The Journal of Supercomputing (2024) 80:13386–13410
https://doi.org/10.1007/s11227-024-05956-7

Accelerating the detection of DNA differentially methylated regions using multiple GPUs

Carlos Reaño[1] · Ricardo Olanda[1] · Elvira Baydal[2] · Mariano Pérez[1] · Juan M. Orduña[1]

Accepted: 3 February 2024 / Published online: 6 March 2024

Abstract

DNA methylation analysis has become an important topic in the study of human health. In previous work, we developed a suite of tools to perform this analysis. It includes HPG-Dhunter, a web-based tool for automatic detection of differentially methylated regions (DMRs) between different samples. The back-end of that tool receives an undefined number of simultaneous requests to detect DMRs on different datasets. Currently, simultaneous requests are queued and processed one at a time. This paper proposes a parallel architecture where multiple daemons serve requests simultaneously. Daemons can also share the same physical GPUs. A scheduler manages requests and forwards them to daemons. The number of daemons per GPU is configurable, thus adapting the architecture to the available hardware. Results show that the proposed parallel architecture hugely reduces the execution time. Furthermore, the speedup increases proportionally to the number of available GPUs (up to 7.47x in our experimental setup).

Keywords DNA methylation analysis · Software as a service · GPU computing

✉ Carlos Reaño
carlos.reano@uv.es

Ricardo Olanda
ricardo.olanda@uv.es

Elvira Baydal
mebaydal@disca.upv.es

Mariano Pérez
mariano.perez@uv.es

Juan M. Orduña
juan.orduna@uv.es

1 Departament d'Informàtica, Escola Tècnica Superior d'Enginyeria (ETSE), Universitat de València, Avinguda de la Universitat s/n, 46100 Burjassot, València, Spain

2 Departament d'Informàtica de Sistemes i Computadors (DISCA), Universitat Politècnica de València, Camí de Vera s/n, 46022 València, València, Spain

1 Introduction

DNA methylation analysis has become an important topic in the study of human health and other biotechnology fields like phytology [1] or organic chemistry [2]. The reason for this trend is that DNA methylation plays key roles in local control of gene expression [3], the establishment and maintenance of cellular identity [4], the regulation of mammalian embryonic development [5], and other biological processes [6]. For example, the methylation process might stop a tumor-causing gene from "turning on," preventing cancer, and it also seems to play a decisive role in other complex diseases like Obesity, Hypertension, and Diabetes Mellitus Type 2 (DM2) development [7].

The human reference genome is composed of 3×10^9 nucleotides. In order to analyze the DNA methylation of a person, his/her DNA samples are added bisulfite, and then, DNA sequencers produce millions of short DNA segments (called *reads*), whose size typically does not exceed hundreds of nucleotides, in a fastq file (text file). Each read in the fastq file must be compared to all locations (nucleotides) in the reference genome to find not only the correct location of the read along the reference genome (this is known as the alignment operation), but also the methylation status of the cytosines in the read (the bisulfite is added with this purpose). Each fastq file coming from a next generation sequencer can easily contain tens or hundreds of millions reads. Also, typical study cases require the analysis and comparison of different biological samples coming from different tissues or different individuals, in order to detect different methylation levels in different regions of the DNA, which are called *differentially methylated regions (DMRs)*. Typically, studies about the effects of methylation on cancer [8] use samples coming from a number of normal persons (control) and another samples coming from the same number of persons suffering from cancer (cases), and they try to find DMRs among control and cases samples. Thus, the huge size of the data involved in DNA methylation analysis requires the use of high-performance architectures to process the samples in a timely manner.

In previous works, we developed a complete suite of tools (available at GitHub [9]) which allow to carry out the complete process of DNA alignment and methylation analysis, creation of methylation maps of the whole genome, and the detection and visualization of DMRs between different samples: HPG-Methyl [10, 11], a tool for providing single-base information of the alignment and the methylation status of each input sequence (or read), HPG-HMapper [12], a tool which uses the methylation information of each base after its alignment to build a DNA methylation map which gives information about the methylation level for each base of the reference genome, and a graphic tool called HPG-DHunter [13, 14] for an efficient detection and visualization of DMRs with a high level of usability. HPG-DHunter can identify and display DMRs of different samples at different levels. Nevertheless, HPG-DHunter was developed as a stand-alone tool to be installed on a high-performance platform server with new generation GPU devices. Therefore, this tool requires knowledge about GPU installation, setting up and maintenance. This requirement limits its use by biomedical researchers with little or no knowledge about this hardware and related software packages. Thus, we developed a web-based version of this tool [15], which allows biomedical researchers the use of a powerful tool for methylation analysis, even for those without GPU knowledge. Note that the management of GPUs and their related software is done by system administrators.

Nevertheless, the back-end of the web-based tool should support an undefined number of simultaneous requests to detect DMRs on different datasets. The DMR detection algorithm is based on procedures carried out in the GPU, and therefore, simultaneous requests may have to spend long waiting times in a FIFO queue of incoming requests, until the GPU becomes idle, greatly reducing the back-end throughput. In this paper, we propose a parallel architecture for the back-end server of the web-based tool. The performance evaluation results show that, regardless the number of physical GPUs available in the computing platform acting as a back-end server, a parallel architecture with several daemons (one serving each simultaneous request) sharing each GPU can fully exploit the parallelism of GPUs. The proposed architecture significantly reduces the execution times in regard to the previous architecture, where only a single daemon sequentially processed the simultaneous requests. As it could be expected, if this architecture is used on platforms with several physical GPUs, the speedup increases proportionally to the number of concurrent daemons until a point where the GPU memory is exhausted. It should be noted that the architecture proposed is implemented in the back-end of the web-based tool. Thus, its use is transparent to final users (i.e., biomedical researchers).

The rest of the paper is organized as follows: Sect. 2 shows some background on proposals using GPUs in the field of DNA analysis. Section 3 describes the proposed architecture for the back-end of the HPG-DHunter service. Next, Sect. 4 shows the performance evaluation of the proposed architecture. Section 5 presents related work on schedulers and their architecture. Finally, Sect. 6 shows conclusion remarks and future work to be done.

2 Background

HPG-DHunter [13] is a graphical tool designed for efficient detection and visualization of DMRs at different levels of resolution. Initially, it was designed as a standalone tool that needs to be installed on a high-performance computer with one GPU device. Since typical biomedical researchers may not have the expert knowledge required for installing and managing this tool and the software required for GPU computing, we have created a web-based version of the tool [15] that can be used without the need for specialized knowledge on GPU and CUDA.

The infrastructure of this web-based application is based on a client–server architecture. It has been designed to separate the front-end (user interface and control logic) from the back-end (server-side infrastructure hosting services for processing front-end requests), as can be seen in Fig. 1. The front-end is implemented with angular framework. It runs on the user's browser using HTML5 and CSS to format the content in the browser interface, and JavaScript for capturing events and communication with the back-end. In order to minimize latency when transferring

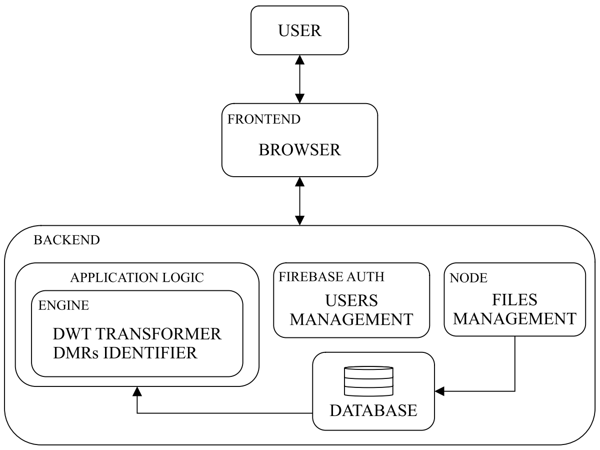

Fig. 1 Previous architecture of the web-based application

large amounts of data, the Web-Socket protocol over a permanent TCP connection is used, with JSON as the data exchange format. The back-end is implemented with Qt framework, being the HPG-DHunter software the core of the system, with a separate module for user access control and file management.

The web-based application has been designed to provide a high degree of security, since the methylation samples provided by users are sensitive data. Individual access is controlled with a username and password, and data uploaded by a given user are stored in independent and isolated folders that are hidden to other users. Furthermore, connections are encrypted using Web-Sockets over TLS (Transport Layer Security).

The front-end interface is intended to provide a level of usability equal or greater than the stand-alone version (HPG-DHunter). The DMR identification is carried out after the user sets the analysis parameters, such as the transformation level, the minimum coverage, a validation threshold, and the minimum percentage of positions with minimum coverage. The system can display any DMR found at any methylation level of every sample uploaded by the user.

The back-end, as shown in Fig. 1, includes the authentication service, the file manager, and the main service of wavelet transformation and identification of DMRs. At the server side, the database module is responsible for file management, and it provides the application logic module with the required data according to the user requests.

The web-based application can handle multiple tasks requested by different users simultaneously. The back-end system manages the availability of RAM memory for loading data from files, and the state of the GPU for computation tasks. A flow diagram illustrating the management of the concurrent requests coming from different customers is shown in Fig. 2. This figure shows three different types of requests, having each

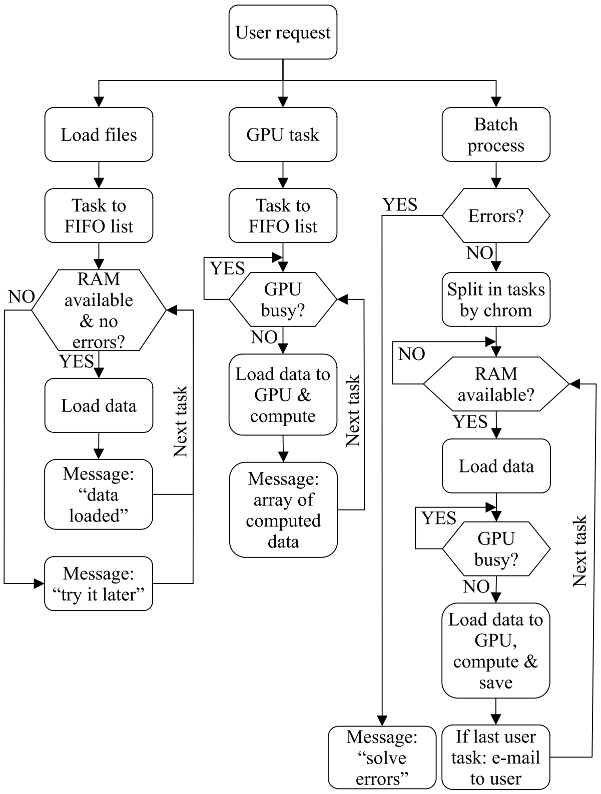

Fig. 2 Flow diagram for the management of concurrent requests in the previous architecture of the web-based tool

one their own FIFO list of tasks. Tasks are launched independently from these lists. The process for loading files does not require GPU availability, but it does require RAM memory, so the availability of RAM memory must be checked before starting this process. GPU tasks needed for visualization require the use of the GPU, so the GPU state must be checked before loading the data array and computing the DWT of the methylation signal. Finally, another process that can be requested by the user is a batch process, which is a long and non-interactive procedure. In this case, the system sends an email to the user when the process has been completed, and the procedure is divided into smaller tasks. All of these strategies are aimed to improve the use of GPU and RAM memory when there are concurrent accesses to the server. In the particular case of GPU tasks, and given that a single GPU is assumed to be available, requests are queued and managed using a first-in, first-out (FIFO) policy. This is a simple and effective way of ensuring that requests are handled in a timely and fair manner, and it guarantees that no request is given priority over any other.

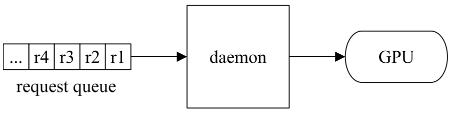

Fig. 3 Current architecture

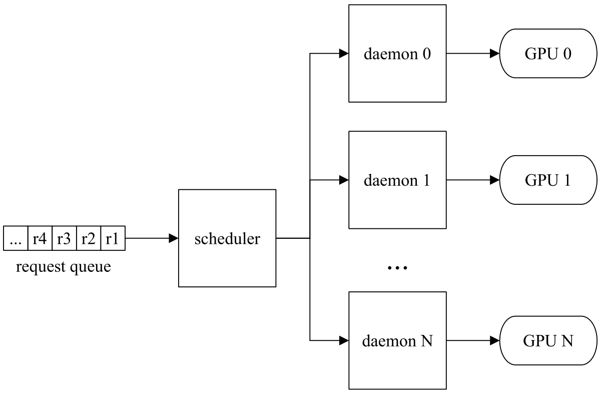

Fig. 4 Proposed architecture for a scenario with one daemon per GPU

3 A new architecture designed for using multiple GPUs

In this section, we first describe the current architecture for the back-end system when managing user requests requiring the use of the GPU (i.e., a GPU task or a batch process, as shown in Fig. 1. For the remainder of the document, we will refer to these user requests as GPU requests). Then, we describe the proposed architecture, designed for using multiple GPUs if they are available in the server computing platform.

Figure 3 shows the current architecture of the system server for serving the user GPU requests. Regardless of the number of GPUs in the underlying computing platform acting as the back-end server, this architecture is only capable of using a single GPU card, which is managed by a single daemon. Therefore, all the GPU requests are queued in a FIFO queue, and they are sequentially processed by a single daemon.

Although this architecture provides parallel processing within the GPU, the processing of different requests is carried out sequentially, and therefore, it can yield very long response times to some user requests if several users are simultaneously accessing the server. Thus, in order to provide a scalable tool with the number of user requests, a new architecture that can take advantage of multiple GPU cards in the back-end server is needed.

Figures 4, 5 and 6 show the proposed architecture of the system server for serving the user GPU requests, along with possible scenarios it enables. This new architecture includes the following elements:

- A request queue storing the requests received from users.

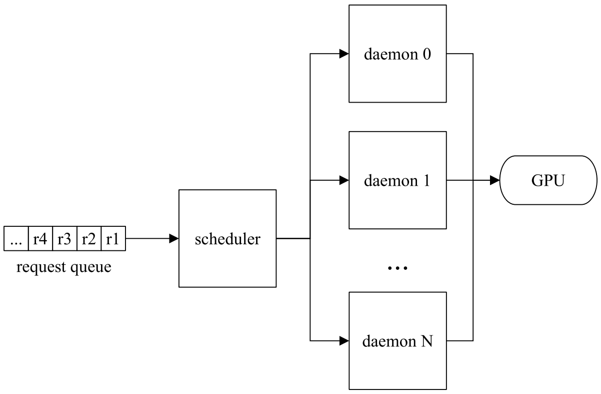

Fig. 5 Proposed architecture for a scenario where multiple daemons share one GPU

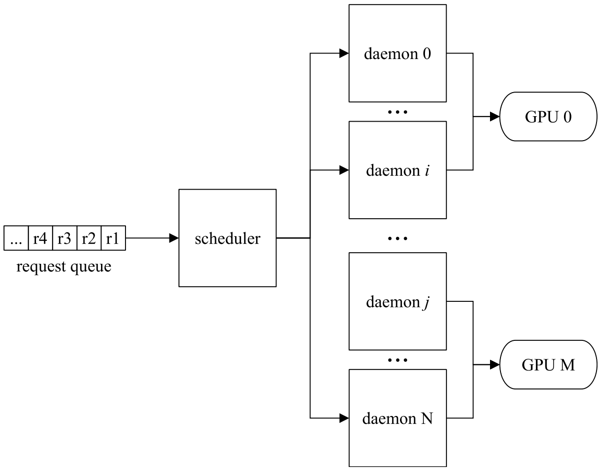

Fig. 6 Proposed architecture for a scenario where multiple daemons share multiple GPUs

- A scheduler, responsible for the management of the queue and the forwarding of the requests to the daemons.
- Daemons for serving the user requests. These daemons should use all the GPUs available in the system.

Regarding the possible scenarios that the proposed architecture enables, we can find the following ones:

- Figure 4: the system has multiple GPUs, and there is one and only one daemon per GPU. Each daemon uses a different GPU to serve a given request.
- Figure 5: the system only has one GPU, and there are multiple daemons sharing that GPU.

- Figure 6: the system has multiple GPUs, and there are multiple daemons sharing those GPUs.

As it can be observed, the main difference between the current architecture (Fig. 3) and the proposed architecture (Figs. 4, 5 and 6) is the scheduler, which enables the utilization of all the GPUs available in the computer platform. The scheduler allows the configuration of the following parameters:

- Scheduler mode. It can be set to `static` or `dynamic`. If the mode is set to `static`, daemons must be manually started by the system administrator. If the mode is set to `dynamic`, daemons are managed by the scheduler. More details about this mode can be found in Sect. 3.1.
- Scheduling policy. Currently, it can be set only to `round_robin`. More details about this policy can be found in Sect. 3.2. If required, other policies can be included in the future.
- IP address and port where the scheduler is listening for user requests. More details about communication and synchronization of the scheduler with the users can be found in Sect. 3.3.
- IP addresses and ports where daemons are listening for requests forwarded by scheduler. More details about communication and synchronization of the daemons with the scheduler can be found in Sect. 3.3.

Currently, these parameters are configured via command line. In the future, we plan to allow also the use of a configuration file. Listing 1 shows how to configure the scheduler parameters via the command line. Listing 2 shows an illustrating example configuring two daemons. As we can see, in this example the scheduler mode is set to dynamic and the scheduling policy is set to round robin. The scheduler and the two daemons are running in the same server (loopback IP), and different ports are used for each of them.

```
./hpg_scheduler                                  \
     <scheduler_mode> <scheduling_policy>  \
     <scheduler_ip>   <scheduler_port>     \
     <daemon1_ip>     <daemon1_port>       \
    [<daemon2_ip>     <daemon2_port>  ...]
```

Listing 1 Configuring the scheduler parameters via the command line.

```
./hpg_scheduler dynamic round_robin \
    127.0.0.1 1234                  \
    127.0.0.1 1235                  \
    127.0.0.1 1236
```

Listing 2 Configuring the scheduler parameters via the command line.

3.1 Dynamic scheduler mode

As commented above, the scheduler currently allows two modes, namely `static` and `dynamic`. If the mode is set to `static`, daemons must be manually started by the system administrator. If the mode is set to dynamic, daemons are automatically started, stopped, and restarted again (when necessary) by the scheduler, depending on the number of user requests. The threshold values for starting, stopping, and restarting again daemons are configurable, as they depend on the computing power of the underlying hardware platform. There are three configurable threshold values or system load levels:

- `daemons_low`: number of daemons started by the scheduler at initialization by default. It is the minimum number of daemons that are running if the number of user requests is low.
- `daemons_mid`: number of additional daemons started (or restarted) by the scheduler when more daemons than in the low level are required to meet the current load. It is also the number of daemons stopped by the scheduler if the number of daemons in the low level is enough to meet the needs. (Note that this last scenario assumes that the scheduler has previously started additional daemons due to an increase in user requests.)
- `daemons_high`: number of additional daemons started (or restarted) by the scheduler when more daemons than in the middle level are required to meet the current load. It is also the number of daemons stopped by the scheduler if the number of daemons in the middle level is enough to meet the needs. (Again, note that this last scenario assumes that the scheduler has previously started additional daemons due to an increase in user requests.)

Algorithm 1 shows the algorithm run by the scheduler to start, stop, and restart again daemons, depending on the system load. As it can be observed in the algorithm, at start-up the scheduler starts the minimum number of daemons. If at any moment more daemons are required, they are started gradually. First, the number of additional daemons configured for a middle-level system load is started. If even more daemons are required, the number of daemons configured for a high-level system load are started (i.e., the maximum). In a similar way, daemons are stopped gradually. If at any moment too many daemons are idle, then some daemons are stopped, until reaching the number of daemons configured for a middle level. If there are still too many daemons idle, some more daemons are stopped, until reaching the number of daemons configured for a low-level system load (i.e., the minimum).

Algorithm 1 Algorithm run by the scheduler in the `dynamic` mode to start, stop, and restart again daemons depending on the system load

```
   // Max. number daemons in mid. level
1  $max\_mid = daemons\_low + daemons\_mid$;
   // Max. number daemons in high level
2  $max\_high = max\_mid + daemons\_high$;
3  if $daemons\_free \leq 1$ then
       // Start more daemons?
4      if $daemons\_started = daemons\_low$ then
5          start $daemons\_mid$ daemons;
6      else if $daemons\_started = max\_mid$ then
7          start $daemons\_high$ daemons;
8      else
           /* Max. number of daemons running. No more daemons can be
              started.                                              */
9      end
10 else if $daemons\_started \neq daemons\_low$ then
       // Stop some daemons?
11     if $daemons\_started = max\_high$ then
12         if $daemons\_free > (daemons\_high + 1)$ then
13             stop $daemons\_high$ daemons;
14         end
15     else if $daemons\_started = max\_mid$ then
16         if $daemons\_free > (daemons\_mid + 1)$ then
17             stop $daemons\_mid$ daemons;
18         end
19     else
           /* Min. number of daemons running. No more daemons can be
              stopped.                                              */
20     end
21 end
```

3.2 Round-Robin scheduling policy

The scheduler currently allows a single scheduling policy, namely `round_robin`. If required, other policies can be included in the future. The behavior of the `round_robin` is as follows: When more than one daemon are available to serve a user request, the scheduler assigns and forwards the first user request in the queue to the first idle daemon, until all the daemons are busy, following a round-robin approach. The scheduler also takes into account the current load of each GPU (i.e., number of daemons free for that specific GPU). The aim of this policy is to distribute the load equally across all the GPUs.

Algorithm 2 shows the algorithm run by the scheduler to select the daemon to serve a user request. As it can be observed in the algorithm, if there are no free daemons, the scheduler waits. A mutex and a condition variable is used for this purpose. Once there are free daemons, the scheduler selects the daemon to serve the request based on the GPU used by the daemon. In first place, the GPU selected is the GPU next to the last GPU used. For instance, if the last GPU used was the GPU 0, the next GPU to use by default is GPU 1. Once the maximum number of GPUs is reached, GPU 0 is selected again. In second place, once the next GPU is selected by default, the scheduler compares the load of the selected GPU with the load of the other GPUs. If there is a GPU with less load (i.e., higher number of daemons free), that GPU will be selected. Following this algorithm, the user requests are evenly distributed across all the GPUs.

Algorithm 2 Algorithm run by the scheduler in the `round_robin` scheduling policy to select the daemon to serve a user request

```
1  while $!free\_daemons$ do
       // Wait for free daemons
2  end
3  $selected\_gpu = (last\_selected\_gpu + 1)\ \%\ NUM\_GPUS$;
4  for $i = 0;\ i < NUM\_GPUS;\ i++$ do
5      if $selected\_gpu = i$ then
6          continue;
7      end
8      if $daemons\_free[selected\_gpu] < daemons\_free[i]$ then
9          $selected\_gpu = i$;
10     end
11 end
12 $last\_selected\_gpu = selected\_gpu$;
```

3.3 Communication and synchronization of the scheduler with users and daemons

The scheduler serves user requests in the IP address and port provided in the configuration parameters. Similarly, daemons will serve requests forwarded by scheduler in the IP addresses and ports provided in the configuration. TCP/IP sockets are used for communication and synchronization of the scheduler with users and daemons. In Fig. 7, we can see a sequence diagram showing the communication and synchronization of the scheduler with users and daemons when serving user requests.

Figure 7 shows that after initialization the scheduler main thread starts daemons for serving requests. In this case, we have assumed that the scheduler mode has been set to `dynamic`. The scheduler then waits for user requests. When a user sends a request, the scheduler main thread creates a new worker thread to handle it. The main thread then uses the previously explained Algorithm 1 to determine if the

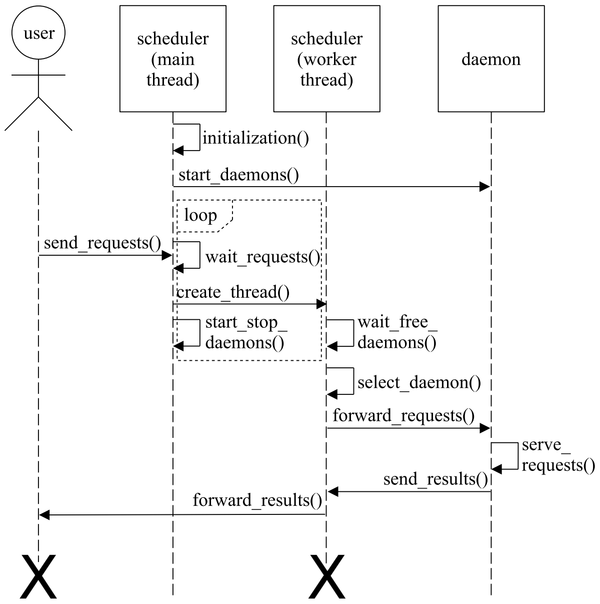

Fig. 7 Sequence diagram showing communication and synchronization of the scheduler with users and daemons when serving user requests. Note that in a real scenario there will be multiple users, scheduler worker threads and daemons

current number of daemons is appropriate for the current system load. At this point, daemons are automatically started or stopped if necessary by the scheduler, depending on the number of user requests in the system. After that, the scheduler main thread waits again for more user requests. A timer ensures that the scheduler periodically checks both the number of daemons and the system load, to stop daemons if necessary when no user requests are received.

Being executed in parallel to the scheduler main thread, the scheduler worker thread handles the user requests. In this case, we assume that the scheduling policy has been set to `round_robin`. In first place, the scheduler worker thread waits until there is an idle daemon. Once there are one or more free daemons, it selects one of them using Algorithm 2. Then, the user request is forwarded to the selected daemon, which serves the request. Finally, the results are sent back to the user. This process is repeated for each user request. When the user has no more requests and ends the session, the associated scheduler worker thread terminates.

4 Results

In this section, we evaluate the performance of the proposed architecture. First, we detail the experimental setup. Then, we compare the performance of the previous architecture with the proposed one when running a single request on a computer platform with a single GPU available. Next, we analyze the performance when receiving multiple, simultaneous requests on the same platform. Finally, we evaluate the performance when using multiple GPUs. Each value shown in the tables and figures included in this section have been computed as the average value of 10 executions of the corresponding test.

4.1 Experimental setup

The system used in the experiments is a rack-mounted 2-socket server BullSequana X450-E5, featuring two 20-core Intel(R) Xeon(R) Gold 6230 CPU at 2.10 GHz, 191 GB RAM, a RAID 1 with two 10.9 TB HDD, and four GPUs Tesla V100 PCIe with 32GB of RAM each. The operating system installed on this platform is Rocky Linux 8.7 (Green Obsidian), including NVIDIA CUDA 12 with NVIDIA driver version 525.60.13.

The test used for the experiments is a very short one that requires very short computation time in the GPU. The reason for selecting such test is that in this paper we focus on the scheduler not on the computation on the GPU, which has already been addressed in previous works [13]. As commented in previous sections, the main difference between the previous architecture and the architecture proposed in this paper is the scheduler. Therefore, a short test is the worst case to test our approach, because the overhead introduce by the scheduler will not be hidden behind a long computation time in the GPU.

4.2 Performance evaluation for a single request

Table 1 presents a comparison of the previous architecture with the proposed architecture described in the previous section. Different values are compared, namely execution time, CPU utilization, CPU memory, GPU utilization, and GPU memory. The maximum relative standard deviation (RSD) observed in these experiments was 0.290 for the execution time of the previous version.

Table 1 shows that the proposed architecture slightly increases the execution time (30 ms) on average. The rest of the metrics show negligible differences (not seen with only two decimal digits). In order to better explain the reason of this increase in the execution time, Figs. 8 and 9 present more detailed charts, where Fig. 8 shows the performance values for the previous version and Fig. 9 shows the ones for the proposed version. These figures show that the increase in the execution time of the proposed version is the time required by the scheduler to forward the request from

Table 1 Performance comparison between the previous and the proposed architecture for a single request and a single GPU

Average values	Previous version	Proposed version
Execution time (s)	9.49	9.52
CPU utilization (%)	1.46	1.46
CPU memory (%)	1.01	1.01
GPU utilization (%)	1.25	1.25
GPU memory (%)	1.02	1.02

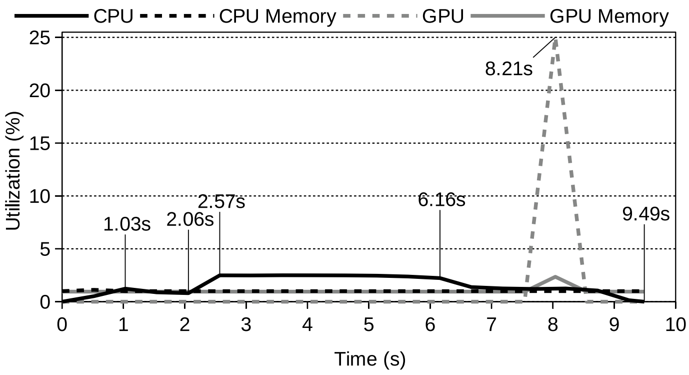

Fig. 8 Detailed execution of the previous architecture (original version) for a single request and a single GPU

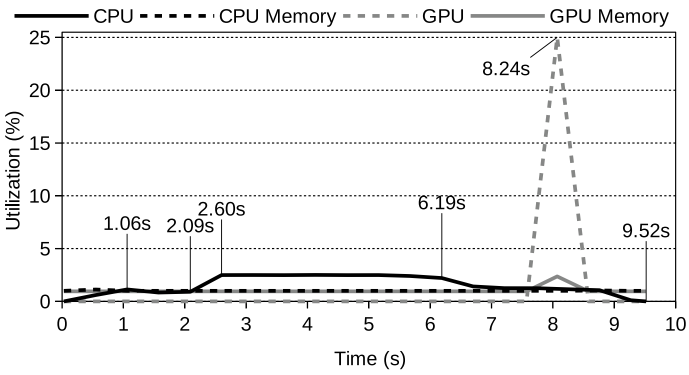

Fig. 9 Detailed execution of the proposed architecture (new version) for a single request and a single GPU

the request queue to the daemon that serves the request. As it can be observed, once that has happened, the new version just mimics the original version with a delay of 30 ms. Several points have been labeled in the figure to make the comparison easier.

It should be noted that in this experiment, only one request was submitted to the system and only one daemon was serving requests. For such reason, the scheduler required very little time to forward the request to the available daemon. In the next section, we evaluate how this time is increased when running multiple requests.

4.3 Performance evaluation for multiple simultaneous requests

In this section, we evaluate the performance when two or more requests are concurrently submitted to the system, and a single GPU is available in the back-end computer platform. For the results of the new version, one daemon is using the single

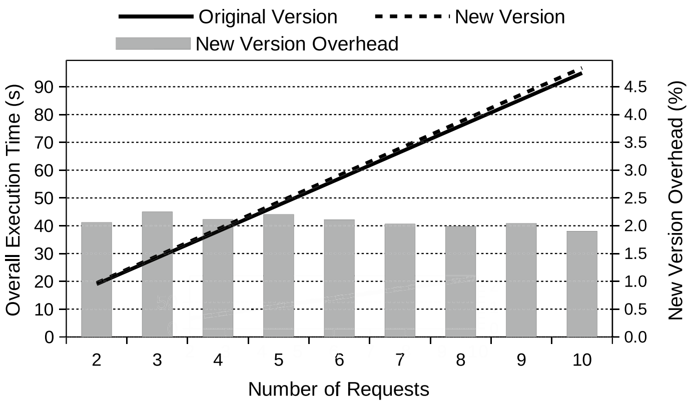

Fig. 10 Evaluation results for multiple simultaneous requests and a single GPU

GPU, i.e., scenario shown in Fig. 4 with only one GPU. Figure 10 shows the results for a number of concurrent requests ranging from 2 to 10.

As it could be expected, in this case the overhead introduced by the scheduler is higher (1.85% on average) than in the case of a single request. It can also be observed that this overhead tends to decrease slightly as the number of requests increases. The maximum RSD observed in these experiments was 2.050, corresponding to the execution time of 3 concurrent requests with the new version. Regarding the execution times, there are not significant differences between the previous architecture and the proposed one. These results show that the proposed architecture does no add a significant overhead for a typical stand-alone computer with a single GPU.

4.4 Performance evaluation using multiple GPUs

In this section, we evaluate the performance when more than one request is concurrently submitted to the system and multiple GPUs are used. That is, we want to measure the benefits of the new proposed architecture when more than one GPU are present in the computer platform acting as a back-end server. For these experiments, we assume that there is one daemon per GPU, as previously shown in Fig. 4. The evaluation results are shown in Figs. 11, 12, 13 and 14.

Figure 11 shows the results for a number of concurrent requests ranging from 2 to 10 when using 2 GPUs. The maximum RSD observed in these experiments was 1.775, for the execution time of 3 concurrent requests with the new version. This figure shows that, as it could be expected, the new version of the architecture clearly takes advantage from the underlying hardware in the computing platform, achieving a speedup of up to 1.96x, quite close to the theoretical maximum.

Figure 12 shows the analog results when using 3 GPUs. The maximum RSD observed in these experiments was 2.358, for the execution time of 7 concurrent requests with the new version. In this case, the speedup achieved with the proposed architecture is 2.84×.

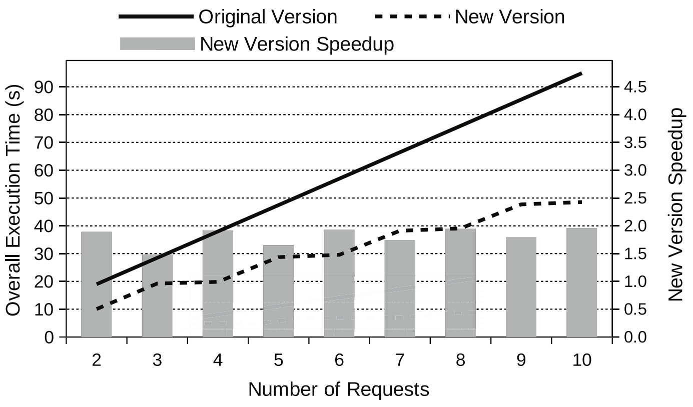

Fig. 11 Evaluation results when multiple requests are concurrently submitted to the system using 2 GPUs and 1 daemon per GPU

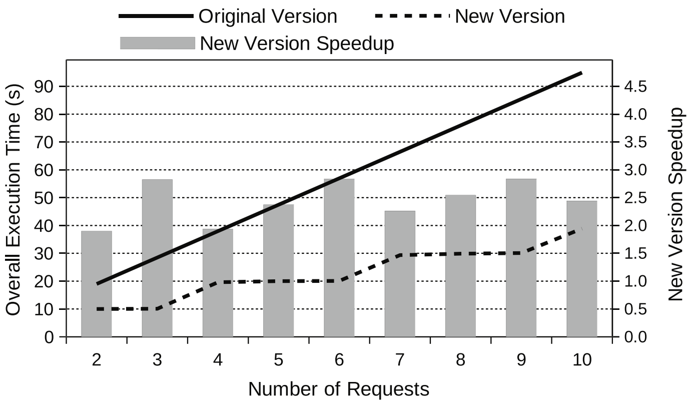

Fig. 12 Evaluation results when multiple requests are concurrently submitted to the system using 3 GPUs and 1 daemon per GPU

Figure 13 shows the results for the case where 4 GPUs are available in our system. The maximum RSD observed in these experiments was 2.313, for the execution time of 2 concurrent requests with the new version. In this case, the speedup achieved by the proposed architecture compared to the previous one is 3.64×.

Finally, Fig. 14 shows the average speedup of these experiments for a different number of GPUs available in the computing platform acting as a back-end server. As we can observe, the proposed architecture scales proportionally to the number of available GPUs.

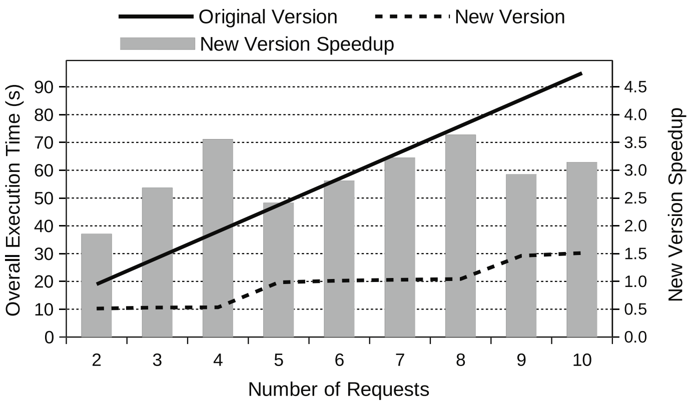

Fig. 13 Evaluation results when multiple requests are concurrently submitted to the system using 4 GPUs and 1 daemon per GPU

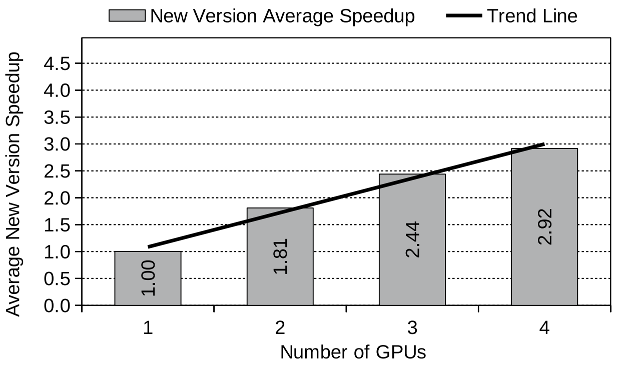

Fig. 14 Average speedup of the new version using multiple GPUs and one daemon per GPU

4.5 Performance evaluation with a single GPU shared by multiple daemons

In this section, we evaluate the performance of the proposed architecture for the case illustrated in Fig. 5. Similarly to the experiments in the previous section, there are simultaneous requests, which are processed by several daemons. However, in this case, all the daemons share a single GPU. The reason for performing this experiment is that Figs. 8 and 9 in Sect. 4.2 show that the GPU utilization and the GPU memory used by a single request are low (1.25 and 1.02% on average, respectively). Thus, for this specific case, it is possible to run multiple daemons (i.e., GPU applications) sharing the same GPU. The performance evaluation results are shown in Figs. 15, 16, 17 and 18.

Figure 15 shows the results varying the number of concurrent request from 2 up to 10 when 2 daemons share the same GPU. The maximum RSD observed in these experiments was 1.618 for the execution time of 4 concurrent requests with the new

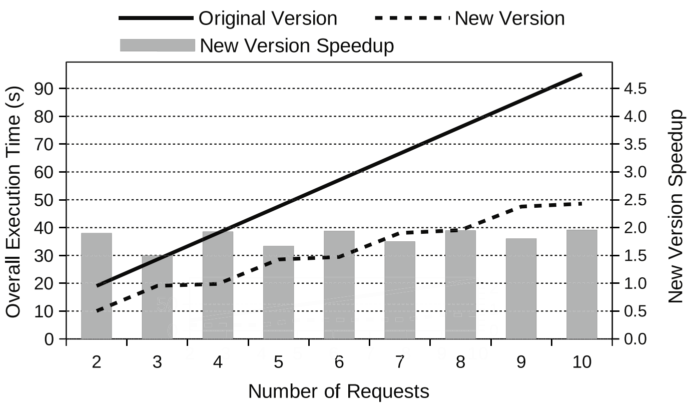

Fig. 15 Evaluation results when multiple requests are concurrently submitted to the system using a single GPU, and 2 daemons sharing a single GPU

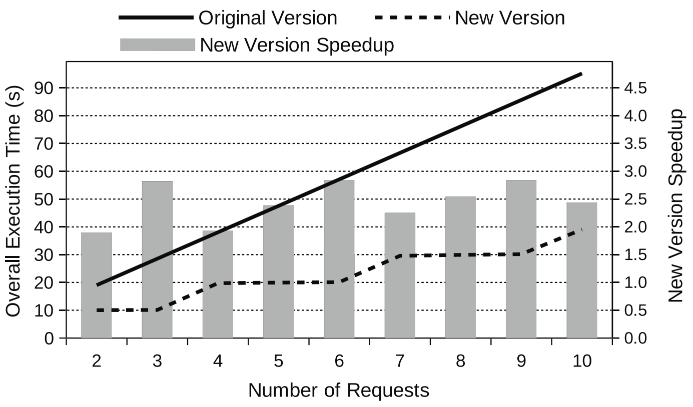

Fig. 16 Evaluation results when multiple requests are concurrently submitted to the system using a single GPU, and 3 daemons sharing a single GPU

version. This figure shows that the proposed architecture achieves an speedup of up to 1.96x in regard to the previous version of the architecture.

Figure 16 shows the results for the same scenario, but this time with 3 daemons sharing each GPU. The maximum RSD observed in these experiments was 2.307, for the execution time of 7 concurrent requests with the new version. In this case, the proposed architecture achieves a speedup of 2.84×. Notice that, in this case and the following ones, when the number of daemons is greater than the number of requests, the remaining daemons are idle.

Figure 17 shows the results for the analog scenario when 4 daemons share the same GPU. The maximum RSD observed in these experiments was 1.559, for the execution time of 3 concurrent requests with the new version. In this case,

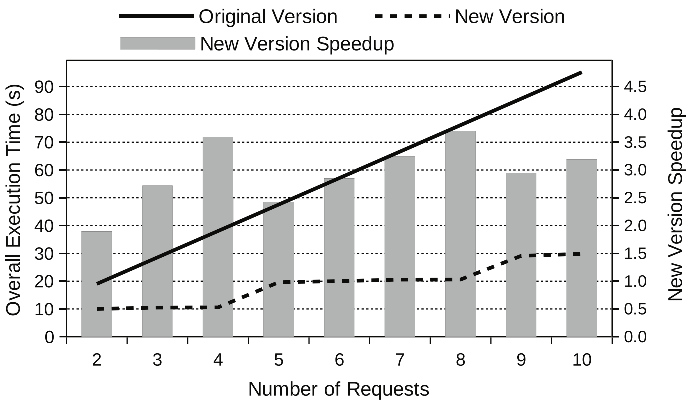

Fig. 17 Evaluation results when multiple requests are concurrently submitted to the system using a single GPU, and 4 daemons sharing a single GPU

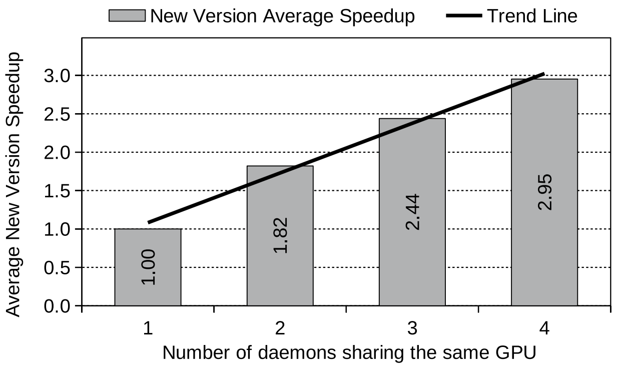

Fig. 18 Average speedup of the new version when multiple daemons share a single GPU

the speedup achieved by the proposed architecture in regard to the previous one reaches 3.70×.

Finally, Fig. 18 shows the average speedup of these experiments, varying the number of daemons that share the same GPU. As we can observe, the proposed architecture scales proportionally to the number of daemons sharing each GPU. These results show that the proposed architecture is also capable of fully exploiting the computing capabilities of the existing GPUs.

If we compare these results with the ones shown in Sect. 4.4, we can observe that the performance is very similar. This is due to the low GPU utilization and the GPU memory required by each daemon for these experiments, which allows to run multiple daemons on the same GPU with no performance loss. Therefore, we can conclude that using multiple daemons per GPU can potentially increase the performance obtained by the proposed architecture. The number of daemons per GPU will depend on the characteristics of the specific test, mainly GPU utilization and GPU memory.

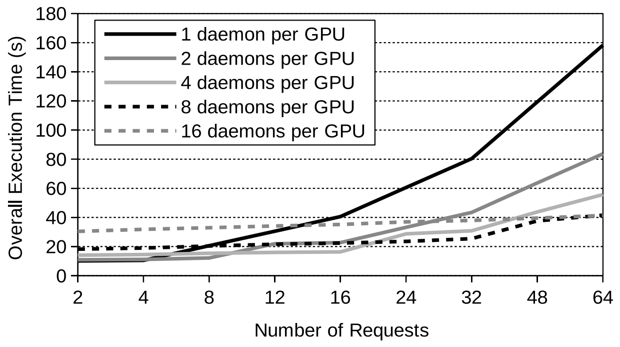

Fig. 19 Execution time of the proposed architecture for the case of multiple GPUs and multiple daemons for each GPU

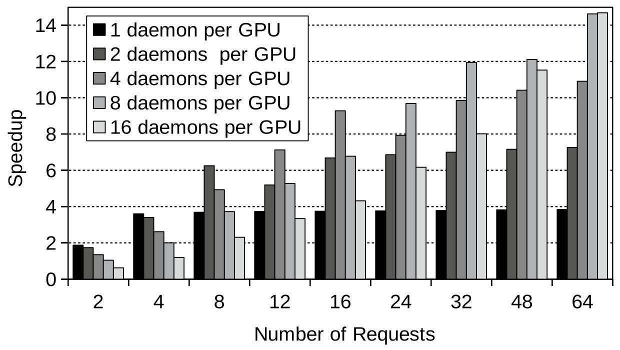

Fig. 20 Speedup of the proposed architecture for the case of multiple GPUs and multiple daemons for each GPU

4.6 Performance evaluation with multiple GPUs and multiple daemons per GPU

In this section, we go one step further and we evaluate the performance when multiple requests are concurrently submitted to the system and multiple daemons share each of the GPUs. The scenario is shown in Fig. 6. In our experimental setup, there are four GPUs available in the computing platform acting as a back-end server (i.e., M is 3). In our experiments, the number of daemons per GPU (i.e., N) ranges from 0 to 15. Results are shown in Figs. 19, 20 and 21. The maximum RSD observed in these experiments was 2.483 for the execution time of 8 concurrent requests, using 4 GPUs and 8 daemons per GPU.

Figure 19 shows the execution time achieved for a different number of simultaneous requests (ranging from 2 to 64), and for a different number of concurrent daemons using each physical GPU. This figure shows three differentiated regions. For up to 8 simultaneous requests, the execution times for the plot corresponding to 2 daemons per GPU obtain the best performance. From over 8 and up to 16 requests, results with 4 concurrent daemons yield the shortest execution time. For the rest

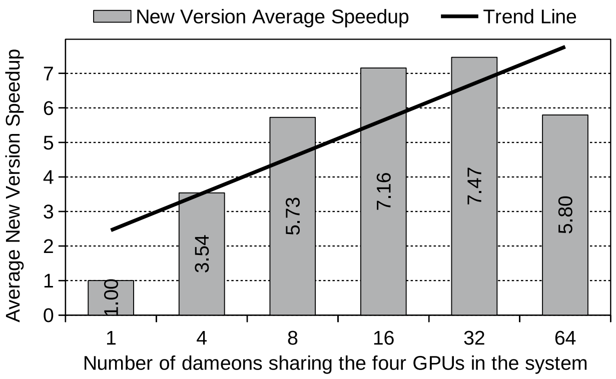

Fig. 21 Average speedup of the proposed architecture for the case of multiple GPUs and multiple daemons for each GPU

of higher numbers of simultaneous requests, the plot corresponding to 8 concurrent daemons provides the best results.

Figure 20 shows the speedup achieved for different numbers of simultaneous requests and concurrent daemons in each physical GPU. It should be noted that when the number of daemons is greater than the number of requests, the remaining daemons are idle. Again, for 8 or less simultaneous requests the best performance, on average, is achieved when there are 2 daemons per GPU. From over 8 and up to 16 requests, the highest speedup is achieved when there are 4 concurrent daemons for each GPU. For higher numbers of simultaneous requests, the highest speedup is achieved with 8 concurrent daemons. Thus, we can conclude that, for our experimental setup, the best number of concurrent daemons is 2 if the number of simultaneous requests is up to 8, 4 daemons per GPU from over 8 to up to 16, and 8 daemons if there are more than 16 requests.

Finally, Fig. 21 shows the average speedup achieved for a different number of daemons sharing each GPU. This figure shows that the average speedup increases proportionally to the number of daemons up to 32 concurrent daemons (7.47×). For more concurrent daemons, it decreases. To find out the reason for this behavior, in Fig. 22 we show the CPU and GPU utilization when running 64 concurrent requests with 4 GPUs and 8 daemons per GPU. As we can observe, in our experimental setup, the CPU required by the daemons is the limiting factor. The CPU required by the scheduler is very low, showing peaks of 2% when the amount of requests to schedule is high. The GPU utilization is also low, showing peaks of up to 25% at some points. In terms of memory, CPU memory required by the daemons has peaks of up to 49.90%, CPU memory required by the scheduler is very low (0.01%), and GPU memory is below 2.2%.

As a summary of all these results, we can state that the best performance depends on the number of simultaneous requests in the server. In the case of our experimental setup, the specific values are:

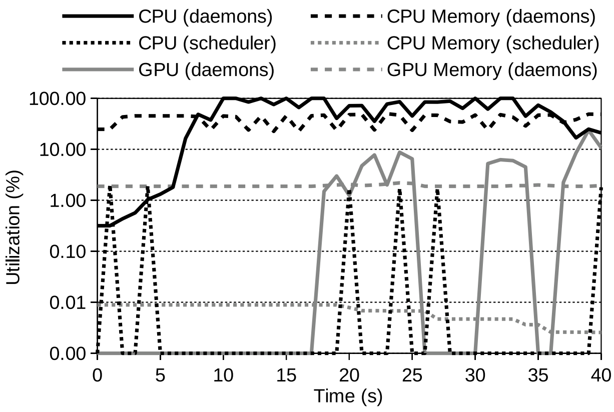

Fig. 22 CPU and GPU utilization when running 64 concurrent requests with 4 GPUs and 8 daemons per GPU. Note that *Y*-axis is in logarithmic scale

- Up to 8 requests: the best results are achieved with 4 GPUs and 2 daemons per GPU.
- Up to 16 requests: the best results are achieved with 4 GPUs and 4 daemons per GPU.
- More than 16 requests: the best results are achieved with 4 GPUs and 8 daemons per GPU.

Therefore, in our case we will configure the scheduler with those thresholds when using the `dynamic` scheduler mode previously detailed in Sect. 3.1 Thus, `daemons_low` will be set to 2 daemons per GPU, `daemons_mid` to 4 daemons per GPU, and `daemons_high` to 8 daemons per GPU. With this configuration, we will maximize the benefits of the proposed architecture.

5 Related work

In this section, we present related work on schedulers and their architecture. We compare our scheduler with popular ones in the literature. We also discuss the novelty of our architecture.

Slurm [16, 17] and PBS [18, 19] are probably one of the most used job schedulers in computing clusters. Both of them present similar features, mainly:

- Resource allocation. They allocate resources (i.e., compute servers or nodes) to users to run their jobs.
- Resource management. They provide a framework for running and monitoring jobs on allocated resources.
- Workload management. They manage the workload of clusters by using queues of jobs for the cluster resources.

In general, their architecture is also similar. Thus, each compute node has a daemon for running and monitoring jobs allocated on that resource. These daemons collaborate with a centralized daemon, which is responsible for resource allocation and workload management.

The parallel architecture proposed in this paper is inspired by these popular approaches. Thus, in our architecture a scheduler manages requests and forwards them to daemons, which serve those requests. In this way, our scheduler would be comparable to the centralized manager, and our daemons to the daemons on compute nodes.

However, note that our proposal presents some novel differences. First, job schedulers operate at inter-node level (i.e., the complete cluster), whereas our approach operates at intra-node level (i.e., one node of the cluster). Second, in our approach multiple daemons can share the same physical GPUs. On the contrary, job schedulers usually assign GPU resources in an exclusive way. Finally, our scheduler can be configured to automatically start, stop, and restart daemons depending on the number of user requests.

6 Conclusions

In this paper, we have proposed a new parallel architecture for improving the performance of a back-end server which processes user requests sent to a web-based tool for automatic DMR detection. In this architecture, there are multiple daemons simultaneously serving requests. In addition, daemons can share the same physical GPUs. A scheduler is responsible of managing requests and forwarding them to daemons for being served. The number of daemons per GPU is configurable, thus adapting to the specific computing capabilities of different back-end servers. Moreover, the new architecture dynamically adapts to the system load by starting or stopping daemons depending on the current number of requests. Results show that the parallel architecture is capable of fully exploiting the parallelism of the back-end server. The proposed architecture significantly reduces execution times compared to the previous architecture, where only a single daemon sequentially processed all the requests. Finally, in our experimental setup using four GPUs and 8 daemons per GPU, we have seen that the speedup increases up to 7.47×.

Acknowledgements This work is funded by Conselleria d'Educació, Universitats i Ocupació, Generalitat Valenciana (Spain) under grant CIGE/2021/132. The authors would also like to thank Enrique Bonet Esteban and Manolo Pérez Aixendri for their invaluable support in setting up and configuring the computing cluster used in the experiments. Last but not least, we would like to express our most sincere and best wishes to Elvira on her retirement.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work is funded by Conselleria d'Educació, Universitats i Ocupació, Generalitat Valenciana (Spain) under grant CIGE/2021/132.

Availability of data and materials The complete suite of tools used in this paper is available at GitHub [9].

References

1. Gallego-Bartolomé J (2020) DNA methylation in plants: mechanisms and tools for targeted manipulation. New Phytol 227(1):38–44. https://doi.org/10.1111/nph.16529
2. Chen Y (2019) Recent advances in methylation: a guide for selecting methylation reagents. Chem Eur J 25(14):3405–3439. https://doi.org/10.1002/chem.201803642
3. Schubeler D (2015) Function and information content of DNA methylation. Nature 517:321–326. https://doi.org/10.1038/nature14192
4. Li S, Chen M, Li Y, Tollefsbol TO (2019) Prenatal epigenetics diets play protective roles against environmental pollution. Clin Epigenetics 11:82. https://doi.org/10.1186/s13148-019-0659-4
5. Fulka H, Mrazek M, Tepla O, Fulka J (2004) DNA methylation pattern in human zygotes and developing embryos. Reproduction 128(6):703–708. https://doi.org/10.1530/rep.1.00217
6. Robertson K (2005) DNA methylation and human disease. Nat Rev Genet 6:597–610. https://doi.org/10.1038/nrg1655
7. Raciti A, Nigro C, Longo M, Parrillo L, Miele C, Formisano P, Béguino F (2014) Personalized medicine and type 2 diabetes: lesson from epigenetics. Epigenomics 6(2):229–238. https://doi.org/10.2217/epi.14.10
8. Shenoy N et al (2019) Ascorbic acid-induced TET activation mitigates adverse hydroxymethylcytosine loss in renal cell carcinoma. J Clin Investig 129(4):1612–1625. https://doi.org/10.1172/JCI98747
9. Networks and Virtual Environments Group (GREV), Universitat de València: HPG-Msuite, the methylation analysis ultimate tools suite (2020). https://grev-uv.github.io/
10. Tárraga J, Pérez M, Orduña JM, Duato J, Medina I, Dopazo J (2015) A parallel and sensitive software tool for methylation analysis on multicore platforms. Bioinformatics 31(19):3130. https://doi.org/10.1093/bioinformatics/btv357
11. Olanda R, Pérez M, Orduña JM, Tárraga J, Dopazo J (2017) A new parallel pipeline for DNA methylation analysis of long reads datasets. BMC Bioinform 18(1):161. https://doi.org/10.1186/s12859-017-1574-3
12. González C, Pérez M, Orduña JM (2019) HPG-HMapper: a DNA hydroxymethylation analysis tool. Int J High Perform Comput Appl. https://doi.org/10.1177/1094342019840792
13. Fernández L, Pérez M, Olanda R, Orduña JM, Marquez-Molins J (2020) HPG-DHunter: an ultra-fast, friendly tool for DMR detection and visualization. BMC Bioinform 21(1):287. https://doi.org/10.1186/s12859-020-03634-y
14. Networks and Virtual Environments Group (GREV), Universitat de València (2020) HPG-DHunter, a tool for detecting differentially methylated regions (DMRs) (2020). https://github.com/grev-uv/hpg-dhunter-batch
15. Fernández L, Olanda R, Pérez M, Orduña JM (2021) A web-based tool for automatic detection and visualization of DNA differentially methylated regions. Electronics. https://doi.org/10.3390/electronics10091083
16. Yoo AB, Jette MA, Grondona M (2003) SLURM: simple linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers. Lecture Notes in Computer Science, vol 2862. Springer, Berlin, pp 44–60. https://doi.org/10.1007/10968987_3
17. SchedMD LLC (2024) The Surm workload manager. https://slurm.schedmd.com/

18. Nitzberg B, Schopf JM, Jones JP (2004). In: Nabrzyski J, Schopf JM, Weglarz J (eds) PBS Pro: grid computing and scheduling attributes. Springer, Boston, pp 183–190. https://doi.org/10.1007/978-1-4615-0509-9_13
19. Altair Engineering Inc. (2024) PBS: portable batch system. https://www.openpbs.org/