Reaño González, C.; Silla Jiménez, F.; Nikolopoulos, DS.; Varghese, B. (2018). Intra-node Memory Safe GPU Co-Scheduling. IEEE Transactions on Parallel and Distributed Systems. 29(5):1089-1102. https://doi.org/10.1109/TPDS.2017.2784428

Intra-node Memory Safe GPU Co-Scheduling

Carlos Reaño, Federico Silla, Dimitrios S. Nikolopoulos and Blesson Varghese

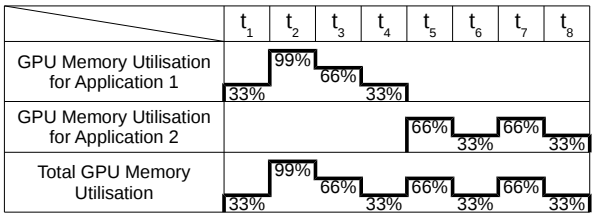
Abstract—GPUs in High-Performance Computing systems remain under-utilised due to the unavailability of schedulers that can safely schedule multiple applications to share the same GPU. The research reported in this paper is motivated to improve the utilisation of GPUs by proposing a framework, we refer to as schedGPU, to facilitate intra-node GPU co-scheduling such that a GPU can be safely shared among multiple applications by taking memory constraints into account. Two approaches, namely a client-server and a shared memory approach are explored. However, the shared memory approach is more suitable due to lower overheads when compared to the former approach. Four policies are proposed in schedGPU to handle applications that are waiting to access the GPU, two of which account for priorities. The feasibility of schedGPU is validated on three real-world applications. The key observation is that a performance gain is achieved. For single applications, a gain of over 10 times, as measured by GPU utilisation and GPU memory utilisation, is obtained. For workloads comprising multiple applications, a speed-up of up to 5x in the total execution time is noted. Moreover, the average GPU utilisation and average GPU memory utilisation is increased by 5 and 12 times, respectively.

Keywords—GPU Co-scheduling, access synchronisation, memory safe, accelerator, under-utilisation, schedGPU

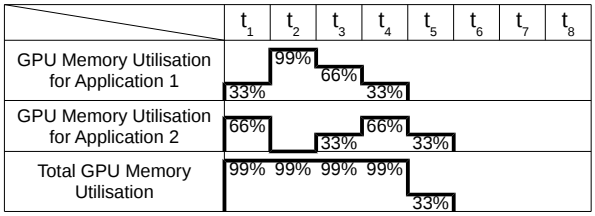
1 Introduction

High-Performance Computing (HPC) systems are becoming heterogeneous in pursuit of exascale computing [1], [2].These systems not only offer CPUs, but also provide accelerators, such as Graphics Processing Units (GPUs). Heterogeneity can be leveraged for improving performance by decomposing compute-intensive components of an application and offloading them on to GPUs. Existing schedulers, such as Slurm [3] and Torque [4], cannot safely schedule multiple applications for sharing the same GPU [5], thereby exclusively locking a GPU for a single application. This results in the under-utilisation of GPUs and will have negative implications on the performance of future exascale systems [6], [7]. Hence, the research in this paper aims to improve the utilisation of GPUs.

One way of addressing accelerator under-utilisation, given that GPUs are usually coupled to a CPU node, is by sharing GPUs among multiple applications that execute on different CPU cores of the same node. However, there is a risk of running out of GPU memory which can cause applications to unexpectedly end. Current techniques that incorporate scheduling [8], [9], [10], kernel-based [11], [12], [13], synchronisation [14] and architectural [15], [16] approaches cannot safely share GPUs among applications while eliminating the above risk. Therefore, applications are currently executed sequentially, although they may use the GPU for a relatively small fraction of the entire execution time, as shown in Figure 1a. This raises the need for a scheduler that can account for GPU memory required by applications to safely share GPUs as indicated in Figure 1b.



(a) Using existing workload schedulers



(b) Using proposed approach

Fig. 1: Execution of applications on a two CPU node with one GPU. In Figure 1a, two applications need access to the GPU, but are executed sequentially using existing workload schedulers. Figure 1b shows the proposed approach that co-schedules applications on the same GPU and accounts for GPU memory to maximise utilisation.

In this paper, we propose an intra-node, memory safe GPU co-scheduling framework, referred to as *schedGPU*. The framework safely handles multiple application requests to access GPUs by ensuring that memory overruns do not occur during execution. Two implementation approaches, namely a client-server and a shared memory approach, are considered. The access of applications to shared memory is synchronised by developing a custom protocol that employs file locks and system signals. This protocol avoids abandonment, the problem that arises when the framework employs other interprocess synchronisation mechanisms, such as mutexes. Four policies

• *C. Reaño and F. Silla are with the Universitat Politècnica de València, Spain. D. Nikolopoulos and B. Varghese are with the Queen's University Belfast, UK. Email:* carregon@gap.upv.es, fsilla@upv.es, {d.nikolopoulos, varghese}@qub.ac.uk

are proposed and investigated in schedGPU to handle applications that wait to be scheduled on the GPU.

The feasibility of schedGPU is validated first using popular GPU benchmark suites and then on three real-world applications that have varying GPU utilisation. Using schedGPU performance gain in terms of average speed-up, average GPU utilisation and average GPU memory utilisation when executing concurrent instances of an application using schedGPU is noted to be up to 10 times over conventional execution. For workloads comprising multiple applications, using Slurm along with schedGPU results in a 5x speed-up. Moreover, the average GPU utilisation and average GPU memory utilisation is increased by 5 and 12 times, respectively, when compared to not using schedGPU.

The research contribution of this paper is an approach for intra-node scheduling at runtime. Conventional schedulers schedule applications ahead-of-time typically over multiple nodes. However, they do not optimise scheduling on each node. The merit of our approach is that scheduling is performed on the fine-grain level at runtime, therefore allowing any application to be executed without knowledge of its GPU requirements prior to execution. Existing schedulers exclusively lock a given GPU for an application. Our novel approach on the other hand is memory safe and shares GPUs to co-schedule multiple applications. For this our approach monitors the GPU resources to service applications.

The remainder of this paper is organised as follows. Section 2 presents the key concepts of our GPU co-scheduling framework. Section 3 provides the implementation approaches considered in our framework. Section 4 describes the typical life cycle of an application using our framework. Section 5 details a set of four policies incorporated in the framework for scheduling applications. Section 6 identifies suitable real-world use-cases for schedGPU. Section 7 evaluates the performance of the framework using two popular GPU benchmark suites and then highlights its benefit for three real-world applications. Section 8 considers the related research. Section 9 concludes this paper and presents future work.

2 GPU Scheduling Framework

Consider a typical server, which comprises multiple CPU cores and one GPU. There are two challenges in executing multiple applications on the same GPU. Firstly, consider a scenario in which a conventional workload scheduler, such as Slurm, is employed to schedule applications from multiple users, then the scheduler will handle multiple requests to the GPU by simply executing the jobs sequentially. While these schedulers can schedule applications on multiple servers they schedule them ahead-of-time, leaving no room for adapting to runtime requirements. Therefore, the jobs are executed sequentially on each server since the scheduler cannot ensure whether the GPU memory requirements of requesting application can be met at any time (for example, whether there is sufficient GPU memory for a second application on the GPU).

Secondly, assume a workload scheduler can schedule multiple applications on the same GPU. While this can improve GPU utilisation, it could lead to potentially terminating the job (for example, if there is insufficient GPU memory an out of memory error will be returned). This is because there is no safe handling of GPU memory requirements for co-scheduling jobs.

In this paper, we address the above challenges by presenting a framework for intra-node GPU scheduling, referred to as schedGPU[1], that facilitates the simultaneous execution of multiple applications on a GPU. Using schedGPU multiple applications can request GPU memory during execution time. schedGPU safely co-schedules the applications by taking memory requirements into account and thereby avoids potential memory allocation errors due to unavailable memory on the GPU. The schedGPU framework is proposed and developed for CUDA-based [17] GPU applications. CUDA is widely used in production and commercial environments when compared to the OpenCL alternative [18].

The features of the GPU scheduling framework are:

(i) *Intra-node scheduling*: most schedulers schedule applications over multiple nodes at the coarse-grain level. However, schedGPU schedules at the fine-grain level to improve GPU utilisation of each node.

(ii) *Scheduling at runtime*: unlike conventional schedulers that schedule applications ahead-of-time, our framework can schedule applications on to the GPU in sub-millisecond timescales during execution.

(iii) *Memory-based safe co-scheduling*: typically schedulers allow for executing an application on multiple GPUs. Our approach facilitates the execution of multiple applications on the same GPU concurrently to improve GPU utilisation. We consider memory requirements of each application and ensure that no application unexpectedly ends due to insufficient GPU resources.

(iv) *Scalability*: the control of most workload schedulers is centralised. On the contrary, the control of schedGPU is distributed on each node hence avoiding single points of failure and use on a large number of nodes.

An application that needs to safely use a GPU through our proposed framework follows a four stage life cycle. The first stage is initialising an instance of schedGPU for the application to allow interaction between the application and the framework. The second stage is reserving GPU memory required by an application, we refer to as pre-allocation. The GPU memory requests made by the application are appropriately handled by the framework. The third stage is releasing reserved GPU memory after the application makes use of the GPU, we refer to as post-free. Applications still waiting for the GPU are potentially serviced. The fourth stage is shutting down the instance of schedGPU that was initialised.

1. The schedGPU framework can be requested for download from http://mural.uv.es/caregon/schedgpu.html

3 Implementation Approaches

The schedGPU framework incorporates two approaches, namely a client-server and a shared memory model[2]. The latter is the focus of this paper. A prototype of the client-server approach was briefly reported elsewhere [19].

The functionality of both the client-server and shared memory implementations is the same, which is to avoid applications running out of GPU memory. However, there are important differences architecturally. On one hand, the first approach follows a client-server architecture and the GPU information is centralised on the server side. On the other hand, the second approach performs a distributed management of the GPU information using shared memory.

3.1 Client-Server

In this approach, each CUDA application is a client that requests GPU memory to a centralised server daemon, both of which are executed on the same node. The server permits the application to continue execution if there is sufficient memory on the GPU, otherwise the client may choose to be either blocked until memory is available or informed using CUDA error codes.

Figure 2 presents the architecture of the client-server model. In this model, the CUDA application is minimally modified by explicitly calling functions from the client library to pre-allocate the GPU memory required by the application. The calls are forwarded to the server using a UNIX domain socket. We chose UNIX domain sockets over TCP Loopback sockets due to the superior performance of the former [20]. The server creates a new thread for each client. A global view of the memory used by all clients is maintained by the server through the NVIDIA Management Library (NVML)[3]. We chose to use NVML instead of the CUDA library to avoid the creation of an additional GPU context that consumes GPU memory. In addition, using NVML the physical devices are accessed instead of using logical devices to avoid any ambiguities in the framework (for example, applications using different identifiers for logical devices referring to the same physical device).

3.2 Shared Memory

One disadvantage of a centralised server is that it may unexpectedly end resulting in the failure of the framework. Therefore, an alternate distributed approach was considered using shared memory in which the clients are responsible for maintaining the global state of the GPU memory in use. The client library makes decisions

2. In this paper, 'shared memory' does not refer to aggregating host and device memory for offering a unified address space. This is because schedGPU schedules the access of applications to the GPU from the host side. Instead, we refer to shared memory as the host memory which is accessible for different applications using schedGPU. We refer to 'shared memory data structure' as the format of the data and its contents that is stored in the shared memory.

3. https://developer.nvidia.com/nvidia-management-library-nvml

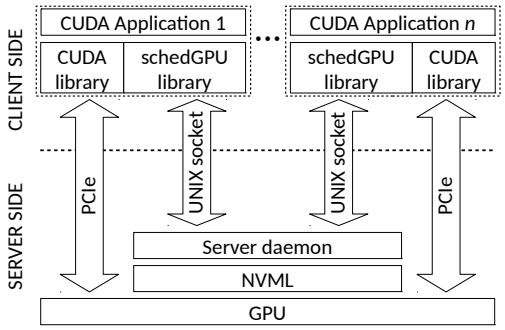


Fig. 2: Architecture of the client-server model

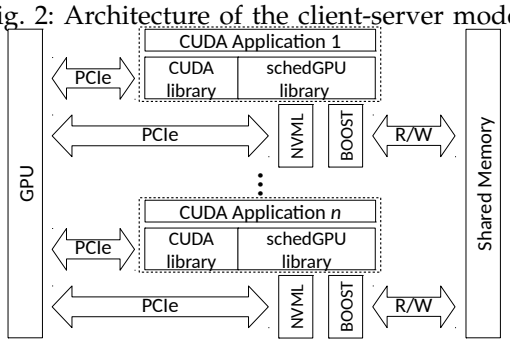


Fig. 3: Architecture of the shared memory model

based on the information available in the shared memory data structure. Figure 3 shows the architecture of the shared memory model. The client library directly makes use of NVML. The shared memory structure is created and managed using the Boost Interprocess library[4].

The shared memory approach not only overcomes the disadvantages of the client-server approach, but also achieves better performance that will be considered in Section 7. Therefore, this paper will focus on schedGPU using the shared memory approach. Four features are incorporated in the shared memory approach to enhance the robustness of the model. Firstly checkpointing by storing a backup of the shared memory structure by each client library when the application ends.

Secondly, an integrity check and recovery. When a new client starts, the library checks whether the shared memory structure is corrupt. If corrupt, then it is recovered from a backup. If there are no backups or if corrupt, then the shared memory structure is freshly initialised.

Thirdly, a sanity check. When any client has ended or is blocked and waiting for free memory, the client library checks that processes with allocated memory are still alive (this is done to free the memory of clients that unexpectedly terminate). If not, the previously allocated memory is freed for the waiting clients.

Fourthly, mitigating abandonment. If a client application unexpectedly ends, the access to the shared memory structure is not blocked and the framework could be used by other clients transparently. If the client had memory allocated, it will be freed.

Shared Memory Data: The data structure used in the shared memory approach comprises the following data:

4. http://www.boost.org/doc/libs/1_59_0_b1/doc/html/interprocess.html

- Total GPU memory: the total installed or physical device memory accessible to the schedGPU framework.
- Total Used memory: the total memory utilised by active schedGPU client applications.
- Itemised Used memory: memory utilised by each client application that is uniquely identified by schedGPU.
- Queue of client applications waiting to access GPU memory: a queue of applications that requested more GPU memory than what was available. The priority of an application is also included depending on the policy in use. Policies will be described in Section 5.

Synchronising Access to Shared Memory: To avoid inconsistencies we synchronise the access of multiple clients to the shared memory data structure using two methods.

The first method is based on using interprocess mutexes and conditions both provided by the same Boost Interprocess Library that manages the shared memory data structure. Access of each client application to the shared memory data is controlled by a mutex which ensures that only one client modifies the shared memory structure at a given time. A condition is associated with the mutex for either notifying clients that memory has been freed or waiting for notifications on freed memory.

Although this is the most common method, it is restricted due to *abandonment*: if a process owning the mutex unexpectedly ends, then the mutex becomes unusable and other processes endlessly wait for it. This can be avoided by the use of lock-free methods such as robust mutexes. Such methods are currently available for intraprocess communication (multi-threads). In the case of schedGPU, multiple applications will need to communicate with schedGPU simultaneously and this additionally requires interprocess communication along with intraprocess communication. However, there are no standard solutions for lock-free synchronisation in interprocess communication[5]. One available solution requires patching the operating system kernel[6].

In order to surmount the challenge of abandonment when using interprocess communication in schedGPU, we developed a second method that employs file locks instead of interprocess mutexes for controlling the access of client applications to the shared memory data structure. If a client owning a file lock unexpectedly ends, then the file lock is still safely used by other clients.

However, conditions cannot be associated with file locks. So a custom protocol using system signals was developed for interprocess communication. The protocol issues a user system signal for notifying a waiting client that memory has been freed. The waiting client on receiving the signal continues execution on the GPU. This method is more appropriate than the first method and is therefore incorporated in the schedGPU framework.

5. http://www.boost.org/doc/libs/1_61_0/boost/interprocess/detail/robust_emulation.hpp

6. http://yurovsky.github.io/2015/06/04/lockfree-ipc/

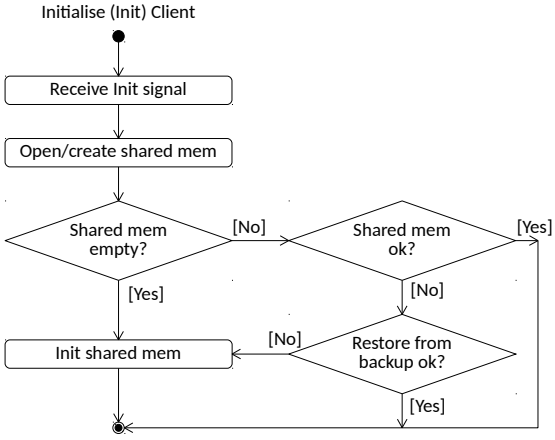


Fig. 4: Initialisation of a schedGPU client

4 The Life Cycle

Three functions are offered by schedGPU that implements the life cycle. They include the initialisation, memory pre-allocation and memory post-freeing functions. The shutdown stage in the framework is implicitly called when the CUDA application terminates execution. schedGPU provides two options for using the functions: (i) implicit memory management - users do not modify the source code, but memory is implicitly managed by the framework using a set of default parameters, and (ii) explicit memory management - users minimally modify the GPU source code by including the schedGPU functions. This provides the developer finer control on memory management.

Initialisation: Figure 4 shows how a schedGPU client is initialised in the shared memory approach using the `schedGPUInit()` function. First of all, the client is made ready to handle system signals that are used for internal notifications. The shared memory data structure is then accessed or created. If the shared memory structure is empty, then it is initialised by gathering information of the GPUs using NVML. If the shared memory structure already contains GPU information, then an integrity check is performed to ensure that the data is not corrupted (recovers from the shared memory backup if the integrity check fails).

Pre-allocation: As shown in Figure 5 for pre-allocating memory using the `preCudaMalloc()` function, the client requests the ownership of the file lock. When the ownership of the lock is obtained, the client checks that the GPU requested is a valid device and the requested memory is available on the GPU. If memory is insufficient, then the client performs a sanity check on whether other clients with pre-allocated memory are still alive. If memory is freed from other clients, then the client rechecks if there is sufficient memory.

If the available memory is still insufficient and provided that the pre-allocation call is non-blocking, then control is returned to the application with an error code (`cudaErrorNotReady`). If the call is blocking, then the

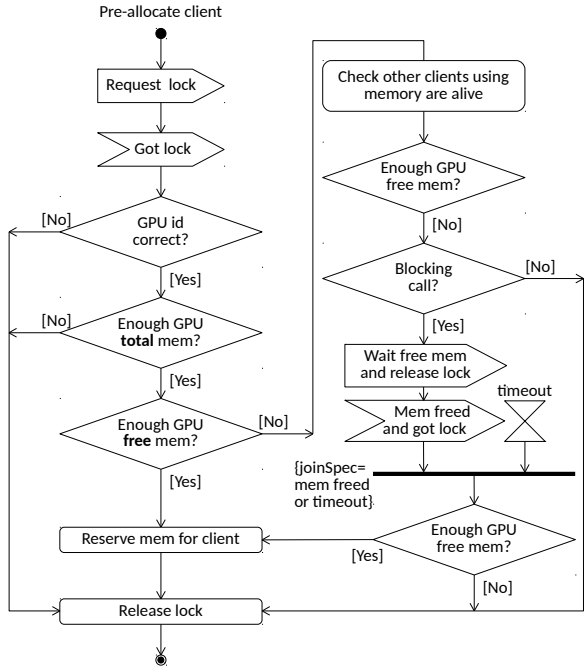


Fig. 5: Pre-allocation of memory by a schedGPU client

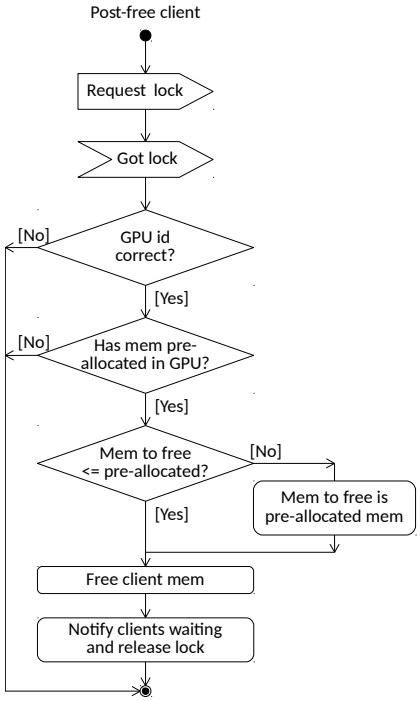


Fig. 6: Post-free of memory by a schedGPU client

client waits for a specified time period[7] defined by the application until free memory is available. In the event that the client does not pre-allocate all free GPU memory, then it does not notify other clients of free memory. This notification is carried out during post-freeing.

Post-free: As shown in Figure 6 for post-freeing memory using the `postCudaFree()` function, the client requests the ownership of the file lock. When the ownership of the lock is obtained the client checks that (i) the GPU requested is a valid device and (ii) the requested memory for freeing is already pre-allocated on the GPU. If memory is freed, then the clients that may be waiting for memory are notified (refer to Section 5).

Shutdown: As shown in Figure 7 the client requests the ownership of the file lock. When the ownership is obtained the client (i) ensures that it has post-freed all pre-allocated memory, and (ii) performs a sanity check whether other clients with pre-allocated memory are still alive. If memory is freed, then the waiting clients are notified (refer to Section 5). The shutdown is implicitly handled by the `postCudaFree()` function.

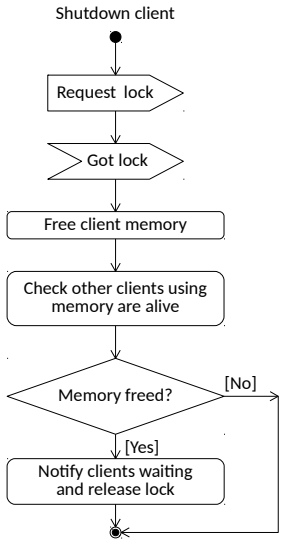


Fig. 7: Shutdown of a schedGPU client

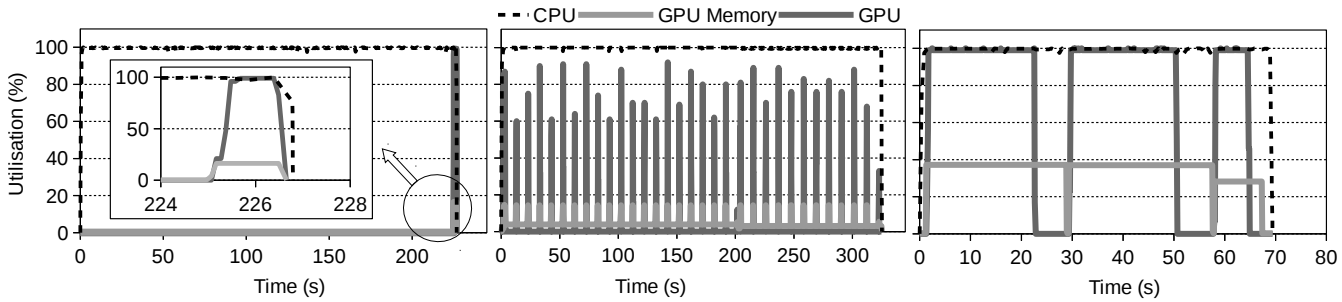
5 Notification Policies

Client applications that wait in a queue for GPU memory are notified when memory is available because another application released it. Policies are required to schedule memory requests of waiting clients. Scheduling policies are reported for managing CPU resources [21], [22]. Popular policies include First-In, First-Out (FIFO) and those that maximise resource utilisation. Priority-based policies are implemented to prioritise execution by the operating system. We adapted these policies in the context of GPU co-scheduling.

The FIFO policy is simple, but is limited in that if the first waiting application's GPU memory request cannot be furnished, then even if there was a subsequent waiting client that could be scheduled to access the GPU has to wait. This potentially reduces the utilisation rate of the GPU. This can be mitigated by using policies based on consumable resources. In the context of GPU co-scheduling, a policy to maximise the usage of memory on the device is ideal, which in turn increases utilisation.

The basic version of both FIFO and Maximum Memory Utilisation (MMU) policies do not consider the quality of service offered to the clients. This requires priority of waiting applications to be accounted for to

7. This avoids a deadlock; the user either has to provide a timeout for the pre-allocation call or receives an instantaneous error message.



(a) ARA (b) MUMmerGPU (c) GPU-BLAST

Fig. 8: CPU and GPU utilisation and GPU memory used for one execution of the applications.

provide preferential services to applications with higher priorities. In this paper, we considered four policies, two basic policies and two priority-based policies.

GPU utilisation can be used as a good priority criterion in policies as in CPU scheduling. However, we have chosen GPU memory utilisation since it is a more stable indicator given that it can be quantified more reliably than GPU utilisation. This is because current techniques employed on GPUs do not provide accurate estimates of utilisation as on the CPU. Therefore, the policies considered in this paper are based on GPU memory.

Policy 1 - First-In, First-Out (FIFO): Consider there are n waiting clients, represented as $C = \{C_1, C_2, \cdots, C_n\}$, and that Δ_i is maximum time client C_i waits for free memory. In this policy, the first waiting client in the queue C_1 is benefited. C_1 waits until there is sufficient free memory. If memory is available, then it is preallocated to C_1, and if there is more free memory then the next client is served. If there is insufficient memory, then client C_1 waits for a maximum time of Δ_1 as it is blocked by the scheduler (also applied for the next 3 policies).

Policy 2 - Maximum Memory Utilisation (MMU): The aim of this policy is to use maximum GPU memory and hence the request of the first client in the queue that can be pre-allocated memory is furnished. If no clients can be serviced, then the clients continue to wait until a subsequent client terminates and memory is available.

Policy 3 - Priority FIFO: This policy is similar to the FIFO policy, but has a priority associated with each client. Therefore, in the queue, the clients with the highest priority are pre-allocated memory. The first client with the highest priority will be served, but if there is insufficient memory to serve this request or there is more memory available after serving a request, then a following client with the same priority is served.

Policy 4 - Priority MMU: This policy is similar to the MMU policy, but has a priority associated with each client. To maximise GPU memory usage the request of the first client with the highest priority in the queue that can be pre-allocated memory is furnished. If no clients in the queue with the highest priority can be attended to, then clients continue waiting until a subsequent client terminates and more memory is available.

The implications of using the above policies for executing workloads is explored in the subsequent sections.

6 Experimental Setup

The hardware platform and the benchmarks and use-cases employed for validating the feasibility of schedGPU are presented in this section.

Hardware Platform: The experimental test-bed used for our experiments is one 1027GR-TRF Supermicro server comprising two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1,600 MHz. One NVIDIA Tesla K20m GPU which has 4,799 MiB of memory is available on the server. The CentOS 6.4 operating system and the CUDA 7.5 with NVIDIA driver 352.39 is used.

Benchmarks: We evaluate the performance of schedGPU using two popular GPU benchmark suites, namely Rodinia [23] and Parboil [24].

Use-cases: Three real-world applications are considered as use-cases in this paper. The first is a catastrophe risk simulation employed in the financial risk industry, referred to as Aggregate Risk Analysis (ARA) [25]. This simulation computes a key risk metric, namely Probable Maximum Loss (PML) on an industry size input comprising 150,000 catastrophic event trials and a collection of one thousand events and their corresponding losses.

The second and third are applications for aligning DNA sequences in bioinformatics. The second application, which is referred to as MUMmerGPU [26], is used for aligning DNA sequence data to a reference sequence which is useful in genotyping and genomics. In our experiments, the search pattern is a sequence length of 25 base pairs that is matched against the reference, which is a complete genome of Bacillus Anthracis allowing up to five differences in an alignment for 500,000 reads.

The third application is referred to as the GPU Basic Local Alignment Search Tool (GPU-BLAST) [27]. The application searches a database of proteins for a nucleotide with a sequence length of 5,000.

The use-cases were chosen based on the following three observations from Figure 8, which shows the CPU and GPU utilisation and the GPU memory in use during execution. Firstly, *low GPU utilisation*. ARA, in Figure 8a, uses GPU acceleration for a short time period at the end of the simulation. For the given input, over 16% of GPU memory is used and therefore, up to a maximum of 6 concurrent instances of the application can be safely executed on this GPU without potential GPU

memory allocation errors (in this paper, we refer to this as 'maximum concurrent instances'). Such concurrent applications that have low GPU utilisation are ideal candidates for schedGPU since the framework can coordinate the access of multiple applications to the GPU, which otherwise would execute sequentially.

Secondly, *moderate GPU utilisation*. MUMmerGPU, in Figure 8b, harnesses GPU acceleration at regular intervals. For the given input, the GPU is used for approximately 50% of the total execution time and the maximum GPU memory used is nearly 15% allowing for up to 6 parallel instances of the application to be reliably executed. Concurrent executions of moderate GPU utilising applications are again ideal candidates for schedGPU since the framework can maximise the number of these applications safely using the GPU.

Thirdly, *high GPU utilisation (and GPU memory is still available)*. GPU-BLAST (Figure 8c) uses the GPU nearly 80% of the total execution time, but for the given input maximum GPU memory used is over 36% of total available memory. This allows for safe execution of up to 2 concurrent application instances. This is not an ideal candidate for schedGPU, however performance gains may be obtained when GPU memory usage drops below 30% towards the end of the execution.

7 Evaluation

In this section, we present the experiments carried out for validating the feasibility of schedGPU. For this we (i) evaluate the overheads associated with the client-server and shared memory approaches, (ii) analyse the performance using popular GPU benchmark suites, (iii) highlight the benefits of employing schedGPU to improve the throughput of concurrent executions of an individual application, and (iv) consider the performance gain of workloads comprising multiple applications.

7.1 Overhead of the approaches

Figure 9 compares different stages of the schedGPU life cycle. Both the client-server and shared memory approaches are considered. For the former, an additional server initialisation and server shutdown stages are required since these are distributed between the client and the server. For the latter, initialisation and shutdown are carried out by the client since no servers are present.

It is observed that the server initialisation and server shutdown stages for the client-server approach are costly in terms of time although they occur only once. The client initialisation and client shutdown stages are however shorter. Regardless, even when excluding the time for initialising and shutting down the server, the total time taken by the client-server is nearly twice as taken by the shared memory approach. This is because communications over UNIX sockets introduce an overhead. Therefore, only the shared memory approach is employed in the experiments considered in subsequent sections.

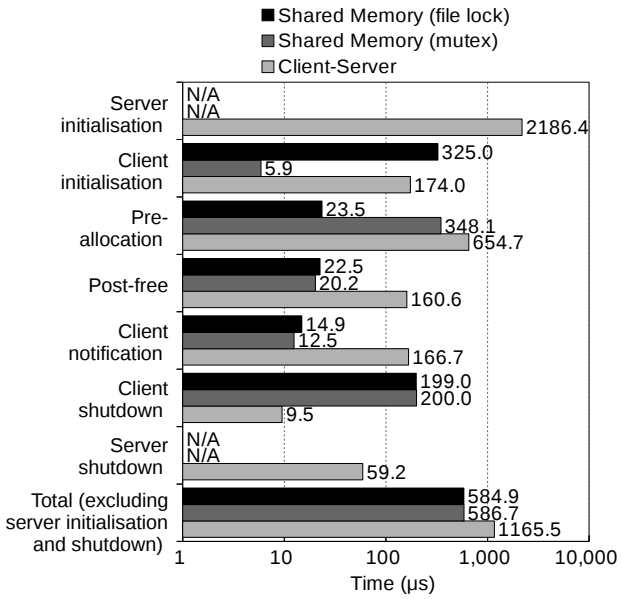


Fig. 9: Comparison of the stages of the schedGPU life cycle for the client-server and shared memory approaches.

Both shared memory approaches using mutexes and file locks offer similar performance. In the initialisation stage, the file locks method requires more time since the notification protocol using system signals needs to be set-up. With mutexes the notification protocol using conditions is set-up in the pre-allocation stage and hence an increase in time for the pre-allocation stage is noted.

Given that both the shared memory approaches have similar performance, in the following sections we consider the file lock method since it is more robust than mutexes by avoiding the problem of abandonment.

The overhead of the different implementations is less than one millisecond. This does not have any impact on long running applications. However, the shared memory implementation does not require an additional service to run on the server. This is valued by administrators of production systems to keep the number of services running on a server to a minimum for security reasons.

7.2 schedGPU with Benchmark Suites

Figure 10 shows the execution time and speed-up of 10 Rodinia and 8 Parboil benchmarks when schedGPU is employed. The tests were run in three different scenarios. The first scenario does not use schedGPU and one instance of the benchmark is safely executed at a time. Hence, if 12 instances of the benchmark were required to be executed, then they are executed sequentially. The second scenario, similarly does not use schedGPU. However, we manually packed tasks to maximise GPU memory usage. This is not realistic, but was pursued for the sake of comparison, as workload schedulers do not know GPU memory required by an application in advance. The third scenario employs schedGPU and safely runs multiple instances of the benchmark.

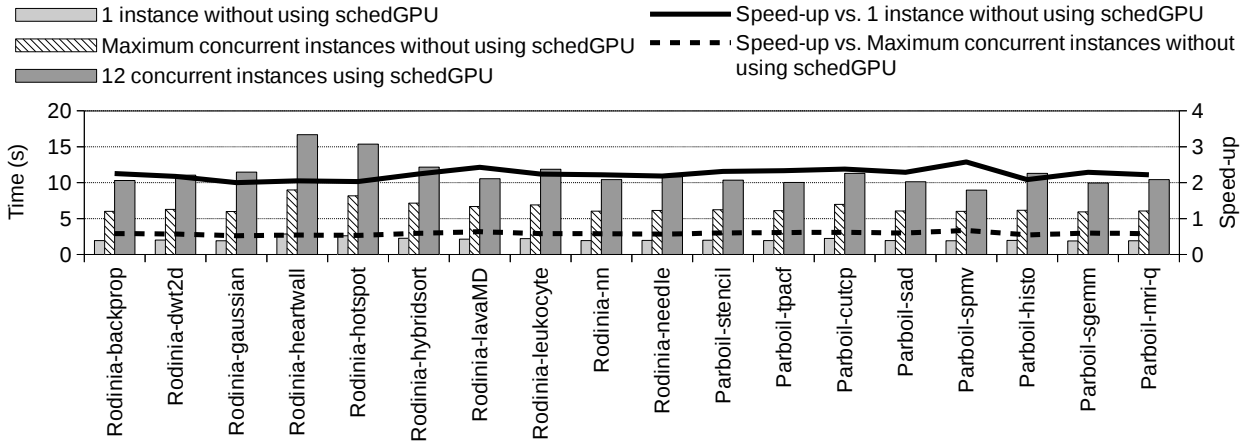


Fig. 10: Speed-up using schedGPU when concurrently running 12 instances of benchmarks selected from Rodinia and Parboil suites.

Multiple instances of an application or multiple applications are executed concurrently using schedGPU. Therefore, the overall execution time of all instances/applications is reduced when compared to running them sequentially, which is referred to as performance speed-up. However, the execution time of an individual instance is not improved and optimising performance of individual applications is not within scope of this paper.

We observe that using schedGPU there is over a 2x speed-up when compared against the first scenario that does not employ schedGPU. On the contrary, compared to the second scenario, the execution time is increased 40% on average when using schedGPU. These benchmarks have short execution times (2 seconds on average), therefore, it is more difficult to compensate the overhead introduced by schedGPU with the potential gain of concurrently using the GPU. However, this is because we have manually packed tasks to maximise the usage of GPU memory, but as previously noted this is not realistic. In the next section, we use production codes instead of benchmarks to demonstrate the benefits of schedGPU.

7.3 schedGPU Performance Gain on Use-cases

We further explore performance in terms of utilisation of GPU resources, speed-up and throughput in the following two ways: (i) on an experimental environment, to study mechanisms to achieve maximum performance of the three use-cases with and without schedGPU, and (ii) on a production environment, to assess the potential of schedGPU using real-world workloads.

7.3.1 Concurrent Execution of Individual Applications

Three schedGPU functions considered in Section 4 were included in the three applications. The initialisation function was included at the beginning of the CUDA program, the pre-allocation function was inserted before the CUDA memory allocations and the post-free function was placed after CUDA release memory calls. Up to 12 instances of each application were concurrently executed (the number of CPU cores in the experimental test-bed).

Figure 11 shows the improvement in execution time and speed-up of the three applications when schedGPU is employed. The three scenarios of Section 7.2 were considered again. It is immediately inferred that when comparing our proposed approach (the third scenario) using schedGPU against (i) the first scenario that does not employ schedGPU there is a 10x speed-up for ARA, nearly 9x speed-up for MUMmerGPU and close to 1.5x speed-up for GPU-BLAST, and (ii) the second scenario the speed-up is approximately doubled when running 12 concurrent instances of ARA and MUMmerGPU. These applications have low and moderate GPU utilisation allowing schedGPU to take advantage of the time periods that the GPU remains under-utilised. During these time periods schedGPU services instances that request the GPU to maximise GPU utilisation. In the second scenario, the execution of large number of instances of an application (more than 6 for ARA and MUMmerGPU respectively) at the same time will not be possible due to insufficient GPU memory. schedGPU still outperforms this unrealistic scenario of manually packing tasks. Not only is it feasible to execute large number of instances using schedGPU, but also a profitable speed-up is noted.

However, there is only a small improvement in performance for GPU-BLAST with the execution of 4 concurrent instances achieving maximum speed-up. The application has high GPU memory utilisation almost during all of its execution (over 30% on average as shown in Figure 8c). Therefore, there is insufficient GPU memory for boosting performance of concurrent instances. Nonetheless, schedGPU yields a small improvement in performance by making use of any spare GPU memory.

Figure 12 shows the average CPU, GPU and GPU memory utilisation when maximum speed-up is obtained for the three applications using schedGPU. The amount of GPU memory utilised by each application is indicated in the figures. GPU utilisation is maximised, which in turn results in an observed speed-up.

Figure 13 shows the frequency distribution of GPU utilisation for the three applications. For all applications it is observed that the amount of time the GPU achieves

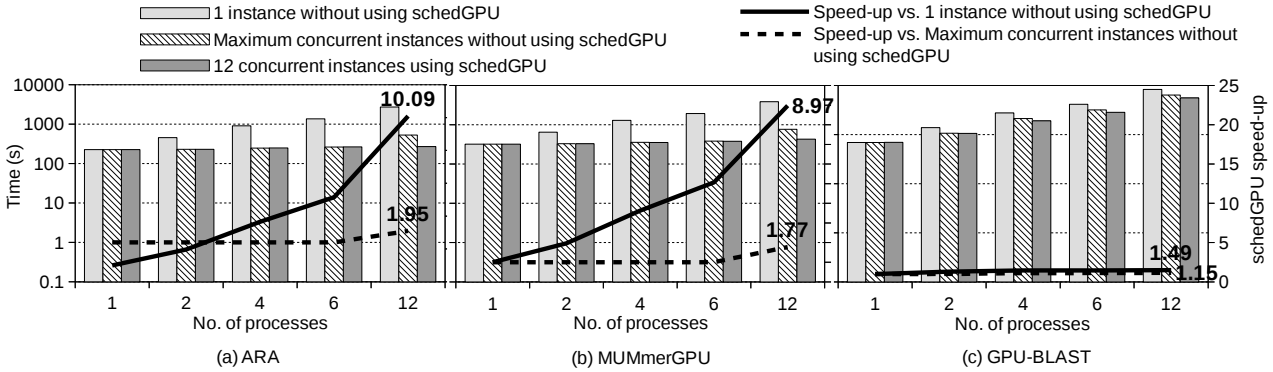


Fig. 11: Speed-up using schedGPU when varying number of instances of an application are concurrently executed.

TABLE 1: Comparison of GPU utilisation and GPU memory utilisation when executing the use-cases

Application	Average GPU Utilisation (%), Average GPU Memory Utilisation (%)		
	1 instance without using schedGPU	Maximum concurrent instances without using schedGPU	12 concurrent instances using schedGPU
ARA	0.51, 0.37	2.74, 0.90	5.26, 2.02
MUMmerGPU	2.46, 4.04	12.73, 26.13	20.96, 41.59
GPU-BLAST	69.48, 33.73	86.28, 63.71	90.24, 70.09

between 91% and 100% utilisation is increased (the time the GPU is not utilised decreases - 0%) and yields a speed up as shown in Figure 11. This validates that schedGPU can improve the utilisation of resources.

Table 1 shows the average GPU utilisation and GPU memory utilisation for the applications when one instance of the application is executed without using schedGPU, running the maximum number of concurrent instances of the application supported without using schedGPU and 12 concurrent instances using schedGPU are employed. It is evident that schedGPU has superior performance since GPU utilisation is improved over 10 times for a single instance and nearly 2 times over six instances for ARA. Similarly, GPU memory utilisation is improved over 10 times for a single instance of MUMmerGPU and nearly 2.25 times over six instances of ARA. The memory utilisation of GPU-BLAST is high without using schedGPU leaving little room for optimisation. However, a small improvement is noted.

Table 2 shows that the individual performance of an application is not improved by co-scheduling. We compare the execution time of the individual application when (i) running 1 instance without using schedGPU and (ii) the average of 1 instance when running 12 concurrent instances using schedGPU. The execution time of an individual instance is not improved by co-scheduling, but there is a collective performance gain when running multiple instances concurrently.

7.3.2 Workloads Comprising Multiple Applications

Our experimental test-bed uses the Slurm [3] workload scheduler for scheduling jobs from multiple users.

TABLE 2: Comparison of execution time when (i) running 1 instance of each use-case without using schedGPU and (ii) the average of 1 instance of the use-case when running 12 concurrent instances using schedGPU

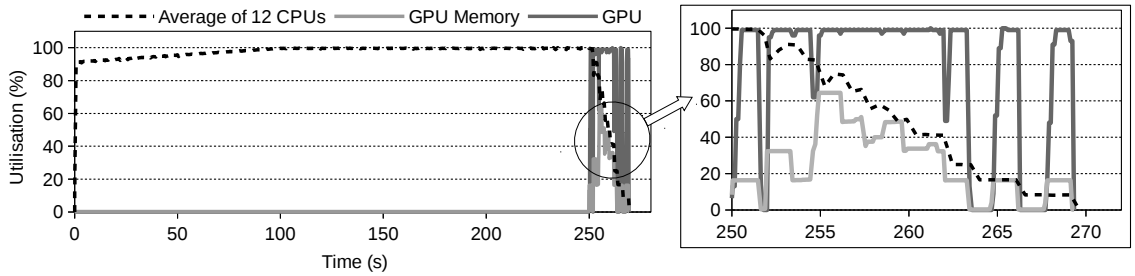
Application	1 instance without using schedGPU	12 concurrent instances using schedGPU
ARA	226.847	260.108
MUMmerGPU	322.799	370.169
GPU-BLAST	69.144	184.751

However, given that one GPU is used in the test-bed, Slurm handles multiple jobs requesting the GPU by sequentially executing them. As expected this results in the under-utilisation of the GPU.

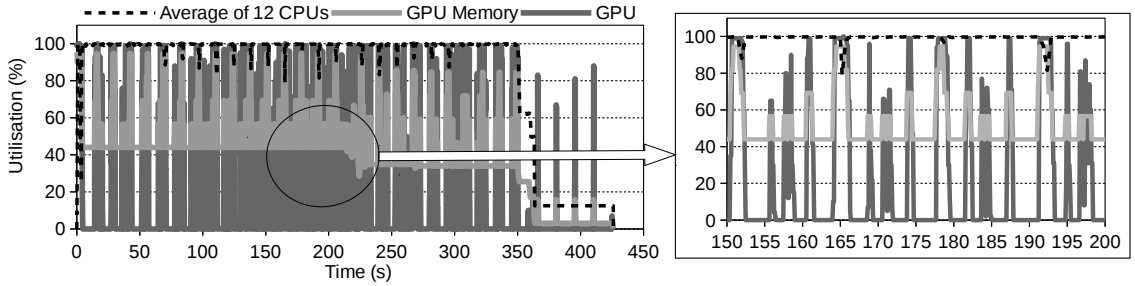
On the other hand, schedGPU can be employed to mitigate the above problem by managing the access of multiple job requests requiring GPUs. If there are m real GPUs and n CPUs, then Slurm is reconfigured (by only making changes to the configuration file) to be in possession of $m \times n$ GPUs. On our test-bed Slurm is reconfigured to have 12 GPUs (1 real GPU $\times$ 12 CPUs). This allows for Slurm to execute up to 12 concurrent jobs as if each CPU had access to a GPU. SchedGPU ensures that the jobs make use of the GPU safely.

A workload comprising 12 concurrent jobs (12 jobs since there are 12 CPUs, each job requires one CPU and one GPU for execution) using 4 instances of ARA, MUMmerGPU and GPU-BLAST applications was submitted to Slurm. The applications have the same input as presented in the previous section. Figure 14 shows average CPU and GPU utilisation and average GPU memory utilisation for the workload. Figure 15 shows CPU utilisation of the cores for each application in the workload (Y-axis shows 0-100% utilisation for each core).

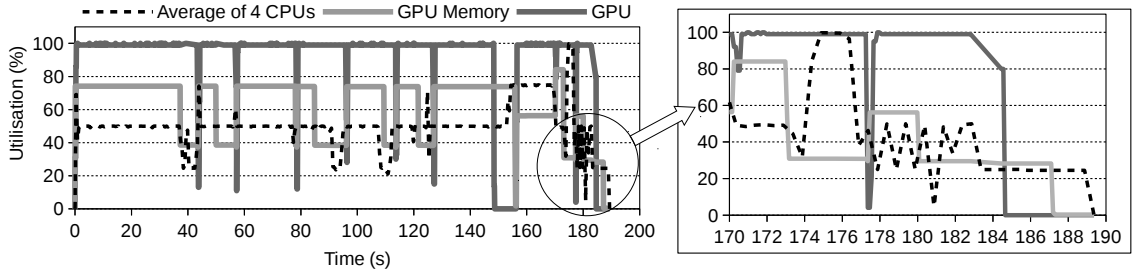
When considering non-priority based policies, it is observed that using FIFO (refer Figure 14a) there are peaks in the GPU and GPU memory utilisation. This is because when sufficient memory is not available to furnish a request, no further requests are considered and hence memory remains under-utilised. However, using the MMU policy (Figure 14b) GPU memory utilisation is more evenly spread out. Requests of waiting clients are immediately furnished to maximise GPU memory



(a) ARA running 12 concurrent instances



(b) MUMmerGPU running 12 concurrent instances



(c) GPU-BLAST running 4 concurrent instances

Fig. 12: CPU and GPU usage when running concurrent instances of the applications using schedGPU.

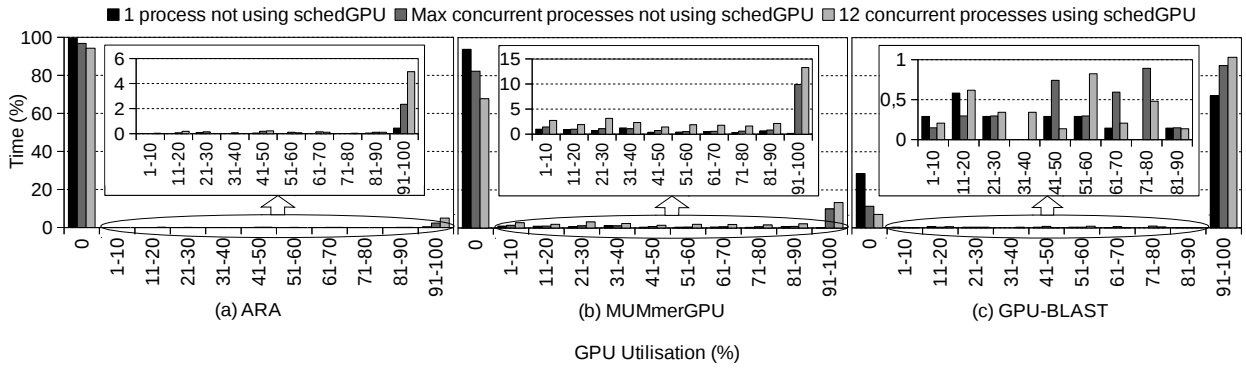


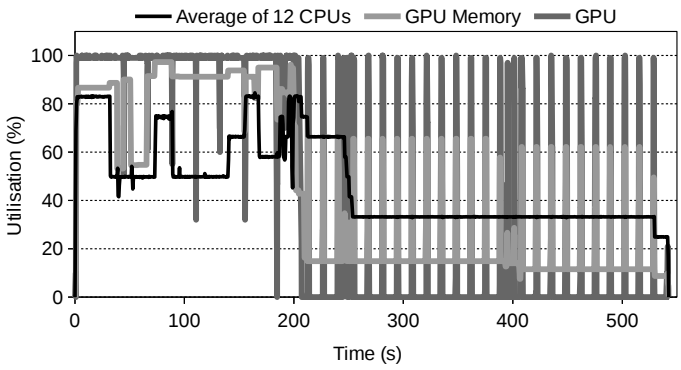
Fig. 13: Frequency distribution of GPU utilisation when executing the application with and without schedGPU.

usage. The MMU policy results in a reduction of nearly 3% in the execution time of the workload over the FIFO policy as shown in Table 3. An improvement of nearly 1.5% is also noted for both the average GPU utilisation and average GPU memory usage for MMU over FIFO.

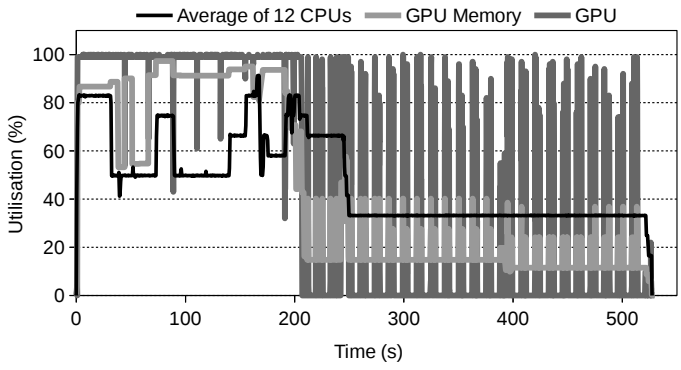
For non-priority based policies it is noted that CPU cores of the jobs waiting for GPU memory remain idle as shown in Figure 15a and Figure 15b. This is noted for MUMmerGPU and GPU-BLAST instances since there is insufficient memory on the GPU to furnish all requests.

Since it is observed that the MUMmerGPU takes the most time for completing execution, all MUMmerGPU instances are assigned a high priority in an attempt to optimise the execution of the workload by further reducing the total execution time. When priority-based policies are taken into account, it is observed that the initial CPU utilisation increases and similar trends to non-priority based policies is observed for GPU utilisation.

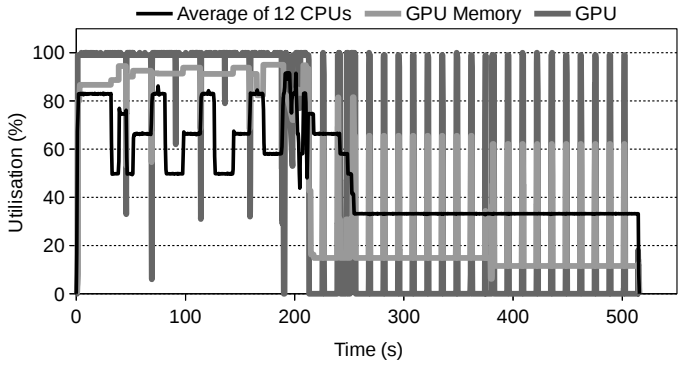
In Figure 15c and Figure 15d, assigning a higher priority to MUMmerGPU instances reduces waiting times for



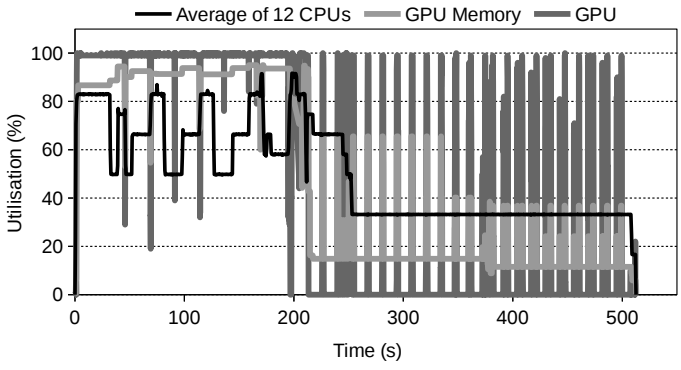
(a) Using FIFO policy



(b) Using MMU policy

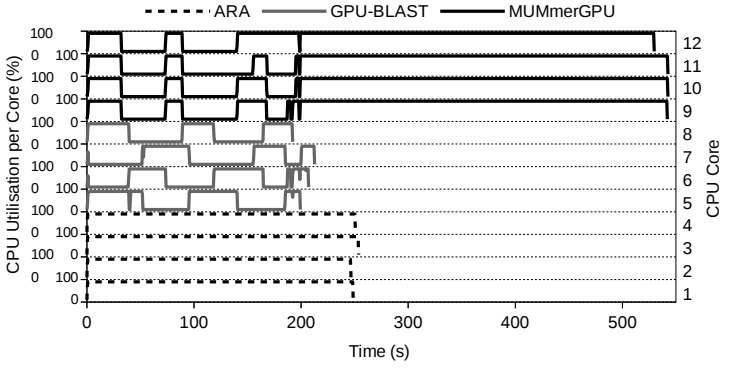


(c) Using Priority FIFO policy

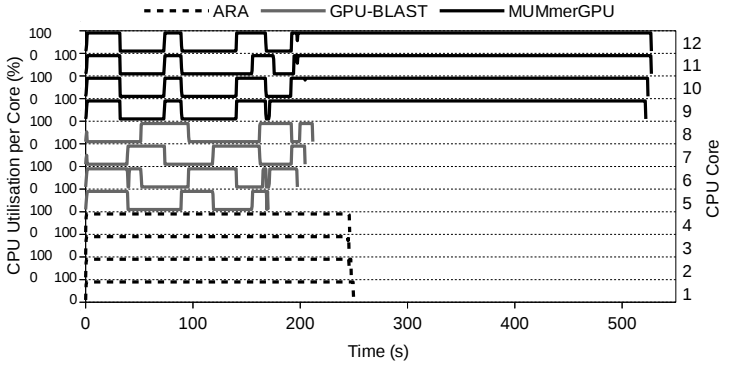


(d) Using Priority MMU policy

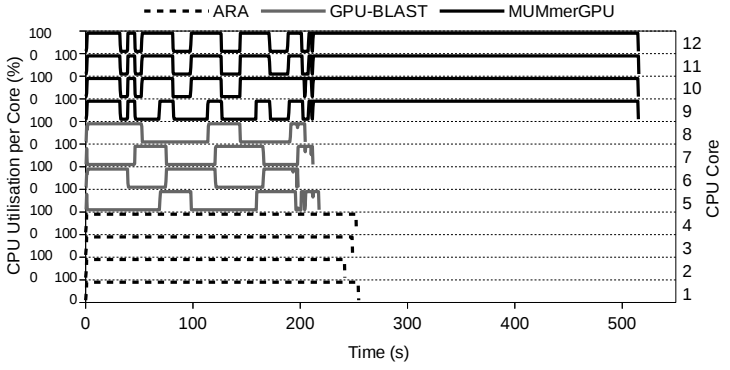
Fig. 14: CPU and GPU usage when running a workload using schedGPU for different client notification policies.



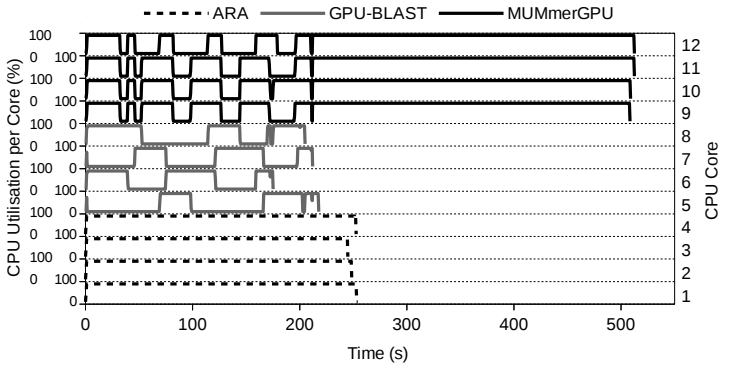
(a) Using FIFO policy



(b) Using MMU policy



(c) Using Priority FIFO policy



(d) Using Priority MMU policy

Fig. 15: Usage per CPU core when running a workload using schedGPU for different client notification policies.

TABLE 3: Comparison of GPU utilisation and GPU memory utilisation when executing a workload comprising multiple applications

Configuration	Time (s)	Average GPU Utilisation (%)	Average GPU Memory Used (%)
Without schedGPU	2,485.20	9.24	3.79
schedGPU FIFO	542.51	43.09	42.65
schedGPU MMU	527.22	43.74	43.25
schedGPU Priority FIFO	515.57	45.59	46.59
schedGPU Priority MMU	512.53	45.75	47.15

free GPU memory, thereby the CPU is idle for shorter periods of time. This translates into a reduction of total execution time using the priority-based policies by 15 seconds over the best case non-priority policy (MMU) as shown in Table 3. Similarly, an improvement of over 4.5% and 9% are noted for GPU utilisation and GPU memory utilisation, respectively, over the MMU policy.

The advantage of using different policies on performance and utilisation is small when compared to the naive FIFO policy. This is because of the generic workload we have chosen in this paper to avoid a bias in our experimental results. Our workload comprises equal number of low, moderate and high GPU utilisation jobs. Even for such a workload there is some benefit in speeding up the overall execution time and utilisation. The benefit of the policies will be more apparent in workloads, for example, where a high memory utilisation job blocks a number of small memory utilisation jobs. A study on the effect of policies on different types of workloads is beyond the scope of this paper.

7.4 Summary

We make three observations from the experiments. Firstly, the overhead of the shared memory approach is significantly less than that of the client-server approach, making it an ideal candidate for facilitating the schedGPU framework (refer Figure 9).

Secondly, the performance gain, measured in terms of average speed-up, average GPU utilisation and average GPU memory utilisation, when concurrently executing individual applications using schedGPU is noted to be up to 10 times better than when not using schedGPU.

Thirdly, for workloads comprising multiple applications, using Slurm along with schedGPU results in a speed-up of up to 5 times in the total execution time. The average GPU utilisation and average GPU memory utilisation is increased by 5 and 12 times, respectively, when compared to not using schedGPU.

8 Related Work

Approaches for efficiently utilising GPUs include (i) scheduling, (ii) kernel-based, (iii) synchronisation, and (iv) architectural approaches. Scheduling approaches include coarse-grain and fine-grain job scheduling. Coarse-grain job scheduling improves the overall throughput by scheduling concurrent applications on to nodes of the cluster [8]. While throughput is improved, the focus is on inter-node scheduling of jobs, without considering a further level of optimisation at the intra-node level. Load balancing is commonly used for fine-grain job scheduling in multiple GPU environments [9], [10]. However, the focus has been to uniformly distribute the executing workload across the GPUs, but not to improve utilisation of the GPUs. This paper focuses on intra-node scheduling at the fine-grain level to maximise GPU utilisation and to improve the overall throughput.

Kernel-based approaches have included event-driven programming models for scheduling on shared GPUs [11]. This approach does not concurrently share the GPU, but interleaves kernel executions on the GPU. A mechanism for concurrent execution of GPU kernels has been proposed [28]. However, the mechanism does not safely handle GPU memory, such that sufficient GPU memory is available for the executing applications. A scheduler to facilitate multiple concurrent kernel executions has been proposed [29]. Only two kernels can be executed and this may lead to potential deadlocks. More complex frameworks for synchronising GPUs have been developed [12], [13]. These require modifying the Linux kernel or GPU drivers, thereby limiting their use in production environments. Kernel-based approaches require extensive modifications, but the schedGPU framework requires no modifications to the source code, if the implicit memory management functionality is used. As considered in Section 4, the framework additionally provides explicit memory management functions requiring the source code to be minimally modified using the API, but offers the developer finer control over memory management. Our approach is simpler than modifying kernels.

Synchronisation approaches manage implicit and explicit synchronisations in GPU hardware and software for improving application concurrency [14]. This approach avoids concurrent GPU operations to be executed sequentially. An application cannot use multiple kernel streams and cannot support unified memory. Our framework achieves synchronisation by a custom protocol we developed using file locks and system signals.

Architectural approaches, such as Multi-Process Service (MPS) [15], [30] or Hyper-Q [16], [31], improves the GPU utilisation by allowing multiple processes, or threads, to simultaneously access a single GPU. However, in this research, GPU memory is not considered, and therefore, jobs fail when GPU memory is not available. The schedGPU framework on the other hand, safely handles GPU memory, and therefore applications do not fail due to insufficient memory but wait in a queue.

Workload schedulers such as Torque [4], PBS [32] or Slurm [3] include mechanisms for scheduling jobs on GPUs. We differentiate our framework from such schedulers in the following two ways. Firstly, schedulers operate at the cluster level (inter-node) and are capable of coarse-grain job scheduling, whereas schedGPU op-

erates at the node level (intra-node) and performs fine-grain job scheduling to share the same physical GPU among multiple CPUs. Secondly, the schedulers work ahead-of-time; the configurations need to be set before execution of the workload. However, schedGPU works just-in-time, such that scheduling is dynamic and occurs during the execution of the workload.

Preemption mechanisms have been developed, but is based on including hardware extensions [33]. More recent GPU architectures provide preemption (for example, on the NVIDIA Pascal architecture) [34]. Preemption prevents long-running applications that block other applications from monopolising the system. Such preemption mechanisms cannot inherently co-schedule applications. Our framework can be employed on both preemptible and non-preemptible GPUs and does not assume the GPU to be either time or space shared. A framework that can be used on non-preemptive accelerators to guarantee a given QoS for an application in terms of time duration by improving GPU utilisation is reported [35]. Although QoS violation due to kernel interference and PCI-e bandwidth contention is minimised, it does not account for GPU memory-based co-scheduling.

9 Conclusions and Future Work

Currently, there are no schedulers that can safely co-schedule multiple GPU applications in terms of memory requirements. This results in the under-utilisation of GPUs in high-performance computing systems. In this paper, we aimed to improve the utilisation of GPUs by proposing an intra-node GPU scheduling framework, referred to as *schedGPU*. We incorporated a client-server and shared memory approach for synchronising the access of multiple applications to the GPU. The schedGPU framework was validated using real-world applications both individually as single applications and collectively as workloads. A gain of over 10 times, as measured by performance speed-up, GPU utilisation and memory utilisation, was obtained for individual applications. For workloads, a speed-up of up to 5 times was noted and the average GPU utilisation and average GPU memory utilisation was increased by 5 and 12 times, respectively.

We intend to pursue the following three areas in our future research on schedGPU. Firstly, exploring the performance of schedGPU against memory paging supported on new GPUs. We note that the memory paging mechanism is only available in the latest NVIDIA GPUs that use the Pascal architecture [34]. Our approach, however, is also compatible with GPUs that do not employ memory paging (pre-Pascal); such GPUs are widely used in current HPC clusters and do not support memory paging. For example in the June 2017 Top500 list, NVIDIA GPUs used in clusters are all pre-Pascal GPUs. Using the NVLink high speed interconnect with Pascal GPUs may outperform schedGPU, but we expect competitive results when compared to the PCI-e version of Pascal GPUs. We anticipate that not all applications will benefit from GPU memory oversubscription and the page migration feature of the Pascal architecture. This is because memory access patterns are sometimes extremely difficult for prefetchers to predict. In these case, schedGPU may be used to complement memory paging for optimising application performance.

Secondly, by considering applications whose GPU memory requirement cannot be known before execution. It may not be always possible to know the total GPU memory required by an application as assumed in this paper. For example, when GPU memory is allocated at runtime; if two or more applications were concurrently executed and gradually increased their GPU memory usage, then when all GPU memory is used some of these applications could require more time to complete execution or may exit with a runtime error. In this case, schedGPU is not beneficial and other ways of improving performance will need to be explored.

Finally, by accounting for applications that do not benefit from sharing the same GPU. For example, consider applications that require large amounts of GPU resources - many kernels, threads and register files per kernel. Even if GPU memory was available, it would not be beneficial to use schedGPU for co-scheduling another application due to the overheads in kernel switching. Running the applications exclusively and sequentially on the GPU may be beneficial. An application can be exclusively allocated to a GPU using schedGPU by simply pre-allocating all GPU memory to the application and then releasing it at the end of execution.

Acknowledgment

This work was funded by Generalitat Valenciana under grant PROMETEO/2017/77.

References

[1] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *IEEE/ACM Conference on Supercomputing*, 2004, pp. 47–.

[2] M. B. Giles and I. Reguly, "Trends in High-Performance Computing for Engineering Calculations," *Philosophical Transactions of the Royal Society of London Series A*, vol. 372, 2014.

[3] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *International Workshop on Job Scheduling Strategies for Parallel Processing*, 2003, pp. 44–60.

[4] Adaptive Computing. (2016) TORQUE Resource Manager. [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/

[5] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. mei Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster," in *10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[6] S. Mittal and J. S. Vetter, "A Survey of Methods for Analysing and Improving GPU Energy Efficiency," *ACM Computing Surveys*, vol. 47, no. 2, pp. 19:1–19:23, 2014.

[7] B. Varghese, J. Prades, C. Reaño, and F. Silla, "Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application," in *11th IEEE International Conference on e-Science*, 2015, pp. 47–56.

[8] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 140–147.

[9] J. I. Agulleiro, F. Vázquez, E. M. Garzón, and J. J. Fernández, "Dynamic Load Scheduling on CPU-GPU for Iterative Tomographic Reconstruction," in *10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2012, pp. 603–608.

[10] L. Chen, O. Villa, and G. R. Gao, "Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems," in *IEEE International Conference on Cluster Computing*, 2011, pp. 386–394.

[11] Y. Suzuki, H. Yamada, S. Kato, and K. Kono, "Towards Multi-tenant GPGPU: Event-driven Programming Model for System-wide Scheduling on Shared GPUs," in *Workshop on Multicore and Rack-scale Systems*, 2016.

[12] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *IEEE Real-Time Systems Symposium*, 2013.

[13] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments," in *USENIX Annual Technical Conference*, 2011, pp. 17–30.

[14] M. Butler, K. Sajjapongse, and M. Becchi, "Improving Application Concurrency on GPUs by Managing Implicit and Explicit Synchronisations," in *21st IEEE International Conference on Parallel and Distributed Systems*, 2015.

[15] P. Sah, "Improving GPU Utilisation with Multi-Process Service (MPS)," in *GPU Technology Conference*, ID S5584, 2015. [Online]. Available: http://on-demand.gputechconf.com/gtc/2015/presentation/S5584-Priyanka-Sah.pdf

[16] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture," in *Zuse Institute Berlin Report*, June 2014, pp. 1–17.

[17] NVIDIA, *CUDA C Programming Guide 8.0*, 2016. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[18] L. Howes, *OpenCL 2.1 Specification*, Khronos OpenCL Working Group, 2015. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf

[19] C. Reaño, F. Silla, and M. J. Leslie, "schedGPU: Fine-grain Dynamic and Adaptive scheduling for GPUs," in *International Conference on High Performance Computing & Simulation*, 2016.

[20] P. K. Immich, R. S. Bhagavatula, and R. Pendse, "Performance Analysis of Five Interprocess Communication Mechanisms Across UNIX Operating Systems," *Journal of Systems and Software*, vol. 68, no. 1, pp. 27–43, 2003.

[21] V. K. Naik, M. S. Squillante, and S. K. Setia, "Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments," in *IEEE/ACM Conference on Supercomputing*, 1993, pp. 824–833.

[22] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronisation Methods of Performance of Parallel Applications," *SIGMETRICS Performance Evaluation Review*, vol. 19, no. 1, pp. 120–132, 1991.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[24] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," in *Center for Reliable and High-Performance Computing*, 2012.

[25] A. K. Bahl, O. Baltzer, A. Rau-Chaplin, and B. Varghese, "Parallel Simulations for Analysing Portfolios of Catastrophic Event Risk," in *SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1176–1184.

[26] C. Trapnell and M. C. Schatz, "Optimising Data Intensive GPGPU Computations for DNA Sequence Alignment," *Parallel Computing*, vol. 35, no. 8-9, pp. 429–440, 2009.

[27] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: Using graphics Processors to Accelerate Protein Sequence Alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[28] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting Concurrent Kernel Execution on Graphic Processing Units," in *International Conference on High Performance Computing and Simulation*, 2011, pp. 24–32.

[29] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-Grained Resource Sharing for Concurrent GPGPU Kernels," in *USENIX Workshop on Hot Topics in Parallelism*, 2012.

[30] NVIDIA, *CUDA Multi-Process Service*, May 2015. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[31] T. Bradley, *Hyper-Q Example*, NVIDIA, 2013. [Online]. Available: https://www.ecse.rpi.edu/~wrf/wiki/ParallelComputingSpring2014/cuda-samples/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[32] B. Nitzberg, J. M. Schopf, and J. P. Jones, "PBS Pro: Grid Computing and Scheduling Attributes," in *Grid Resource Management*, 2004, pp. 183–190.

[33] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Annual International Symposium on Computer Architecture*, 2014, pp. 193–204.

[34] NVIDIA, *Tesla P100*, 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[35] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 681–696.

Carlos Reaño received the MSc (2012) and PhD (2017) degrees in Computer Engineering from Technical University of Valencia, Spain. He is currently a postdoctoral researcher at the Department of Computer Engineering of that university, focusing his research on the virtualisation of scarce resources, mainly GPUs. More information is available from http://mural.uv.es/caregon.

Federico Silla received the MS and PhD degrees from Technical University of Valencia, Spain. He is currently an associate professor at the Department of Computer Engineering of that university. He has previously worked for the Intel Corporation. His research addresses high performance on-chip and off-chip interconnection networks, distributed memory systems and remote GPU virtualisation mechanisms. More information is available from http://www.disca.upv.es/fsilla.

Dimitrios S. Nikolopoulos is Professor and Head of the School of Electronics, Electrical Engineering and Computer Science, at Queen's University of Belfast. He holds the Chair in High Performance and Distributed Computing. His research explores scalable computing systems for data-driven applications and new computing paradigms at the limits of performance, power and reliability. More information is available from http://www.cs.qub.ac.uk/~D.Nikolopoulos/.

Blesson Varghese is a Lecturer at Queens University Belfast and an Honorary Lecturer at the University of St Andrews. He obtained a PhD in Computer Science (2011) and MSc in Network Centred Computing (2008), both from the University of Reading, UK, on international scholarships. Blesson's interests are in developing and analysing state-of-the-art parallel and distributed systems. More information is available from www.blessonv.com.