GYAN: Accelerating Bioinformatics Tools in Galaxy with GPU-Aware Computation Mapping

Gulsum Gudukbay* , Jashwant Raj Gunasekaran*, Yilin Feng*, Mahmut T. Kandemir*,
Anton Nekrutenko*, Chita R. Das*, Paul Medvedev*, Björn Grüning†, Nate Coraor*, Nathan Roach‡, Enis Afgan‡
*The Pennsylvania State University, PA, USA† University of Freiburg, Germany,
‡ Galaxyworks, Johns Hopkins University, MD, USA

***Abstract*—Galaxy is an open-source web-based framework that is widely used for performing computational analyses in diverse application domains, such as genome assembly, computational chemistry, ecology, and epigenetics, to name a few. The current Galaxy software framework runs on several high-performance computing platforms such as on-premise clusters, public data centers, and national lab supercomputers. These infrastructures also provide support for state-of-the-art accelerators like Graphical Processing Units (GPUs). When coupled with accelerator support, the tools executing in Galaxy can benefit from massive performance gains in terms of computation time, thereby allowing a more robust computational analysis environment for researchers. Despite tools having GPU capabilities, the current Galaxy framework does not support GPUs, and thus prevents tools from taking advantage of the performance benefits offered by GPUs. We present and experimentally evaluate GYAN, a GPU-aware computation mapping and orchestration functionality implemented in Galaxy that allows the Galaxy tools to be executed on a GPU-enabled cluster. GYAN has the capability of identifying GPU-supported tools and scheduling them on single or multiple GPU nodes based on the availability in the cluster. GYAN supports both native and containerized tool execution. We performed extensive evaluations of the implementation using popular bio-engineering tools to demonstrate the benefits of using GPU technologies. For example, the Racon consensus tool executes ∼2× faster than the regular baseline CPU-only jobs, while the Bonito base calling tool shows ∼50× speedup.**

I. Introduction

Galaxy is a web-based open-source framework [1] widely used by thousands of researchers [21] for a variety of compute-intensive applications, including computational chemistry [13], genome assembly, epigenetics, metagenomics, machine learning, and drug discovery [34]. Galaxy can be installed on various compute platforms, such as local clusters, public datacenters, and national lab supercomputers [20], and is also available as a world-wide network of managed free services (known as *usegalaxy.**). As a result of its accessibility, Galaxy has been cited over 10,000 times in the last decade [22]. Galaxy allows users to access tools, manage workflows, reproduce, store and share experimental results with the community with these deployment options.

As evidenced by prior research [30], many of the tools that are used in Galaxy, have parallelization opportunities, potentially enabling substantial performance improvements when executed on hardware accelerators. An example is PyPaSWAS [39], which is a sequence alignment application that shows a 33× speedup with GPU compared to CPU. Within the broad spectrum of hardware-accelerators including GPUs, Field-Programmable-Gate-Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs), GPUs have been dominating the landscape due to their multi-faceted nature of supporting modern-day applications [30]. GPUs have become an essential part of modern computing systems with their use reaching far beyond their initial target domain (computer graphics) to many parallel application domains such as bioinformatics, nuclear physics, deep learning, and others [4], [6], [11], [24]. With the rapid increase in programmability as well as compute and storage capabilities of GPUs, there has been ongoing development of a growing set of applications and problems that have been mapped to GPUs. For instance, Argonne National Laboratory's researchers have accelerated a COVID-19 vaccine study that simulates an important part of a protein spike on coronavirus made up of 1.5 million atoms. By using the latest V100 GPUs, they were able to achieve 5× speedup compared to CPU-only execution [10].

With the proliferation of parallel bioinformatics applications/tools, deployment platforms like HPC supercomputers and public datacenters have begun to include GPUs as a part of their mainstream hardware infrastructure. For instance, the Brookhaven National Lab has expanded its supercomputers to include a 200-node GPU cluster [17]. Despite GPUs being available in today's hardware infrastructure, the current Galaxy framework does not support GPUs. This apparent deficiency in Galaxy motivates the central premise of our work: *can we make the Galaxy framework "GPU-aware" such that GPU-enabled tools can leverage the flexibility offered by executing in Galaxy for enhancing performance?*

However, it is non-trivial to integrate GPUs into the current Galaxy framework without affecting the user experience (i.e., users should be able to retain their original tool deployment method, while the tools should be able to leverage GPUs when applicable). Towards addressing this problem, in this paper, we present *GYAN*, an enhanced Galaxy framework with support for executing GPU-enabled tools. We achieve this through minimal code enhancements and user-agnostic modifications to the Galaxy framework, and we further test our modifications on an in-house Galaxy deployment.

This research is generously supported by NSF Award #1931531

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00037

In this paper, we make the following key contributions:

1) We make Galaxy GPU-aware such that the tools with GPU capability can seamlessly execute in Galaxy alongside CPU-enabled tools. We support both bare-metal (locally running) and containerized tools to be executed in Galaxy.
2) We design an intelligent GPU-aware orchestration policy such that a given tool can be executed on a CPU or a GPU based on the availability. Furthermore, we provide multi-GPU support that facilitates the spreading of highly compute-intensive tools across multiple GPUs. The GPU selection is designed based on the availability and utilization of all GPUs in a cluster.
3) We evaluate the effectiveness of the proposed GPU support in Galaxy using two widely used tools: *Racon* and *Bonito*. Racon is a genome consensus [37] tool and Bonito performs base calling [36].
4) The results from our experiments indicate that the GPU versions of these two tools show $\sim$2$\times$ (for Racon) and $\sim$50$\times$ (for Bonito) speedups over their original CPU-based counterparts.

The remainder of this paper is structured as follows. Section II presents the necessary background information on Galaxy, bioinformatics tools, GPUs, and containerization. We then explain, in Section III, the motivation along with the challenges in the design. The details of the implementation of the enhancements we made to the Galaxy framework to allow GPU-aware job mapping and orchestration are given in Section IV. The experimental results are presented and discussed in Section V, followed by the conclusions.

II. Background and Related Work

We present a brief background on Galaxy, containerization support, parallelism, and GPUs.

A. Galaxy Software Framework

Galaxy is an open-source web-based framework, which is maintained by a large, world-wide community. Galaxy enables thousands of researchers without informatics expertise to perform computational analyses [20]. Galaxy framework consists of two main components. The first is *The Galaxy Software Framework*, which is a web-based application for computational analysis. The framework interacts with the underlying computational infrastructure, which is hidden from the user. This infrastructure can be a conventional cluster, cloud, or a hybrid system that combines the two. The second component is called *usegalaxy.** and it is a set of hosted, free servers around the works that allow thousands of users to use a variety of tools. These servers are typically hosted on national cyberinfrastructure (e.g., Texas Advanced Computing Center (TACC) as part of the CyVerse project [20] in US, NeCTAR academic cloud in Australia).

The hosted Galaxy instances offer a set of popular, commonly-used tools. Tools are the applications that are run on Galaxy instances. These tools are used by an end-user, installed as a "Galaxy Admin", and developed by a tool-developer. When a user wants to execute a tool, it is submitted as a "Galaxy Job". A single job can be a single tool instance or a workflow consisting of a sequence of multiple tools. Galaxy tools have XML files which are called "tool configuration files" or "wrapper files" and these files are automatically rendered into the web user interface for the tool. The wrapper files create a bridge between the tools and Galaxy to inform Galaxy on how to execute the tool, what options to pass as parameters, and what output file(s) will be generated [23].

B. Containerization Support in Galaxy

Galaxy can also take advantage of containerized tools by launching jobs as "containers". A container is a "standard unit of software that packages up code and its dependencies" [18]. Containerization allows seamless reliable execution from one platform to another and makes end users' jobs easier by managing all the dependencies. Docker containers [18] are instantiated, at runtime, from Docker Container Images, which are lightweight and standalone packages that include the implementation, tools, libraries, and other settings necessary for that application to be executed independent of the operating system [18]. Singularity is a tool that works almost the same as Docker; however, it has different permission configurations which allow it to be executed on HPC clusters easily [28]. Galaxy currently supports both Docker and Singularity-containerized tools. Galaxy also supports "Biocontainers". Biocontainers [33] is an open-source project that helps manage bioinformatics packages for applications and allows to deploy them as containers. Biocontainers include Docker containers which are built from Dockerfile recipes and Conda based containers that first develop a Conda package and then build a Docker container from the package [33].

C. Parallelism and GPUs

GPUs are designed to be powerful engines for computationally demanding applications. They deliver a great performance for many types of parallel computations. A GPU is a highly-parallel-programmable processor with large arithmetic capabilities and memory bandwidth, therefore it is not only meant for graphics purposes. So for many parallel applications, it yields high performance advantages over its CPU counterpart. Further, with parallelism and throughput being increasingly more important than latency in many application domains, GPU architectures have developed substantially over the years. Especially the recent NVIDIA GPUs can achieve thousands of GFLOPS (giga-floating point operations per second) [30].

In an NVIDIA GPU architecture, an application code is parallelized by using CUDA parallel programming model [15], which maps threads, blocks, grids and warps to the GPU architecture. In this model, GPUs are referred to as "devices" and CPU is referred to as "host". The functions that run in GPUs are called "device kernels". In a device kernel, the threadIdx gives the ID of the current thread, blockIdx gives the ID of the current block, blockDim gives the size of each dimension of the current block, and lastly, gridDim gives the

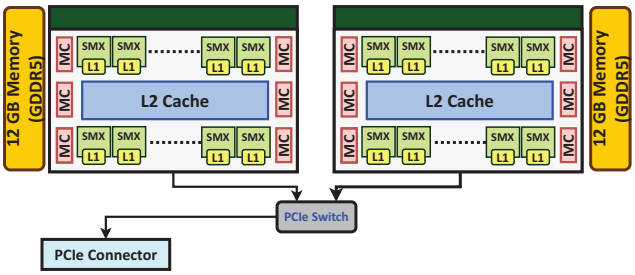


Figure 1. Tesla K80 GPU architecture.

size of each dimension of the current grid [15]. GPUs can run many device kernels in parallel, and each device kernel accesses a grid to access threads and blocks. The number of grids, blocks per grid, and threads per block depends on the GPU's compute capability. Threads are organized in 1, 2, or 3-dimensional blocks, and similarly, blocks are organized in 1, 2, or 3-dimensional grids [15]. Within a block of threads, the threads are executed in groups of 32, which are named as "warps" and all threads in a warp execute the same thing [7]. Each thread in a warp accesses a shared memory.

NVIDIA GPUs contain Streaming Multiprocessors (SMs) and each SM contains Streaming Processors (SPs) or CUDA cores. SMs execute the device kernels in block(s) (therefore several warps) one after another. In the current GPU architectures, each SM has several warp schedulers [25]. This shows the dynamic scheduling nature of these GPUs, which allows scalability. So higher number of blocks used in a device kernel allows better scaling across any GPU architecture.

To evaluate GYAN, we used a machine with two Tesla K80 GPUs. The Tesla K80 GPU has two Tesla GK210 GPUs as seen in Fig. 1. Both GPUs have 2,496 processor cores with a core clock of 560 MHz to 875 MHz; the memory bandwidth is 480 GB/sec, and the total board memory is 24 GB [26]. In this GPU, the number of threads per warp is 32; the maximum number of threads per block is 2048; and the maximum number of warps per SM is 64. There are 15 SMs, each containing 4 warp schedulers, allowing 4 warps to be executed simultaneously. The SMs in this GPU have improved performance for double-precision workloads [25].

D. Related Work

We briefly describe the infrastructure and software enhancements to Galaxy and bioinformatics tools over the years. On the infrastructure front, cloud computing can offer on-demand access to elastic computational infrastructure, however, it is not available for "as is" usage for biologists. "Galaxy CloudMan" was developed [2] to allow researchers to manage an arbitrarily sized compute cluster on Amazon EC2. This system does not require informatics knowledge as it allows the creation of configured compute cluster within five minutes and it makes the entire biological tools suite available for immediate usage.

Besides the cloud computing support, there have been several other application-level enhancements for the Galaxy Framework. One of which is related to expanding Galaxy's reference data [38]. For many bioinformatics analyses, the proper management of reference datasets is an important task. Refgenie [38] is a reference management system that enables this task and it is integrated into the Galaxy platform with a graphical user interface. Similarly, Galaxy was recently enhanced with another platform related to long-read sequencing, which has become popular, allows sequencing long contigs at low cost and minimal preparation. "NanoGalaxy" [5] was developed and it is a freely available Galaxy-based toolkit for analyzing long-read sequencing data. PyPasWAS is a Python-based multi-core GPU and CPU sequence alignment tool. While PyPasWAS achieves 33× speedup in execution time using GPUs for alignment, it is not a platform for running different sequence alignment tools and it does not provide infrastructure nor an interface with Galaxy. Another recent work [16] designed infrastructure for NGS analyses using Galaxy with GPU support. While providing limited implementation details, this work is an in-house framework that uses Galaxy as a middleware to run jobs along with the Slurm scheduler. It is not integrated into the main Galaxy repository.

III. Motivation

While the current implementation of Galaxy does not allow GPU-enabled tools to run, prior research demonstrates that the GPU versions of many tools currently running in Galaxy can potentially achieve significant speedups (compared to their CPU versions). The speedups for a few life sciences applications are as follows: Direct Coulomb Summation ∼45× [30]; Cutoff Pair Potentials application ∼17× [35]; Fluorescence Microphotolysis ∼11× [3]; and Multi-Level Summation Method Short-Range application ∼25× [9].

As seen in these examples, using GPUs can improve performance in the important research areas for human life, which shaped the accelerator development in recent years. At Argonne National Laboratory, researchers study a COVID-19 vaccine, where a 24 DGX A100 system cluster empowers them to accelerate the simulations, enabling a faster understanding of how this virus infects humans [10]. Furthermore, The National Energy Research Scientific Computing Center uses A100 for AI-based simulations [10]. They had speedups up to 5× (V100 GPU vs. CPU) in different areas, and they expect more gains with A100 [10]. These results show that these use cases and tools are highly parallelizable and can yield significant performance improvements when coupled with GPUs. Further, these tools are often embarrassingly parallel, allowing them to scale across multiple GPUs.

Scientific Impact: Galaxy is used by thousands of researchers who significantly contribute to various important domains, where they execute hundreds of thousands of both compute and data-intensive experiments. With GPU infrastructure support, these experiments will benefit from massive speedups due to the inherent parallelism they offer. This advantage unilaterally motivates the need for enabling GPU-aware tool execution in the Galaxy framework.

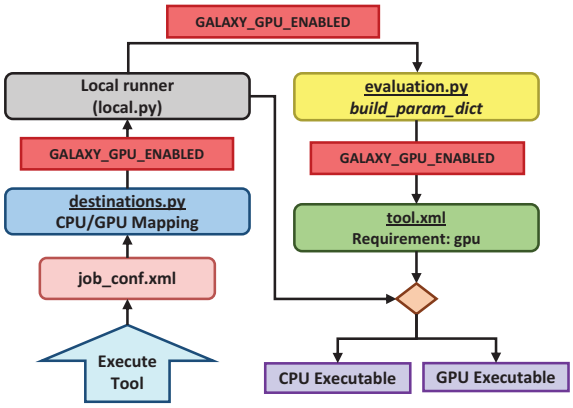


Figure 2. The system flow diagram for Galaxy tool execution.

A. Challenges in Bringing GPU-Support

To design and develop the GPU support, we first investigate the four main steps involved with Galaxy's tool execution flow. As seen in Fig. 2, first, users trigger a job submission through the Galaxy web-interface. Galaxy parses the tool requirements from the wrapper file (which includes references to the required libraries and hardware). Second, Galaxy executes the submitted job via a *runner*, which maps the job to a destination using a job configuration file. Third, Galaxy submits the job to a job scheduler, or executes it locally as a dedicated process. Finally, Galaxy collects the results from the job execution and presents them to the user via the web-interface. While inspecting this tool execution workflow, we identified four critical challenges about the first two steps in the workflow.

The first challenge, *Challenge-I*, is to include a new compute requirement for GPU (in the form of XML tags) in the tool configuration file. The current Galaxy implementation does not provide explicit hardware requirement specification tags. Hence, it is non-trivial to include a new hardware specification while retaining the original Galaxy execution flow. The second challenge, *Challenge-II*, lies in exposing the GPU availability to the Galaxy runner. The runner makes use of dynamic destination mapping to map jobs into physical hosts (destinations). The runner needs to extract information from the new GPU requirement, such as GPU availability, GPU model, etc., to dynamically map GPU jobs alongside CPU jobs. Additionally, if GPUs are unavailable, the runner needs to switch jobs to CPU nodes in a user-agnostic fashion.

Since we need to support both bare-metal and containerized tools, *Challenge-III*, is to enable GPU-support for containerized tools. Although Galaxy supports launching tools as Docker containers, it does not support NVIDIA Docker-based GPU containers. To enable this support, the primary challenges lie in defining a new compute requirement (in *Challenge-I*) as a part of the existing container launch script.

The final challenge *Challenge-IV*, concerns the design of multi-GPU-aware computation mapping support. To efficiently enable multi-GPU support, we need the following: (i) allow the end-user to specify the IDs of GPUs as a requirement; (ii) obtain the real-time GPU information such as GPU IDs, the number of executing processes, and memory usage and (iii) design a GPU device allocation strategy without (or minimally) affecting the currently running processes. These four challenges highlight the design complexities involved in enabling GPU support in Galaxy.

IV. Design and Implementation of GYAN

To address the challenges discussed in Section III-A, we design and implement GYAN – a GPU-aware computation mapping support for Galaxy. While retaining the original execution flow of Galaxy, the essential features of GYAN are (i) minimal to no user involvement, (ii) easily extensible code enhancements to Galaxy's core framework, (iii) minimal overheads for cluster administrators, and (iv) under-the-hood automated decision-making process. We further explain in detail below, the individual design components of GYAN.

A. GPU-Aware Computation Mapping

```
1 <macros>
2   <xml name="requirements">
3     <requirements>
4       <requirement type="package" version="@VERSION@
      ">racon</requirement>
5       <requirement type="compute">gpu</requirement>
6     </requirements>
7   </xml>
8   ...
9 </macros>
```

Code 1. The macros.xml file. It specifies the requirements of the tool and is imported into the racon.xml file. The requirement of type "gpu" is at line 5, allowing Galaxy to recognize that the tool requires GPU to be executed.

As mentioned in the previous section, *Challenge-I* is the design of a new compute requirement of type GPU in the form of XML tags. To overcome this challenge, we designed and implemented a new *parser* which interprets the new requirement type. The new requirement type is to be used in the tool wrapper file as shown in Code 1. The new requirement type is utilized when deciding if GPU is required in the upcoming steps of the Galaxy execution flow. The values of the compute requirement type can be "gpu" or "cpu" (default).

```
1 <job_conf>
2   <plugins>
3     ...
4     <plugin id="dynamic" type="runner">
5       <param id="rules_module">galaxy.jobs.rules</
      param>
6     </plugin>
7   </plugins>
8   <destinations default="gpu_cpu_decision">
9     <destination id="gpu_cpu-decision" runner="
      dynamic">
10      <param id="type">python</param>
11      <param id="function">dynamic_map</param>
12    </destination>
13  </destinations>
14 </job_conf>
```

Code 2. The "job_conf.xml" configuration file which specifies the dynamic job destination decision making dynamic_destination.py script as the runner.

Recall that *Challenge-II* is about exposing the GPU availability to the Galaxy runner. We added a new job rule to address this challenge, which allows us to dynamically map between CPU or GPU destinations according to different conditions. The job rule obtains the system GPU availability and the

number of GPUs using the "pynvml" Python library. If the tool's wrapper file has the compute requirement of type "gpu" and if there is at least one GPU available, then the destination is configured to be "local GPU". At the same time, a boolean environment variable called "GALAXY_GPU_ENABLED" is introduced to Galaxy; it is set to "true" if the "local GPU" destination is configured, and "false" otherwise. The Galaxy administrators configure the job execution destinations by using the "job_conf.xml" file. The new job rule is used in the "job_conf.xml" file as a "destination" (shown in Code 2).

In the Galaxy framework, the backend Python variables are exposed to the tool developer with the dictionary data structure, which is the output of the "build_param_dict" function. This function resides in the "evaluation.py" script and serves as a bridge between the Galaxy backend and the tool developer. Using this information, we exposed the "GALAXY_GPU_ENABLED" environment variable to the tool wrapper file with the insertion of a dictionary entry. Hence, we assign the "GALAXY_GPU_ENABLED" value to "__galaxy_gpu_enabled__" in the tool-config file.

```
1  <tool id="racon" name="Racon" version="@VERSION@">
2    ...
3    <command detect_errors="exit_code"><![CDATA[
4      ...
5      #if $__galaxy_gpu_enabled__=="true":
6        racon_gpu
7      #else:
8        racon_cpu
9      #end if#
10     ...
11   ]]></command>
12 </tool>
```

Code 3. The "racon.xml" wrapper file for the Racon tool. The wrapper files allow users to specify the executable and include the parameters that the end-user can set. This file is necessary for Galaxy to recognize the tool. The "GALAXY_GPU_ENABLED" environment variable is accessed via the parameter dictionary entry "galaxy_gpu_enabled", as shown in line 5.

Code 3 shows how the tool wrapper file utilizes the "__galaxy_gpu_enabled__" key from the parameter dictionary. The tool wrapper checks the value of "__galaxy_gpu_enabled__" and decides on which executable to use. By default, in CUDA programming, if the tool does not specify any GPU device preference, all the GPUs are made available. However, if the tool has a specific GPU preference within the requirements XML, then that GPU is used. In case that the GPU is busy, the tool will be offloaded to another GPU device based on availability.

B. GPU-Awareness for Containerized Tools

Challenge-III is about enabling GPU support for containerized tools. Although Galaxy supports launching tools as containers and a tool's container has support for GPU, Galaxy does not launch the containers with GPU support. To overcome this challenge, we need to modify the container launch script to utilize the GPU compute requirement that the tool developer specifies using the wrapper file.

In the original Galaxy framework implementation, when the job_conf.xml file has the "docker_enabled" parameter set to "true" [19], the Docker runner takes effect. Next, the Galaxy container launching script reads the required container ID from the tool wrapper file and pulls the container from the docker-hub or bioconda. Subsequently, the script executes the container by assembling a bash command. While assembling this command, GPU support for both Docker and Singularity-containerized tools can be added using an additional flag. The machine or cluster that hosts Galaxy should have NVIDIA-Docker installed so that the user driver components and the GPU devices into the container are mounted to the container at launch.

To add the GPU support, we must first obtain information about GPU availability and the tool's GPU requirement. To this end, we use a similar approach to the one described in Section IV-A. The destination to execute the tool on is changed to "docker" destination in the job_conf.xml configuration file. If there is no GPU available, the NVIDIA-Docker library will not work. Therefore, when we are adding the new GPU flag with the `command_part.append("--gpus all")` line to the Docker run command, we first check the environment variable that is set according to the GPU availability and requirement using the statement `if os.environ['GALAXY_GPU_ENABLED'] == "true"`.

We added GPU support to Singularity-containerized tool execution similar to the GPU support design for Docker-containerized tool execution. If the "GALAXY_GPU_ENABLED" environment variable value is "true", the GPU support is added to the Singularity container launch command using the `command_part.append("--nv")` statement. Theoretically, this addition should suffice for GPU support for Singularity containers to work. However, in the current Galaxy framework implementation, when volumes are mounted to the Singularity image, the "rw" and "ro" flags are given for the read-write or read-only permissions. These two flags are removed with the GYAN enhancements, because Singularity's new version (Version 3.1) does not support these flags when adding the GPU flag.

C. Multi-GPU-Aware Computation Mapping

In addition to the single GPU support, we also provide a multi-GPU computation mapping support, which executes a given tool using multiple GPUs, provided that there is more than one GPU available on the host. To this end, the first part of *Challenge-IV* is to enable the end-users to specify the IDs of GPUs for tools as requirements in the wrapper files. The non-trivial challenge here is with allowing the end-user to specify the GPU ID along with the already-defined GPU compute requirement. To solve this problem, we used the existing "version" XML tag of the tool wrapper file's requirement object. Therefore, the "version" tag corresponds to the GPU minor ID(s) in our design. The second part of this challenge involves obtaining the real-time GPU information such as GPU IDs, executing processes, and memory usage for each GPU. To solve this part of the challenge, we propose an algorithmic design that determines the processes executing on each GPU. This information is obtained by executing a GPU query command from the Python local runner script ("local.py"). This command uses the "nvidia-smi" query and then returns the output as XML. The "BeautifulSoup" library

is used to process the XML output to extract the GPU IDs and PIDs of the processes executing on each GPU. As shown in Pseudocode 1, a dictionary with the key-value pairs is created by processing the output, where keys are GPU minor IDs and values are process IDs.

Pseudocode 1: The "get_gpu_usage" function which resides in the "local.py" script. This functions captures the executing processes for each device and returns a list of available GPUs and all GPUs in the system.

```
Input: None
Output: avail_gpus, all_gpus
proc_gpu_dict = {};
avail_gpus = [];
all_gpus = [];
bash_cmd = "/bin/bash -c 'nvidia-smi –query -x'";
out, err = subprocess.Popen(...);
soup = bs(out, "lxml");
gpu_find = soup.find("nvidia_smi_log").find_all("gpu");
process_find = p.find("processes").find_all("process_info");
for ( p in gpu_find ) {
    minor_id = p.find("minor_number");
    for ( proc in process_find ) {
        pid_proc = proc.find("pid");
        proc_gpu_dict[minor_id].append(pid_proc);
    }
}
for ( x,y in proc_gpu_dict ) {
    all_gpus.append(x);
    if y is empty then
        avail_gpus.append(x);
}
```

The next part of *Challenge-IV* is to design a GPU device allocation strategy without affecting the currently executing processes. To solve this challenge, we introduce two methodologies explained below.

1) Process ID Approach: The "__command_line" function, located in the "local.py" script, assembles a command to execute the tool submitted by the end-user and launches the command as a sub-process. The "get_gpu_usage" function is called in the "__command_line" function. It returns the lists of available GPUs along with all of the GPU IDs on the host machine. If the list of available GPUs contains the required tool GPU IDs, the value of the "CUDA_VISIBLE_DEVICES" environment variable is set to those GPU IDs. Next, it is exported to the local runner as shown in Pseudocode 2.

Like the earlier approach, the GPU support for containerized tools with multi-GPU support is developed by exposing GPU information to containers. Note that, we have not used the "–gpus x" command which is meant to expose the desired GPUs because it did not work as intended. Instead, we have exported the "CUDA_VISIBLE_DEVICES" environment variable according to the GPU availability. Then, we have used the "–gpus all" flag to obtain all of the GPU IDs that the environment variable exposed. These GPU IDs are determined with the algorithm shown in Pseudocode 2. This flag is necessary for the Docker runtime to support the GPU-enabled containerized tools.

2) Process Allocated Memory Approach: The "Process ID Approach" is not efficient for some scenarios. For example, if all GPUs are executing at least one process and if an incoming task is distributed to all GPUs, some GPUs can have very high memory utilization. This situation may cause stalling due to context switching between tasks. Instead, with the "Process Allocated Memory Approach," we place the upcoming job on a GPU which has the least device memory allocated for the executing process(es). This approach uses the "nvidia-smi" call as mentioned in Section IV-C1. Instead of obtaining the process IDs from this query, we get the "fb_memory_usage.used" of each GPU. We then expose the GPU ID which has the minimum memory usage, to the upcoming job.

Pseudocode 2: The "__command_line" function which resides in the "local.py" script.

```
input : self, job_wrapper
output: CUDA_VISIBLE_DEVICES
if job_wrapper.tool exists then
    ...
    for ( req in reqmnts ) {
        if req.type = "compute" and req.name = "gpu" then
            if req.version and req.version != "" then
                gpu_id_to_query = req.version;
            flag = 1;
    }
    if gpu_flag and gpu_count > 0 and flag then
        GALAXY_GPU_ENABLED = "true";
    avail_gps, all_gps = get_gpu_usage();
    for ( dev in all_gps ) {
        all_gps_str += dev;
    }
    if gpu_id_to_query in avail_gps then
        gpu_dev_to_exec = gpu_id_to_query;
    else
        gpu_dev_to_exec = "";
        for ( dev in avail_gps ) {
            gpu_dev_to_exec += dev;
        }
    CUDA_VISIBLE_DEVICES = gpu_dev_to_exec;
```

3) GPU Hardware Usage Script: To evaluate the design choices and enhancements mentioned in the previous sections, we implemented a GPU hardware usage script, which allowed us to monitor GPU utilization and GPU memory utilization chronologically throughout the tool executions. This script is embedded inside the Galaxy Framework implementation and is executed when a tool execution starts. It is essential for understanding the tool characteristics.

V. Evaluation Framework

In this section, we discuss the evaluation of GYAN using the experiments conducted in Galaxy hosting on a Chameleon Cloud test-bed, which has GPU nodes. We compared runtimes of CPU-only executions and GPU-supported executions along with the multi-GPU enhanced Galaxy executions for two tools. With the use of GYAN, running GPU-supported tools on Galaxy does not introduce any extra overhead, because GYAN executes and schedules jobs to GPUs without adding another layer of software stack. Therefore, to evaluate GYAN, we showed how GPU-supported tools have speedup over CPU-only versions to motivate the need for GPU computation mapping for Galaxy. These speedups can increase the the

number of compute and data-intensive experiments, leading to increased research bandwidth.

A. Workloads

We focus on two major workloads for testing GYAN: *Racon* and *Bonito*. These are popular tools for processing next-generation sequencing data [12] from the two most popular long-read technologies – PacBio [27] and Oxford Nanopore Technologies [31]. We chose them because they capture a broad range of sequencing data analysis, covering multiple platforms and multiple stages of the data processing pipeline and they are highly amenable to GPU-based parallelization.

Sequencing data typically goes through a processing pipeline before it can lead to biological insight. One of the earliest steps in the pipeline is "basecalling" [14]. A basecaller is an algorithm that converts sequencing data from its raw form, which captures the complex combination of optical sensors, hardware, and chemistry underlying the technology, into a sequence of individual nucleotides. Oxford Nanopore Technologies provides a PyTorch-based basecaller for its data, called Bonito [36]. Bonito is inspired by the usage of convolutional neural networks (CNNs) in speech recognition. It has several functionalities, like training a bonito model (*bonito train*), converting an hdf5 training file into a bonito format (*bonito convert*), evaluating a model performance (*bonito evaluate*), downloading pre-trained models and training datasets (*bonito download*), and basecaller which obtains a fasta format output from .fast5 files (*bonito basecaller*). Bonito has both GPU and CPU execution support. It also has automatic mixed-precision support for accelerating the training tool [36].

Basecalled reads are often used to perform a de novo assembly. An assembler outputs long reference sequences for shorter read segments as it predicts sources of these reads. The assembler first constructs a draft backbone sequence of the reference. It then aligns the reads to that backbone and corrects each position in the backbone according to the consensus of the nucleotides that align to it. Racon performs this step for PacBio sequencing data. While this is a compute-intensive process, it leads to significantly better quality assemblies [37]. Racon uses the mapping data to construct a partial-order alignment with single-instruction, multiple-data (SIMD) support to accelerate the consensus generation an order of magnitude faster than state-of-the-art methods [37].

B. Experimental Setup

We used a machine with an Intel Xeon E5-2670 processor with 48 CPUs and two NVIDIA Tesla K80 GPUs to conduct our experiments. The GPU driver version is 455.45.01, and the CUDA version is 10.2. The Python version we used to execute Galaxy is Python 3.6.9. We created different experiments to analyze all of the contributions. The first set of experiments execute the Racon tool with the local runner and compare the performance with different parameters and the GPU vs. CPU execution. Next, we created a Docker image for the Racon-GPU tool and used that to compare the execution times of runs with different parameters and GPU vs. CPU. Lastly, to evaluate the multi-GPU functionality, we created different cases to test the multi-GPU process scheduling.

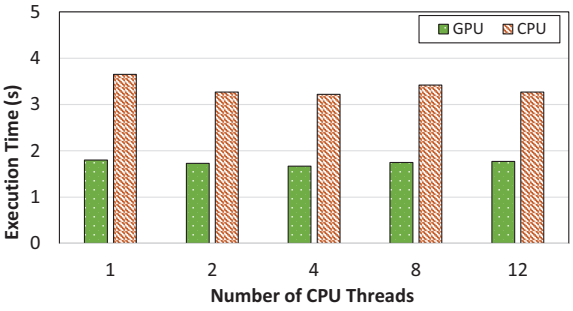


Figure 3. Performance across different thread numbers for the Racon tool comparing the GPU and CPU-only versions.

C. GPU Hardware Usage Script

We implemented a script that allows us to collect statistics about the jobs. This script obtains the GPU utilization, GPU memory utilization, and PCIe link generation information for every second, including minima, maxima, and average. It is executed when a job is submitted and stopped when a job is either killed or stops. Whenever it stops, a post-processing function is executed, and it generates .csv files and other log and statistic files that are aggregated from the chronological data for each job. The GPU query used for this script is shown in Code 4. This script captures the increasing/decreasing trends of the GPU memory usage and SM utilization of the executing tool. It demonstrates how beneficial GPU usage is and yields the design of multi-GPU computation mapping support, which is explained in Section IV-C.

```
1 bash_command = "/bin/bash -c 'nvidia-smi 00query-gpu
      =utilization.gpu, utilization.memory, memory.
      total, memory.free, memory.used, pcie.link.gen.
      max, pcie.link.gen.current --format=csv -l 1'"
2 sp = subprocess.Popen(bash_command, shell=True,
      stdout=File_object, stderr=subprocess.PIPE).pid
```

Code 4. GPU metric query for obtaining hardware usage metrics. This code snippet is added to the "queue_job" function of the "local.py" runner script.

VI. Results and Analysis

A. GPU-Aware Computation Mapping

To test the impact of GYAN, we ran the Racon-GPU tool on Galaxy. We used a 17 GB Alzheimers NFL Dataset, which contains the polished sequencing results for the Alzheimer human brain transcriptome [32]. After experimenting with different batch sizes and CPU thread numbers with the Racon tool, the best performance configuration was 4 threads and 1 batch without banding approximation with 1.72s. Among the experiments that use banding approximation, 4 threads and 16 batches performed the best with 1.67s. The CPU-only execution using 4 threads took 3.22 seconds, nearly 2× slower when compared to GPU execution, as seen in Fig. 3.

To understand the speedup reasons and the nature of the Racon-GPU tool, we used the GPU hardware usage script shown in Section V-C. We found that the Racon-GPU tool does

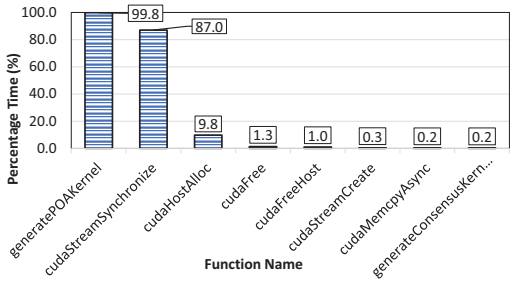


Figure 4. Hotspot functions obtained from NVProf analysis on Racon-GPU.

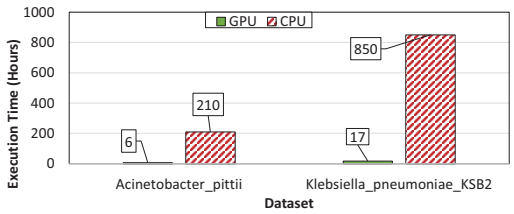


Figure 5. Bonito CPU vs. GPU execution times for two datasets.

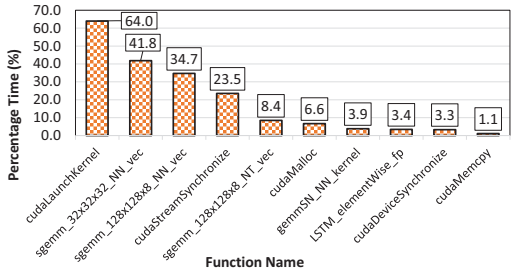


Figure 6. Bonito hotspot functions obtained by doing NVProf analysis.

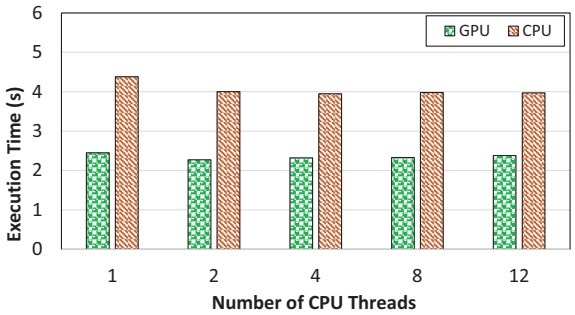


Figure 7. Performance across different thread numbers and batch sizes for Racon-GPU with banding approximation executed on Docker containers.

the "polishing" portion of the consensus generation in GPU. While the CPU-only polishing execution takes 117 seconds, using GPU, this execution time is reduced to 15 seconds (2s for GPU memory allocation + 13s of GPU polishing + 0.0001s of additional CPU polishing for the remaining portion of the reads that could not be polished in GPU). Thus, the CPU end-to-end execution takes ∼410s for the Isoseq NFL Alzheimers dataset, and the GPU end-to-end execution takes ∼200s. So, even though polishing time is reduced from 117 to 13 s, there is an overhead of ∼40s due to CUDA API calls to transfer input data and results from and to GPU for kernel computation of polishing and CUDA kernel synchronization.

To see the hotspots and understand how much the API calls affect the performance, we performed NVProf analysis on the running job. As plotted in Fig. 4, the majority of the calls are kernel synchronization calls, memory transfer API calls (which send the 17 GB dataset in chunks that fit in GPU memory to device and back to host), and lastly, ClaraGenomics library kernel calls, which are "generatePOAKernel" and "generateConsensusKernel". To understand the bottlenecks, we did an NVProf stall analysis on Racon and found that there is ∼70% memory dependency stall and ∼20% execution dependency stall, which are also reasons why we cannot get further performance improvements.

We also experimented with the Bonito Basecalling tool (pip package version 0.3.2) using Acinetobacter_pittii (1.5 GB) and Klebsiella_pneumoniae_KSB2 (5.2 GB) datasets [29]. The GPU support reduced the execution time significantly, as can be seen in Fig. 5. In Fig. 5, the execution times for CPU are shown as approximate results, because the CPU execution time for the smaller dataset (Acinetobacter_pittii) lasted more than 210 hours, and the larger dataset is approximated to last 4× longer than the smaller dataset (more than 850 hours). Therefore, the speedup for GPU vs. CPU execution time is more than 50× for the Bonito Basecaller tool. We did NVProf analysis for the Bonito Basecaller tool. The main hotspot functions were found to be CUDA kernel launcher, kernel synchronizer functions, and GEneral Matrix to Matrix Multiplication (GEMM) functions, which are a critical part of neural networks (Bonito Basecaller uses a pre-trained network).

B. GPU-Awareness for Containerized Tools

We used the Racon-GPU tool and the Alzheimers NFL Dataset for testing the GPU support for containerized tools. We experimented with the same parameters and settings to compare the bare-metal and containerized version of Racon to infer the container launching overhead and the speedup between the CPU and GPU-aware containerized execution of Racon. We created a Racon-GPU Docker container that can

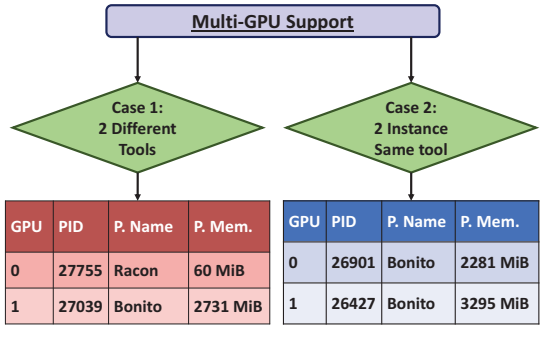
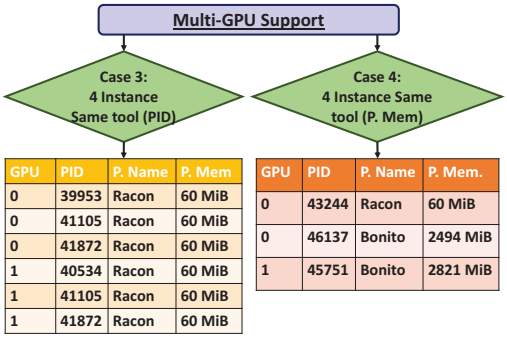


Figure 8. Multi-GPU support Cases 1 and 2.



GPU	PID	P. Name	P. Mem
0	39953	Racon	60 MiB
0	41105	Racon	60 MiB
0	41872	Racon	60 MiB
1	40534	Racon	60 MiB
1	41105	Racon	60 MiB
1	41872	Racon	60 MiB

GPU	PID	P. Name	P. Mem.
0	43244	Racon	60 MiB
0	46137	Bonito	2494 MiB
1	45751	Bonito	2821 MiB

Figure 9. Multi-GPU support Cases 3 and 4.

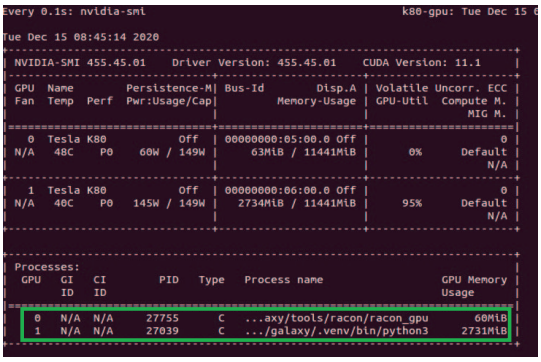


Figure 10. Multi-GPU support Case 1 console output.

be accessed online [8], or can be pulled using the "docker pull gulsumgudukbay/racon_dockerfile" command. We used this container as a requirement in the tool wrapper. For the Racon experiments that did not use banding approximation, the best performance configuration was with 2 CPU threads and 4 batches for accelerated GPU polishing. For the experiments that used banding approximation, the best configuration was using 2 CPU threads and 8 batches for accelerated GPU polishing, as seen in Fig. 7. We compared the two best-performing configuration performance numbers and calculated that approximately 0.6s (36%) of the time was spent on container launching and cold start overhead.

C. Multi-GPU-Aware Computation Mapping

To evaluate the multi-GPU-aware computation mapping, we used both Racon-GPU and Bonito Basecaller tools and executed different experiments with them on Galaxy.

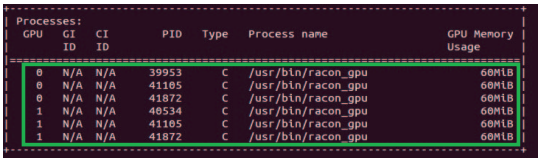


Figure 11. Multi-GPU support Case 3 console output.

1) Case 1: Two Different Tools: This experiment involved testing the most basic functionality of multi-GPU-aware computation mapping, which is checking whether two jobs will be scheduled to their own required devices. If a user specified Device 0 as a compute requirement of the Racon tool and Device 1 as a compute requirement of Bonito tool, the experiment checked if the jobs were scheduled to their correct GPUs. Fig. 8 shows this case under Case 1 along with the "nvidia-smi" query output of the experiment, which shows the processes and the GPUs allocated to them. Fig. 10 shows the console output for this case. Racon process runs on GPU 0, and the Bonito process runs on GPU 1, which indicated that the basic functionality is working as intended. Therefore, we can conclude that two different tools can be executed in parallel in separate GPUs without performance degradation, running in their original execution times.

2) Case 2: Two Instance of the Same Tool: This experiment involved scheduling a second instance of the same tool to another GPU if the first required GPU is busy with an instance of the tool. If Bonito's required GPU ID is 1, the instance should be scheduled in Device 1. If another instance of Bonito is started, with the same GPU ID 1, it will be scheduled to a GPU that is not busy (GPU 0) as seen in Fig. 8 Case 2.

3) Case 3: Four instance of the Same Tool GPU Allocation Using PID: This experiment involved executing more than 2 instances of the containerized Racon-GPU tool (showing that both "PID GPU allocation" and "multi-GPU support for containerized tools" are working). As can be observed from Fig. 9 and the console output in Fig. 11, the scheduling works as intended: the first Racon instance (PID 39953) is scheduled to GPU 0, the second (PID 40534) to GPU 1, and then, since both GPUs are busy, the upcoming processes with PIDs 41105 and 41872 are scattered to both GPUs.

4) Case 4: Four instance of the Same Tool GPU Allocation Using Process Memory: This experiment involved executing instances of both Racon and Bonito tools and then another instance of Bonito. Fig. 9 Case 4 shows that Racon (PID 43244) is scheduled to GPU 0, Bonito (PID 45751) is scheduled to GPU 1, and the second instance of Bonito (PID 46137) is scheduled to GPU 0. This is because, at the time that the user executes a the second instance of Bonito, the GPU with minimum memory usage was GPU 0 (with 60 MiB usage). This case is a better approach than the previous one because it allows more efficient device allocation than distributing the 3rd process to all GPUs and introducing multi-GPU overhead for tools that do not have multi-GPU support.

VII. Conclusion

The explosion of parallel bioinformatics applications has driven deployment platforms like HPC supercomputers and public data centers to embrace GPUs as a part of their hardware ecosystem. Nevertheless, the most popular bioinformatics application framework, Galaxy, does not support GPU-based applications. To fill this void, we design and test GYAN, an enhanced version of Galaxy with GPU-support, which allows the GPU-capable tools to execute in Galaxy. More specifically,

we added an intelligent GPU-aware computation mapping and orchestration support to Galaxy, for researchers to execute the tools in both CPU (or) GPU based on the tool requirements. Furthermore, the GPU-aware computation mapping was extended to multi-GPU application mapping and orchestration. Furthermore, we also enabled GPU containerization support for both Docker and Singularity containers. We performed experiments using two tools. Our experiments revealed that the GPU support through GYAN leads to $\sim 2\times$ improvement in performance using the Racon-GPU tool with a 17 GB dataset, and $\sim 50\times$ improvement for the Bonito base calling tool. Also, the intelligent multi-GPU-aware computation mapping allowed us to efficiently allocate GPUs to jobs according to the states/occupancy of each GPU and execute many instances of different tools at the same time without degradation in performance. GYAN's source code will be open-sourced and it will soon be merged to the public Galaxy's repository.

ACKNOWLEDGMENT

This research was supported by NSF award #1931531. We thank NSF Chameleon Cloud project CH-819640 for their generous compute grant.

REFERENCES

[1] Afgan, E., Baker, D., Batut, B., van den Beek, M., Bouvier, D., Čech, M., Chilton, J., Clements, D., Coraor, N., Grüning, B.A., Guerler, A., Hillman-Jackson, J., Hiltemann, S., Jalili, V., Rasche, H., Soranzo, N., Goecks, J., Taylor, J., Nekrutenko, A., Blankenberg, D.: The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. Nucleic Acids Research **46**(W1), W537–W544 (05 2018). https://doi.org/10.1093/nar/gky379, https://doi.org/10.1093/nar/gky379

[2] Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A., Taylor, J.: Galaxy CloudMan: Delivering cloud compute clusters. BMC Bioinformatics **11**, S4, 6 pages (2010)

[3] Arkhipov, A., Hüve, J., Kahms, M., Peters, R., Schulten, K.: Continuous fluorescence microphotolysis and correlation spectroscopy using 4Pi microscopy. Biophysical Journal **93**(11), 4006–4017 (2007)

[4] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)

[5] de Koning, W., Miladi, M., Hiltemann, S., Heikema, A., Hays, J.P., Flemming, S., van den Beek, M., Mustafa, D.A., Backofen, R., Grüning, B., Stubbs, A.P.: NanoGalaxy: Nanopore long-read sequencing data analysis in Galaxy. GigaScience **9**(10), Article no. giaa105, 7 pages (2020)

[6] Giles, M., Lazlo, E., Reguly, I., Appleyard, J., Demouth, J.: GPU implementation of finite difference solvers. In: Proceedings of the 7th Workshop on High Performance Computational Finance. pp. 1–8. WHPCF'14 (2014)

[7] Giles, M.: An introduction to GPU programming. https://people.maths.ox.ac.uk/gilesm/old/pp10/lec2_2x2.pdf

[8] Gudukbay, G.: gulsumgudukbay/racon_dockerfile. https://hub.docker.com/r/gulsumgudukbay/racon_dockerfile

[9] Hardy, D.J., Stone, J.E., Schulten, K.: Multilevel summation of electrostatic potentials using graphics processing units. Parallel Computing **35**(3), 164–177 (2009)

[10] Harris, D.: Amped Up: HPC Centers Ride A100 GPUs to Accelerate Science. https://blogs.nvidia.com/blog/2020/05/15/hpc-supercomputers-a100-gpus/ (2020)

[11] Hung, C., Tang, C.Y.: Bioinformatics tools with deep learning based on GPU. In: Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine. pp. 1906–1908. BIBM'17 (2017). https://doi.org/10.1109/BIBM.2017.8217950

[12] Illumina, I.: Next-Generation Sequencing (NGS). https://www.illumina.com/science/technology/next-generation-sequencing.html

[13] Katharina Hildebrandt, A., Stockel, D., Fischer, N., de la Garza, L., Kruger, J., Nickels, S., Rottig, M., Scharfe, C., Schumann, M., Thiel, P., Lenhof, H.P., Kohlbacher, O., Hildebrandt, A.: Ballaxy: Web services for structural bioinformatics. Bioinformatics **31**(1), 121–122 (2015)

[14] Ledergerber, C., Dessimoz, C.: Base-calling for next-generation sequencing platforms. Briefings in Bioinformatics **12**(5), 489–497 (2011)

[15] Mahmud, S.: Parallel Programming With CUDA Tutorial (Part-2: Basics). https://saadmahmud14.medium.com/parallel-programming-with-cuda-tutorial-part-2-96f6eaea2832

[16] Manconi, A., Moscatelli, M., Gnocchi, M., Armano, G., Milanesi, L.: A GPU-based high performance computing infrastructure for specialized NGS analyses. PeerJ Preprints **4**, e2175v1 (2016), https://doi.org/10.7287/peerj.preprints.2175v1

[17] Manglaviti, A., Genzer, P.: Brookhaven Lab Named an NVIDIA GPU Research Center. https://www.bnl.gov/newsroom/news.php?a=111821

[18] Docker, Inc.: What is a Container? — App Containerization — Docker. https://www.docker.com/resources/what-container

[19] Galaxy Committers: Docker Integration. https://galaxyproject.org/admin/tools/docker/

[20] Galaxy Committers: Galaxy 101 - What is Galaxy? https://galaxyproject.org/tutorials/g101/

[21] Galaxy Committers: Galaxy Project Stats. https://galaxyproject.org/galaxy-project/statistics/

[22] Galaxy Committers: Galaxy Publication Library. https://galaxyproject.org/publication-library/

[23] Galaxy Committers: Galaxy Tool XML File. https://docs.galaxyproject.org/en/master/dev/schema.html

[24] NVIDIA Corp.: High-Performance Computing Products and Solutions. https://www.nvidia.com/en-us/high-performance-computing/

[25] NVIDIA Corp.: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210. Tech. rep., NVIDIA Corporation (2014)

[26] NVIDIA Corp.: Tesla k80 gpu accelerator: Board specification. Tech. rep., NVIDIA Corporation (2015)

[27] Pacific Biosciences of California, Inc.: PacBio Company — About Us. https://www.pacb.com/company/about-us

[28] Sylabs.io: Singularity. https://sylabs.io/

[29] Monash University: Raw fast5s. https://bridges.monash.edu/articles/dataset/Raw_fast5s/7676174

[30] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE **96**(5) (2008)

[31] Oxford Nanopore Technologies. https://nanoporetech.com

[32] PacBio: Alzheimer IsoSeq 2016. https://downloads.pacbcloud.com/public/dataset/Alzheimer_IsoSeq_2016/

[33] Perez-Riverol, Y.: BioContainers. https://biocontainers-edu.readthedocs.io/en/latest/what_is_biocontainers.html

[34] Raghava, G.P.: OSDDlinux: A Customized Operating System for Drug Discovery. http://osddlinux.osdd.net

[35] Rodrigues, C.I., Hardy, D.J., Stone, J.E., Schulten, K., Hwu, W.M.W.: GPU acceleration of cutoff pair potentials for molecular modeling applications. In: Proceedings of the 5th Conference on Computing Frontiers. p. 273–282. CF'08, ACM, New York, NY (2008)

[36] Seymour, C.: nanoporetech/bonito: Bonito - A PyTorch Basecaller for Oxford Nanopore Reads. https://github.com/nanoporetech/bonito

[37] Vaser, R., Sović, I., Nagarajan, N., Šikić, M.: Fast and accurate de novo genome assembly from long uncorrected reads. Genome Research **27**(5), 737–746 (2017)

[38] VijayKrishna, N., Joshi, J., Coraor, N., Hillman-Jackson, J., Bouvier, D., van den Beek, M., Eguinoa, I., Coppens, F., Golitsynskiy, S., Stolarczyk, M., Sheffield, N.C., Gladman, S., Cuccuru, G., Grüning, B., Soranzo, N., Rasche, H., Langhorst, B.W., Bernt, M., Fornika, D., de Lima Morais, D.A., Barrette, M., van Heusden, P., Petrillo, M., Puertas-Gallardo, A., Patak, A., Hotz, H.R., Blankenberg, D.: Expanding the Galaxy's reference data. bioRxiv, https://doi.org/10.1101/2020.10.09.327114 (2020)

[39] Warris, S., Timal, N.R.N., Kempenaar, M., Poortinga, A.M., van de Geest, H., Varbanescu, A.L., Nap, J.P.: pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment. PLOS One **13**(1), Article no. e0190279, 9 pages (2018)