for these results is 256 for all GPU devices.

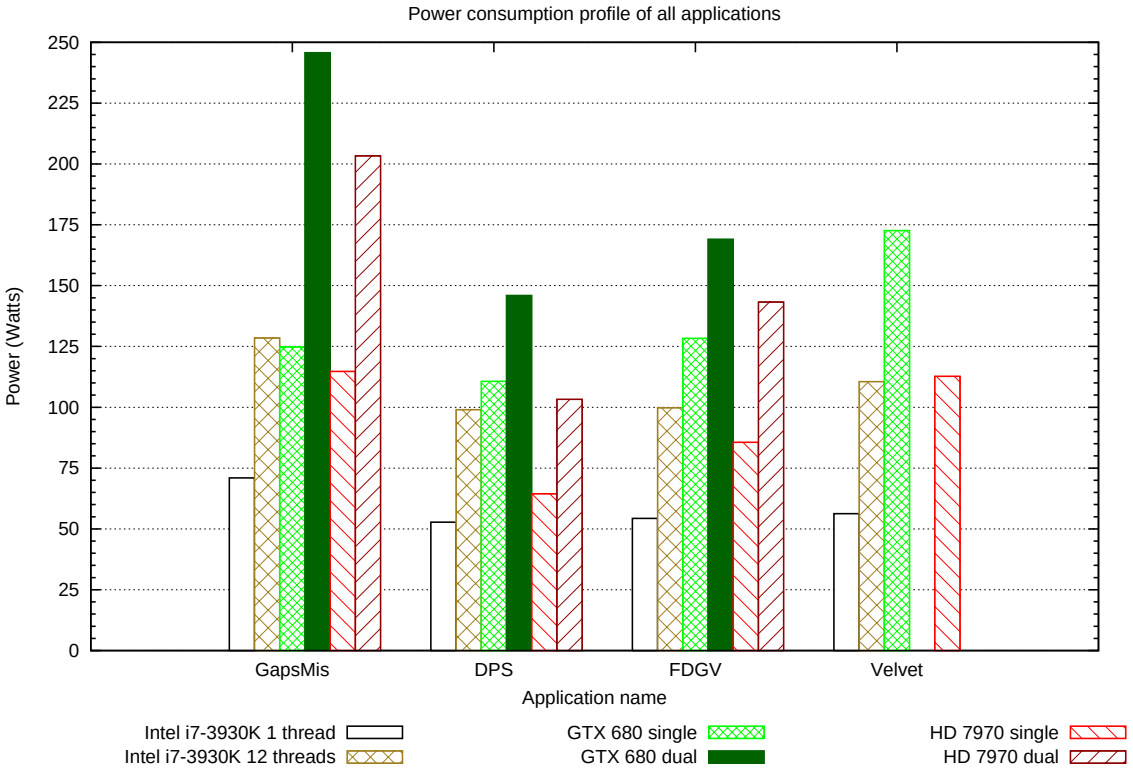


(a) Throughput



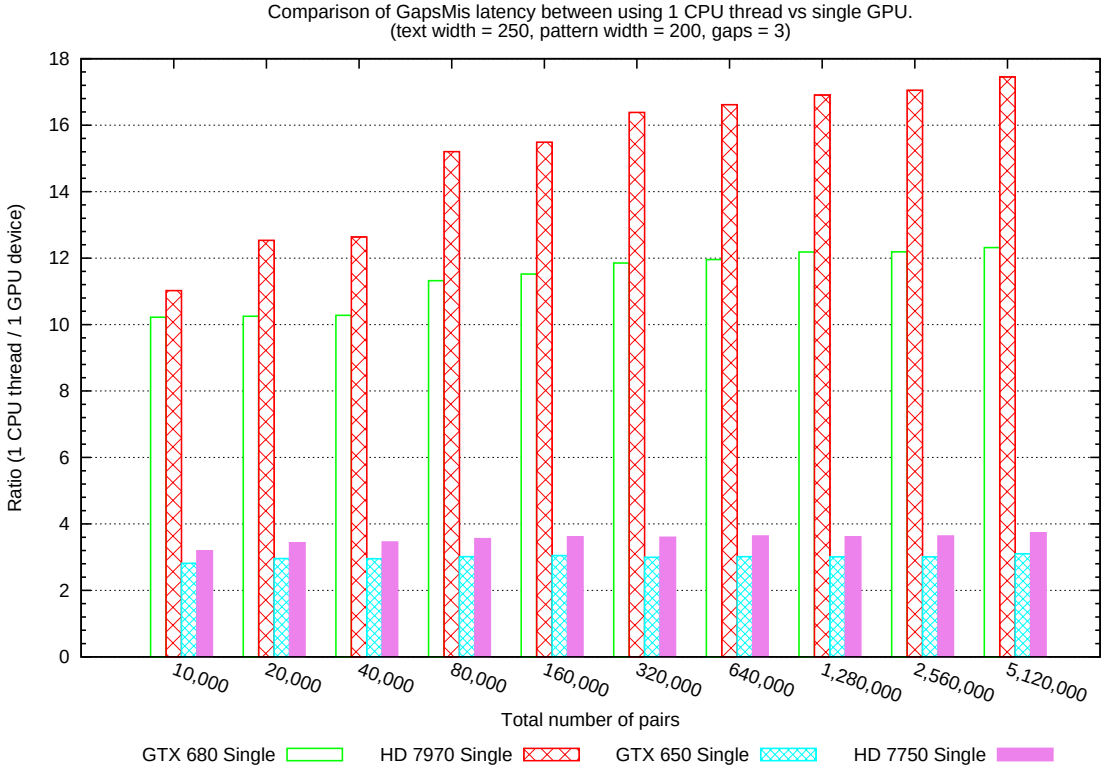
(b) Ratio

FIGURE 6.56: Comparison of how `FDGV-`d scales with the addition of a second GPU device. The results for NVIDIA GPUs are obtained using a work-group size of 512 and using local memory. The AMD GPUs use a work-group size of 256 and without using local memory. Throughput is measured in billions of floating-point operations per second.

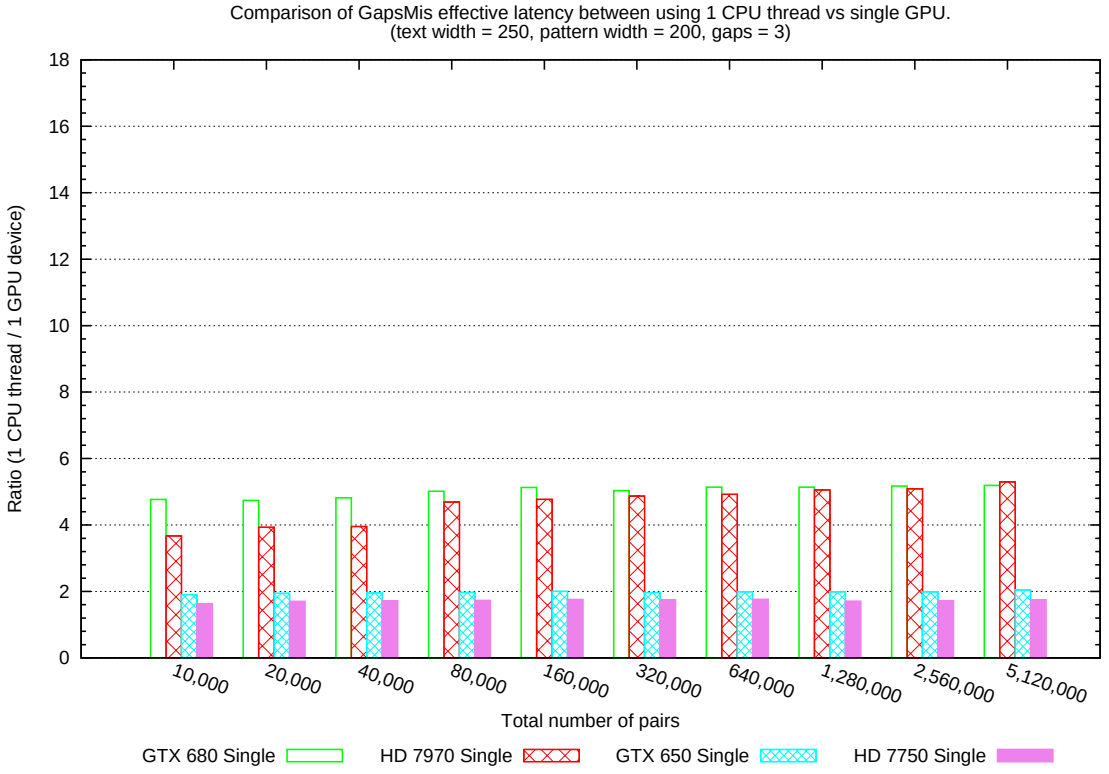


(a) Power consumption in Watts

FIGURE 6.57: Results of power consumption profiling for each application on each device configuration.

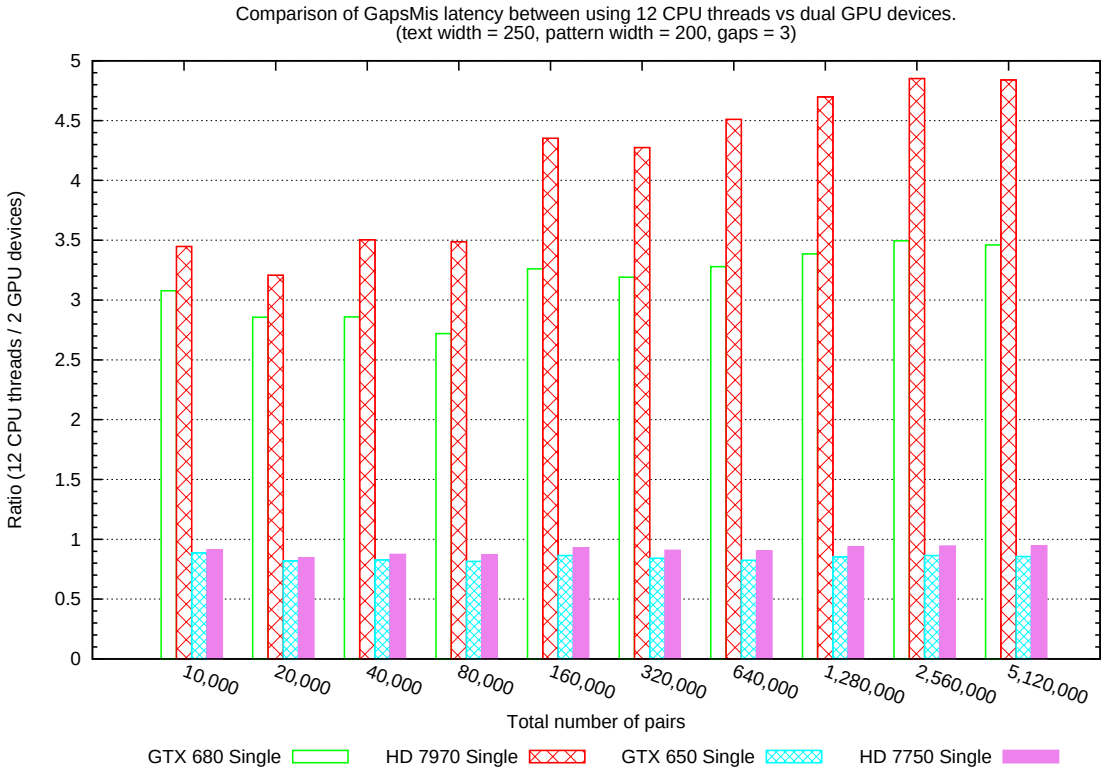


(a) Latency (1 CPU thread vs 1 GPU)

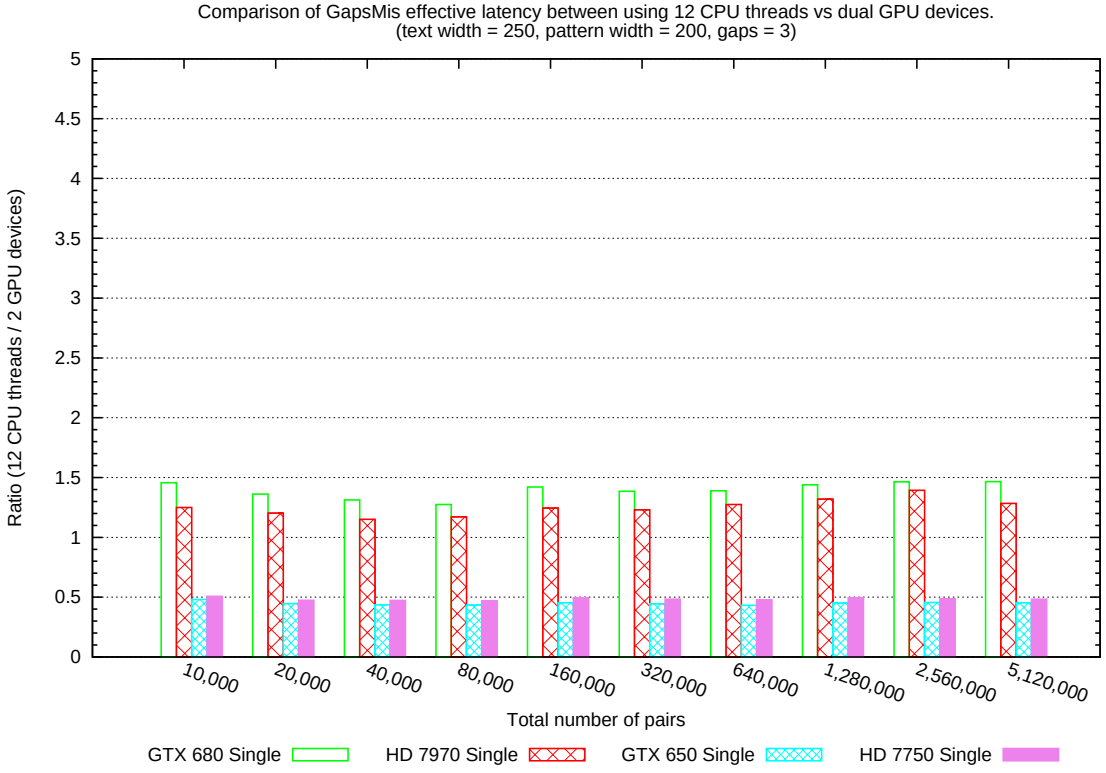


(b) Effective latency (1 CPU thread vs 1 GPU)

FIGURE 6.58: Latency performance of `GapsMis-`s vs `GapsMis-`d on single GPU for a 3-gap alignment. The length of target sequences is 250 and 200 for query sequences.

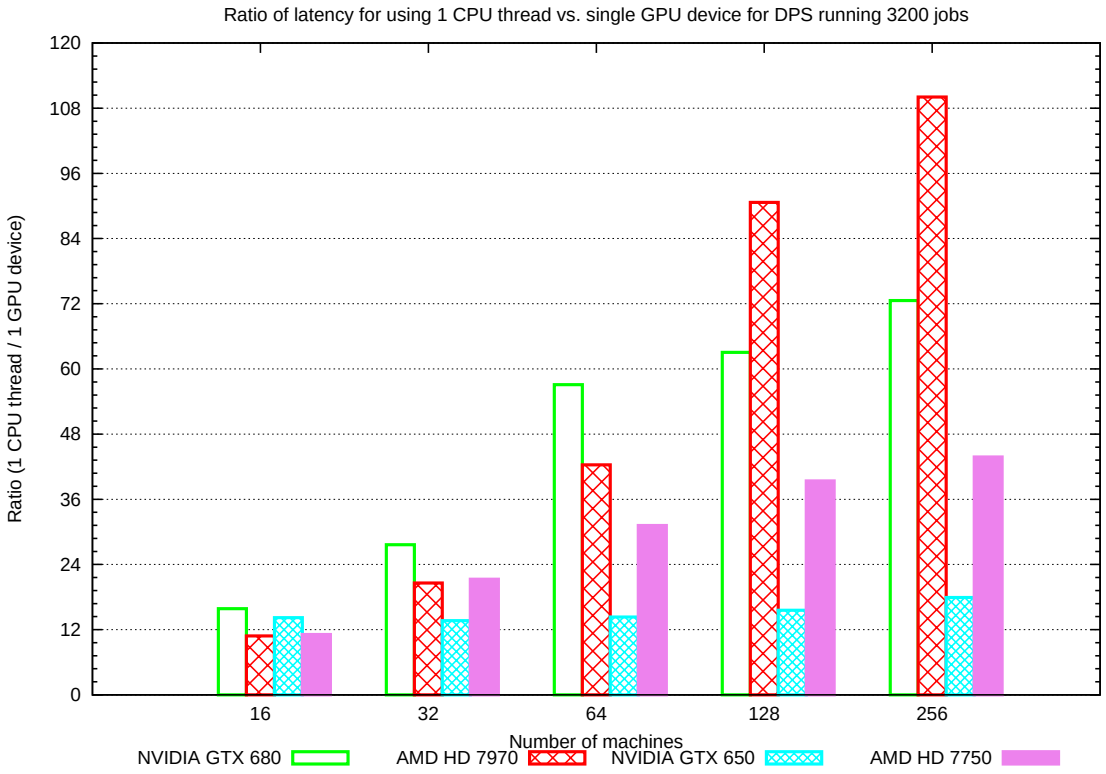


(a) Latency (12 CPU threads vs 2 GPUs)

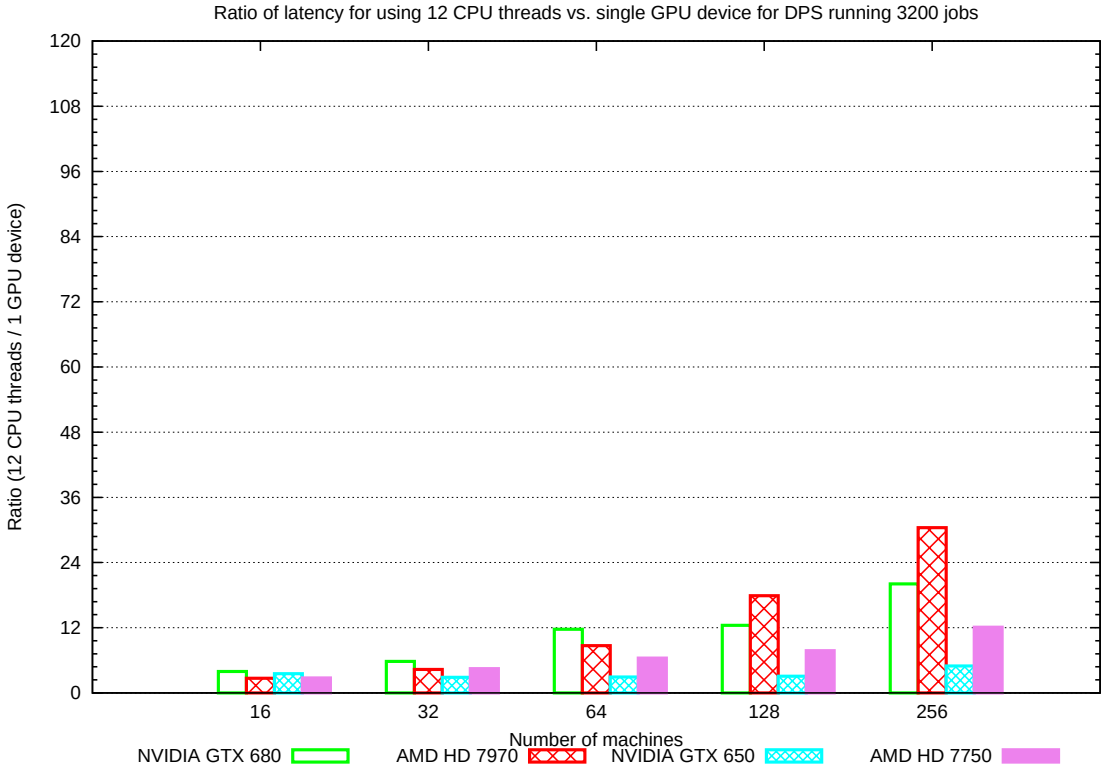


(b) Effective latency (12 CPU threads vs 2 GPUs)

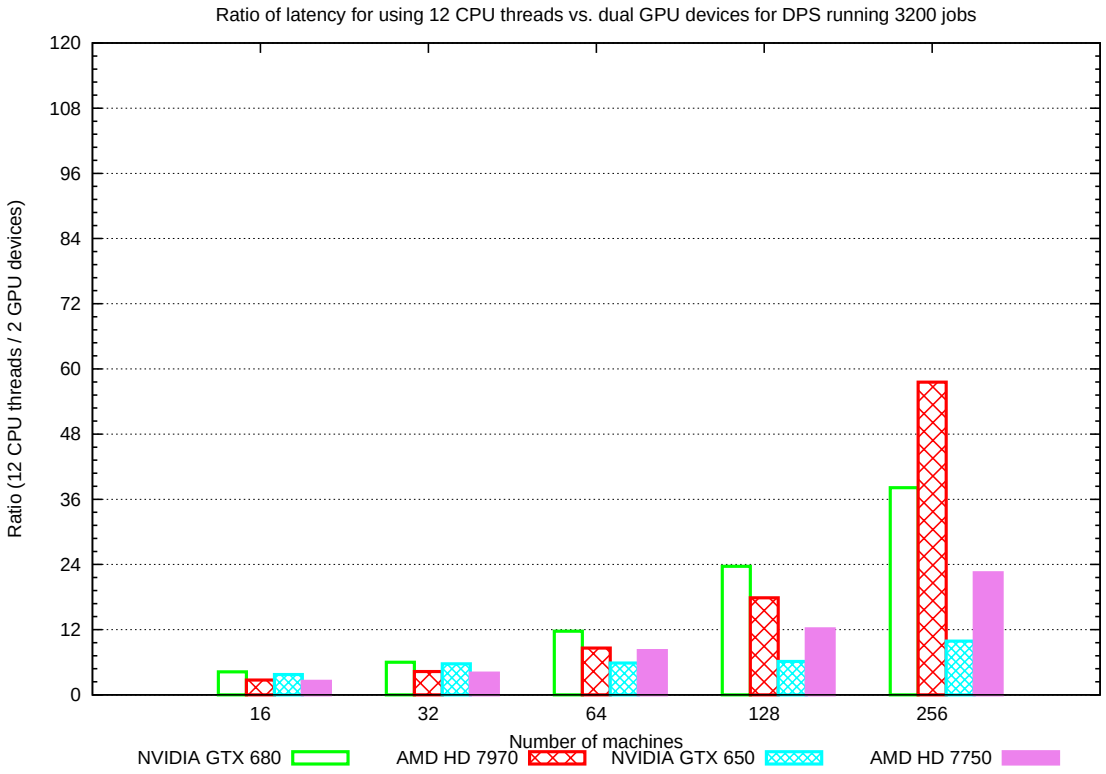
FIGURE 6.59: Latency performance of `GapsMis-`t with 12 CPU threads vs `GapsMis-`d on dual GPUs for a 3-gap alignment. The length of target sequences is 250 and 200 for query sequences.



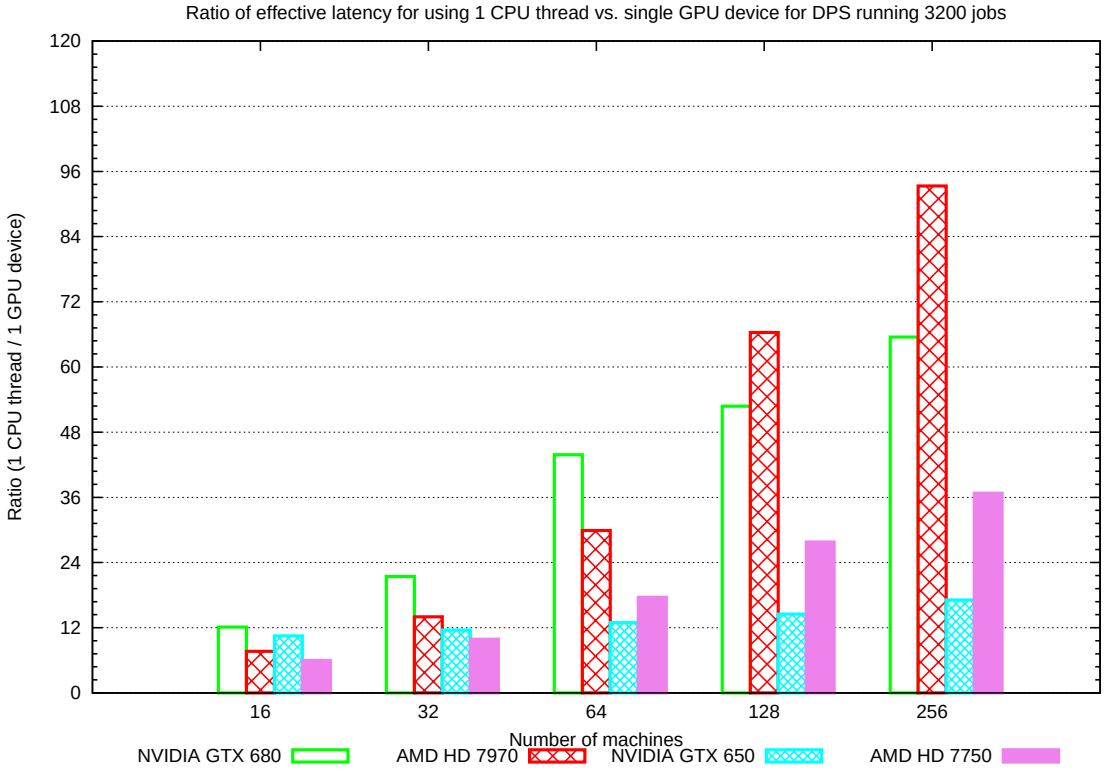
(a) Latency (1 CPU thread vs 1 GPU)



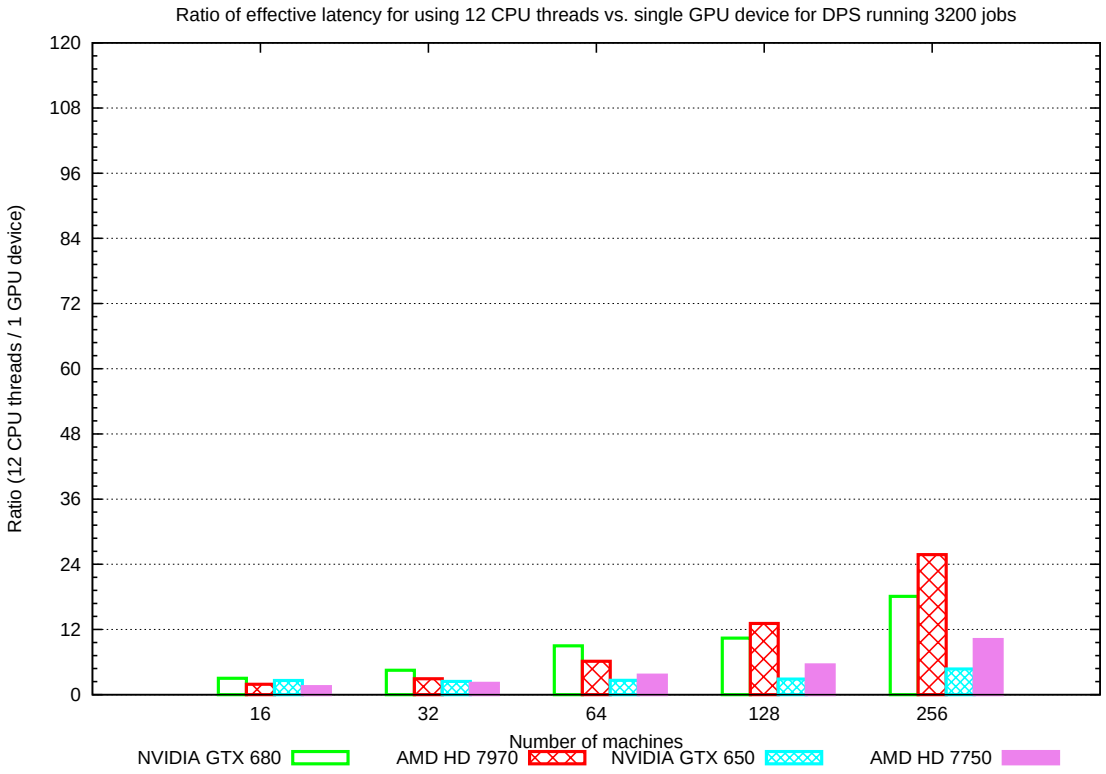
(b) Latency (12 CPU threads vs 1 GPU)



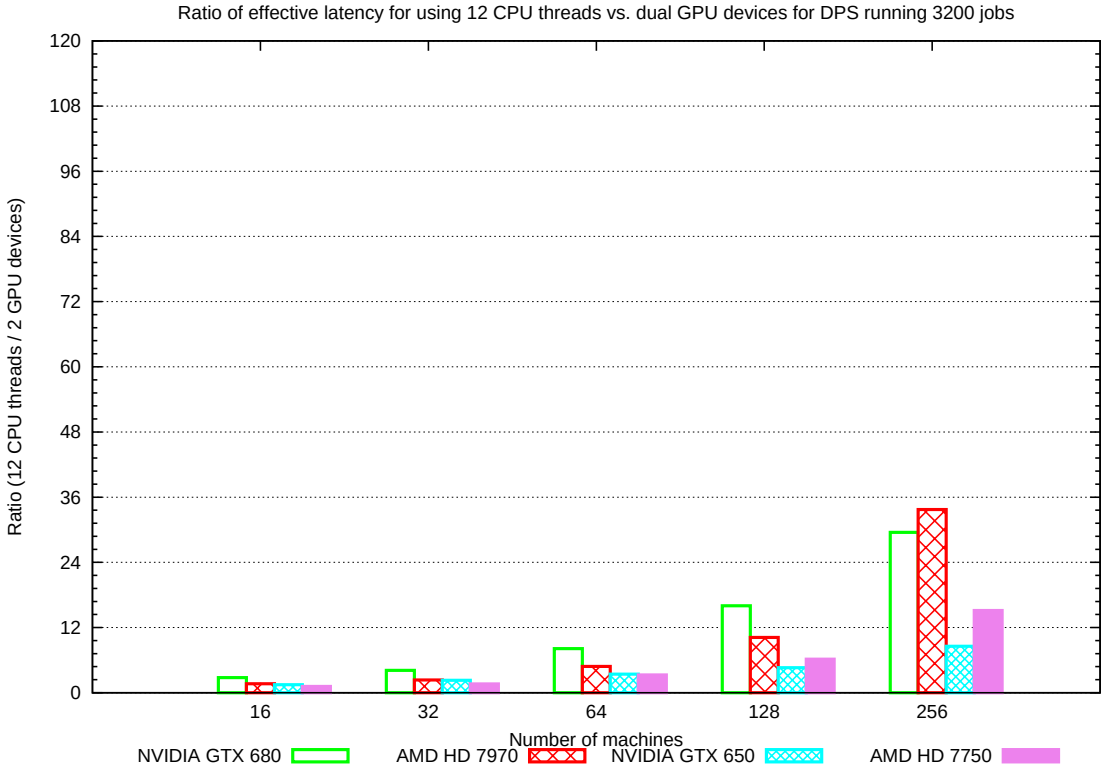
(c) Latency (12 CPU threads vs 2 GPU)



(d) Effective latency (1 CPU thread vs 1 GPU)

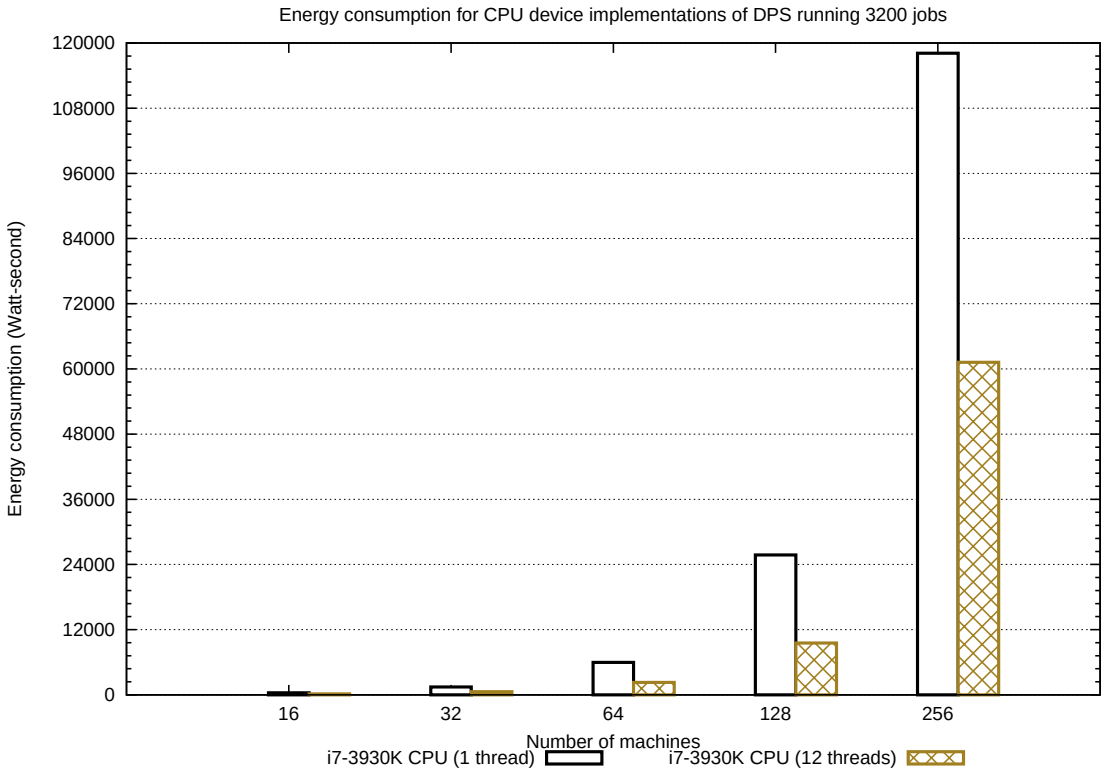


(e) Effective latency (12 CPU threads vs 1 GPU)

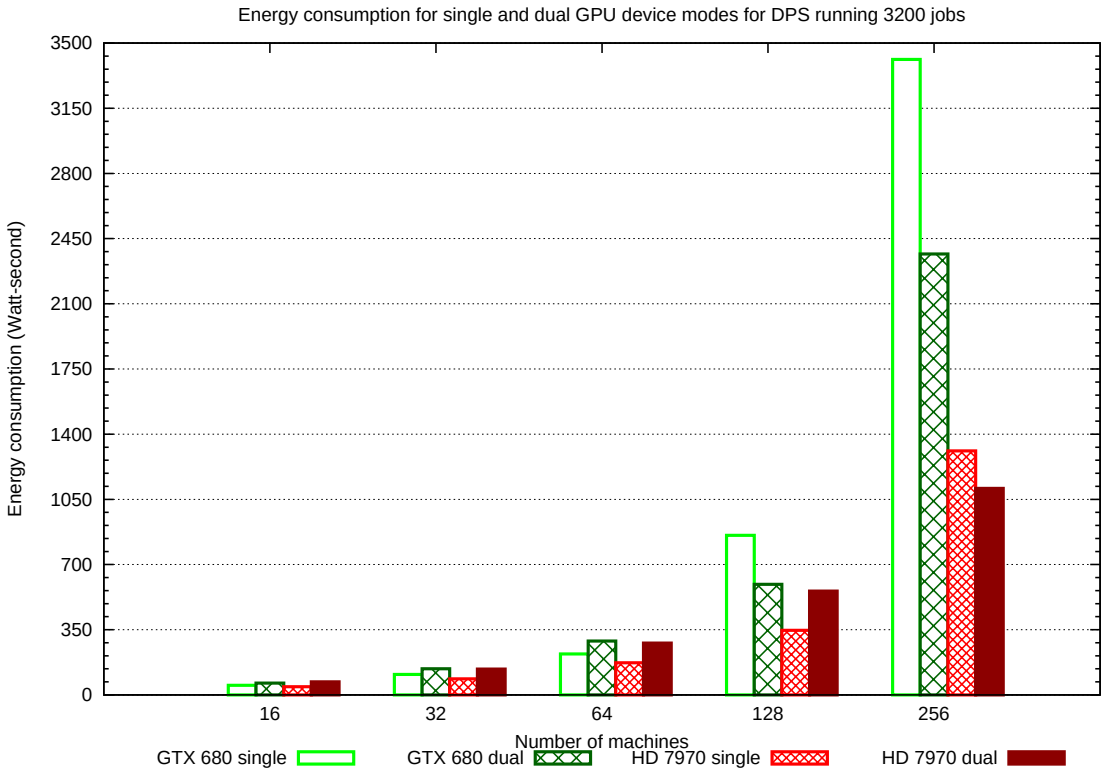


(f) Effective latency (12 CPU threads vs 2 GPU)

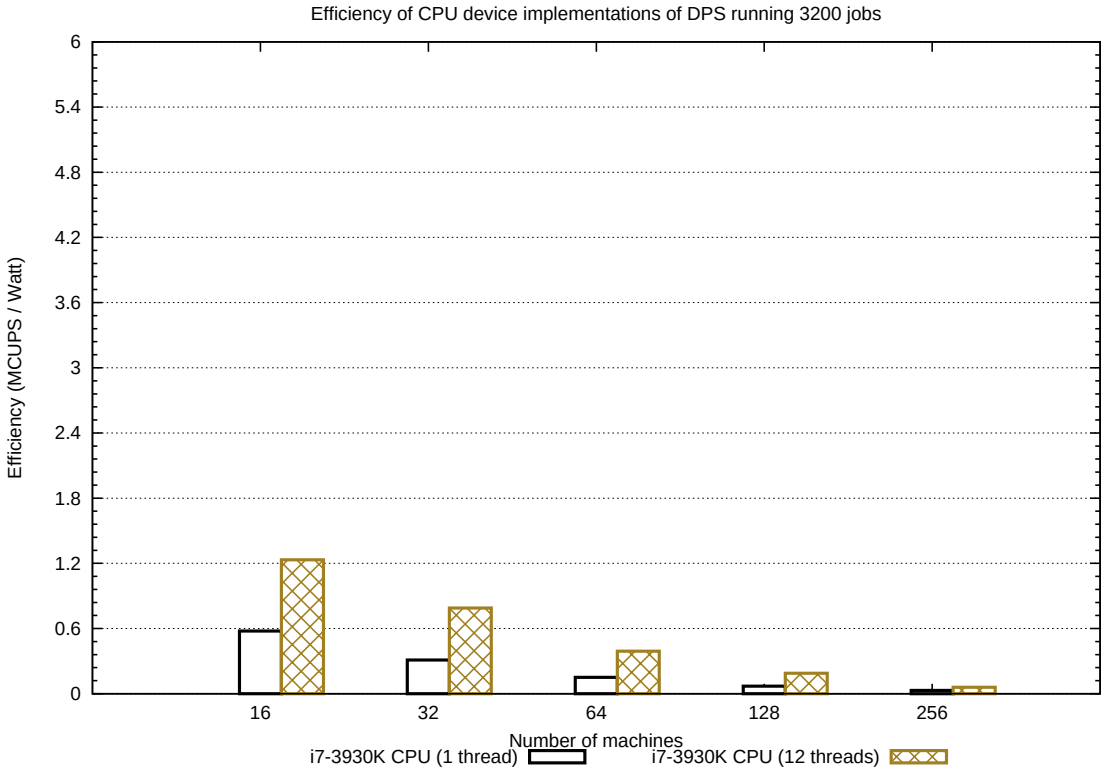
FIGURE 6.57: Comparison of CPU vs GPU performance of `DPS` for a problem size consisting of 3200 jobs.



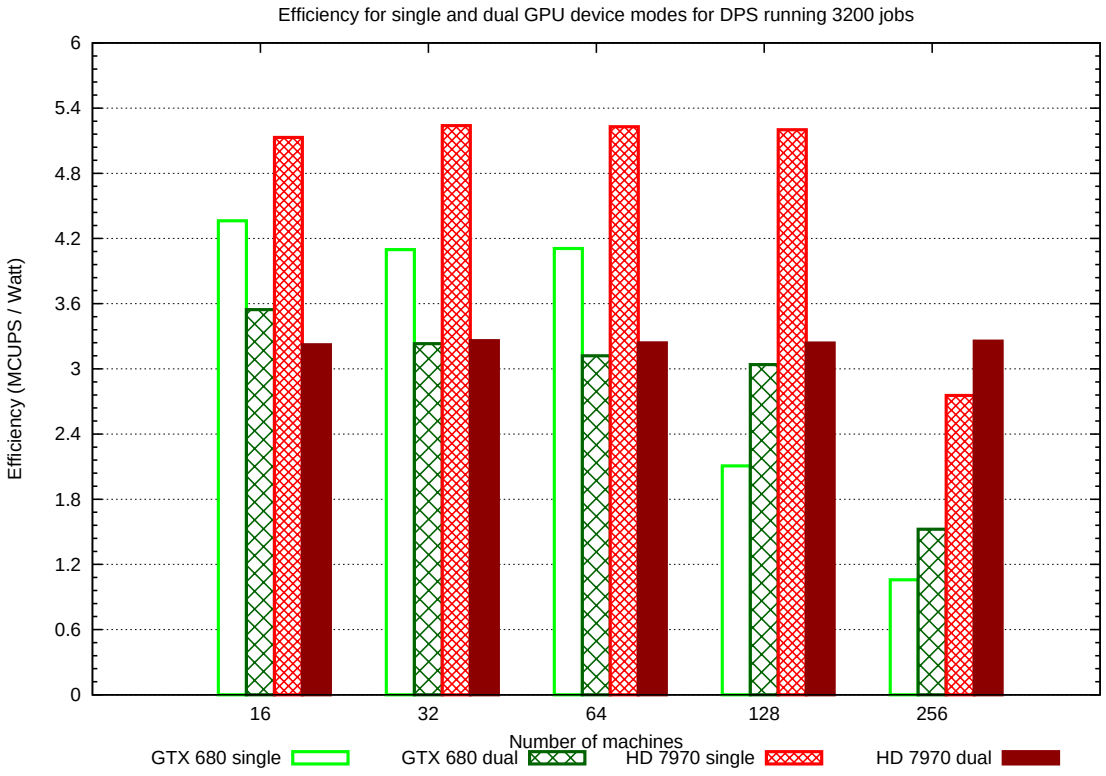
(a) Energy consumption (CPU)



(b) Energy consumption (GPU)

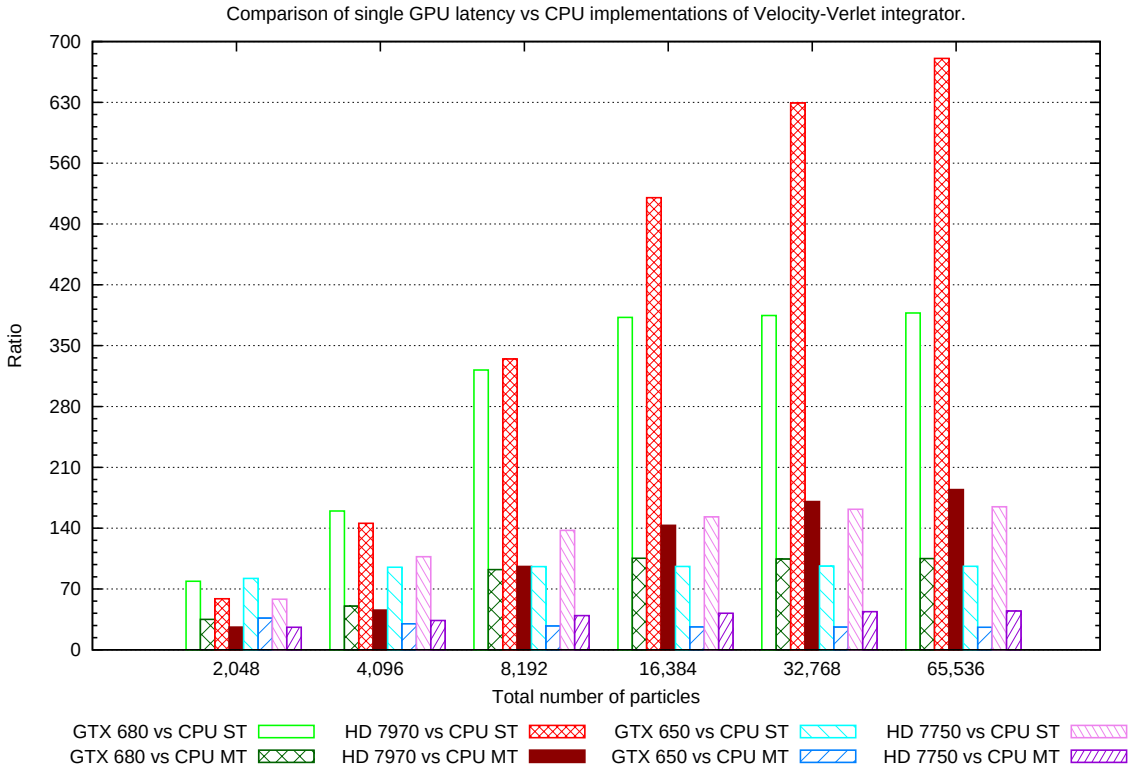


(c) Efficiency (CPU)

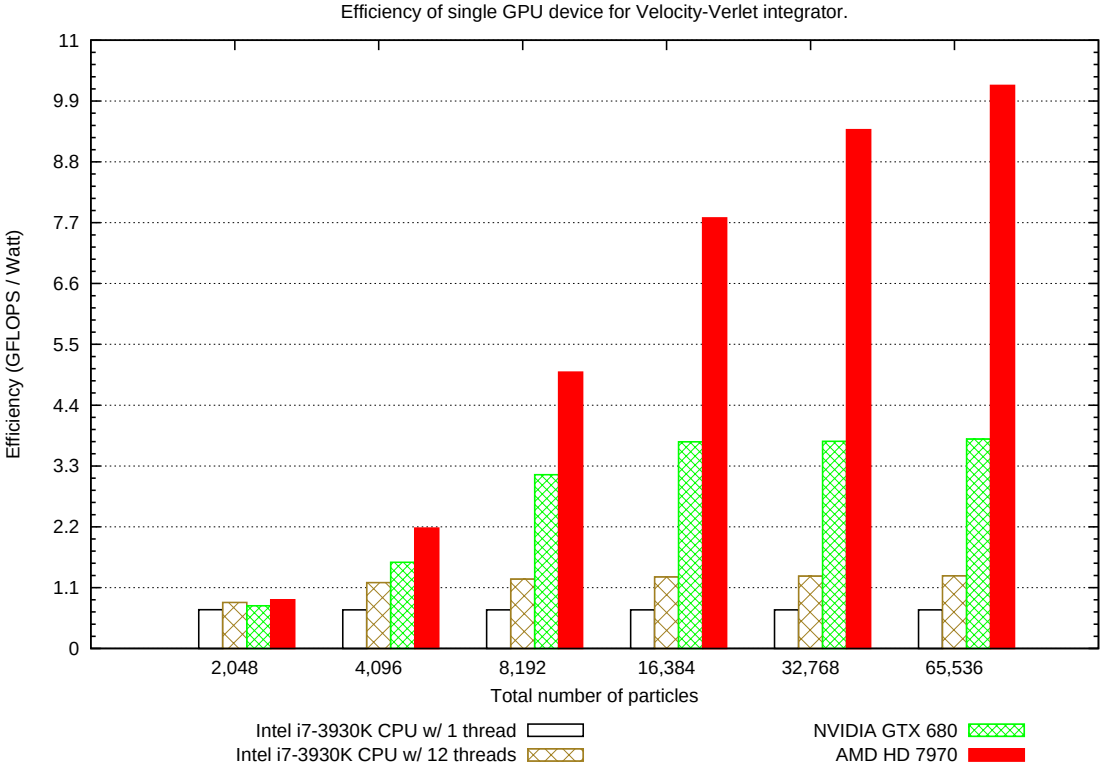


(d) Efficiency (GPU)

FIGURE 6.57: Comparison of energy consumption and efficiency for CPU and GPU devices for `DPS`. Energy consumption is given in Watt-second while efficiency is given in millions of cell updates per second per Watt.

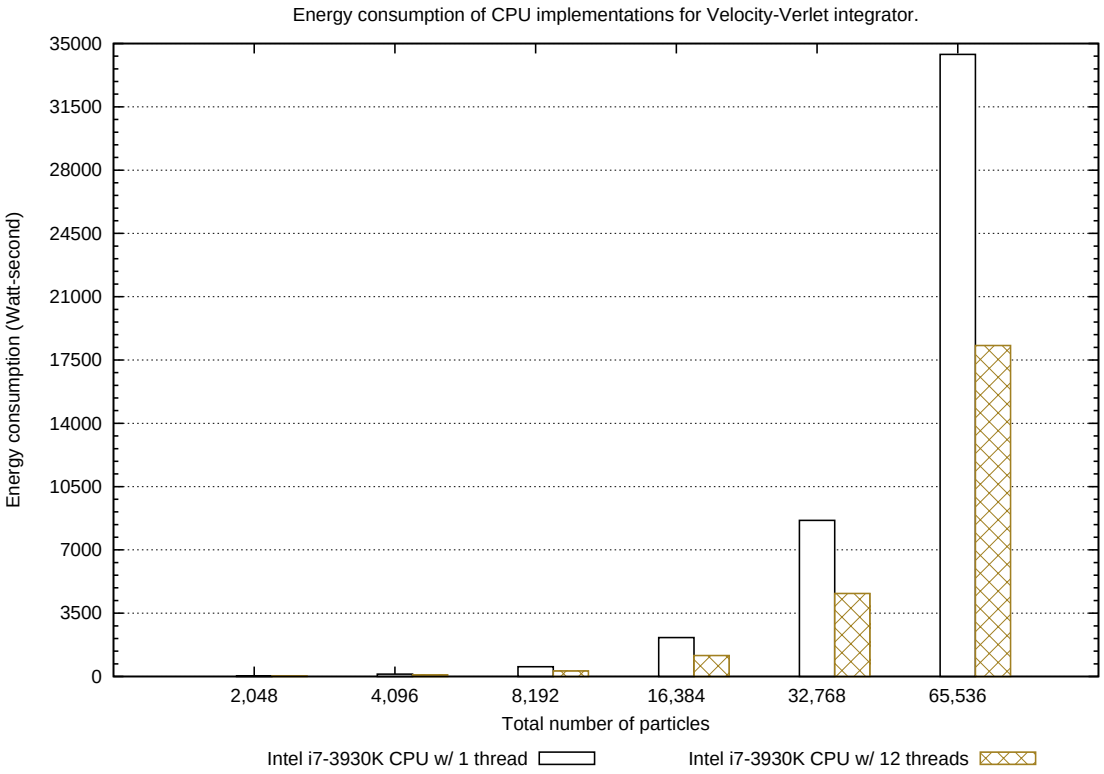


(a) Performance ratio for latency

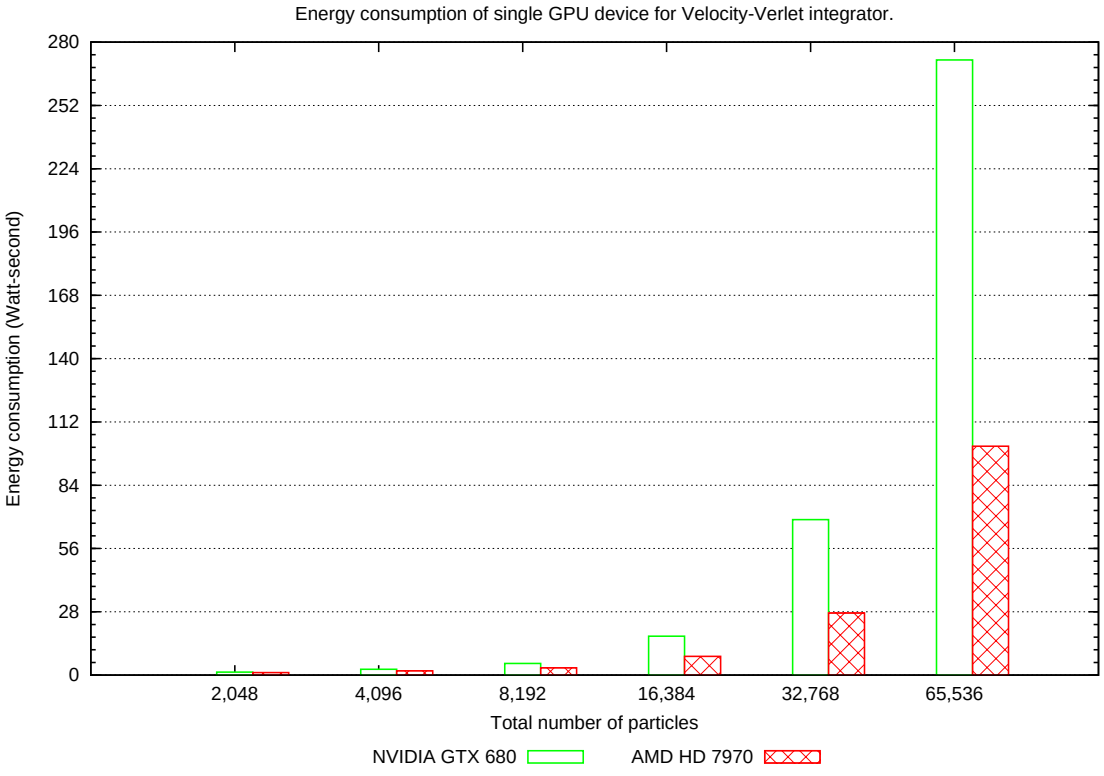


(b) Efficiency

FIGURE 6.57: (a) Ratio of CPU performance to single GPU performance with respect to latency. (b) Comparison of CPU vs. GPU in terms of efficiency measured in billions of floating-point operations per Watt.

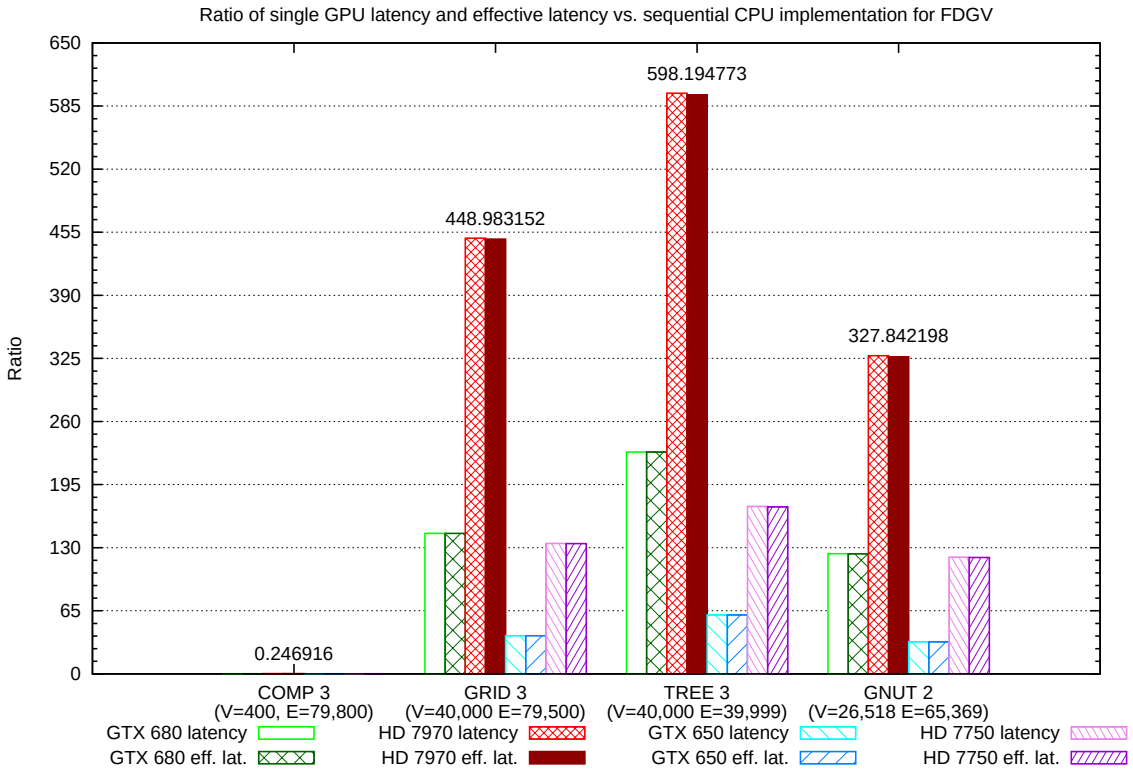


(a) CPU energy consumption

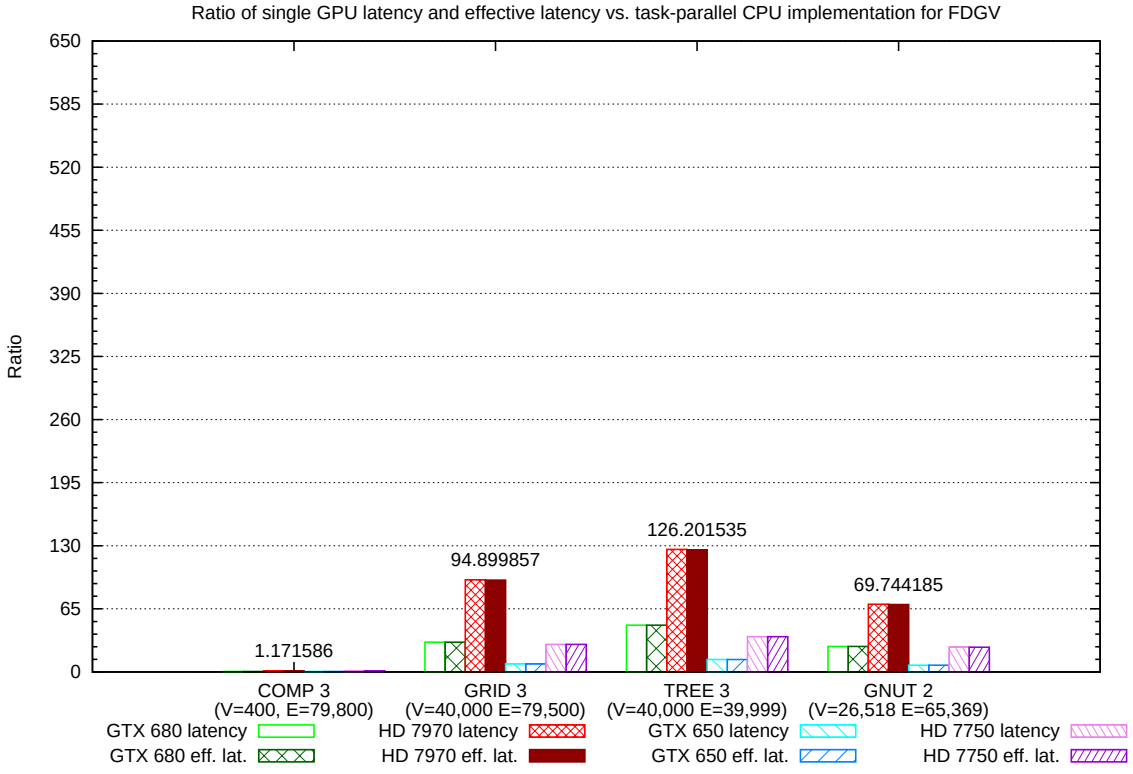


(b) GPU energy consumption

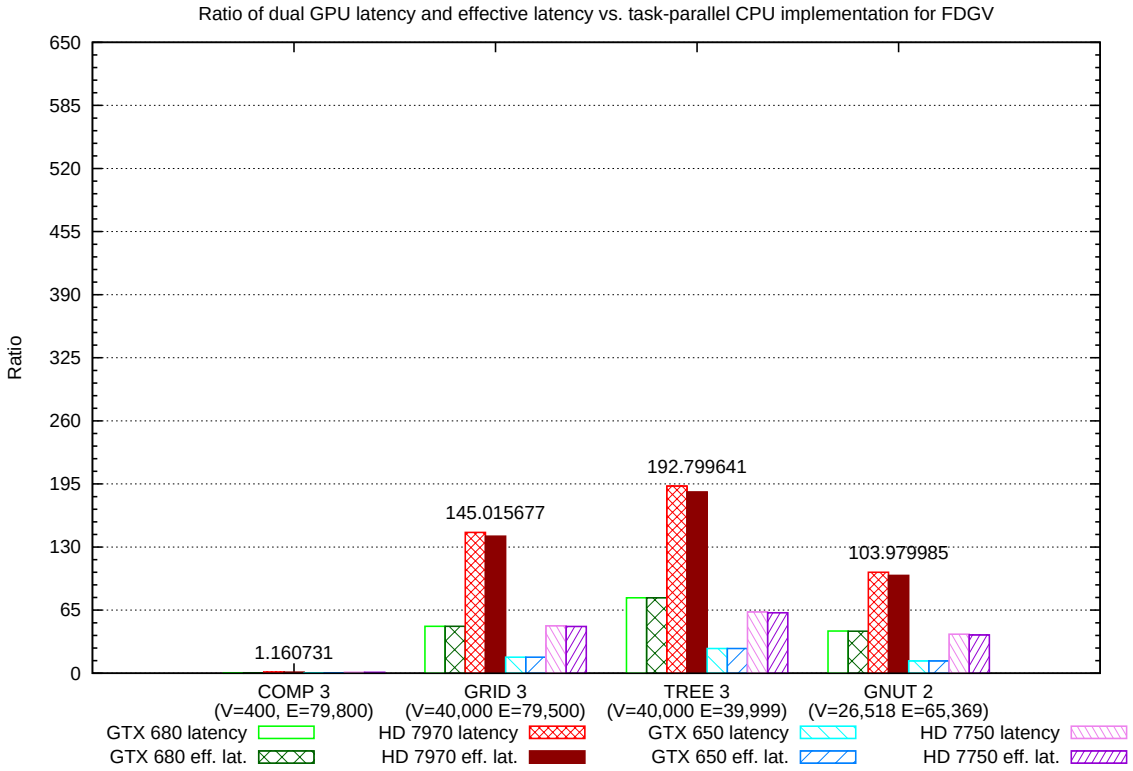
FIGURE 6.57: Comparison of energy consumption for CPU and GPU devices for `Velvet`. Energy consumption is measured in Watt-second.



(a) 1 CPU thread vs. 1 GPU

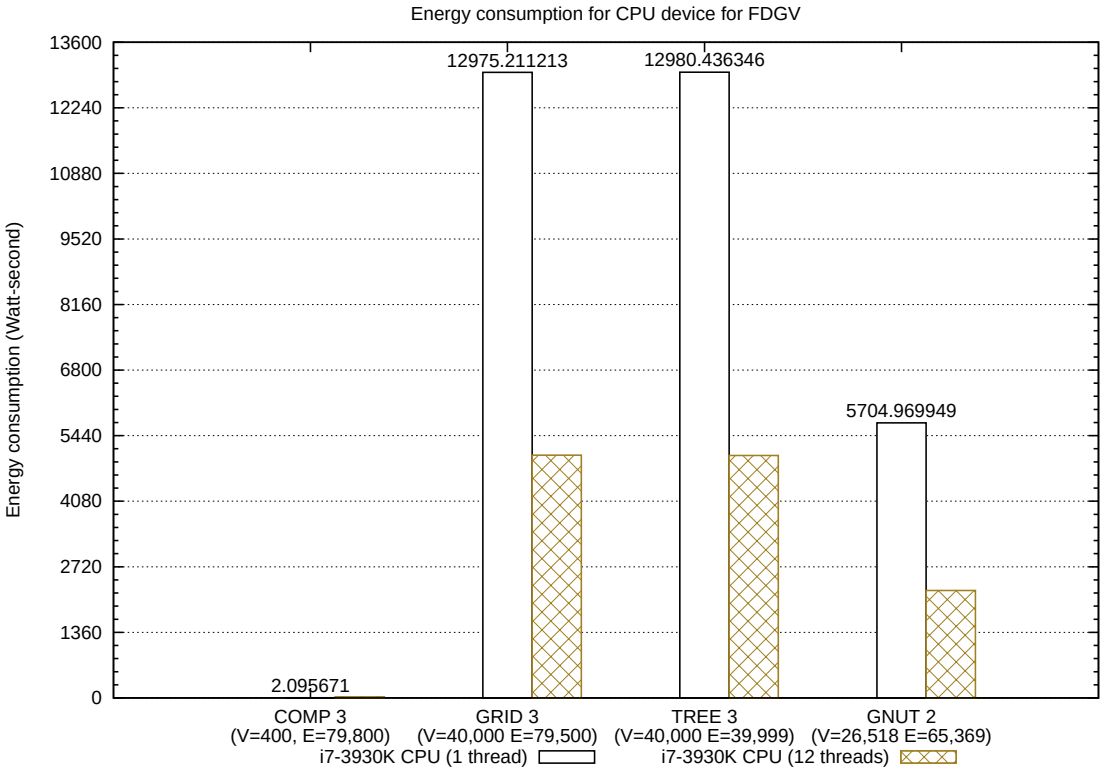


(b) 12 CPU threads vs. 1 GPU

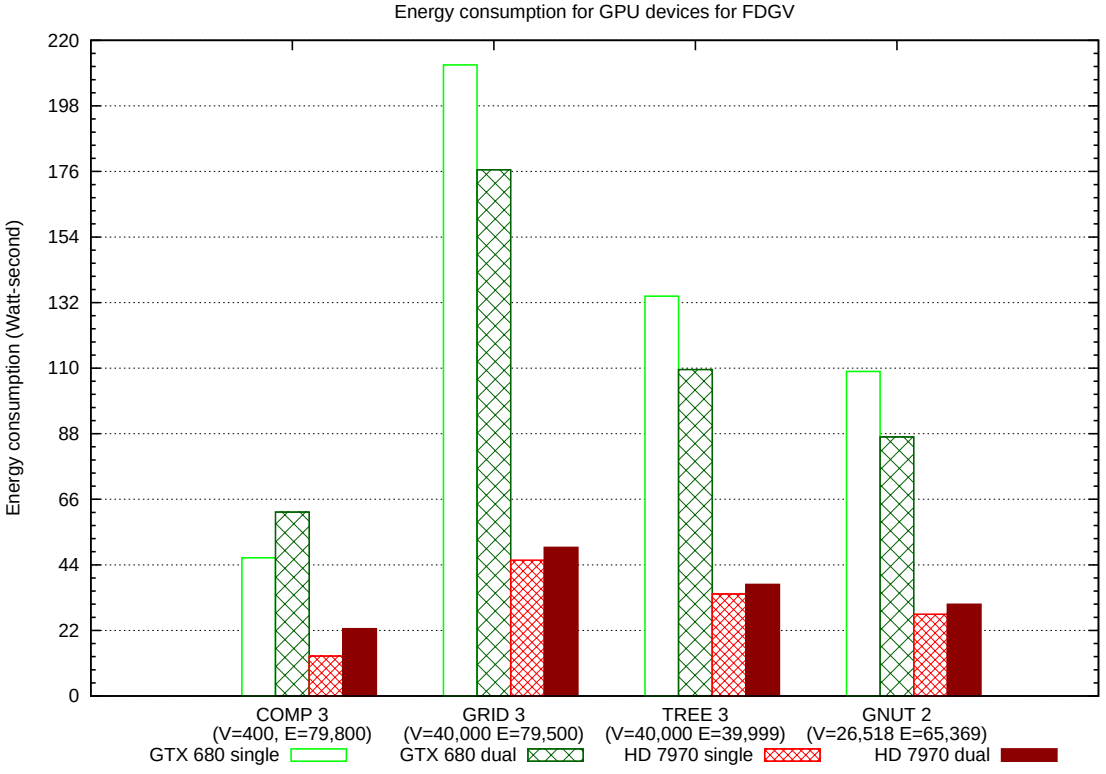


(c) 12 CPU threads vs. 2 GPUs

FIGURE 6.57: Comparison of CPU vs. GPU execution times for `FDGV`. The largest value for each graph is shown in the labels within the plot.

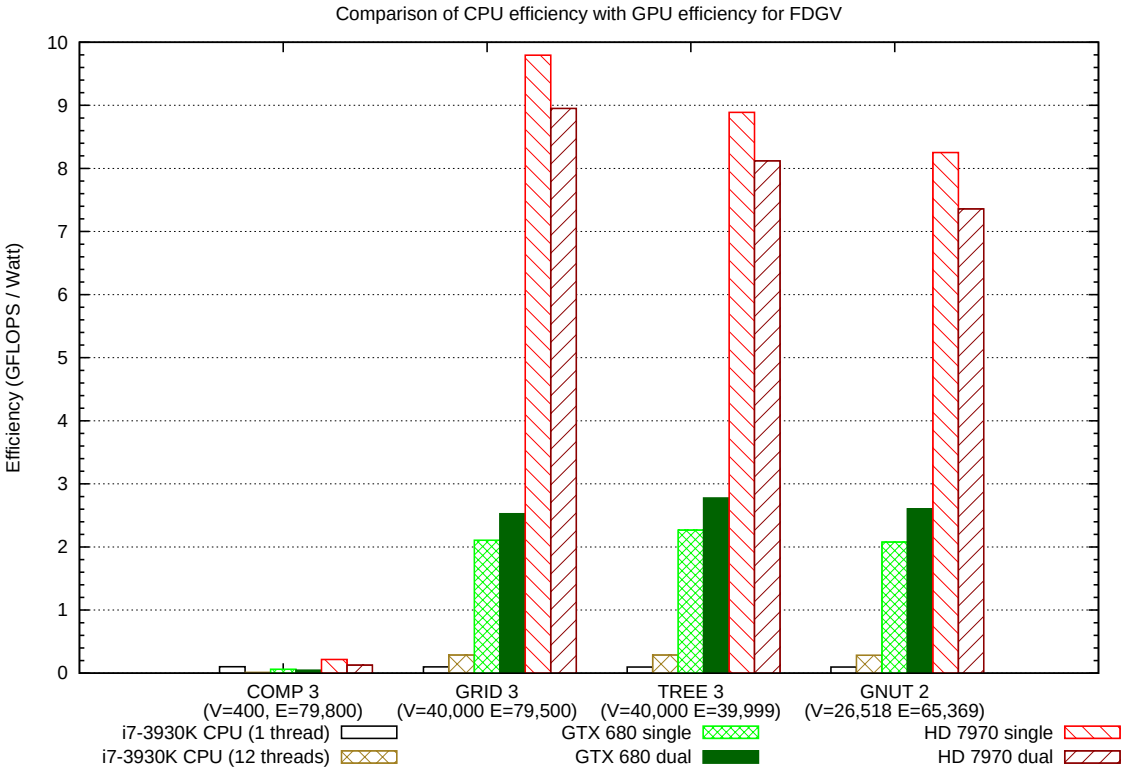


(a) Energy consumption for CPU



(b) Energy consumption for GPU

It is interesting to observe that energy consumption is lower when using a single AMD HD 7970 GPU compared to using two of them. This also translates to higher efficiency



(c) Efficiency of CPU and GPU

FIGURE 6.57: Comparison of CPU vs. GPU energy consumption and efficiency for `FDGV`. Energy consumption is measured in Watt-second and efficiency is measured in GFLOPS/Watt.

as well in terms of performance-per-watt. In general, and as expected, both GPU devices offer superior energy efficiency than the CPU for the graphs with large number of vertices. Figure 6.58(c) shows that the GPU devices are able to achieve efficiency of between 7.4 and 9.8 GFLOPS per Watt while the CPU achieves less than 0.5 GFLOPS per Watt even in multi-threaded configuration.

Conclusion In this experiment, we compare the performance of sequential and task-parallel implementations on the CPU with the data-parallel implementation on GPU in order to evaluate the amount of performance gain we can achieve with the GPU, evaluate energy consumption of both devices and efficiency. We have shown that the applications benefit greatly from using the GPU in saving both time and energy for almost all problem instances. This fact is somewhat better appreciated when we consider the amount of work the GPU devices are able to do with respect to power consumed (performance-per-watt). Finally, we have also shown that a device performs better when the compute demands of an application is better suited to the architecture of that device, as is demonstrated by the performance of the GPU devices and the n-body method applications.

6.10 Conclusion and future work

In this chapter we presented a data-parallel implementation of `GapsMis` on the GPU. The `GapsMis` tool is a practical application that could someday be integrated into the pipeline of already existing sequencing and alignment tools. Unfortunately, due to the time constraint in carrying out this PhD project, we were unable to fully explore some possible improvements to our implementation and so there is room for improvement regarding the heterogeneous implementation that uses both CPU and GPU devices.

The first aspect of this improvement has to do with executing the back-tracking routine on the results returned by the GPU for each batch. In our current implementation, this phase is single-threaded, hence, back-tracking is done sequentially by a single CPU thread for all results in a batch. This process can be greatly improved by managing the thread pool more intelligently. So if there are more than one available CPU threads, apart from the CPU threads responsible for coordinating the GPU command queues, the back-tracking work-load should be shared by these threads.

The other aspect involves the GPU architecture and parallel computing framework. The `GapsMis` application is written with OpenCL 1.0 support in order to provide support for both NVIDIA and AMD GPUs. However, a future direction will be to have a version optimized for each hardware using CUDA from version 6 [81] for NVIDIA GPUs and the OpenCL from version 2.0 [40] for AMD GPUs (both CUDA and OpenCL versions were released very recently). One of the new features in the latest versions of CUDA and OpenCL allows the new GPU devices, like the AMD Kaveri APUs and NVIDIA 700 series GPUs, to have a unified view of memory with the CPU device. This helps to greatly minimize the amount of explicit copy operations necessary between compute device and host since both devices now have the same view of host memory to some considerable extent.

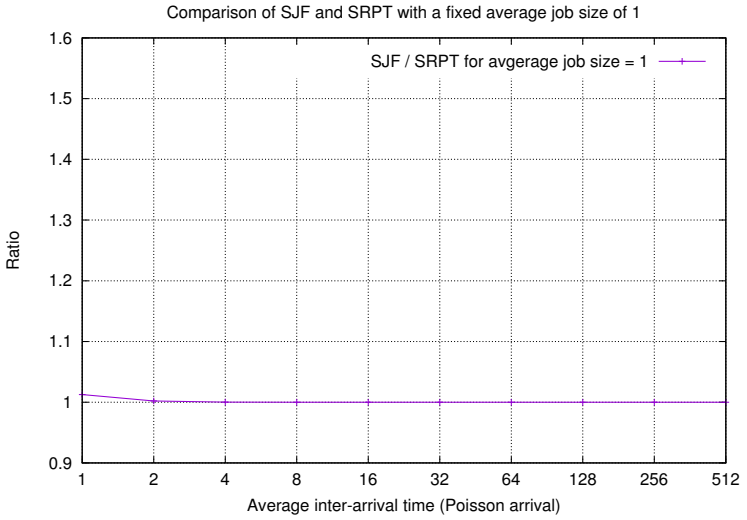
Another aspect we were unable to explore in our project is the use of *Streaming SIMD Extensions* (SSE) in our implementation of `Velvet` and `FDGV` on the CPU. SSE enables modern CPUs to handle SIMD data instructions which is beneficial when the same instruction needs to be performed on multiple data, similar to what the GPU does. It will be interesting to see how using SSE will affect the performance of both CPU implementations.

Appendix A

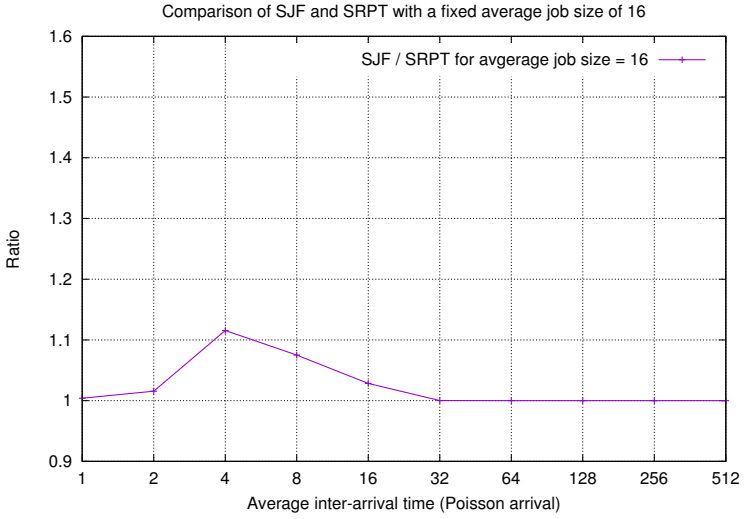
More Experiment Results for Energy-Efficient Flow Time Scheduling

A.1 Results on job selection strategies

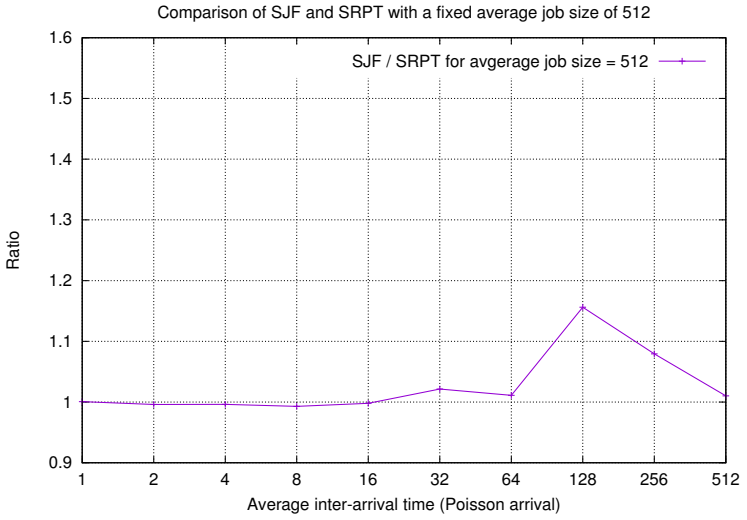
A.1.1 Single processor simulations



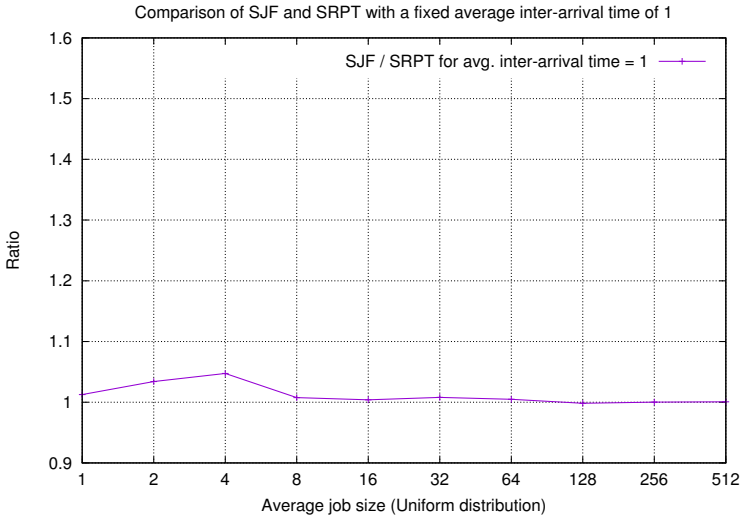
(a) Performance ratio for average job size of 1.



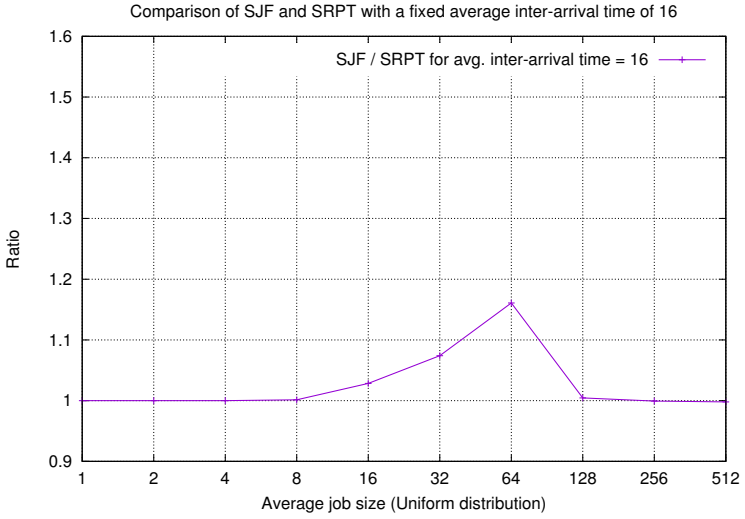
(b) Performance ratio for average job size of 16.



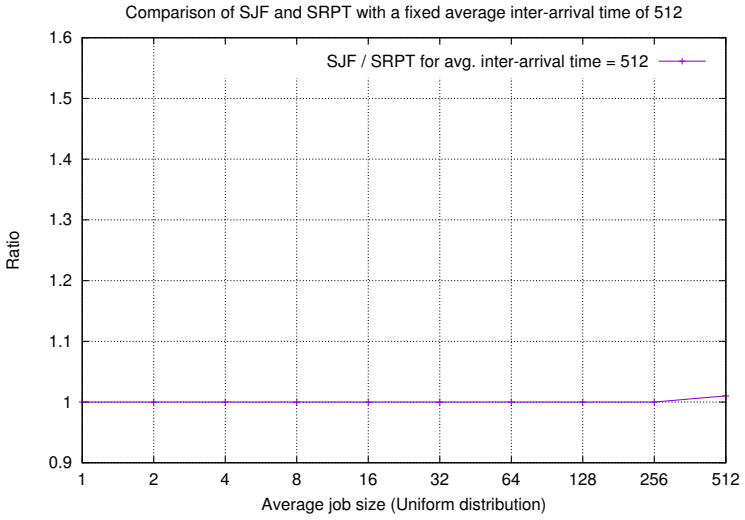
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

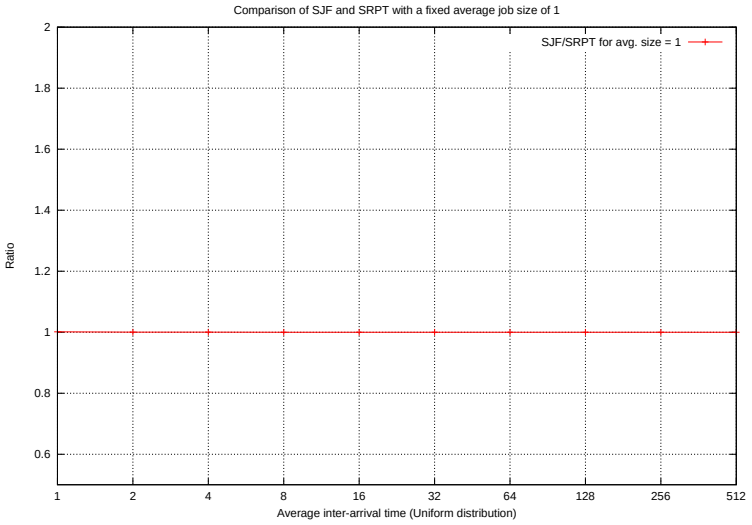


(e) Performance ratio for average inter-arrival time of 16.

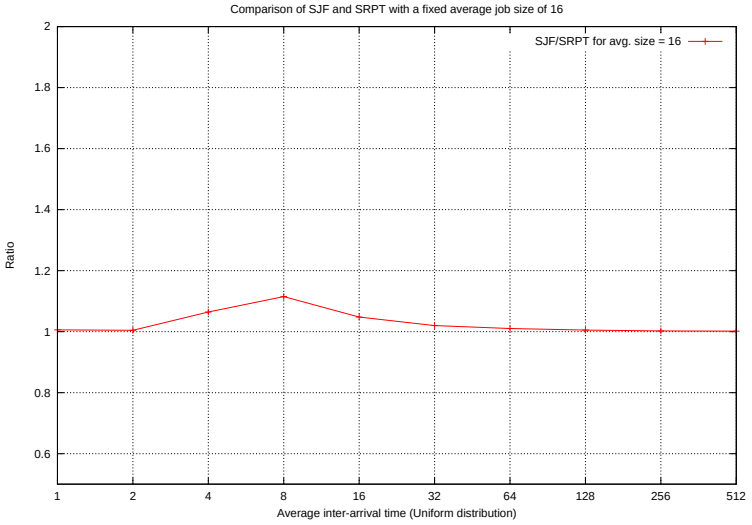


(f) Performance ratio for average inter-arrival time of 512.

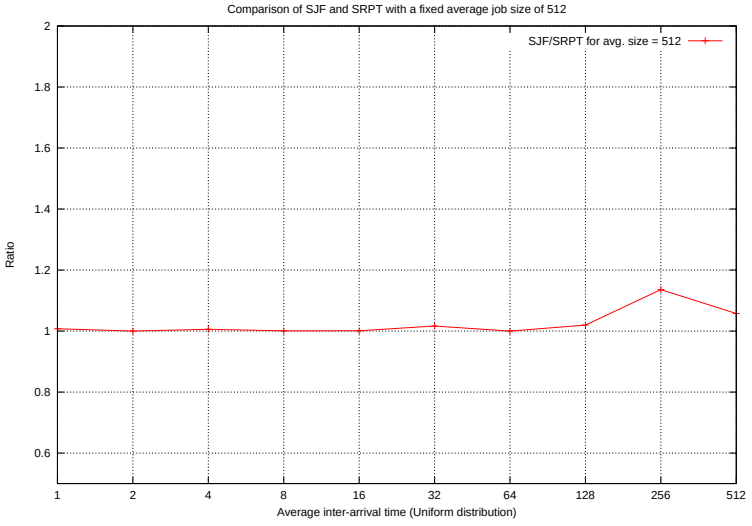
FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy of AJC when using SJF vs. SRPT on a single processor. Inter-arrival times are given by Poisson distribution and job sizes are given by uniform distribution.



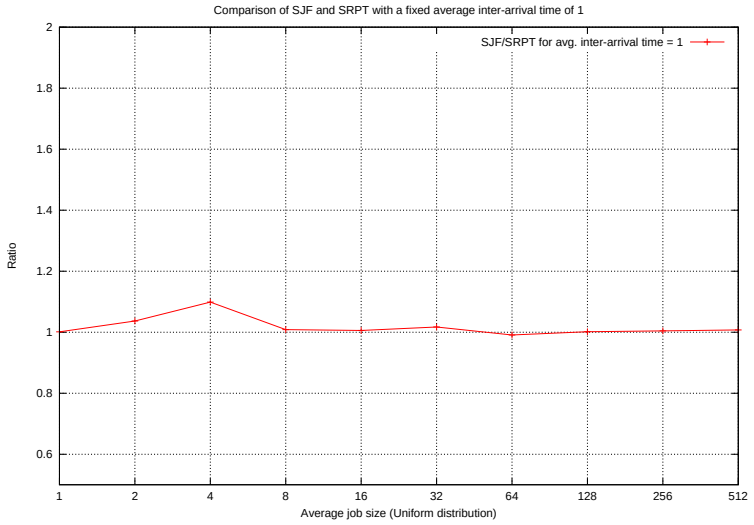
(a) Performance ratio for average job size of 1.



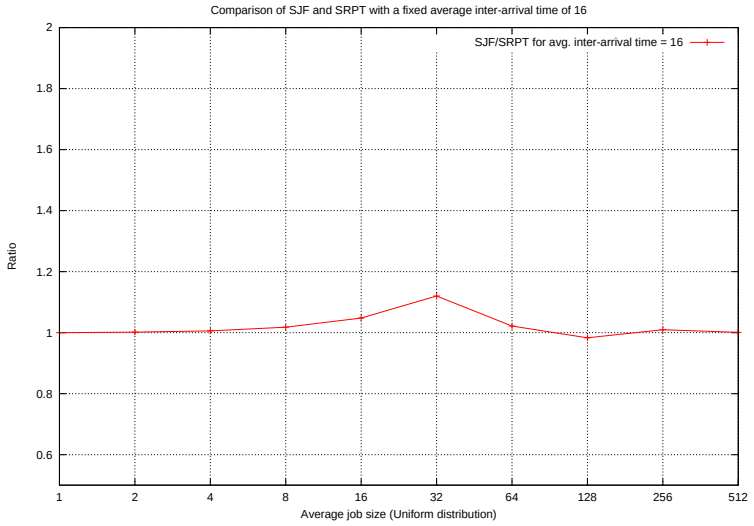
(b) Performance ratio for average job size of 16.



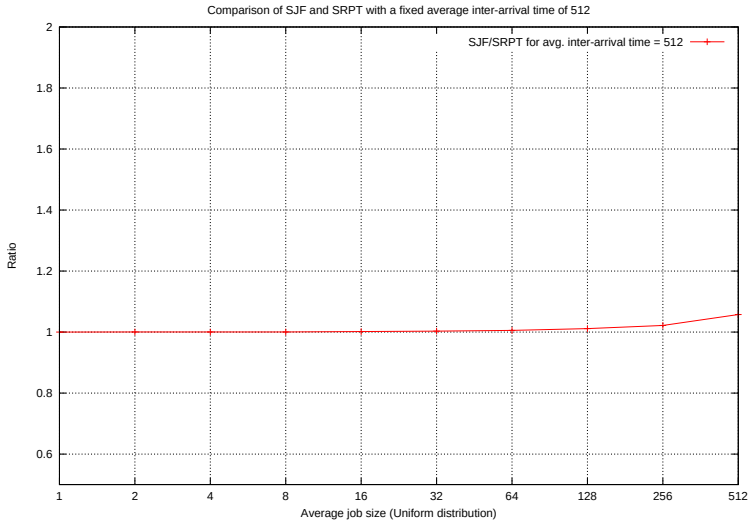
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

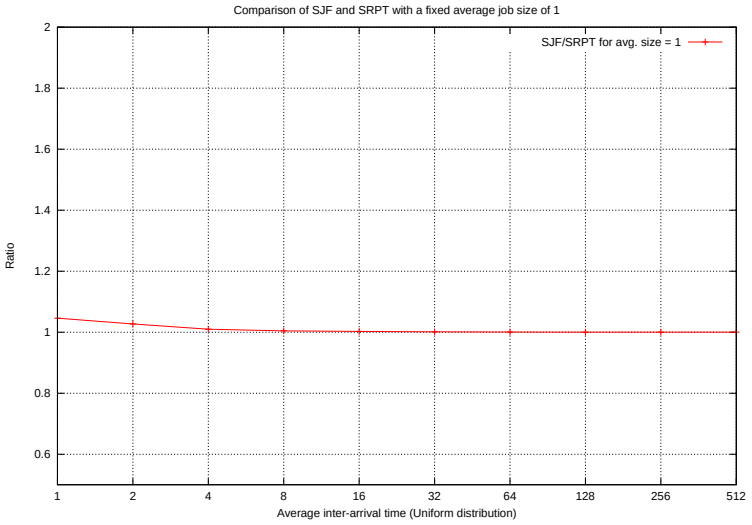


(e) Performance ratio for average inter-arrival time of 16.

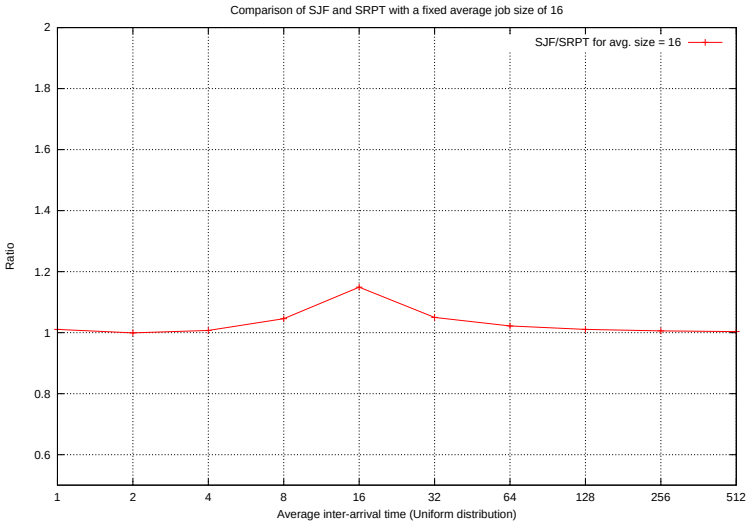


(f) Performance ratio for average inter-arrival time of 512.

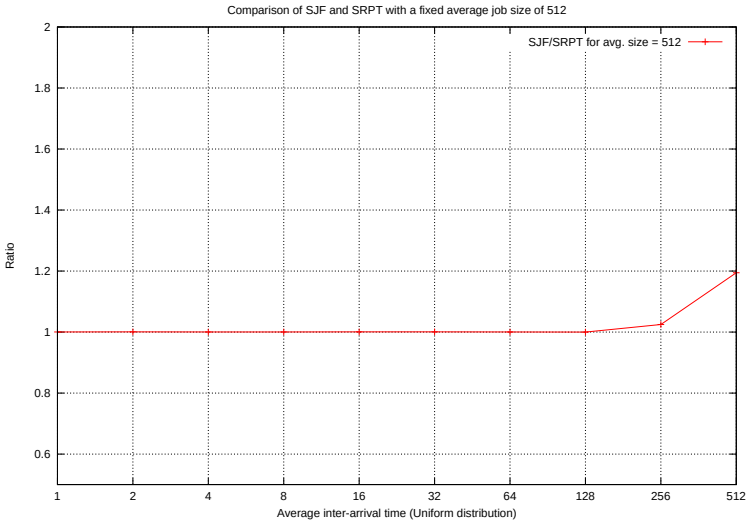
FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy of AJC when using SJF vs. SRPT on a single processor. Uniform distribution is used for both inter-arrival times and job sizes.



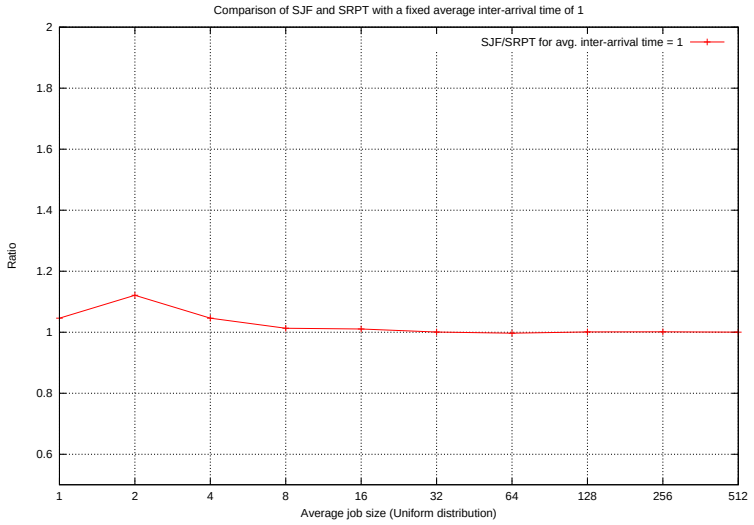
(a) Performance ratio for average job size of 1.



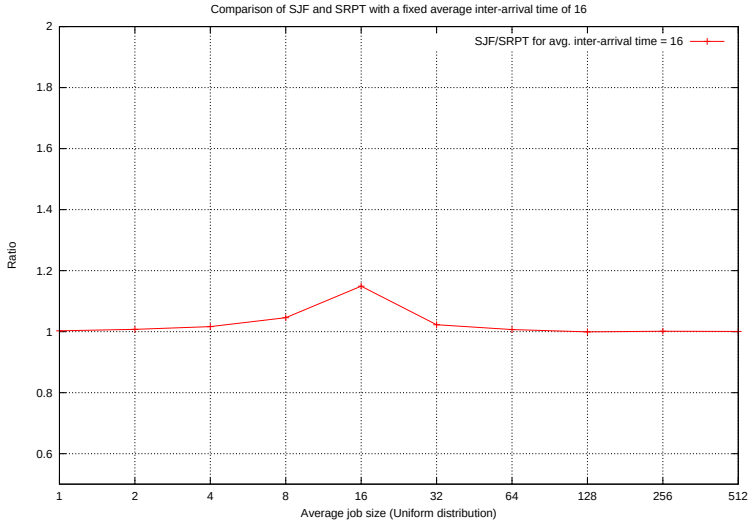
(b) Performance ratio for average job size of 16.



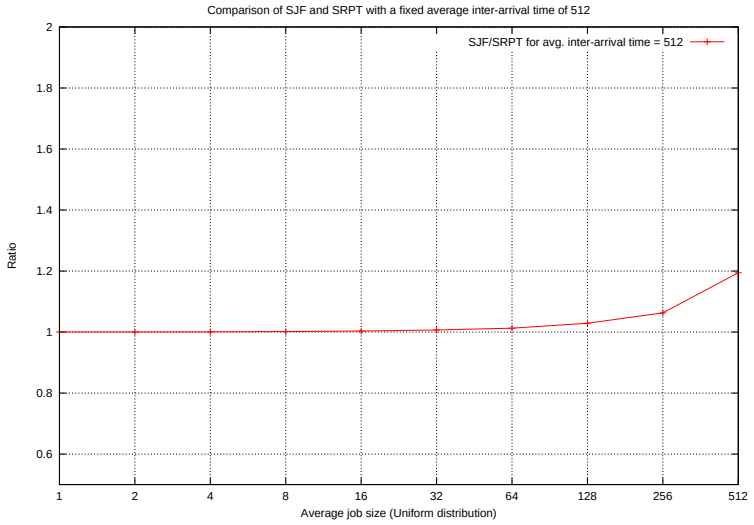
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.



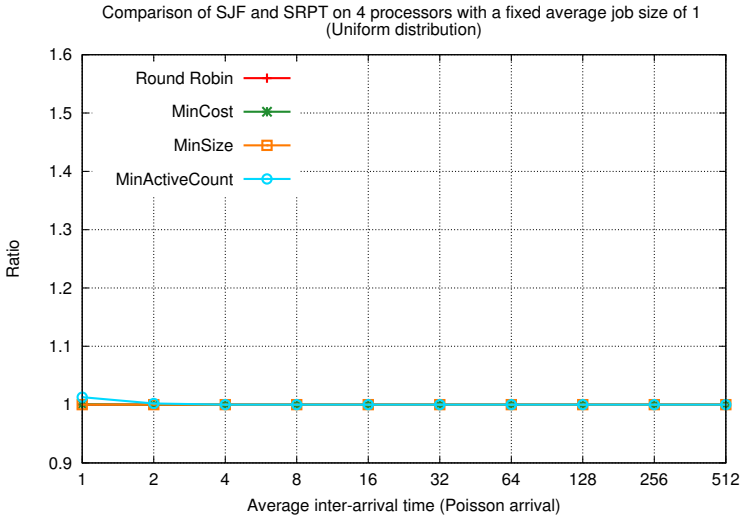
(e) Performance ratio for average inter-arrival time of 16.



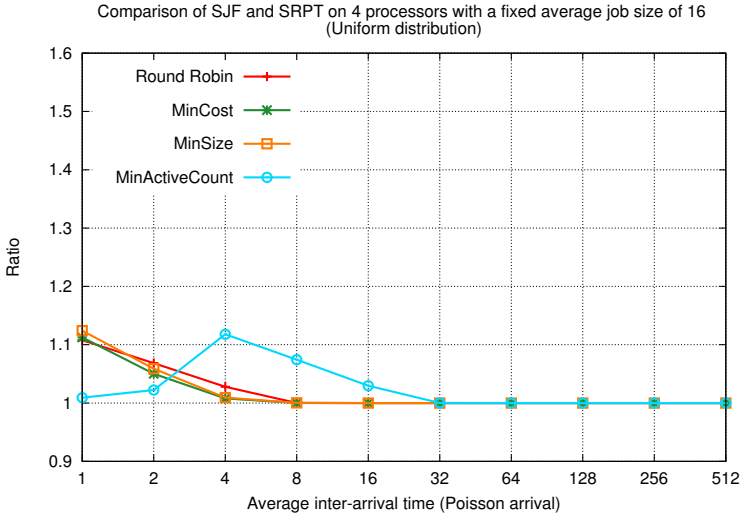
(f) Performance ratio for average inter-arrival time of 512.

FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy of AJC when using SJF vs. SRPT on a single processor. Uniform distribution is used for job sizes while Poisson distribution is used for inter-arrival times.

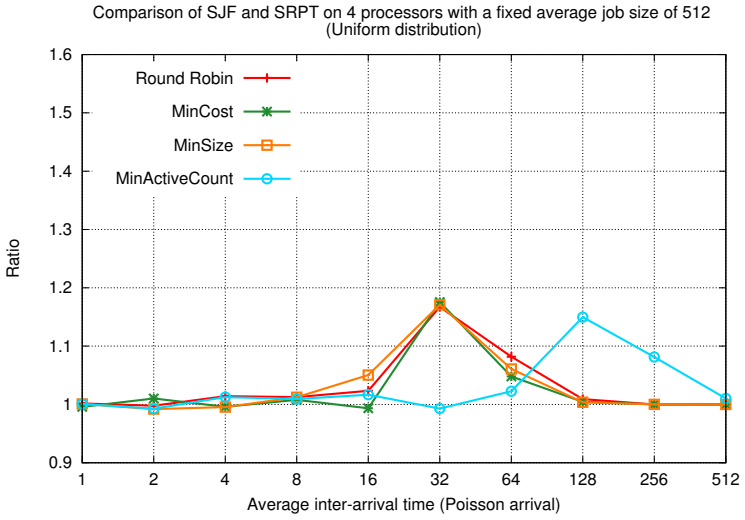
A.1.2 Multi-processor simulations



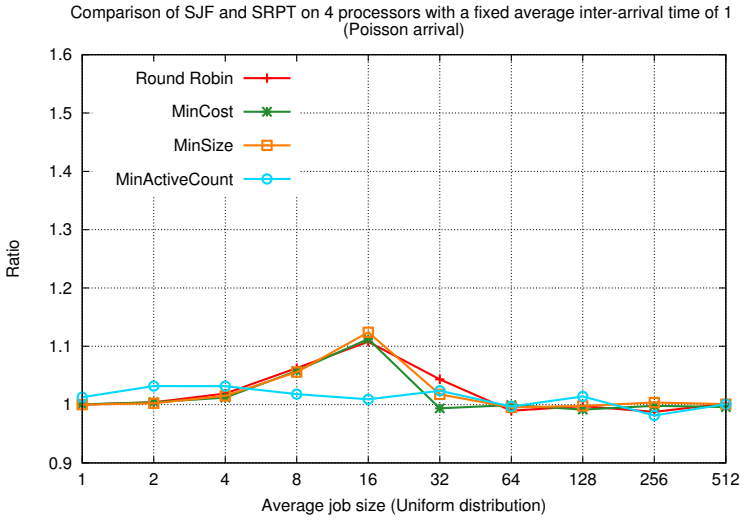
(a) Performance ratio for average job size of 1.



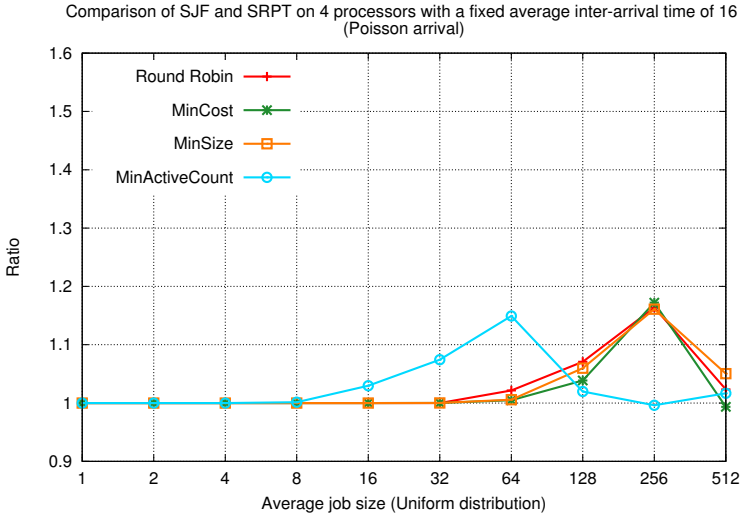
(b) Performance ratio for average job size of 16.



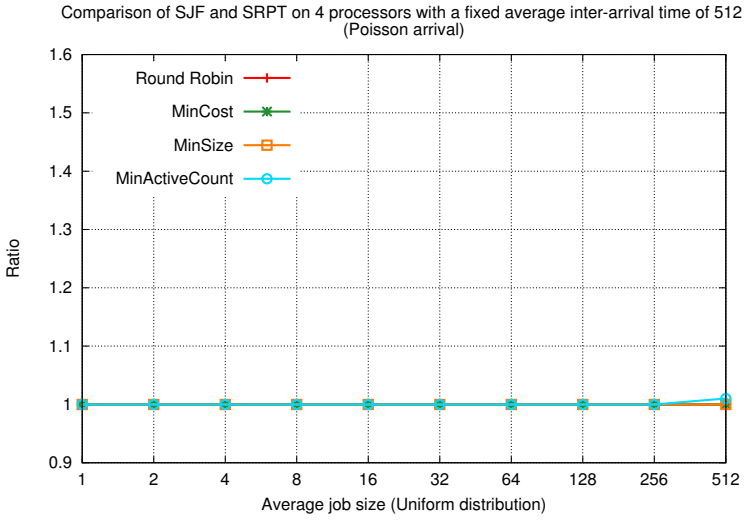
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

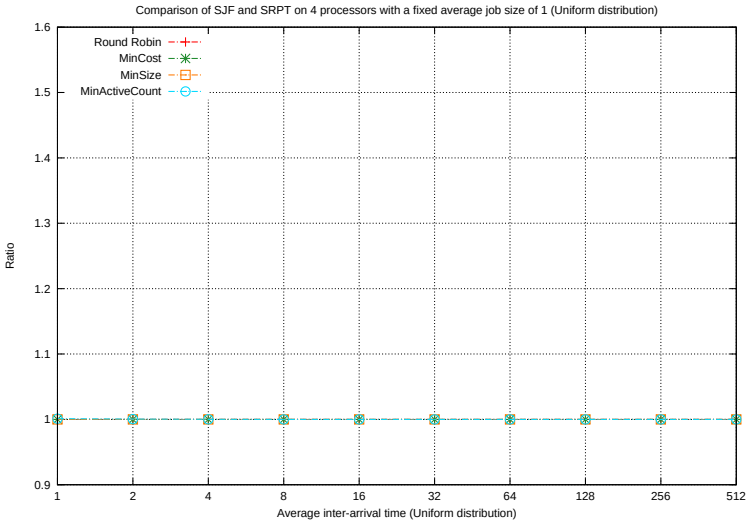


(e) Performance ratio for average inter-arrival time of 16.

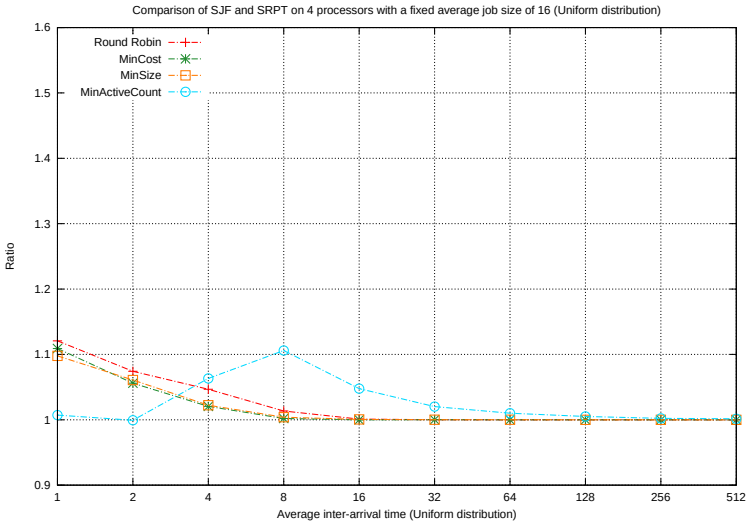


(f) Performance ratio for average inter-arrival time of 512.

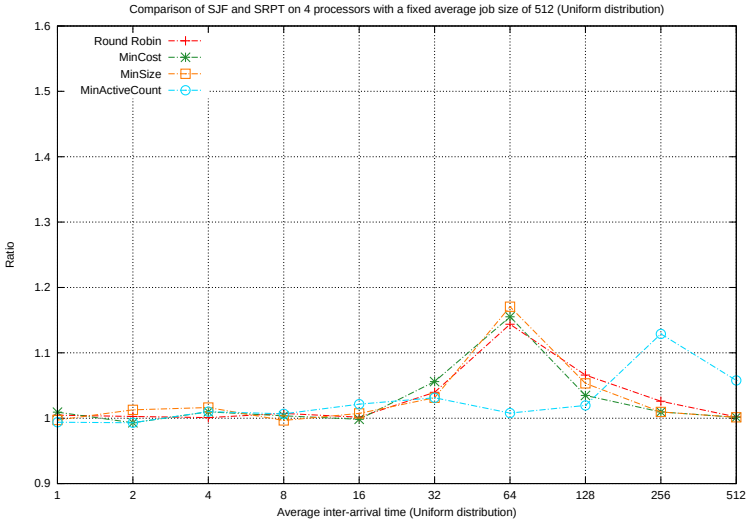
FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy for AJC when using SJF vs. SRPT on 4 processors. Poisson distribution is used for the inter-arrival times while uniform distribution is used for the jobs sizes.



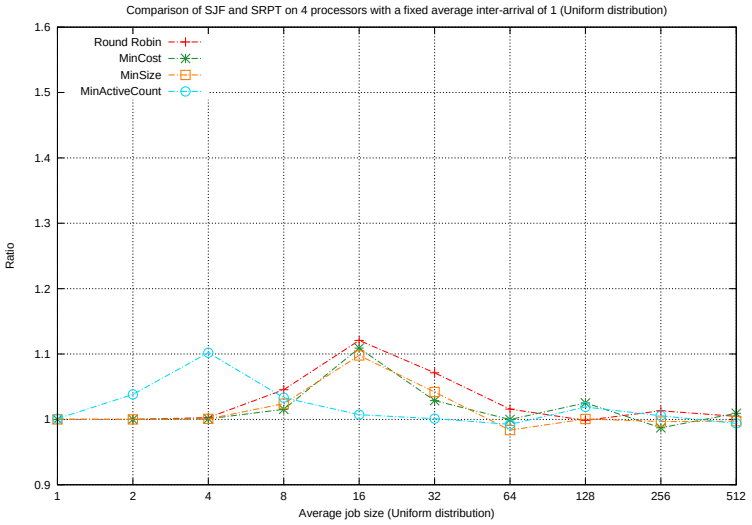
(a) Performance ratio for average job size of 1.



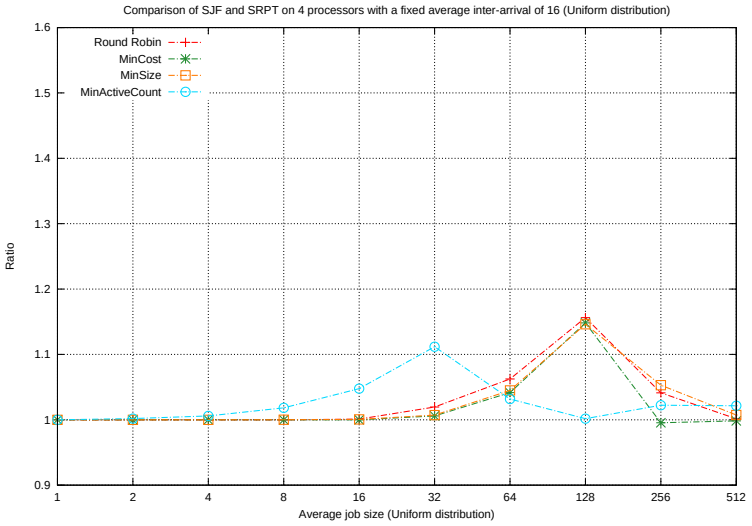
(b) Performance ratio for average job size of 16.



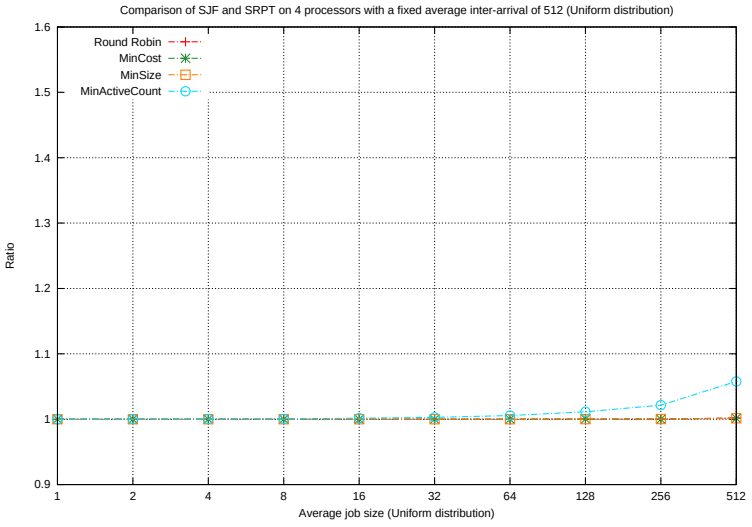
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

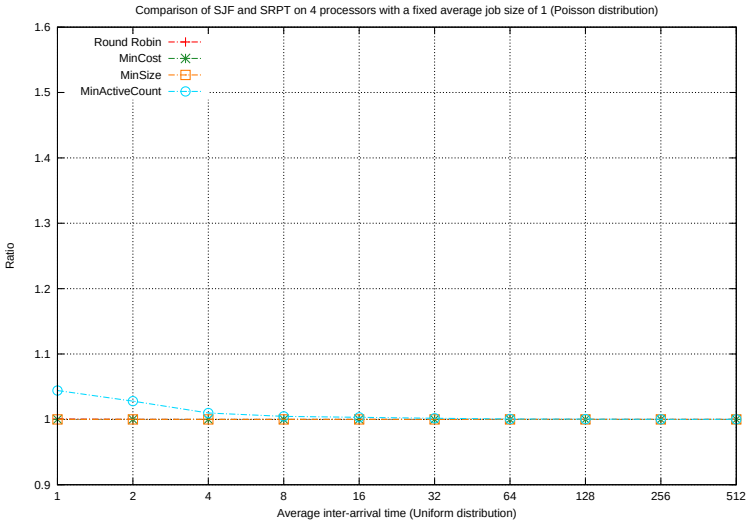


(e) Performance ratio for average inter-arrival time of 16.

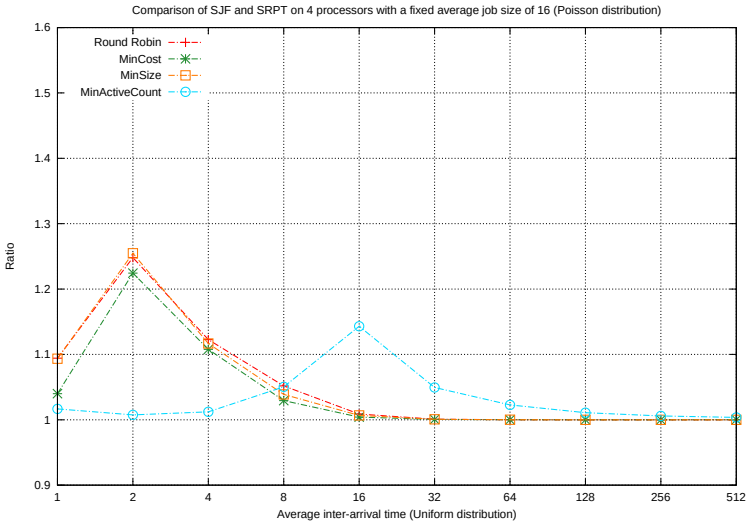


(f) Performance ratio for average inter-arrival time of 512.

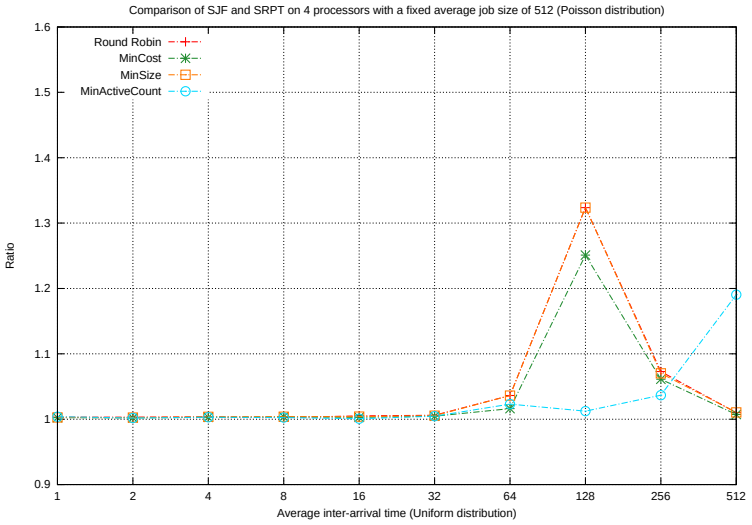
FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy for AJC when using SJF vs. SRPT on 4 processors. Uniform distribution is used for both the inter-arrival times and jobs sizes.



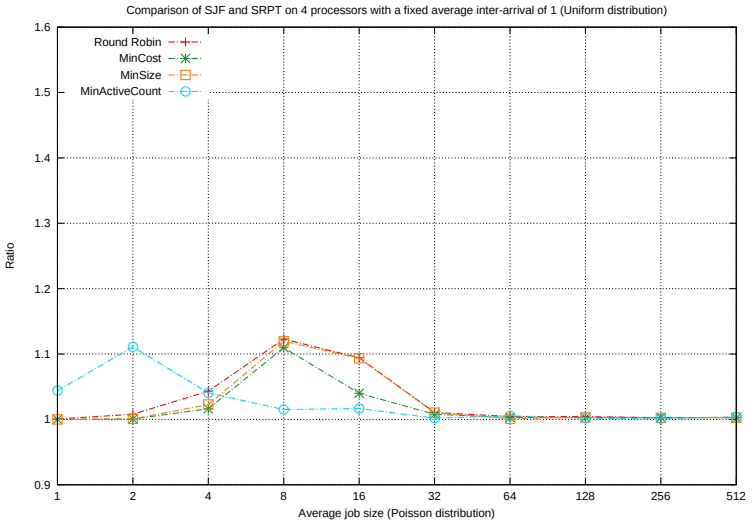
(a) Performance ratio for average job size of 1.



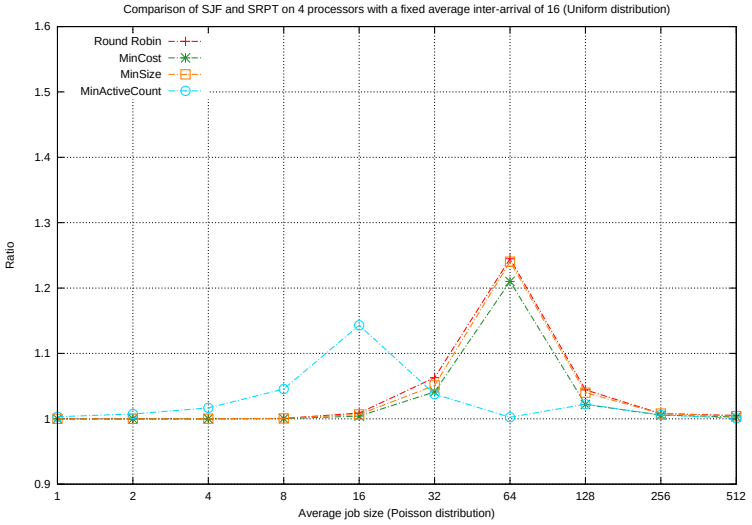
(b) Performance ratio for average job size of 16.



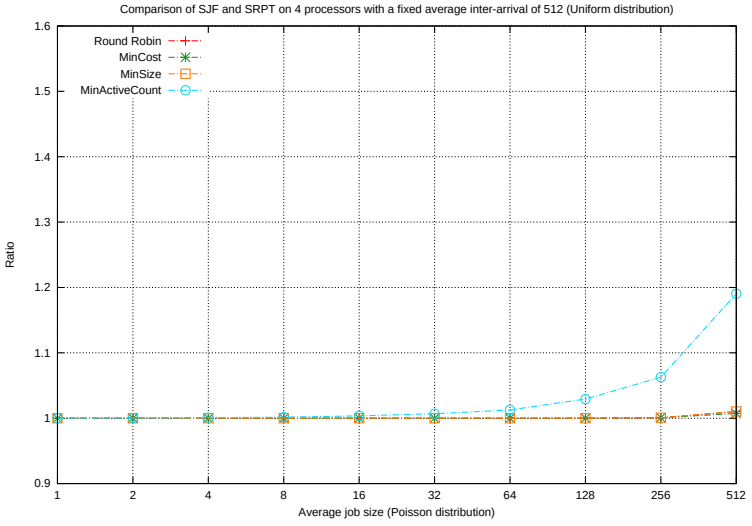
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.



(e) Performance ratio for average inter-arrival time of 16.

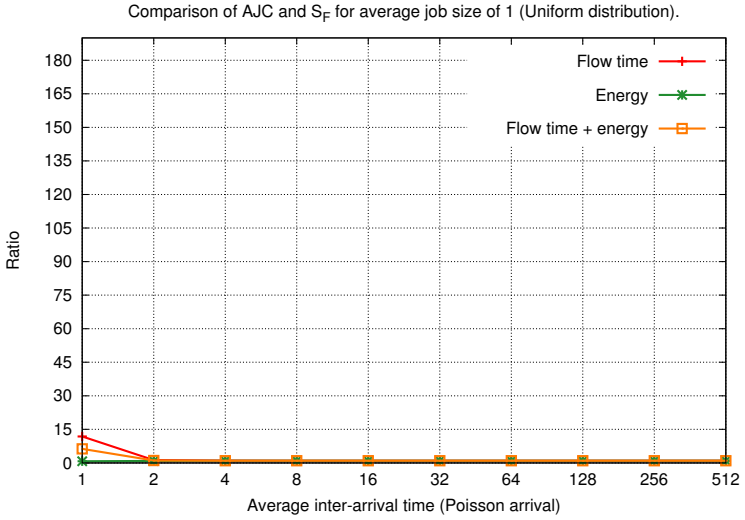


(f) Performance ratio for average inter-arrival time of 512.

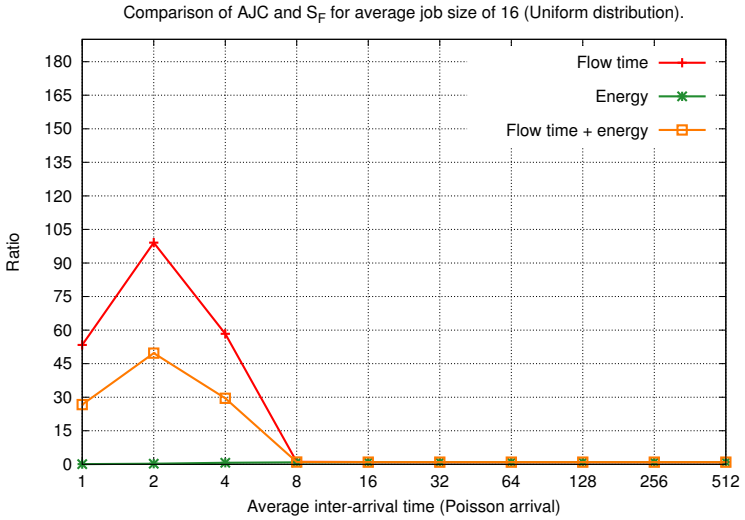
FIGURE A.0: Comparison of the performance ratio based on total flow time plus energy for AJC when using SJF vs. SRPT on 4 processors. Uniform distribution is used for the inter-arrival times and Poisson distribution is used for jobs sizes.

A.2 Results on speed functions

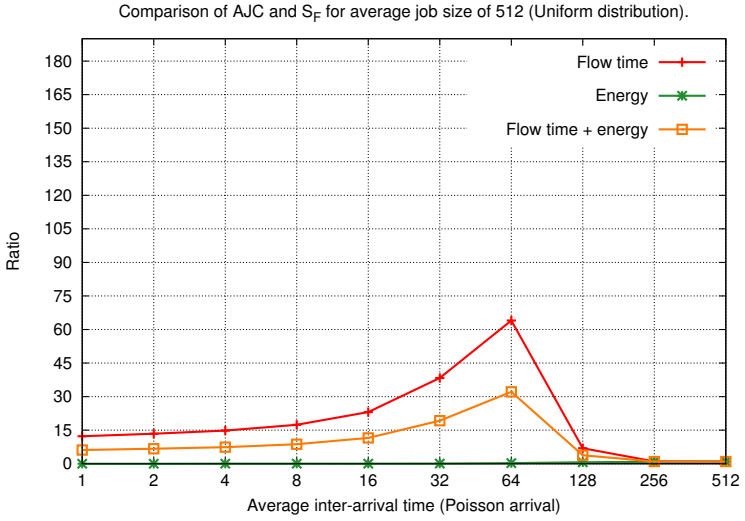
A.2.1 Effectiveness of speed scaling



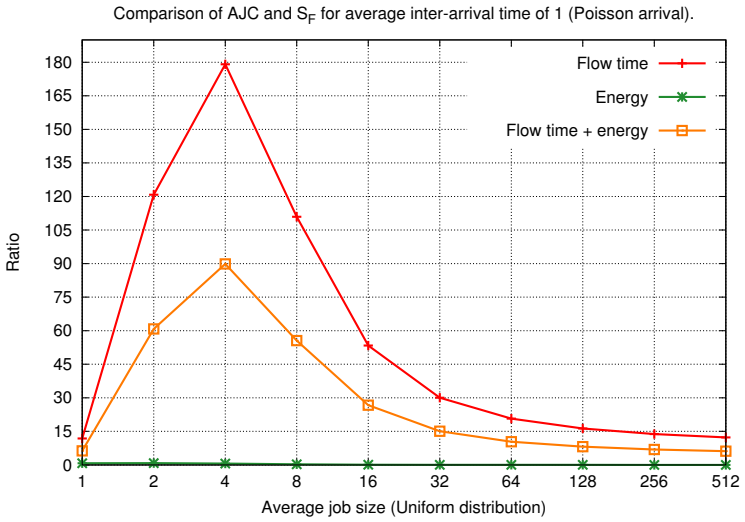
(a) Performance ratio for average job size of 1.



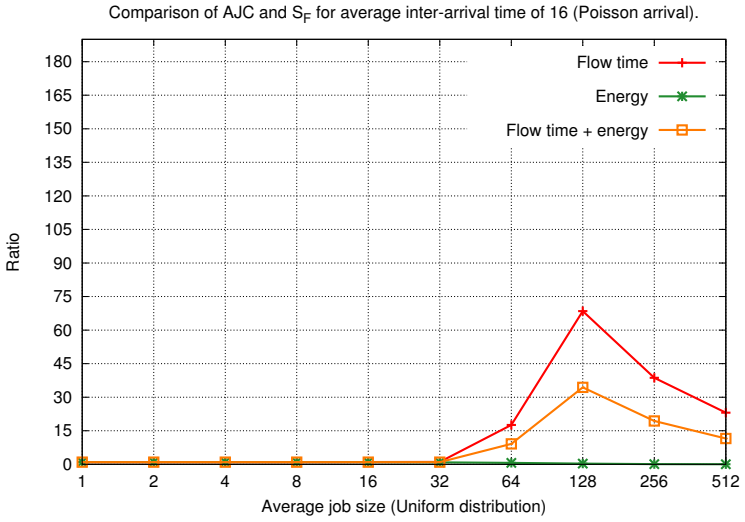
(b) Performance ratio for average job size of 16.



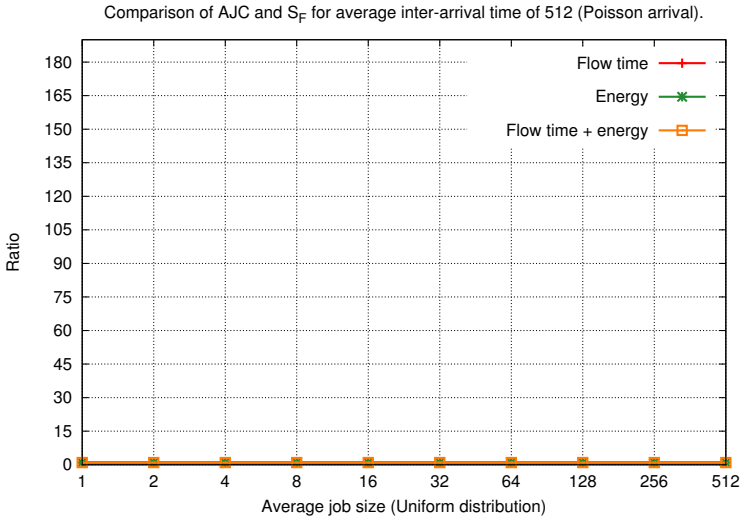
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

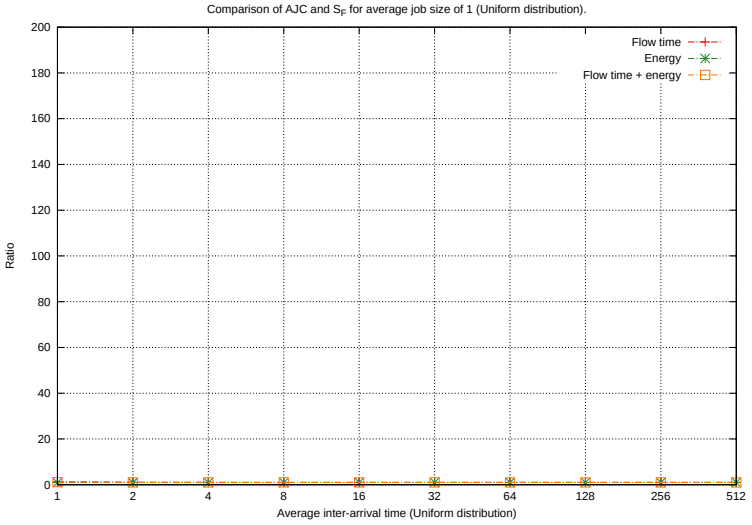


(e) Performance ratio for average inter-arrival time of 16.

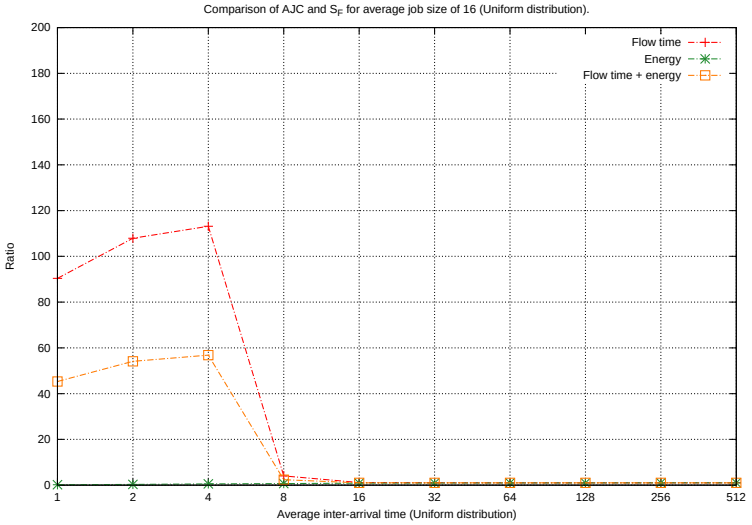


(f) Performance ratio for average inter-arrival time of 512.

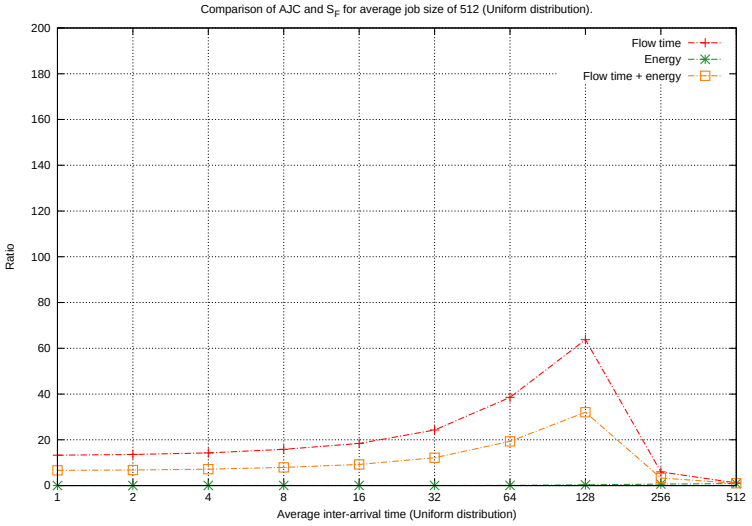
FIGURE A.0: *Effectiveness of speed scaling*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed heuristic that uses a fixed speed of 1 on a single processor. Poisson distribution is used for inter-arrival times and uniform distribution is used for job sizes.



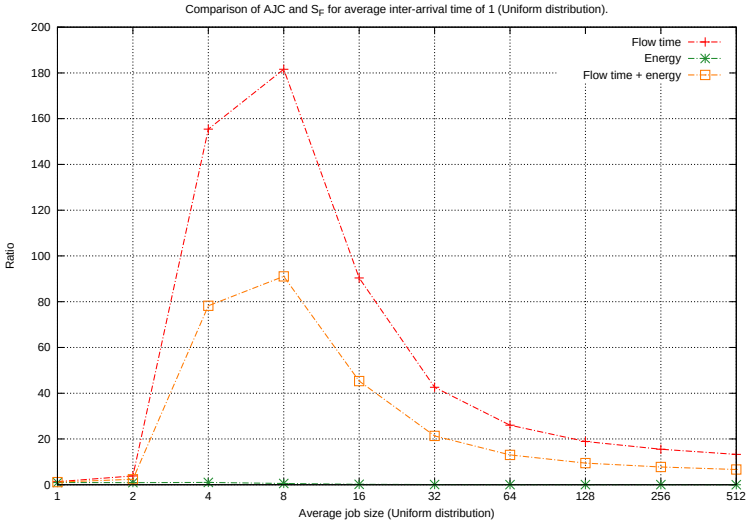
(a) Performance ratio for average job size of 1.



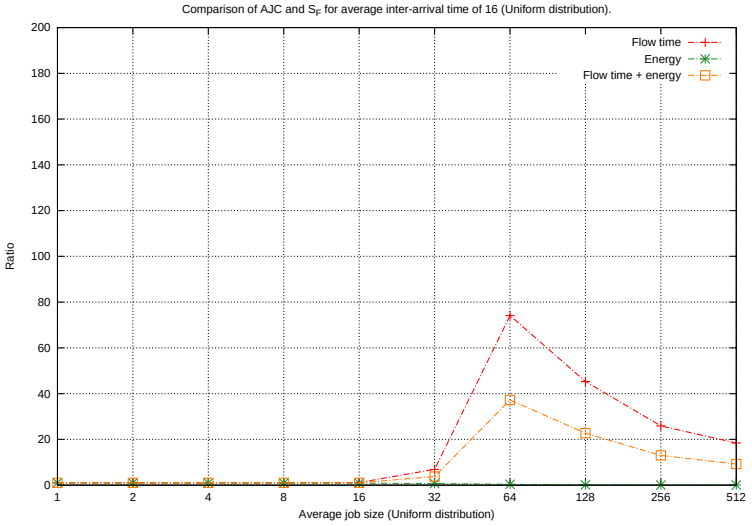
(b) Performance ratio for average job size of 16.



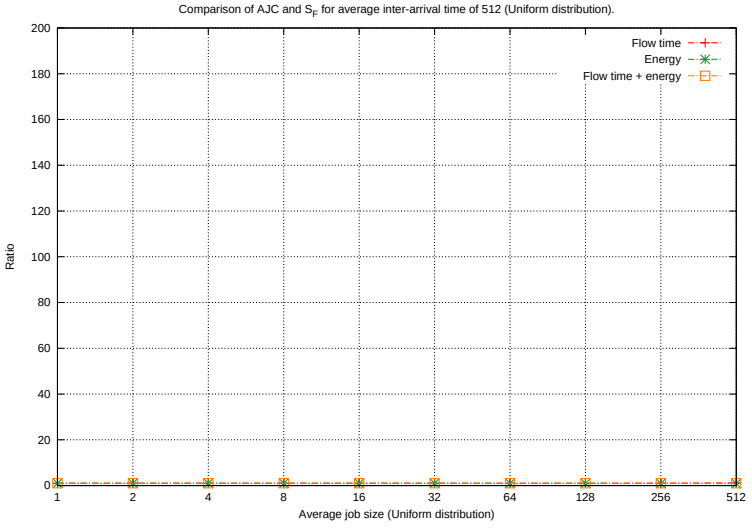
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

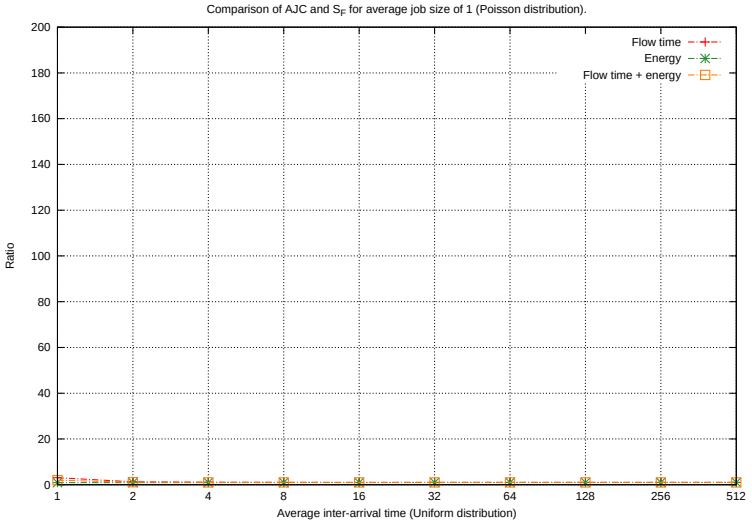


(e) Performance ratio for average inter-arrival time of 16.

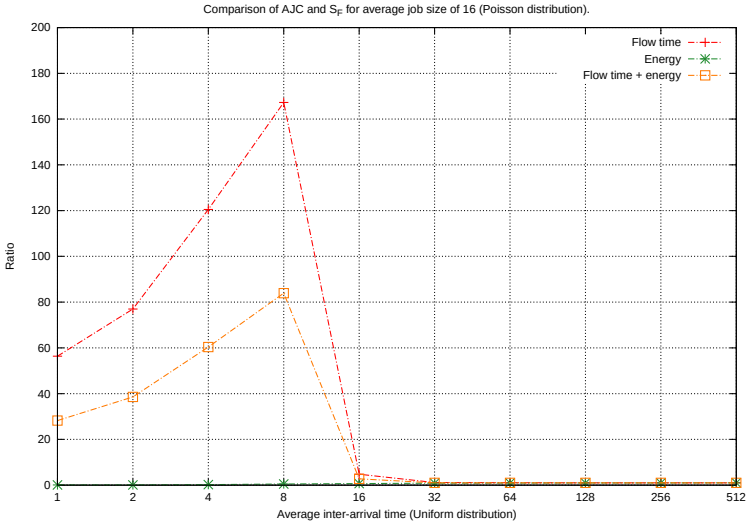


(f) Performance ratio for average inter-arrival time of 512.

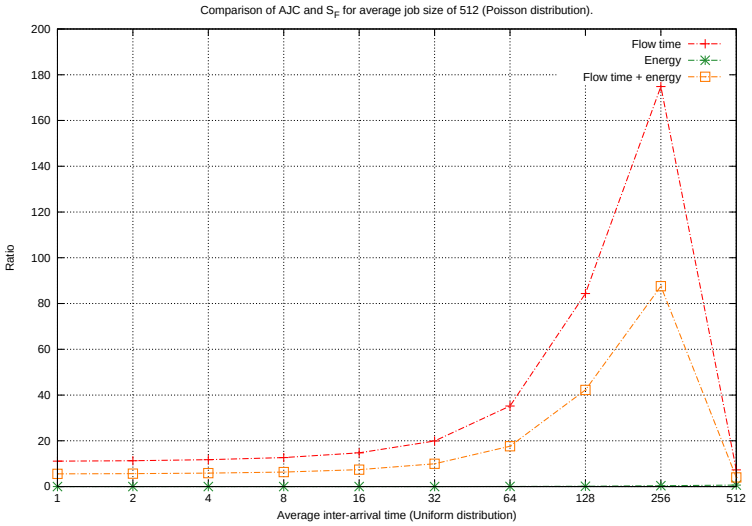
FIGURE A.0: *Effectiveness of speed scaling*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed heuristic that uses a fixed speed of 1 on a single processor. Uniform distribution is used for both inter-arrival times and job sizes.



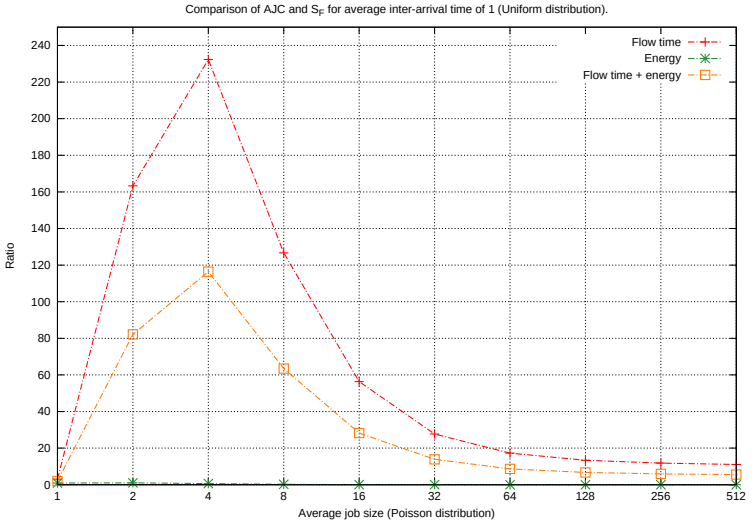
(a) Performance ratio for average job size of 1.



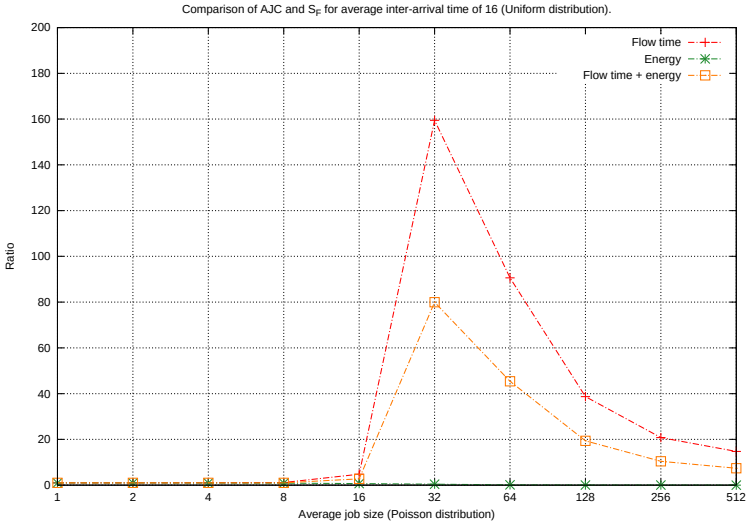
(b) Performance ratio for average job size of 16.



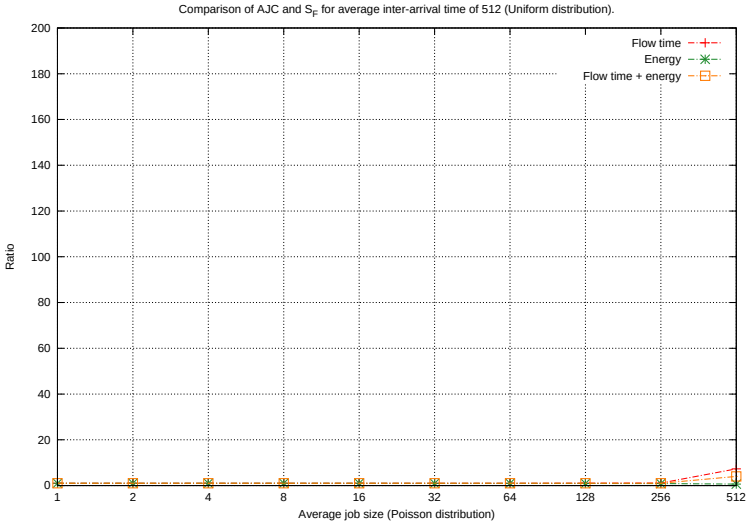
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.



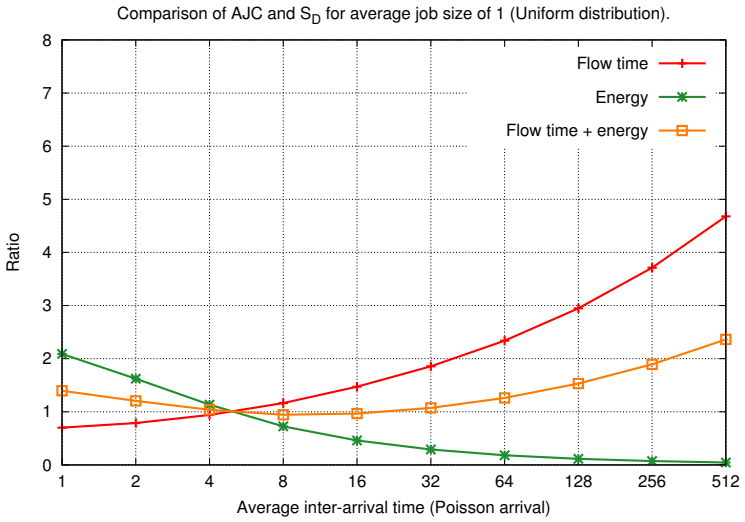
(e) Performance ratio for average inter-arrival time of 16.



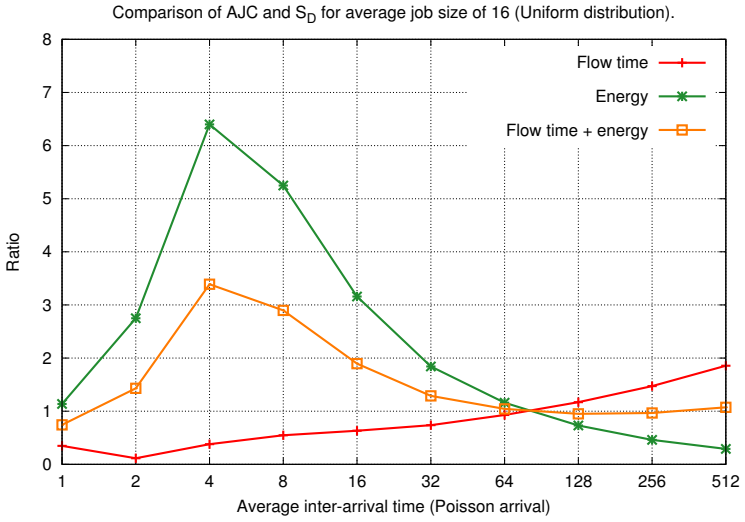
(f) Performance ratio for average inter-arrival time of 512.

FIGURE A.0: *Effectiveness of speed scaling*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed heuristic that uses a fixed speed of 1 on a single processor. Uniform distribution is used inter-arrival times and Poisson distribution is used for job sizes.

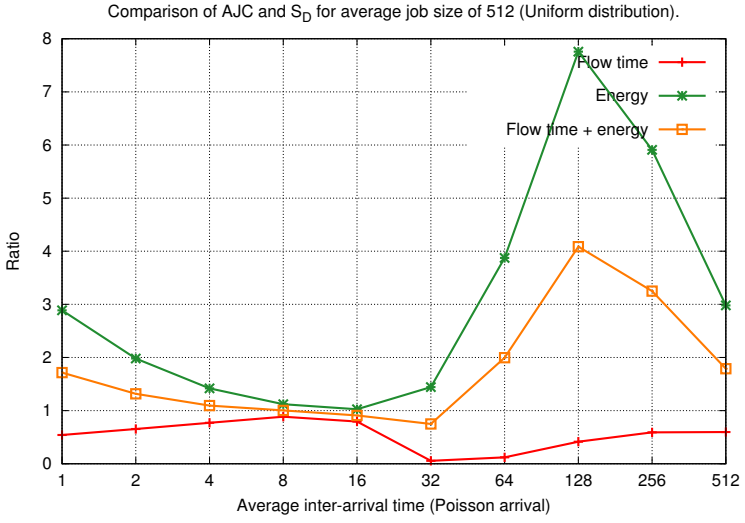
A.2.2 Speed scaling vs. semi-clairvoyant fixed speed function



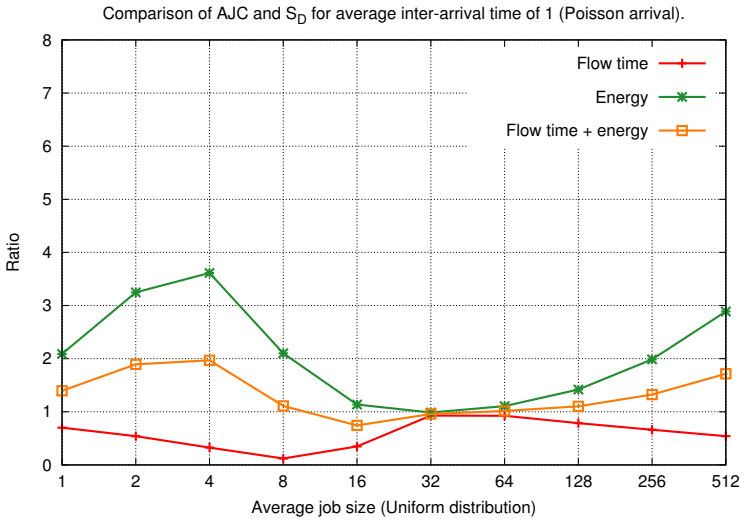
(a) Performance ratio for average job size of 1.



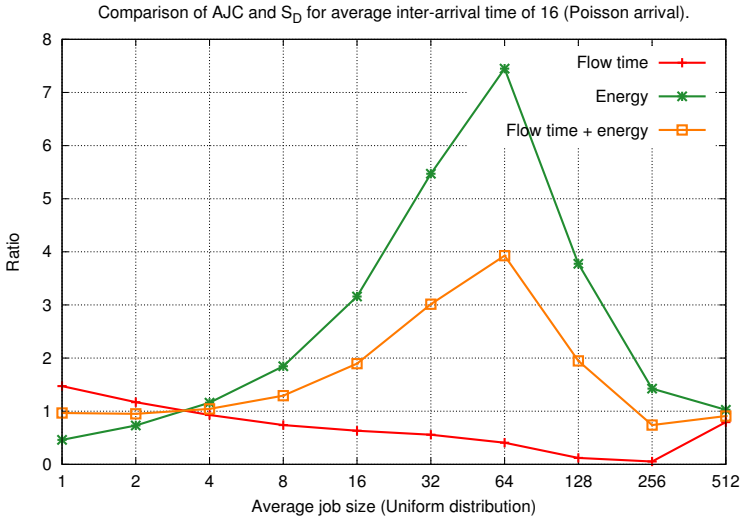
(b) Performance ratio for average job size of 16.



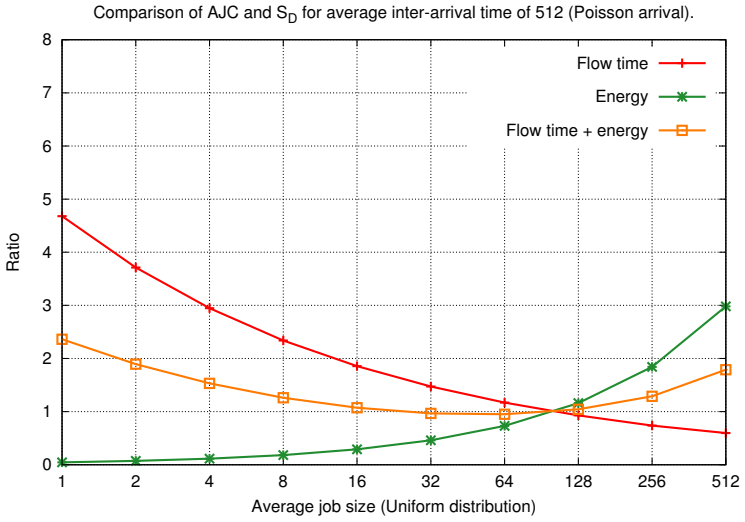
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

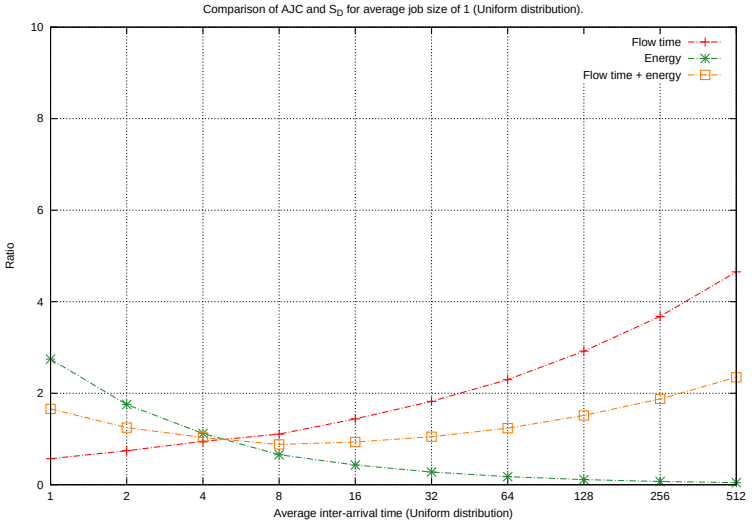


(e) Performance ratio for average inter-arrival time of 16.

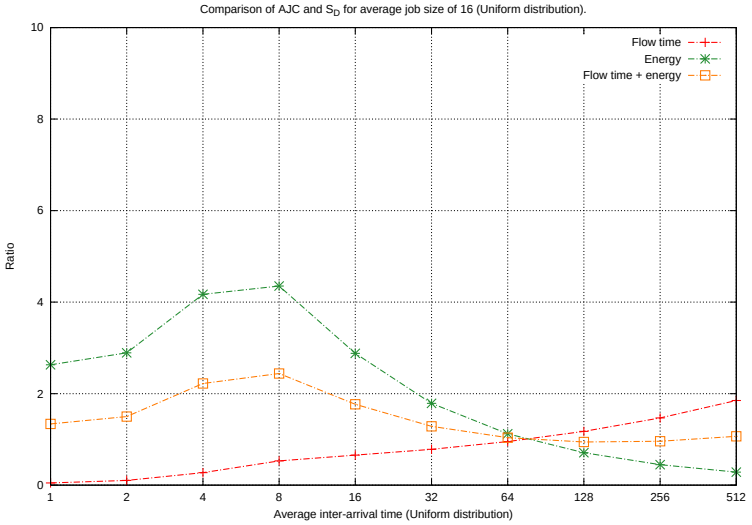


(f) Performance ratio for average inter-arrival time of 512.

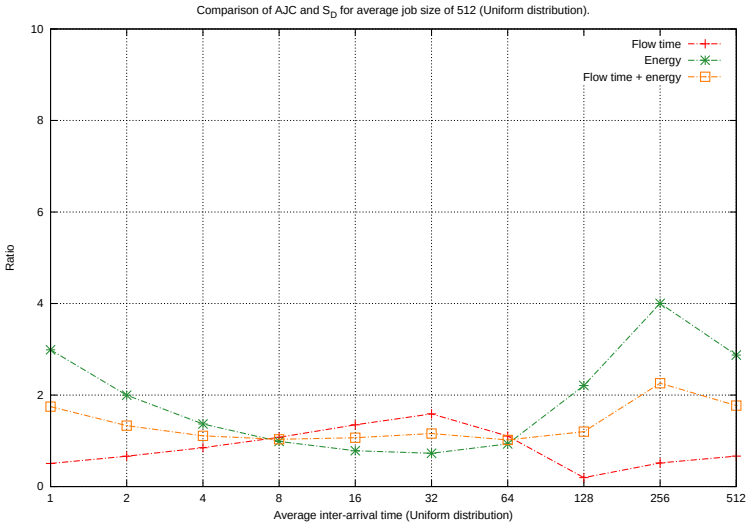
FIGURE A.0: *Speed scaling vs. semi-clairvoyant fixed speed function*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed function that has some information about the job set. Poisson distribution is used for inter-arrival times and uniform distribution is used for job sizes.



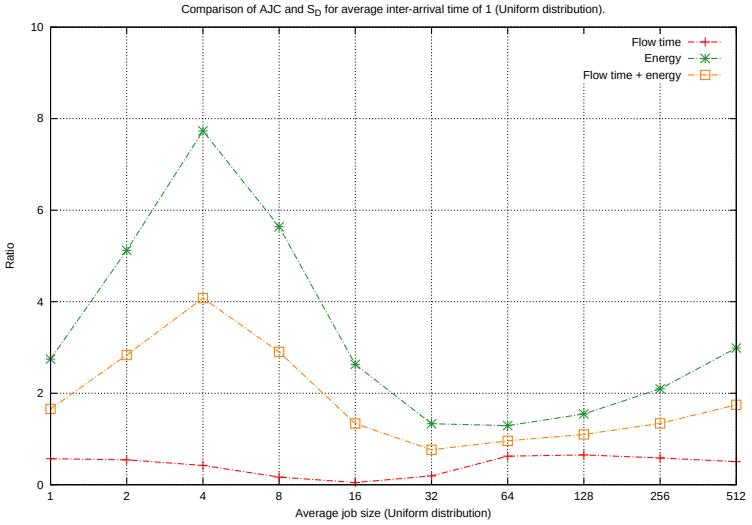
(a) Performance ratio for average job size of 1.



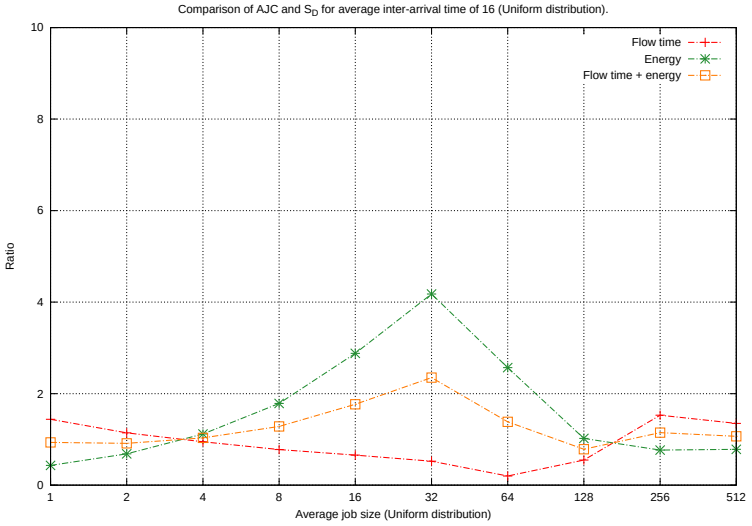
(b) Performance ratio for average job size of 16.



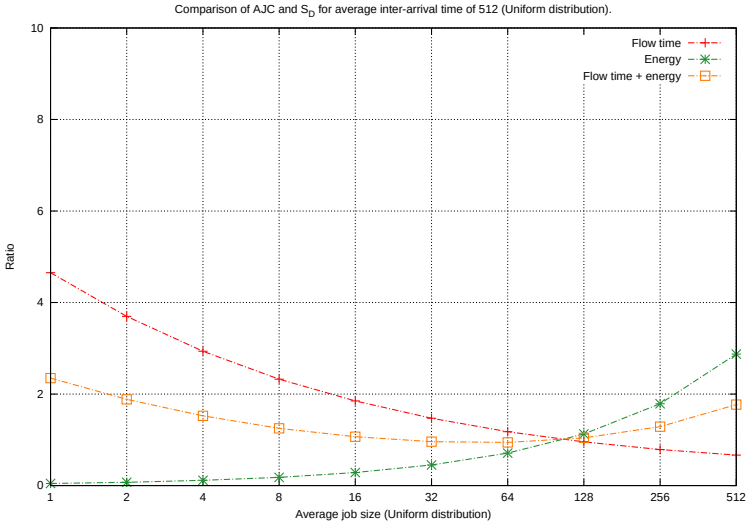
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

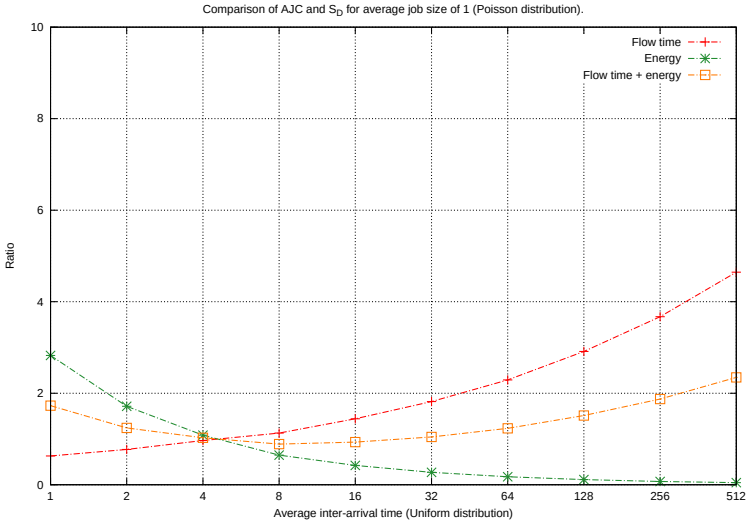


(e) Performance ratio for average inter-arrival time of 16.

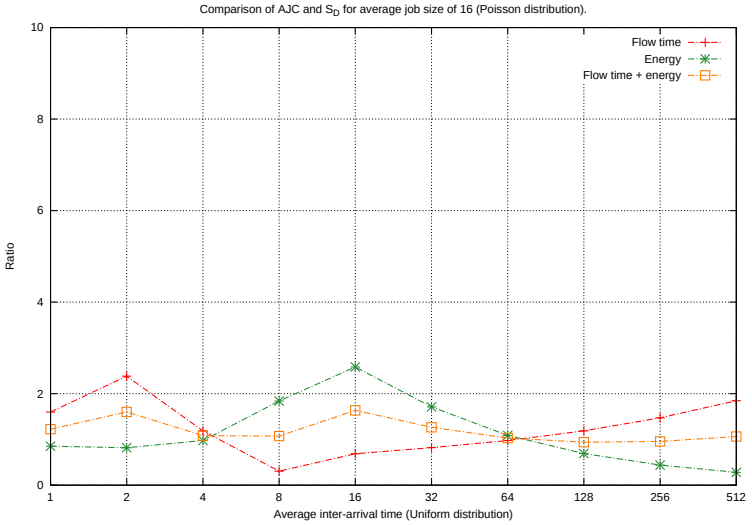


(f) Performance ratio for average inter-arrival time of 512.

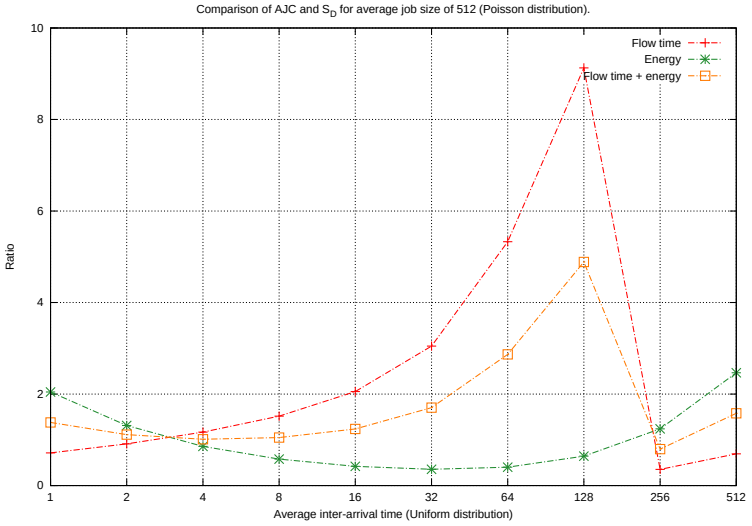
FIGURE A.0: *Speed scaling vs. semi-clairvoyant fixed speed function*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed function that has some information about the job set. Uniform distribution is used for both inter-arrival times and job sizes.



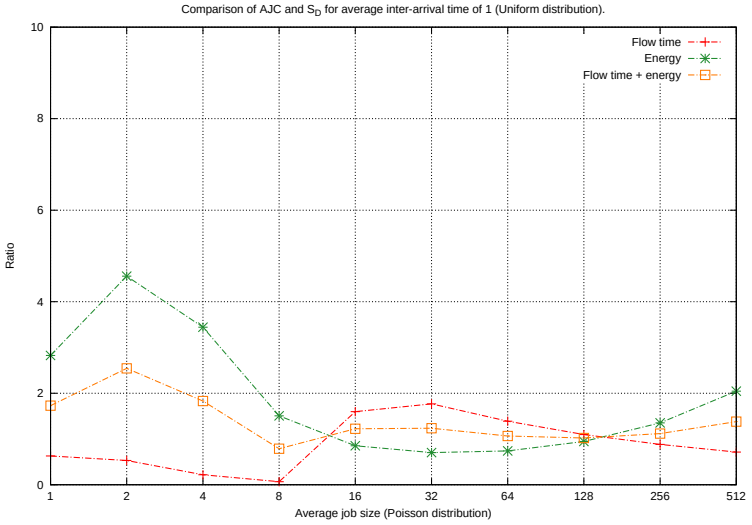
(a) Performance ratio for average job size of 1.



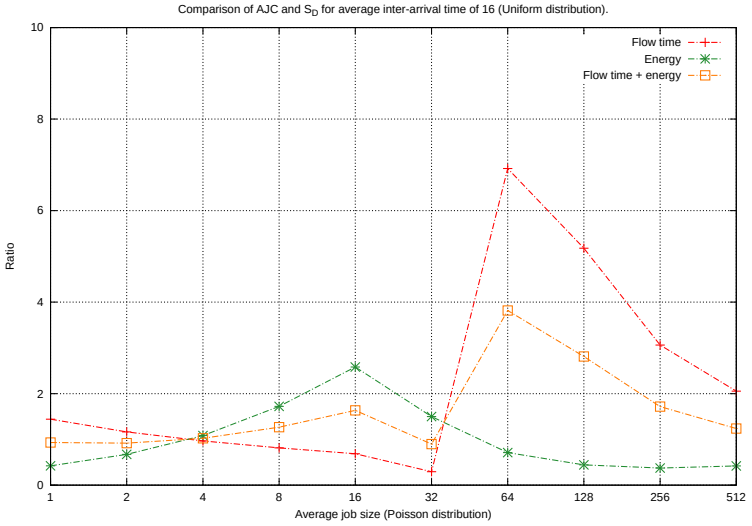
(b) Performance ratio for average job size of 16.



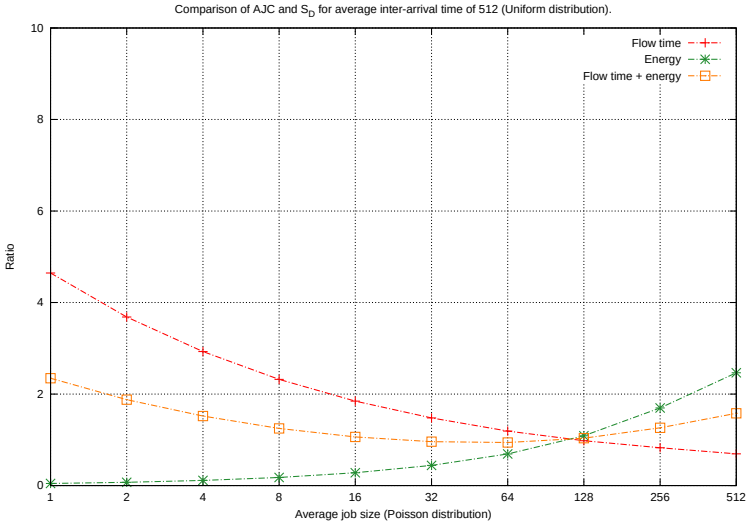
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.



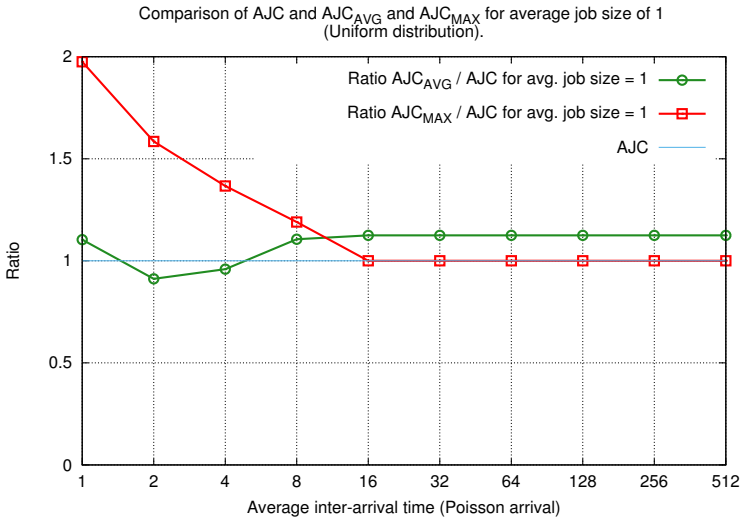
(e) Performance ratio for average inter-arrival time of 16.



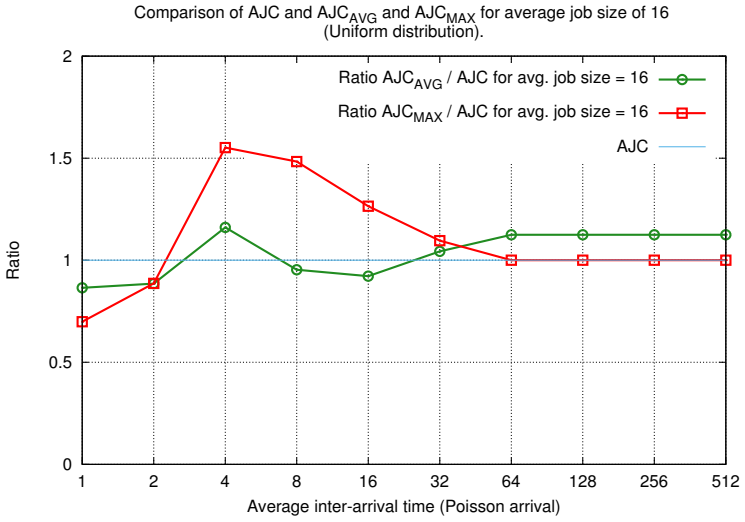
(f) Performance ratio for average inter-arrival time of 512.

FIGURE A.0: *Speed scaling vs. semi-clairvoyant fixed speed function*: Comparison of the performance ratio based on total flow time plus energy between AJC and a fixed speed function that has some information about the job set. Uniform distribution is used for inter-arrival times and Poisson distribution is used for job sizes.

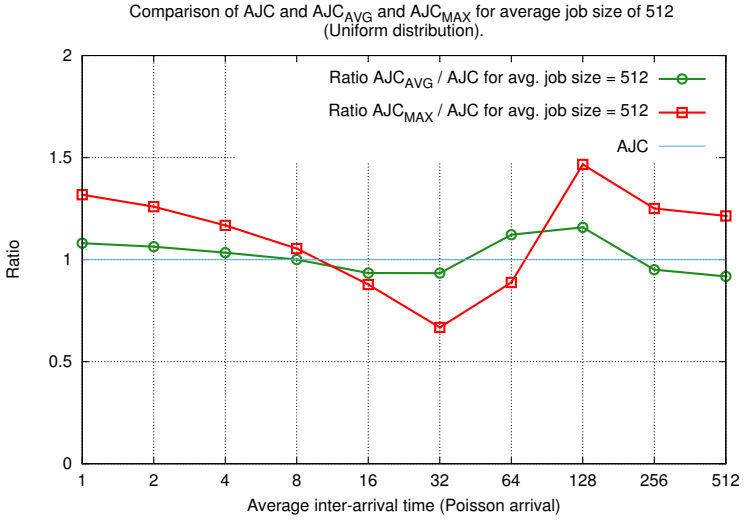
A.2.3 Effectiveness of AJC speed spectrum



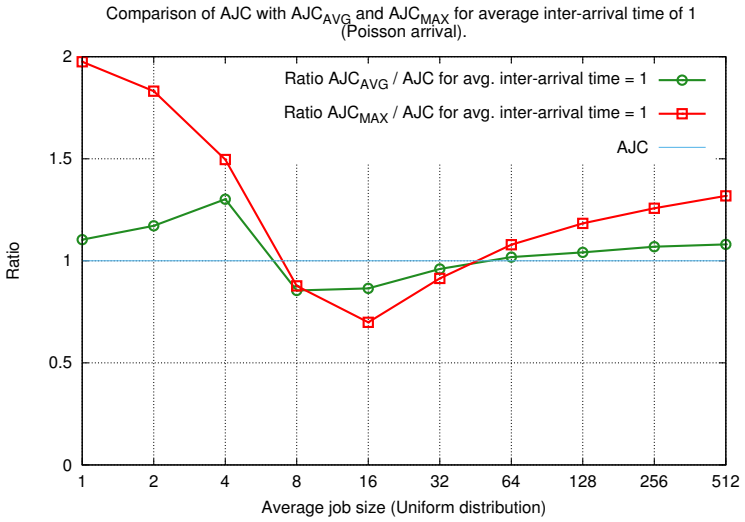
(a) Performance ratio for average job size of 1.



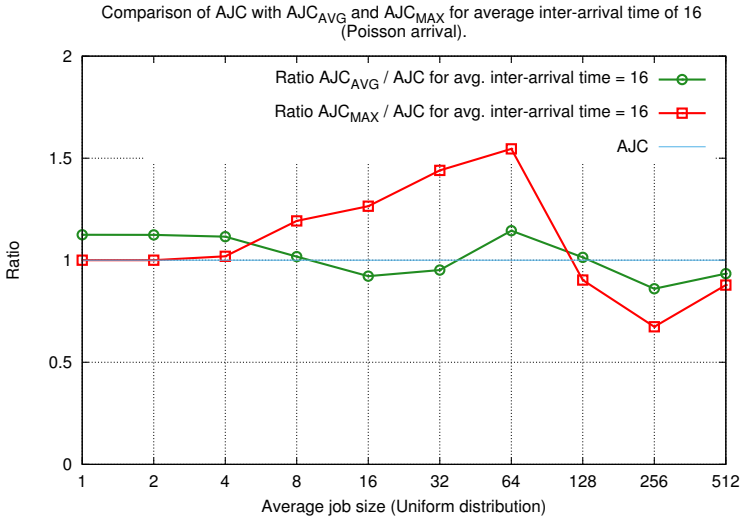
(b) Performance ratio for average job size of 16.



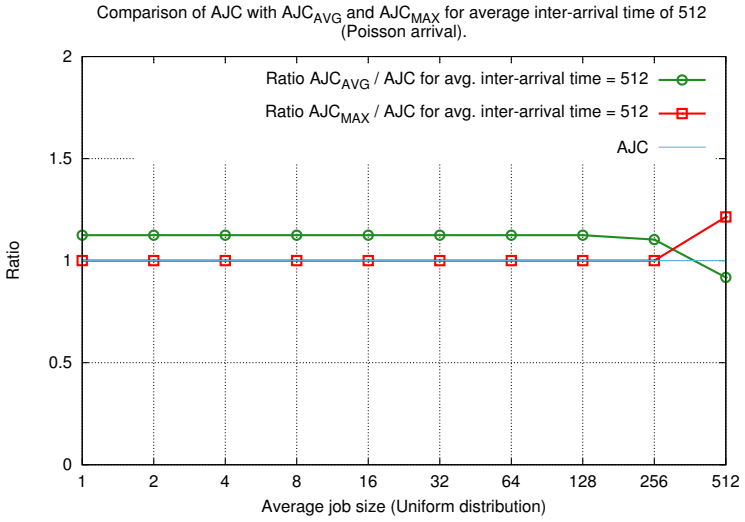
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

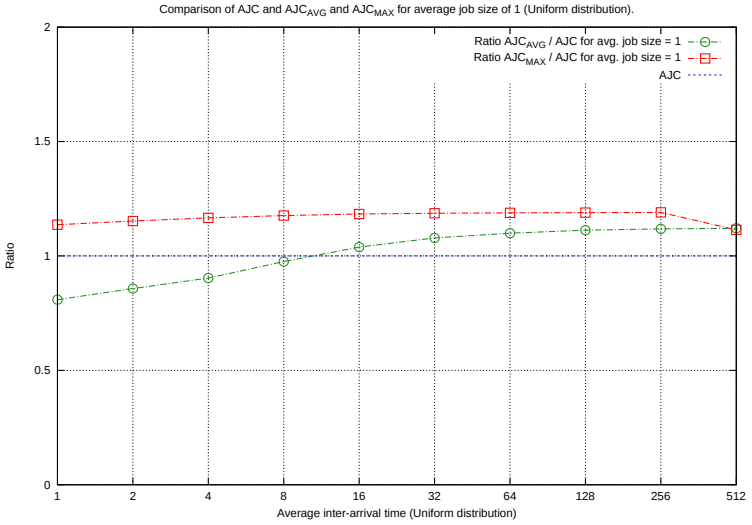


(e) Performance ratio for average inter-arrival time of 16.

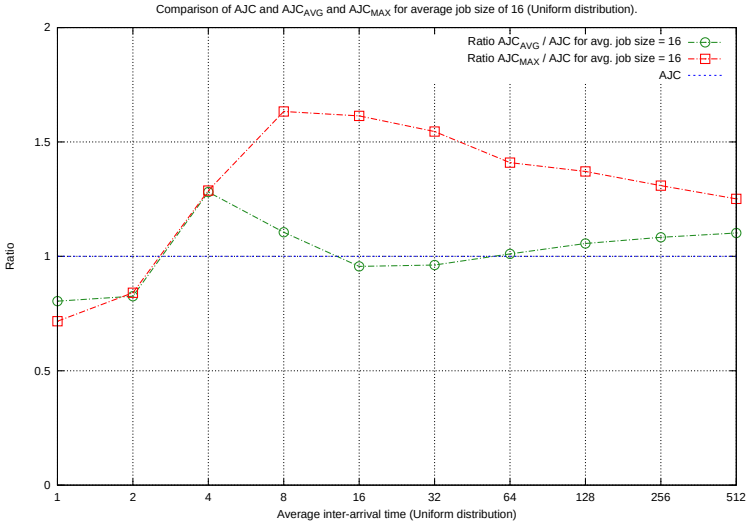


(f) Performance ratio for average inter-arrival time of 512.

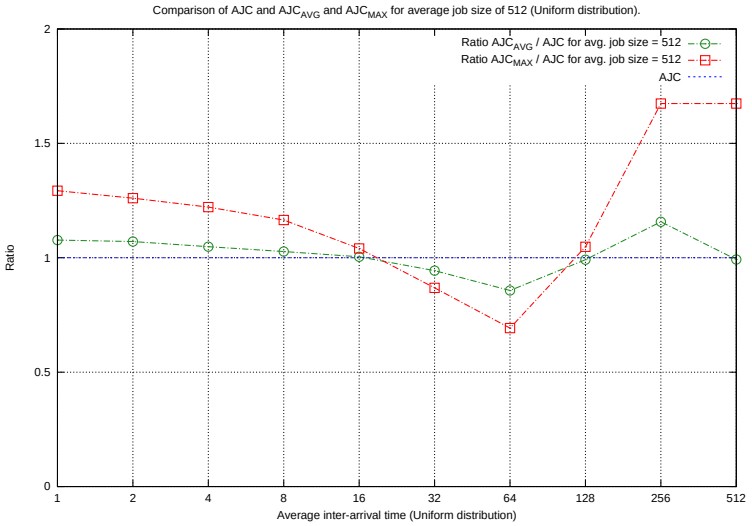
FIGURE A.0: *Effectiveness of AJC speed spectrum*: Comparison of AJC to a fixed speed function that uses, as fixed speed values, the average and maximum speeds obtained from a prior AJC run. Results show the performance ratio of the total flow time plus energy of fixed speed functions vs. AJC. Poisson distribution is used for inter-arrival times and uniform distribution is used for job sizes.



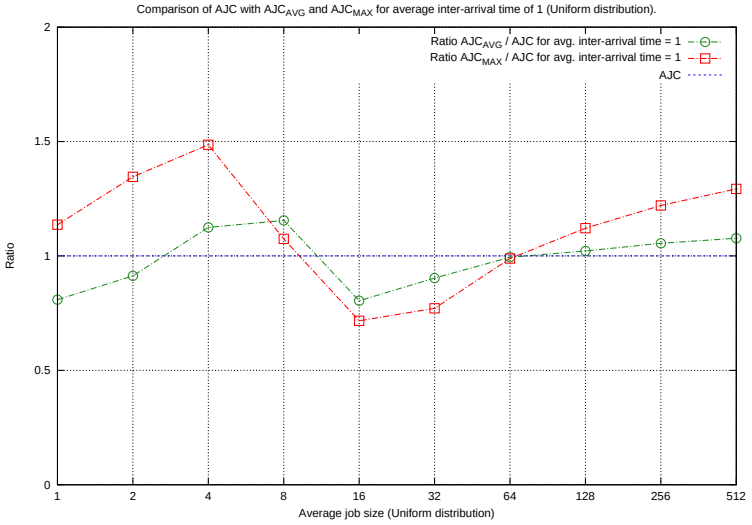
(a) Performance ratio for average job size of 1.



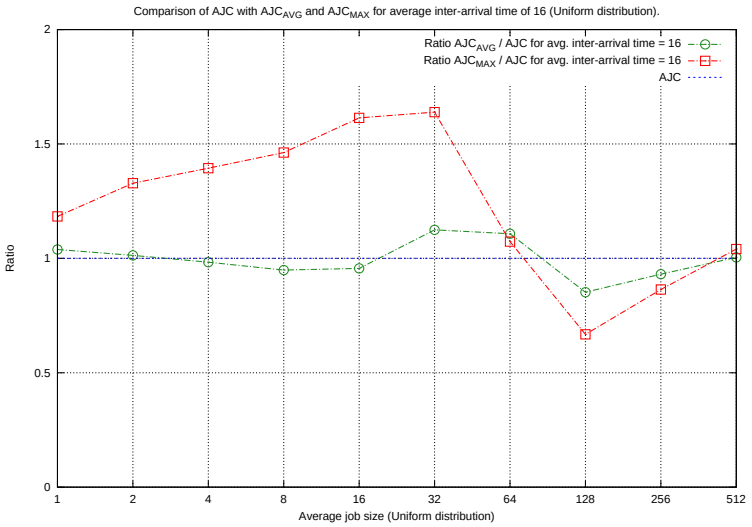
(b) Performance ratio for average job size of 16.



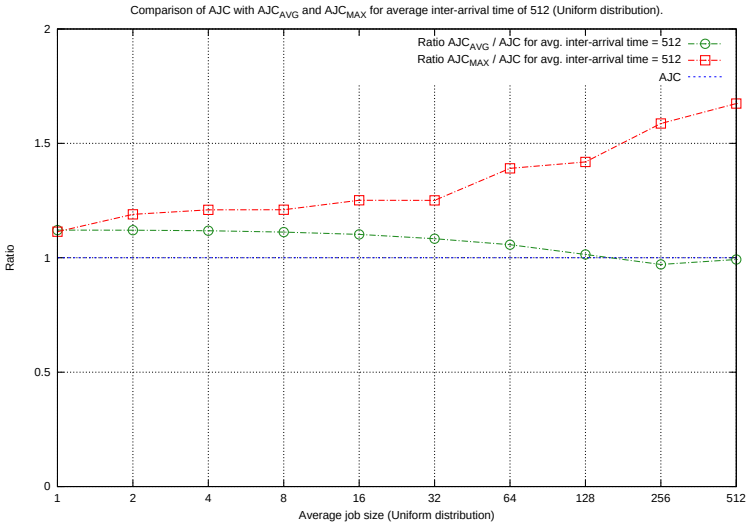
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.

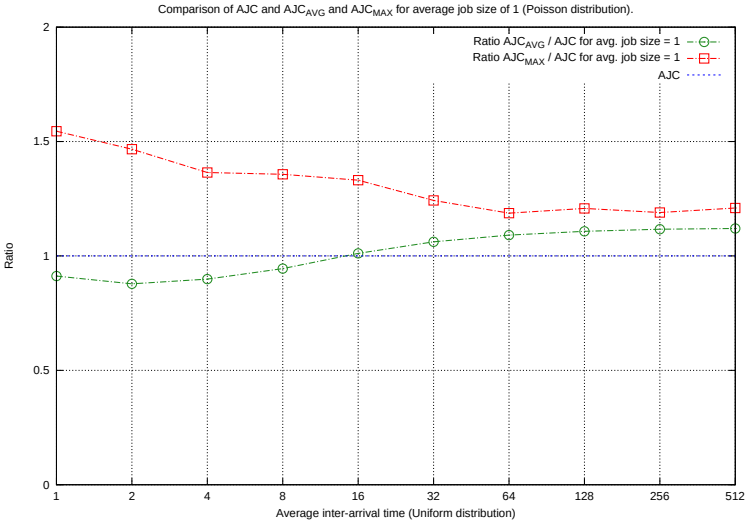


(e) Performance ratio for average inter-arrival time of 16.

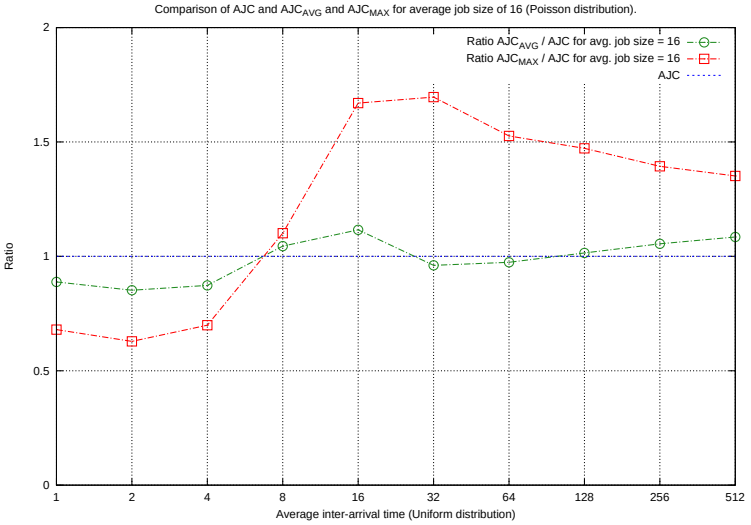


(f) Performance ratio for average inter-arrival time of 512.

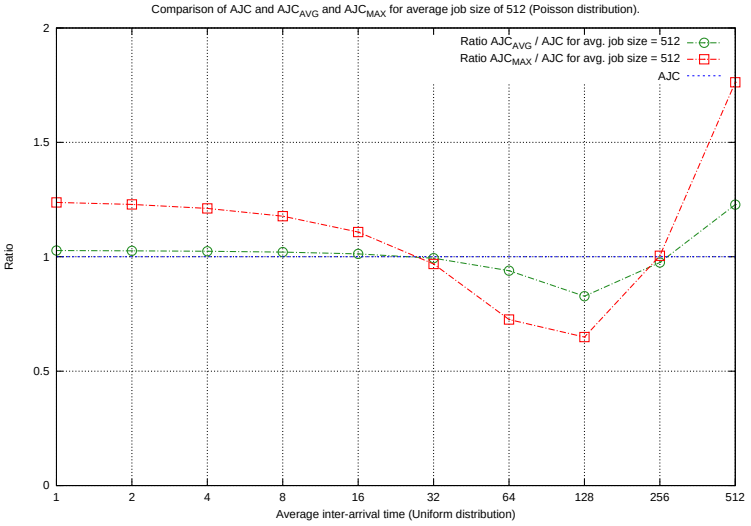
FIGURE A.0: *Effectiveness of AJC speed spectrum*: Comparison of AJC to a fixed speed function that uses, as fixed speed values, the average and maximum speeds obtained from a prior AJC run. Results show the performance ratio of the total flow time plus energy of fixed speed functions vs. AJC. Uniform distribution is used for both inter-arrival times and job sizes.



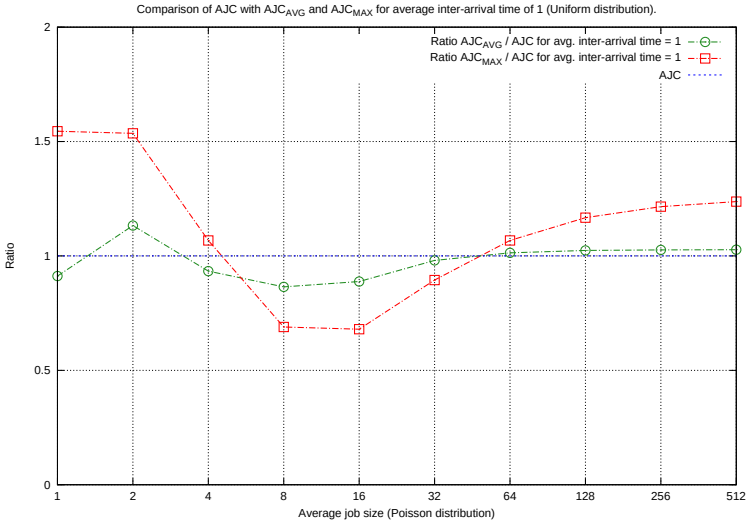
(a) Performance ratio for average job size of 1.



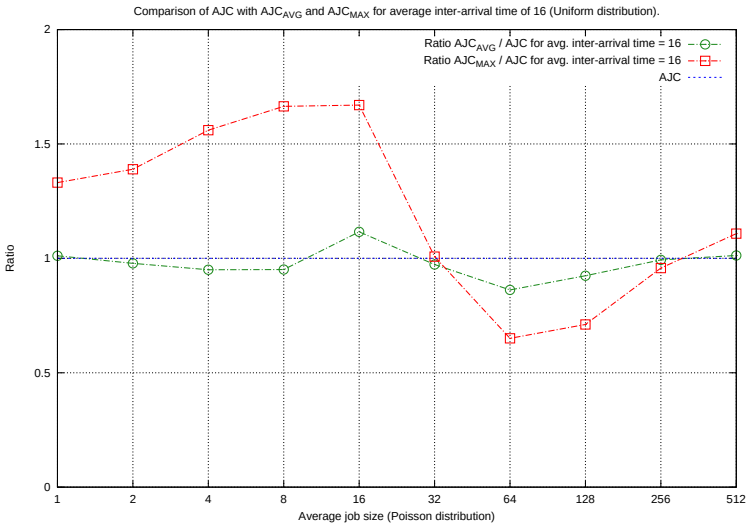
(b) Performance ratio for average job size of 16.



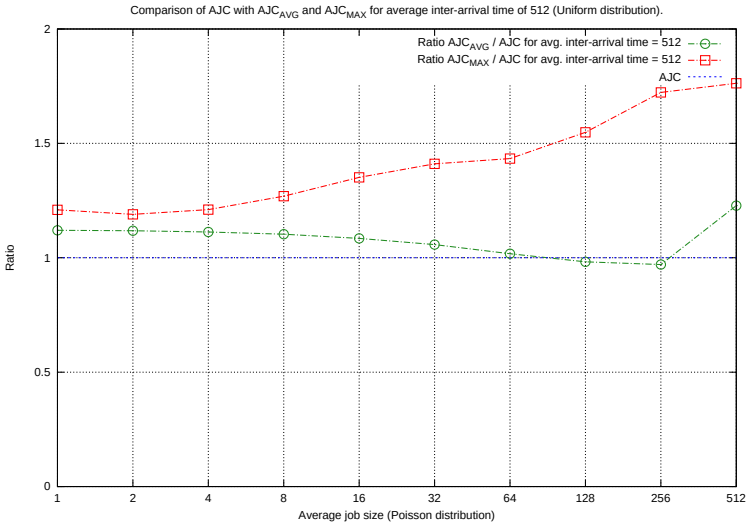
(c) Performance ratio for average job size of 512.



(d) Performance ratio for average inter-arrival time of 1.



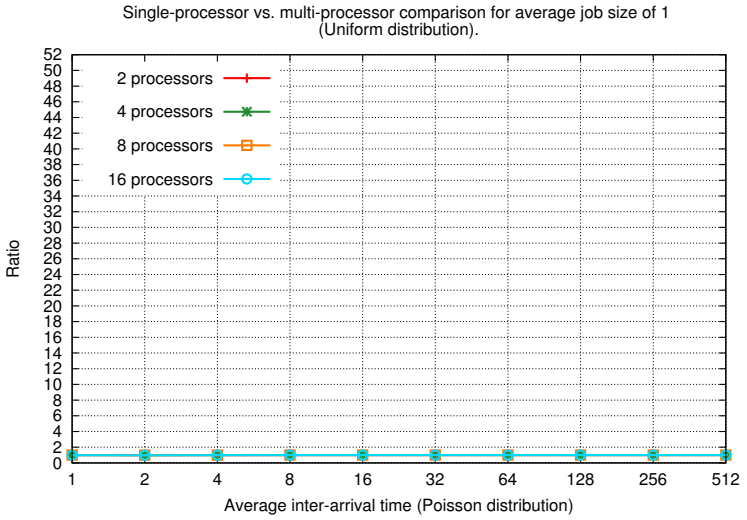
(e) Performance ratio for average inter-arrival time of 16.



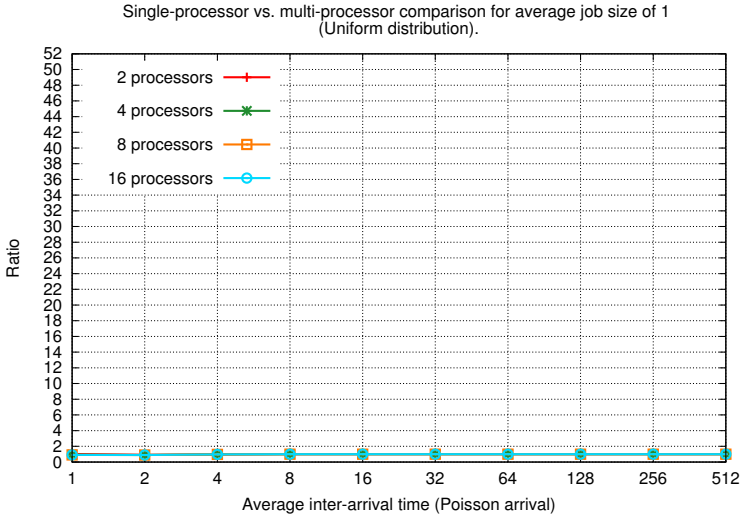
(f) Performance ratio for average inter-arrival time of 512.

FIGURE A.0: *Effectiveness of AJC speed spectrum*: Comparison of AJC to a fixed speed function that uses, as fixed speed values, the average and maximum speeds obtained from a prior AJC run. Results show the performance ratio of the total flow time plus energy of fixed speed functions vs. AJC. Uniform distribution is used for inter-arrival times and Poisson distribution is used for job sizes.

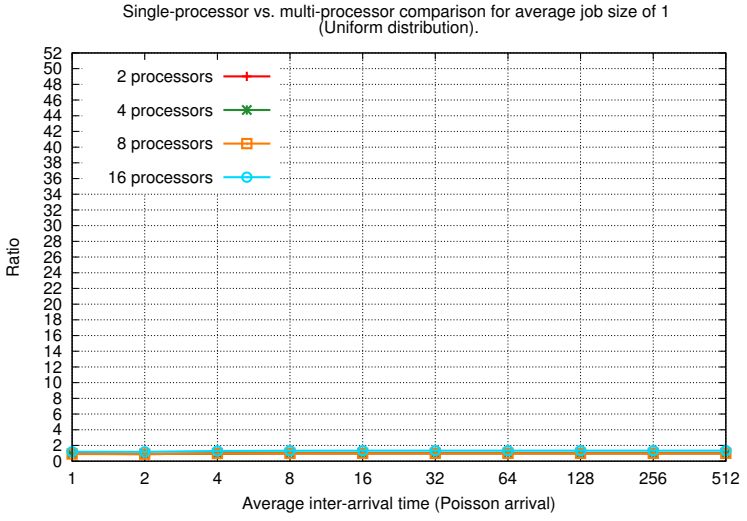
A.3 Results on processor allocation strategies



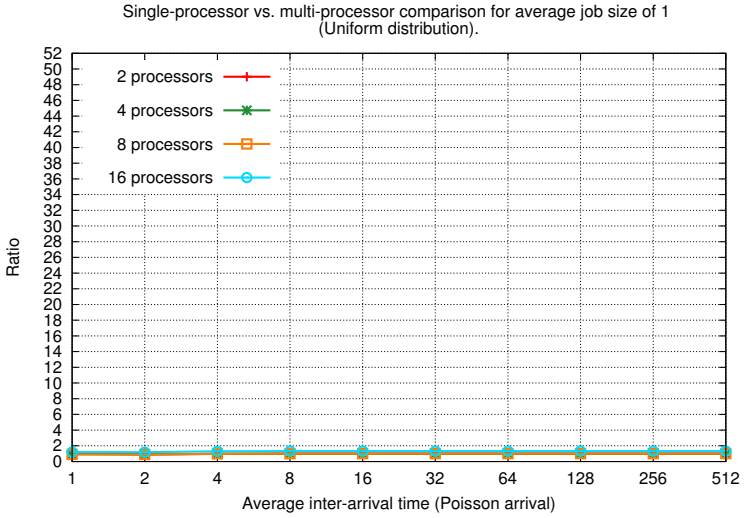
(a) *MinActiveCount



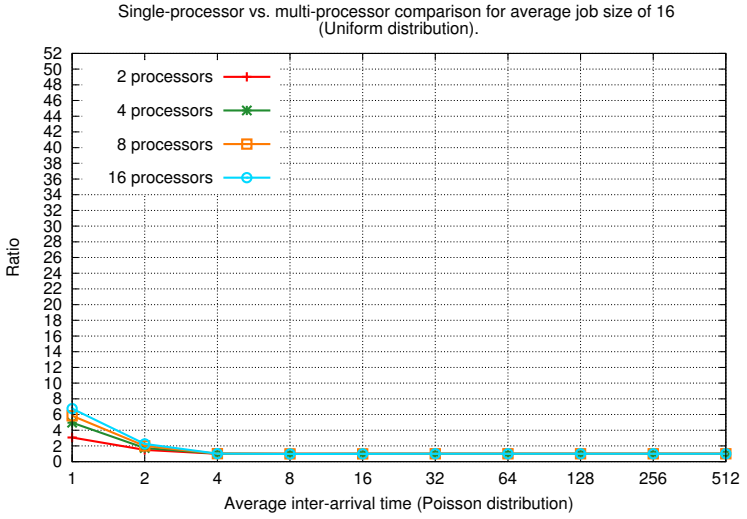
(b) *MinCost



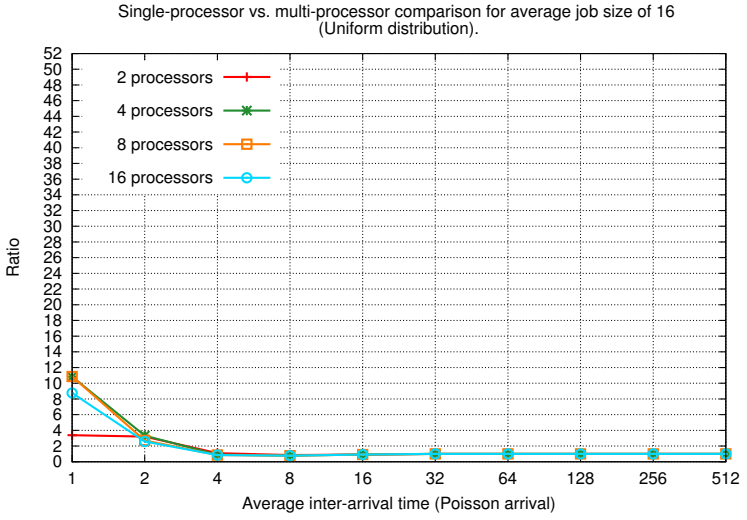
(c) *MinSize



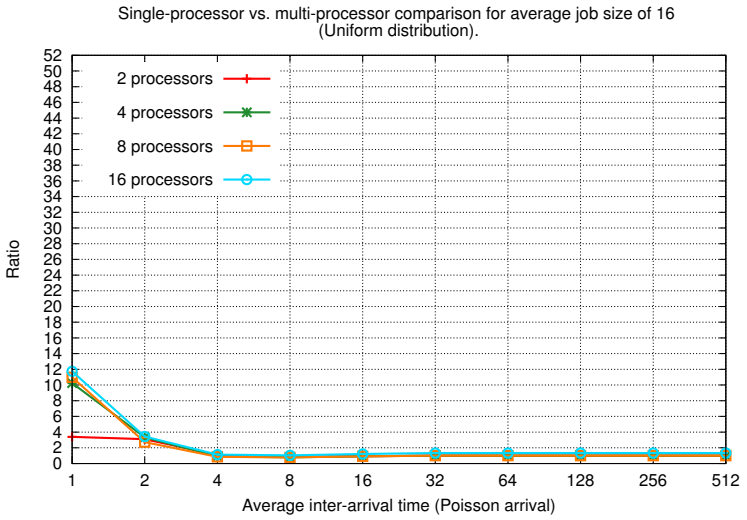
(d) RoundRobin



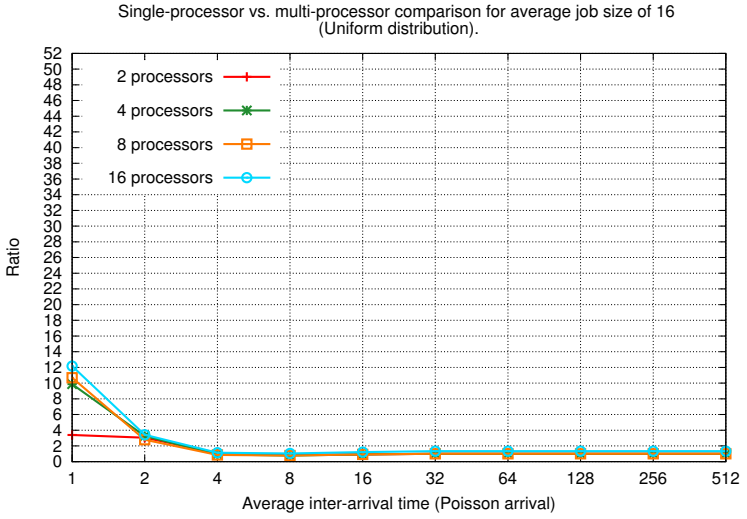
(e) *MinActiveCount



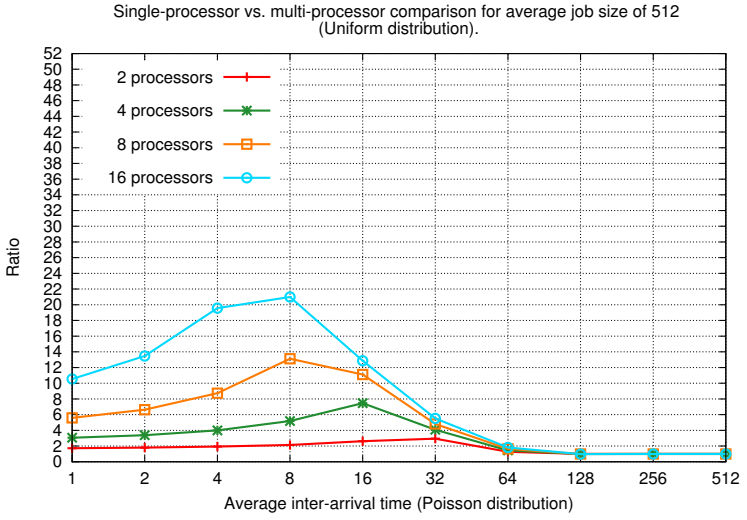
(f) *MinCost



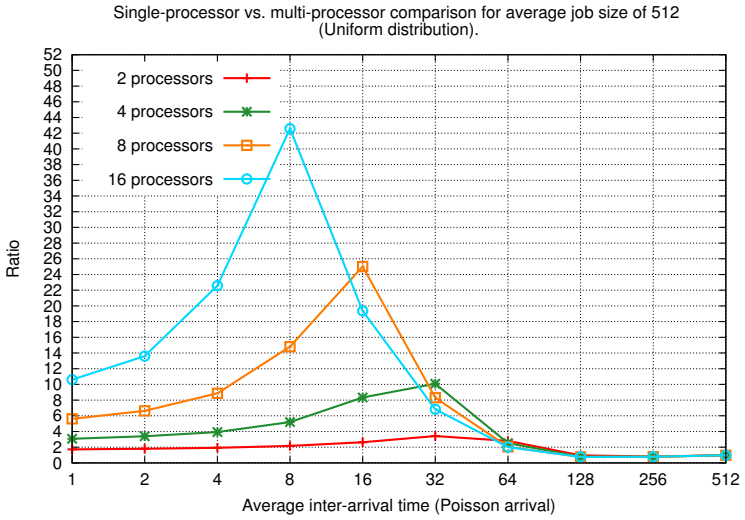
(g) *MinSize



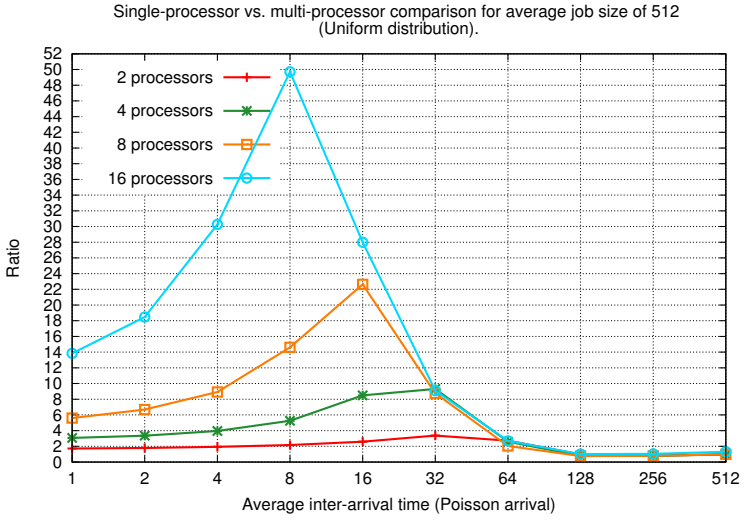
(h) RoundRobin



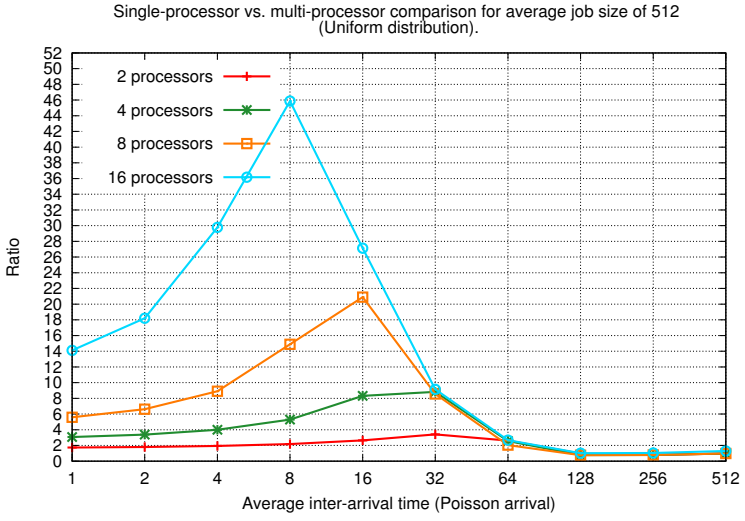
(i) *MinActiveCount



(j) *MinCost

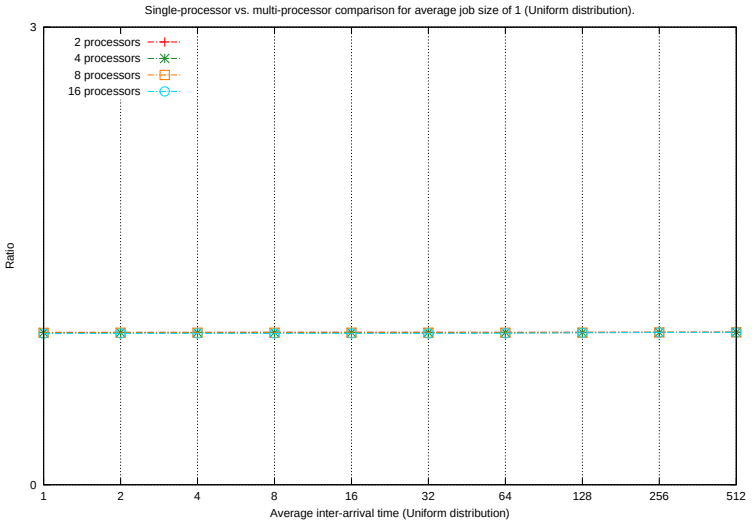


(k) *MinSize

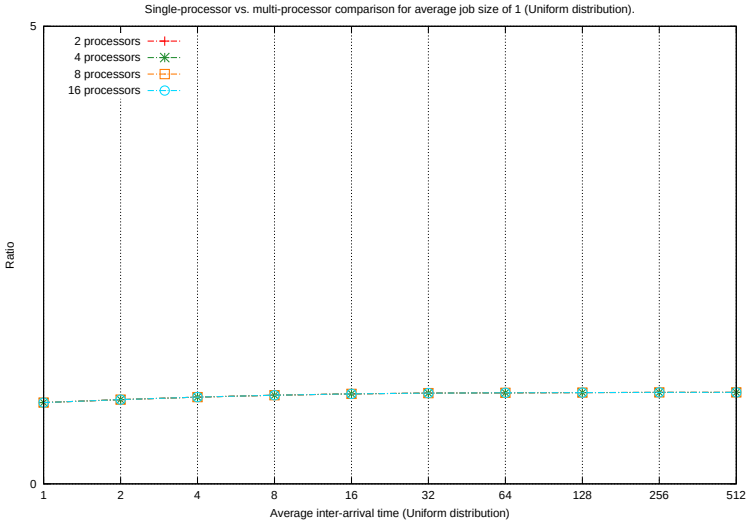


(l) RoundRobin

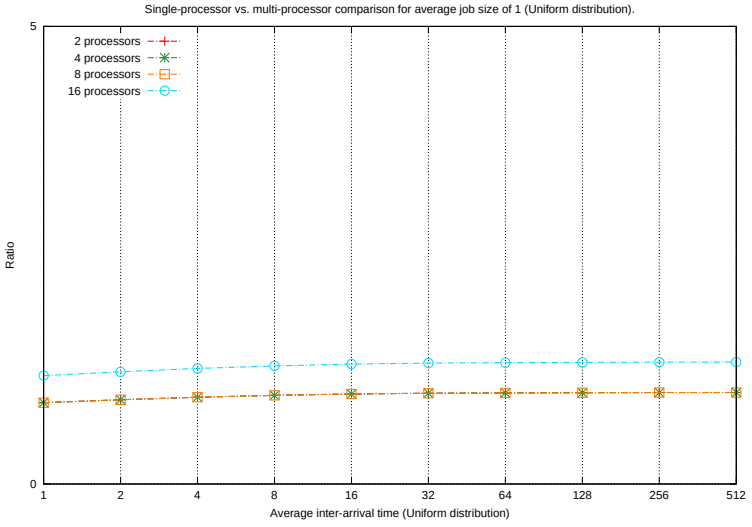
FIGURE A.-2: *Results on processor allocation strategies in terms of average job size*: Figures A.1(a) to A.1(d) show the performance ratios for average job size of 1. Results for average sizes of 16 and 512 are shown in Figures A.0(g), A.0(h), A.1(e) and A.1(f) and Figures A.-1(j) to A.-1(l) and A.0(i) respectively. Results measure the performance ratio of total flow time plus energy for a single processor vs. multiple processors. Poisson distribution is used for inter-arrival times and uniform distribution is used for job sizes.



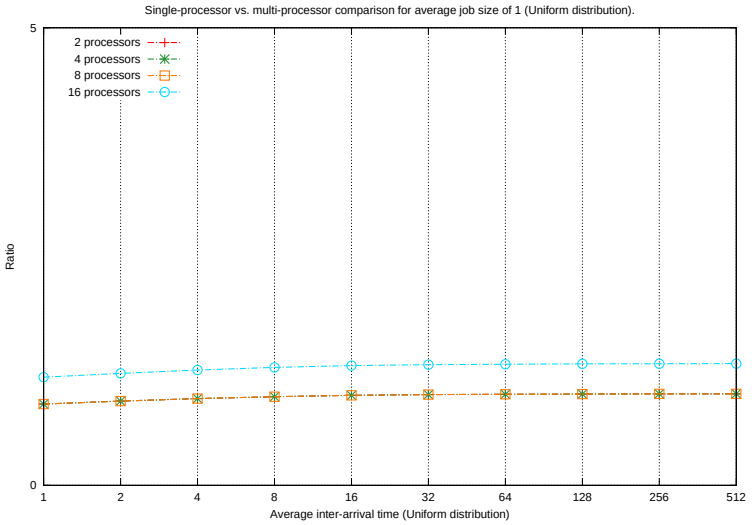
(a) *MinActiveCount



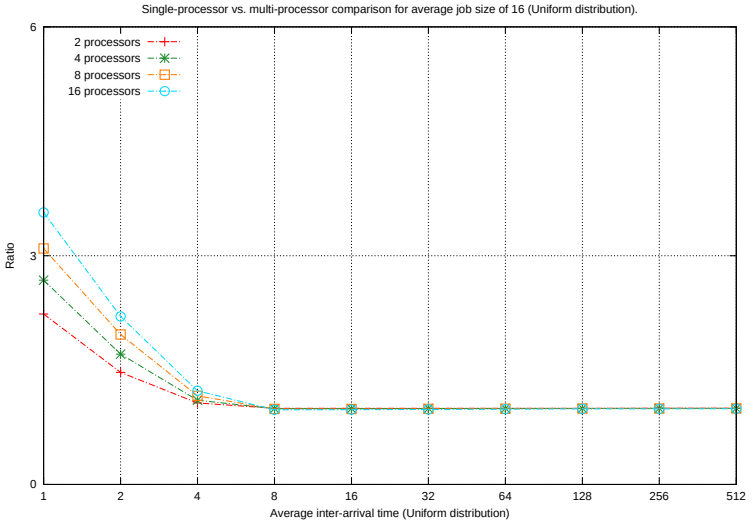
(b) *MinCost



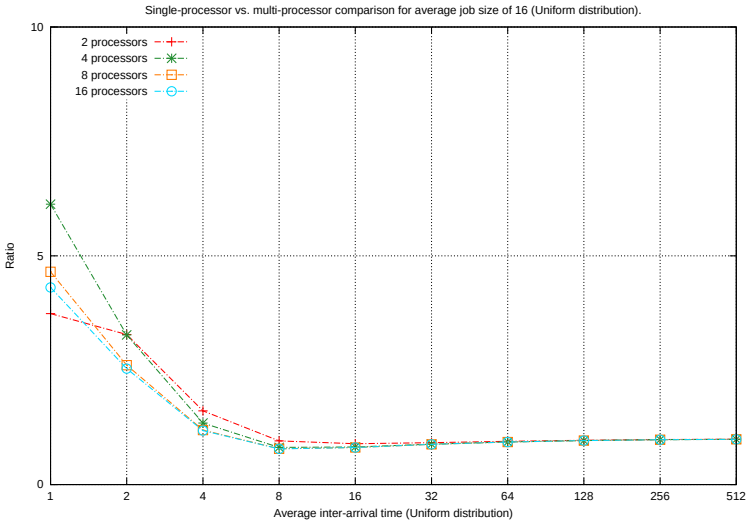
(c) *MinSize



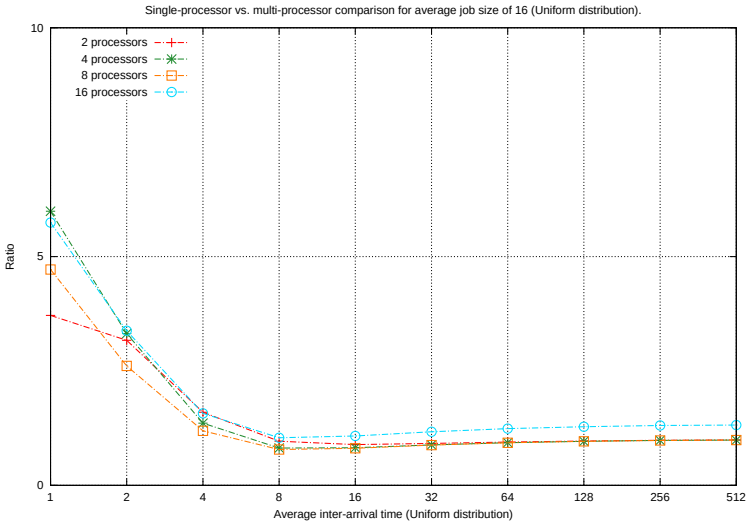
(d) RoundRobin



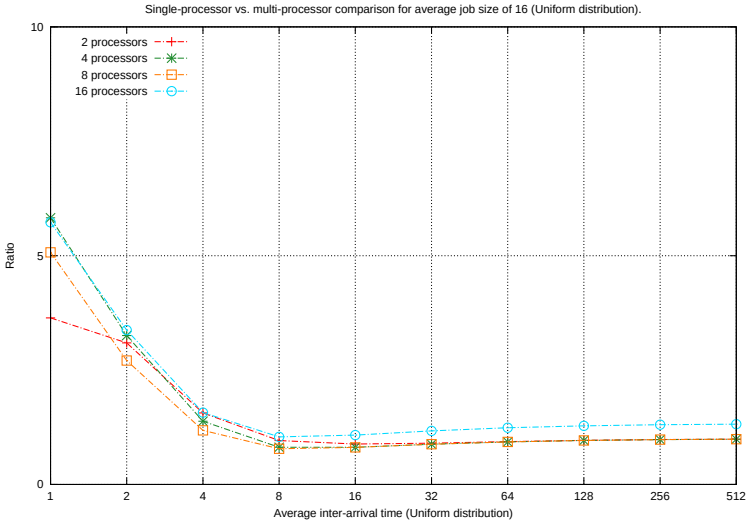
(e) *MinActiveCount



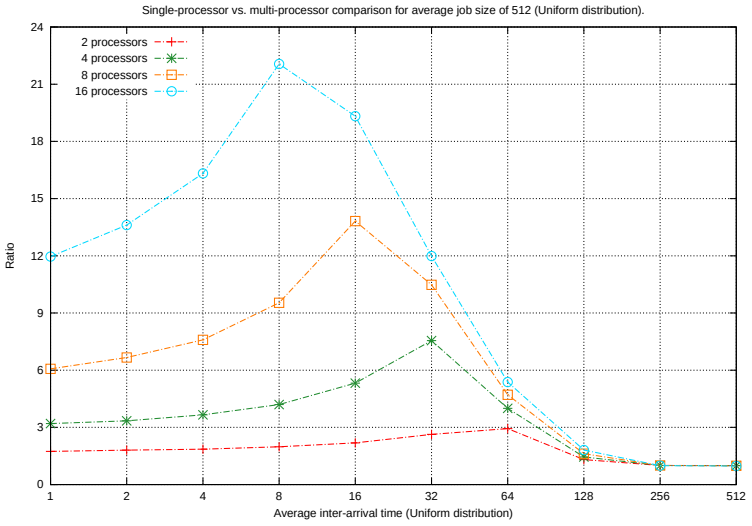
(f) *MinCost



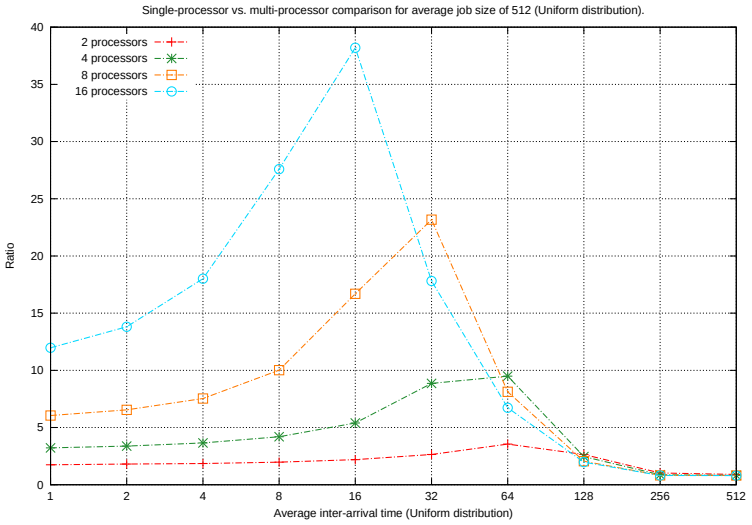
(g) *MinSize



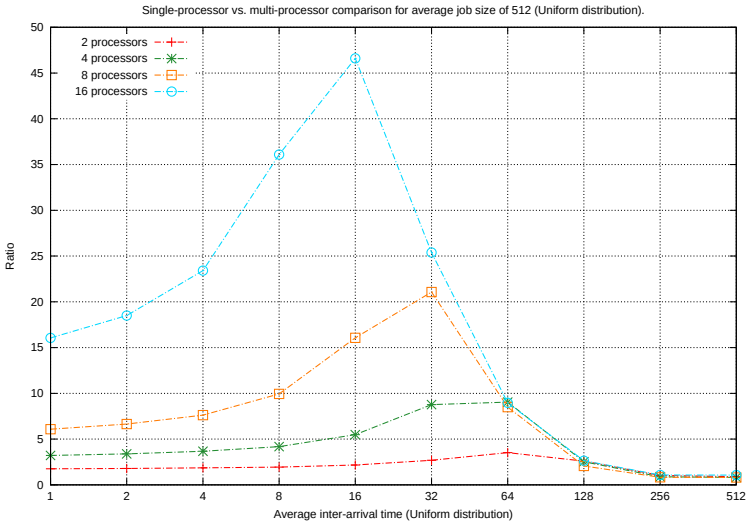
(h) RoundRobin



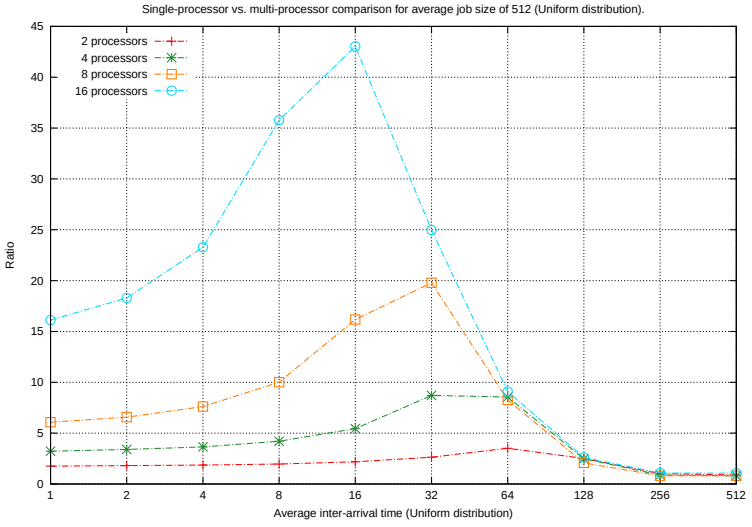
(i) *MinActiveCount



(j) *MinCost

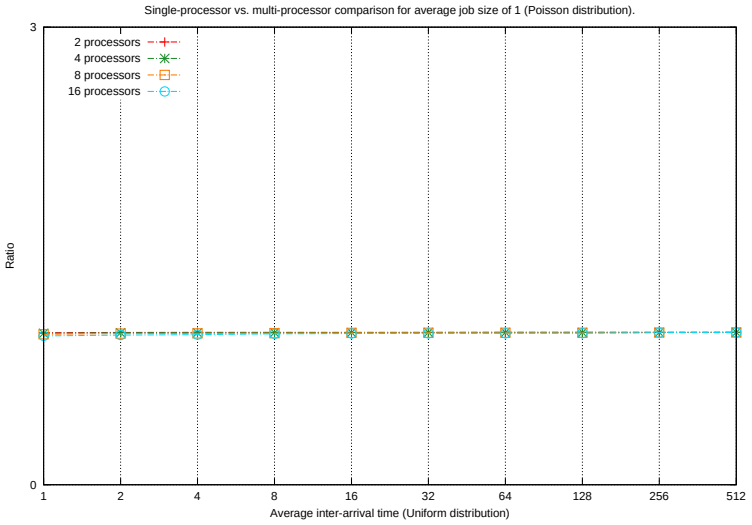


(k) *MinSize

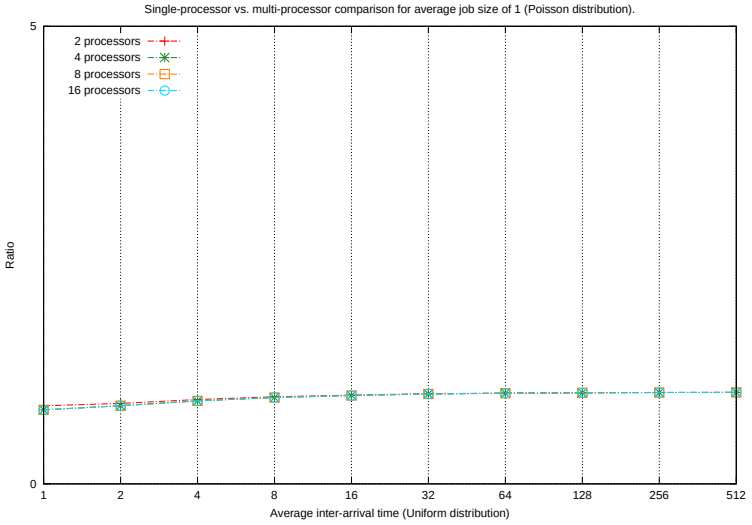


(l) RoundRobin

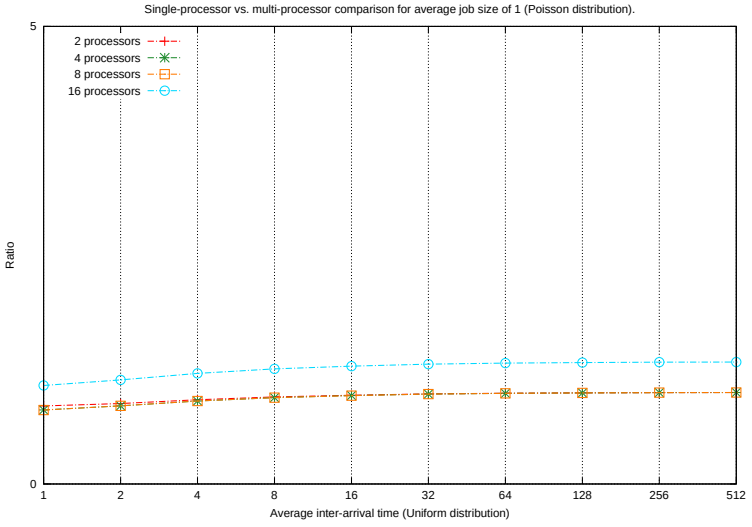
FIGURE A.-4: *Results on processor allocation strategies in terms of average job size*: Figures A.-1(a) to A.-1(d) show the performance ratios for average job size of 1. Results for average sizes of 16 and 512 are shown in Figures A.-2(g), A.-2(h), A.-1(e) and A.-1(f) and Figures A.-3(j) to A.-3(l) and A.-2(i) respectively. Results measure the performance ratio of total flow time plus energy for a single processor vs. multiple processors. Uniform distribution is used for both inter-arrival times and job sizes.



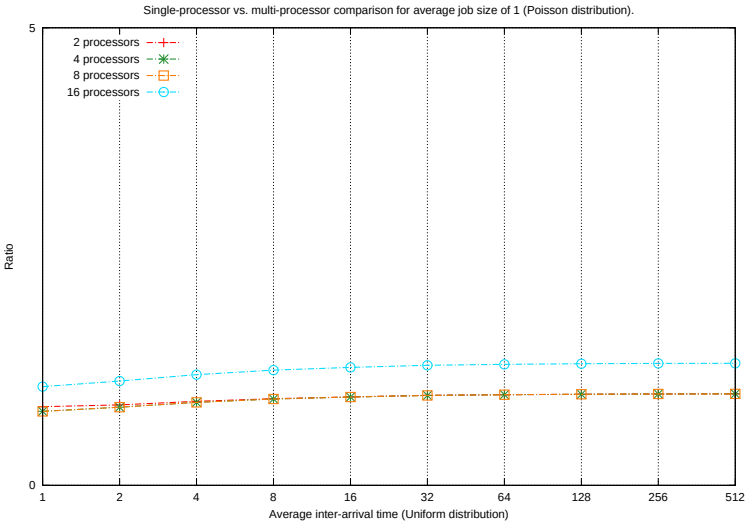
(a) *MinActiveCount



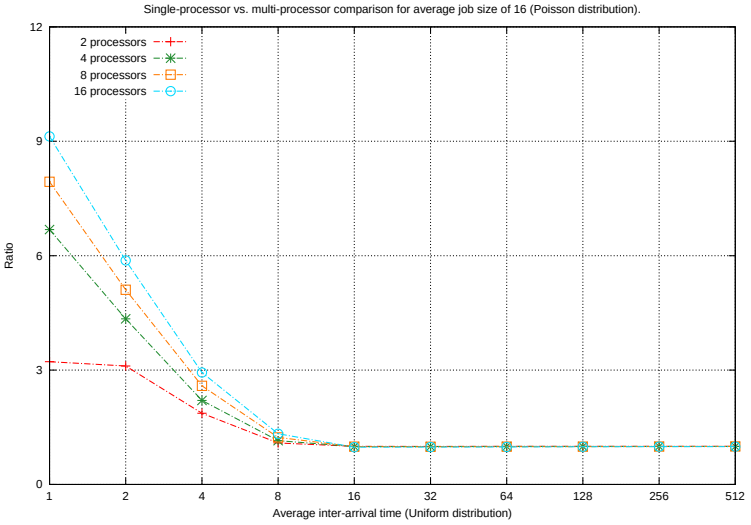
(b) *MinCost



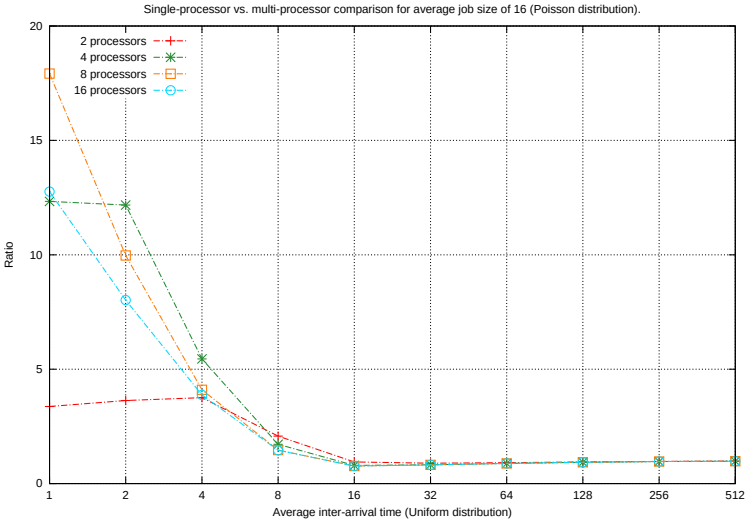
(c) *MinSize



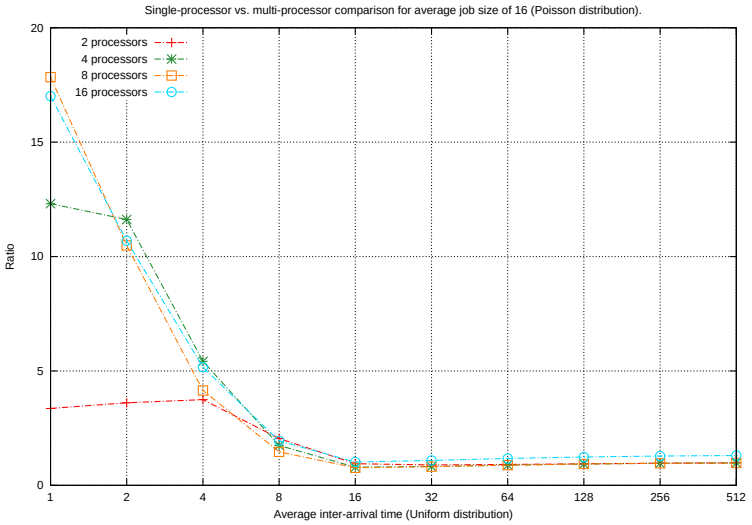
(d) RoundRobin



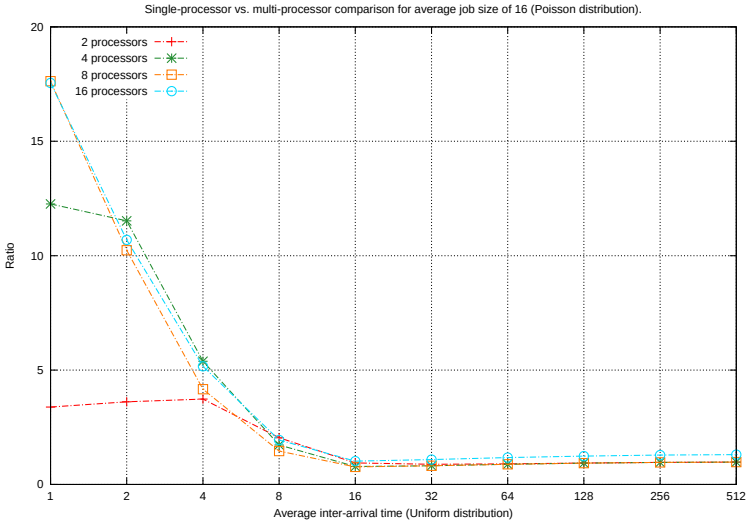
(e) *MinActiveCount



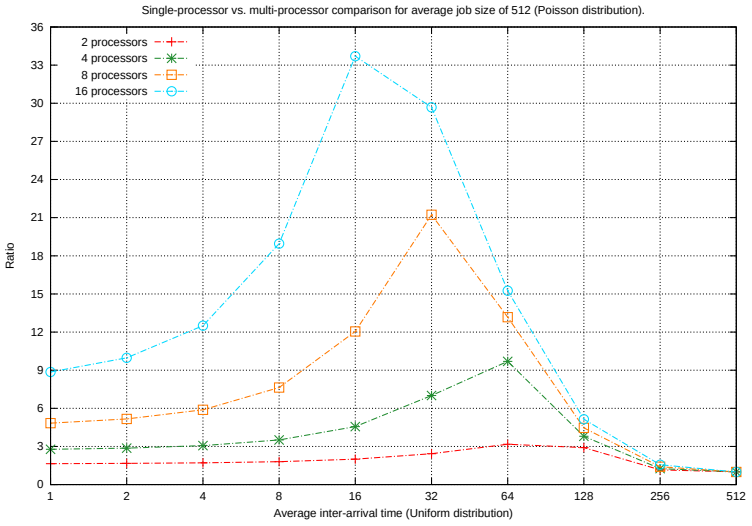
(f) *MinCost



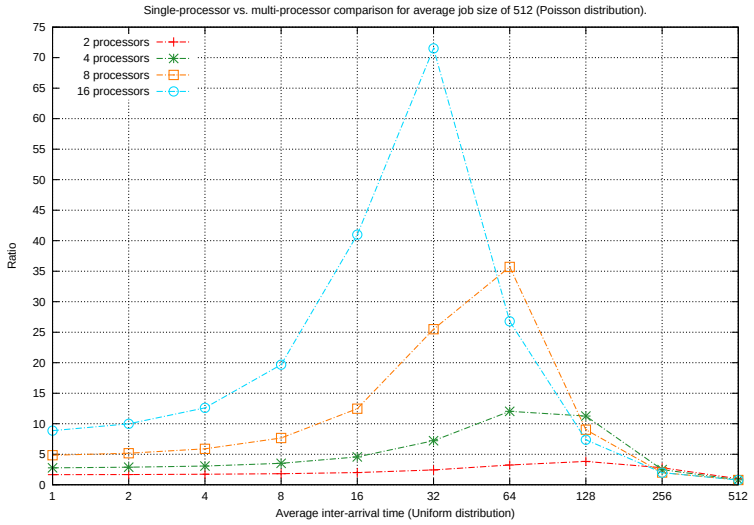
(g) *MinSize



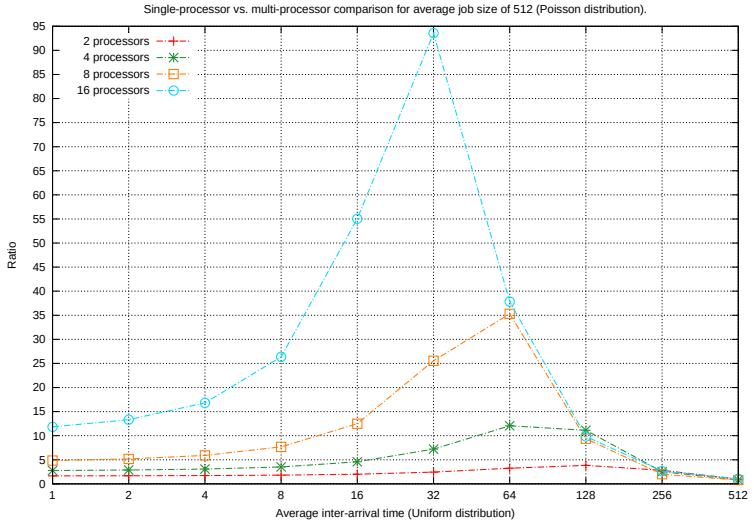
(h) RoundRobin



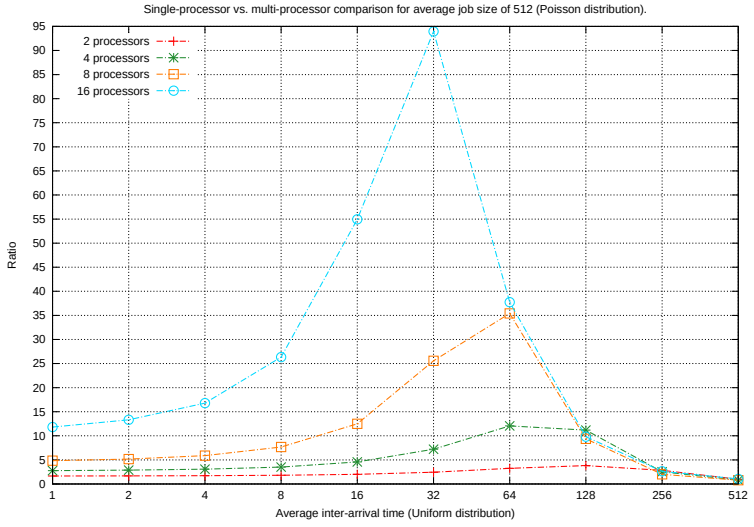
(i) *MinActiveCount



(j) *MinCost



(k) *MinSize



(l) RoundRobin

FIGURE A.-6: *Results on processor allocation strategies in terms of average job size*: Figures A.-3(a) to A.-3(d) show the performance ratios for average job size of 1. Results for average sizes of 16 and 512 are shown in Figures A.-4(g), A.-4(h), A.-3(e) and A.-3(f) and Figures A.-5(j) to A.-5(l) and A.-4(i) respectively. Results measure the performance ratio of total flow time plus energy for a single processor vs. multiple processors. Uniform distribution is used for inter-arrival times and Poisson distribution is used for job sizes.

Bibliography

[1] Advanced Micro Devices Inc. AMD Cool 'n' Quiet™ technology. http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx. Online; 2012.

[2] Advanced Micro Devices Inc. (2013). AMD Accelerated Parallel Processing OpenCL Programming Guide (chapter 2.1.3).

[3] Alachiotis, N., Berger, S., Flouri, T., Pissis, S. P., and Stamatakis, A. (2013). libgapmis: extending short-read alignments. *BMC bioinformatics*, 14(Suppl 11):S4.

[4] Albers, S. and Fujiwara, H. (2007). Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms (TALG)*, 3(4).

[5] Alidaee, B. and Womer, N. K. (1999). Scheduling with time dependent processing times: Review and extensions. *Journal of the Operational Research Society*, 50(7):711–720.

[6] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., et al. (2006). The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[7] Atkins, L. (2014). Algorithms for power savings. *PhD Thesis*, 3:32–57.

[8] Avrahami, N. and Azar, Y. (2003). Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of the fifteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–18. ACM.

[9] Awerbuch, B., Azar, Y., Leonardi, S., and Regev, O. (1999). Minimizing the flow time without migration. In *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*, pages 198–205. ACM.

[10] Bader, D. A. and Agarwal, V. (2007). Fftc: fastest fourier transform for the ibm cell broadband engine. In *High Performance Computing–HiPC 2007*, pages 172–184. Springer.

[11] Bansal, N., Chan, H., Lam, T., and Lee, L. (2008). Scheduling for speed bounded processors. *International Colloquium on Automata, Languages and Programming*, pages 409–420.

[12] Bansal, N., Pruhs, K., and Stein, C. (2007). Speed scaling for weighted flow time. In *18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 805–813.

[13] Barton, C., Flouri, T., Iliopoulos, C. S., and Pissis, S. P. (2013). Gapsmis: Flexible sequence alignment with a bounded number of gaps. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 402:402–402:411, New York, NY, USA. ACM.

[14] Baskiyar, S. and Abdel-Kader, R. (2010). Energy aware dag scheduling on heterogeneous systems. *Cluster Computing*, 13(4):373–383.

[15] Bender, M. A., Clifford, R., and Tsichlas, K. (2008). Scheduling algorithms for procrastinators. *Journal of Scheduling*, 11(2):95–104.

[16] Brooks, D., Bose, P., Schuster, S., Jacobson, H., Kudva, P., Buyuktosunoglu, A., Wellman, J., Zyuban, V., Gupta, M., and Cook, P. (2000). Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 20(6):26–44.

[17] Browne, S. and Yechiali, U. (1990). Scheduling deteriorating jobs on a single processor. *Operations Research*, 38(3):495–498.

[18] Bunde, D. (2006). Power-aware scheduling for makespan and flow. In *Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 190–196. ACM.

[19] Chekuri, C., Goel, A., Khanna, S., and Kumar, A. (2004). Multi-processor scheduling to minimize flow time with ε resource augmentation. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing*, pages 363–372. ACM.

[20] Chekuri, C., Khanna, S., and Zhu, A. (2001). Algorithms for minimizing weighted flow time. In *Proceedings of the thirty-third annual ACM Symposium on Theory of Computing*, pages 84–93. ACM.

[21] Chen, T., Raghavan, R., Dale, J. N., and Iwata, E. (2007). Cell broadband engine architecture and its first implementation—a performance view. *IBM Journal of Research and Development*, 51(5):559–572.

[22] Cheng, M. B. and Sun, S. J. (2007). A heuristic mbls algorithm for the two semionline parallel machine scheduling problems with deterioration jobs. *Journal of Shanghai University*, 11(5):451–456.

[23] Cheng, T. C. E. and Ding, Q. (1998). The complexity of single machine scheduling with release times. *Information Processing Letters*, 65(2):75–79.

[24] Cheng, T. C. E., Ding, Q., and Lin, B. M. T. (2004). A concise survey of scheduling with time-dependent processing times. *European Journal of Operational Research*, 152(1):1–13.

[25] Donald, E. K. (1999). The art of computer programming. *Sorting and searching*, 3:426–458.

[26] Farkas, K., Grunwald, D., Levis, P., Morrey III, C., and Neufeld, M. (2000). Policies for dynamic clock scheduling. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 6–6. USENIX Association.

[27] Feng, W.-c., Lin, H., Scogland, T., and Zhang, J. (2012). Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 291–294. ACM.

[28] FinalWire. AIDA64 Extreme. http://www.aida64.com/product/aida64-extreme/overview. Online; Accessed 29-April-2014.

[29] Flouri, T., Frousios, K., S Iliopoulos, C., Park, K., P Pissis, S., and Tischler, G. (2013). Gapmis: a tool for pairwise sequence alignment with a single gap. *Recent patents on DNA & gene sequences*, 7(2):84–95.

[30] Foundation, P. S. Python wrapper for OpenCL. https://pypi.python.org/pypi/pyopencl. Online; Accessed 29-April-2014.

[31] Frishman, Y. and Tal, A. (2007). Multi-level graph layout on the gpu. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1310–1319.

[32] Fruchterman, T. M. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164.

[33] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco.

[34] Gawiejnowicz, S. (2008). *Time-Dependent Scheduling*. Springer-Verlag, Berlin.

[35] Graham, R., Lawler, E., Lenstra, J., and Kan, A. R. (1977). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Proceedings Discrete Optimization*.

[36] Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

[37] Greb, A. and Zachmann, G. (2006). Gpu-abisort: Optimal parallel sorting on stream architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE.

[38] Green500. The Green500 List - June 2014. *http://www.green500.org/news/green500-list-june-2014*. [Online; accessed 17-Sept-2014].

[39] Gregg, C. and Hazelwood, K. (2011). Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE.

[40] Group, K. O. W. (2014). The opencl specification. (version: 2.0. document revision: 26). *https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf*. [Online; accessed 15-December-2014].

[41] Gupta, J. N. D. and Gupta, S. K. (1988). Single facility scheduling with nonlinear processing times. *Computers and Industrial Engineering*, 14(4):387–393.

[42] Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer.

[43] Ho, K. I., Leung, J. Y., and Wei, W. (1993). Complexity of scheduling tasks with time-dependent execution times. *Information Processing Letters*, 48(6):315–320.

[44] igraph core team. igraph c library by the igraph core team. *http://igraph.org/c/*. [Online; accessed 15-December-2015].

[45] Intel Corporation. Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor. http://download.intel.com/design/network/papers/30117401.pdf. Online white paper; Accessed 27-May-2014.

[46] Intel Corporation. Intel Hyper-Threading Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html. Online; Accessed 21-May-2014.

[47] Ishihara, T. and Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 197–202. IEEE.

[48] JogAmp. Java Binding for the OpenCL API. http://jogamp.org/jocl/www/. Online; Accessed 29-April-2014.

[49] Kang, L. Y. and Ng, C. T. (2007). A note on a fully polynomial-time approximation scheme for parallel-machine scheduling with deteriorating jobs. *International Journal of Production Economics*, 109(1):180–184.

[50] Karp, R. M. and Ramachandran, V. (1989). A survey of parallel algorithms for shared-memory machines.

[51] Kononov, A. (1997). Scheduling problems with linear increasing processing times. In Zimmermann U, e. a., editor, *Operations Research Proceedings 1996. Selected Papers of the Symposium on Operations Research (SOR 96)*, pages 208–212, Berlin. Springer.

[52] Kruskal, C. P., Rudolph, L., and Snir, M. (1990). A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95 – 132.

[53] Kunnathur, A. S. and Gupta, S. K. (1990). Minimizing the makespan with late start penalties added to processing times in a single facility scheduling problem. *European Journal of Operational Research*, 47(1):56–64.

[54] Kurzak, J., Buttari, A., and Dongarra, J. (2008). Solving systems of linear equations on the cell processor using cholesky factorization. *Parallel and Distributed Systems, IEEE Transactions on*, 19(9):1175–1186.

[55] Kurzak, J. and Dongarra, J. (2009). Qr factorization for the cell broadband engine. *Scientific Programming*, 17(1):31–42.

[56] Kwon, W. and Kim, T. (2005). Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):211–230.

[57] Lam, T., Lee, L., To, I., and Wong, P. (2012). Improved multi-processor scheduling for flow time and energy. *Journal of Scheduling (JoS)*, 15(1):105–116.

[58] Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management science*, 19(5):544–546.

[59] Lebak, J., Reuther, A., and Wong, E. (2005). Polymorphous computing architecture (pca) kernel-level benchmarks. Technical report, DTIC Document.

[60] Lee, W., Wu, C., and Chung, Y. (2008). Scheduling deteriorating jobs on a single machine with release times. *Computers and Industrial Engineering*, 54(3):441–452.

[61] Lenstra, J. K. and Rinnooy Kan, A. (1978). Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35.

[62] Leonardi, S. and Raz, D. (1997). Approximating total flow time on parallel machines. In *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 110–119. ACM.

[63] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[64] Leung, J. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*, volume 1. CRC Press.

[65] Li, M. and Yao, F. (2005). An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal on Computing*, 35:658–671.

[66] Liu, C., Wong, T., Wu, E., Luo, R., Yiu, S., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., et al. (2012). SOAP3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879.

[67] Liu, Y., Wirawan, A., and Schmidt, B. (2013). Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):117.

[68] Ma, Y., Chu, C., and Zuo, C. (2010). A survey of scheduling with deterministic machine availability constraints. *Computers & Industrial Engineering*, 58(2):199–211.

[69] Maeurer, T. and Shippy, D. (2005). Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604.

[70] Marsaglia, G. (1972). Choosing a point from the surface of a sphere. *Ann. Math. Statist.*, 43(2):645–646.

[71] McCullough, J. and Torng, E. (2004). SRPT optimally utilizes faster machines to minimize flow time. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 350–358. Society for Industrial and Applied Mathematics.

[72] Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM.

[73] Mosheiov, G. (1991). V-shaped policies for scheduling deteriorating jobs. *Operations Research*, 39(6):979–991.

[74] Mosheiov, G. (1994). Scheduling jobs under simple linear deterioration. *Computers & operations research*, 21(6):653–659.

[75] Mosheiov, G. (1998). Multi-machine scheduling with linear deterioration. *INFOR: Information Systems and Operational Research*, 36(4):205–214.

[76] Mudge, T. (2001). Power: A first-class architectural design constraint. *Computer*, 34(4):52–58.

[77] National Center for Biotechnology Information (NCBI) (2014a). ftp://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62.

[78] National Center for Biotechnology Information (NCBI) (2014b). GenBank FTP. ftp://ftp.ncbi.nih.gov/ncbi-asn1.

[79] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453.

[80] Ng, C., Li, S., Cheng, T. E., and Yuan, J. (2010). Preemptive scheduling with simple linear deterioration on a single machine. *Theoretical Computer Science*, 411(40):3578–3586.

[81] NVIDIA. Unified memory in cuda 6. *http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/*. [Online; accessed 15-December-2014].

[82] Nvidia Corporation (2013). CUDA C Programming Guide.

[83] Ojiaku, J., Thomas, D., and Wong, P. (2013). Energy-efficient flow time scheduling : An experimental study. In *11th Workshop on Models and Algorithms for Planning and Scheduling Problems*.

[84] OpenMP®. The OpenMP API specification for parallel programming. http://openmp.org/wp/. Online; Accessed 31-Jan-2014.

[85] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879–899.

[86] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library.

[87] Phillips, C., Stein, C., Torng, E., and Wein, J. (1997). Optimal time-critical scheduling via resource augmentation. In *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, page 149. ACM.

[88] Pillai, P. and Shin, K. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 89–102. ACM.

[89] Project, C. Cloo. http://cloo.sourceforge.net/. Online; Accessed 29-April-2014.

[90] Pruhs, K. (2007). Competitive online scheduling for server systems. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):52–58.

[91] Pruhs, K., Sgall, J., and Torng, E. (2004). Online scheduling. In Leung, J., editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 15.1–15.42. Chapman and Hall, Boca Raton.

[92] Ren, C. R. and Kang, L. Y. (2007). An approximation algorithm for parallel machine scheduling with simple linear deterioration. *Journal of Shanghai University*, 11(4):151–154.

[93] Rice, P., Longden, I., and Bleasby, A. (2000). Emboss: the european molecular biology open software suite. *Trends in genetics*, 16(6):276–277.

[94] Satish, N., Harris, M., and Garland, M. (2009). Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE.

[95] Schmidt, G. (2000). Scheduling with limited machine availability. *European Journal of Operational Research*, 121(1):1–15.

[96] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. (2007). Scan primitives for gpu computing. In *Graphics Hardware*, volume 2007, pages 97–106.

[97] Sharifi, M., Shahrivari, S., and Salimi, H. (2013). Pasta: a power-aware solution to scheduling of precedence-constrained tasks on heterogeneous computing resources. *Computing*, 95(1):67–88.

[98] Shi, H., Schmidt, B., Liu, W., and Muller-Wittig, W. (2009). Accelerating error correction in high-throughput short-read dna sequencing data with cuda. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE.

[99] Sidney, J. B. (1975). Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23(2):283–298.

[100] Sintorn, E. and Assarsson, U. (2008). Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388.

[101] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

[102] Smith, R. AMD's Graphics Core Next Preview: AMD's New GPU, Architected For Compute. http://www.anandtech.com/show/4455/amds-graphics-core-next-preview-amd-architects-for-compute. Online; Accessed 22-May-2014.

[103] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm.

[104] Swope, W. C., Andersen, H. C., Berens, P. H., and Wilson, K. R. (1982). A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649.

[105] The Khronos Group. Khronos OpenCL Registry. https://www.khronos.org/registry/cl/. Online; Accessed 29-April-2014.

[106] Top500. Top 500 List - June 2014. *http://www.top500.org/list/2014/06/*. [Online; accessed 17-Sept-2014].

[107] Vineet, V. and Narayanan, P. (2008). Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8. IEEE.

[108] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. (2006). The potential of the cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM.

[109] Woligroski, D. AMD Radeon HD 7970: Promising Performance, Paper-Launched. http://www.tomshardware.com/reviews/radeon-hd-7970-benchmark-tahiti-gcn,3104.html. Online; Accessed 22-May-2014.

[110] Xbit Laboratories. Intel Pentium 4 3.06GHz CPU with Hyper-Threading Technology: Killing Two Birds with a Stone... http://www.xbitlabs.com/articles/cpu/display/pentium4-3066.html. Online; Accessed 21-May-2014.

[111] Yao, F., Demers, A., and Shenker, S. (1995). A scheduling model for reduced cpu energy. In *36th IEEE Symposium on Foundations of Computer Science*, pages 374–382.

[112] Yu, S., Ojiaku, J., Wong, P., and Xu, Y. (2012). Online makespan scheduling of linear deteriorating jobs on parallel machines. In *Theory and Applications of Models of Computation*, pages 260–272. Springer Berlin Heidelberg.

[113] Yung, L. S., Yang, C., Wan, X., and Yu, W. (2011). Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies. *Bioinformatics*, 27(9):1309–1310.

[114] Zhou, Y., Liepe, J., Sheng, X., Stumpf, M. P., and Barnes, C. (2011). Gpu accelerated biochemical network simulation. *Bioinformatics*, 27(6):874–876.