Rocket: Efficient and Scalable All-Pairs Computations on Heterogeneous Platforms

Heldens, S.; Hijma, P.; van Werkhoven, B.; Maassen, J.; Bal, H.; van Nieuwpoort, R.

DOI
10.1109/SC41405.2020.00105

Publication date
2020

Document Version
Final published version

Published in
Proceedings of SC20: The International Conference for High Performance Computing, Networking, Storage and Analysis

SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00105

Rocket: Efficient and Scalable All-Pairs Computations on Heterogeneous Platforms

Stijn Heldens*†, Pieter Hijma†‡, Ben van Werkhoven*, Jason Maassen*, Henri Bal‡, Rob van Nieuwpoort*†
*Netherlands eScience Center, †University of Amsterdam, ‡Vrije Universiteit Amsterdam
{s.heldens, b.vanwerkhoven, j.maassen, r.vannieuwpoort}@esciencecenter.nl, {pieter, bal}@cs.vu.nl

***Abstract*—All-pairs compute problems apply a user-defined function to each combination of two items of a given data set. Although these problems present an abundance of parallelism, data reuse must be exploited to achieve good performance. Several researchers considered this problem, either resorting to partial replication with static work distribution or dynamic scheduling with full replication. In contrast, we present a solution that relies on hierarchical multi-level software-based caches to maximize data reuse at each level in the distributed memory hierarchy, combined with a divide-and-conquer approach to exploit data locality, hierarchical work-stealing to dynamically balance the workload, and asynchronous processing to maximize resource utilization. We evaluate our solution using three real-world applications, from digital forensics, localization microscopy, and bioinformatics, on different platforms, from desktop machine to a supercomputer. Results shows excellent efficiency and scalability when scaling to 96 GPUs, even obtaining super-linear speedups due to a distributed cache.**

***Index Terms*—all-pairs computation; heterogeneous computing; GPU; work-stealing; data reuse; distributed cache**

I. Introduction

All-pairs compute problems, which evaluate a function for each pair of items of a data set, are prevalent in many scientific domains including radio astronomy [1], microscopy [2], bioinformatics [3], digital forensics [4], computer vision [5], data mining [6], information retrieval [7], and biometrics [8]. These problems typically involve calculating some measure, such as the distance or similarity, between pairs of data items, such as images or objects. In general, all-pairs compute problems are computationally demanding because of the quadratic nature of the problem. Additionally, maximizing data reuse is necessary to achieve optimal performance, which in turn requires careful consideration of the workload and data distribution.

The coarse-grained parallelism in all-pairs compute problems scales quadratically with the size of the data set. However, many all-pairs applications [1], [2], [4] also exhibit a large amount of fine-grained parallelism, within the pair-wise function, that can be exploited using GPUs. This makes distributed clusters equipped with GPUs a suitable platform for these applications: pairs can be processed in parallel across the different nodes while the computations for each individual pair are parallelized using the GPU. Since GPUs evolve rapidly, these clusters often upgrade in stages throughout their lifetime, leading to highly heterogeneous platforms containing different GPUs.

For all-pairs compute problems it is important that we maximize data reuse on all levels in the system since loading an item is expensive as it requires accessing remote files, unpacking, pre-processing, filtering, and transferring data. After an item has been loaded, the resulting data should ideally be used for as many pair-wise comparisons as possible.

We see a clear gap in related work when considering workload distribution and data reuse in existing distributed all-pairs compute frameworks. Some work applies static scheduling assuming the opportunities for data reuse are known in advance [9], [10], [11], [12], but this approach is not suitable if the pair computations are irregular or if the platform is highly heterogeneous. Others use dynamic workload distribution combined with full replication to overcome load-imbalance [13], [14], [15], but replicating all data across all nodes is expensive and only feasible for small data sets that fit in local storage.

In this paper, we present a framework called Rocket for efficiently executing all-pairs compute problems on heterogeneous platforms. Our solution avoids full replication while exploiting dynamic load balancing and works for data sets that do not fit into memory. We achieve excellent performance by:

- offering a dedicated system for all-pairs computations that provides a clear separation of concerns between the user's application code and the Rocket runtime system;
- using a software-based multi-level (distributed) cache to maximize data reuse at all levels;
- using a divide-and-conquer approach to efficiently exploit data locality, combined with hierarchical random work-stealing to dynamically distribute the workload; and
- exploiting asynchronous processing to overlap all data movement with useful computation.

We implemented three scientific applications (from digital forensics, localization microscopy, and bioinformatics) using our framework and evaluate their performance on different types of platforms (from a desktop machine to a supercomputer having $\sim$100 GPUs). We propose a performance model to analyze the results and show that Rocket achieves $88.5-99.2\%$ efficiency compared to the modeled lower-bound on the run time. Efficiency increases further when scaling the number of nodes due to a communication scheme that exploits the larger distributed memory capacity, leading to super-linear speedups.

This paper is structured as follows: Section II presents related work, Section III and IV explain the design and implementation of our solution, Section V describes the three applications, Section VI evaluates the performance of our framework, and Sections VII and VIII present future work and conclusions.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

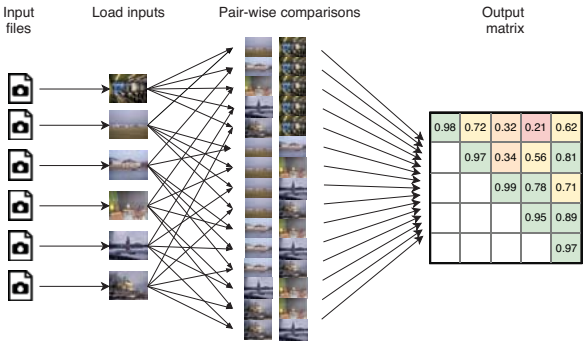

Fig. 1. Example of an all-pairs compute problem: calculating the pairwise similarity between images. Photos from the Dresden image database [16].

II. Motivation and Related Work

Several researchers have considered the problem of all-pairs compute problems on distributed systems. Some have focused on static work distribution where each node is assigned some subset of the pairs to be computed. This means that the opportunities for data reuse are known in advance. For instance, processing pairs (a, b) and (a, c) requires loading items a, b, and c in memory, where item a can be used twice.

Static distribution implies that the n items can be partially replicated since each node can predetermine the subset of items it requires. The question now becomes how to equally divide the $\binom{n}{2}$ possible pairs over p nodes such that the number of times each item is to be replicated is minimal. For example, Zhang et al. [11] consider all-pairs applications on Hadoop, and they use a heuristic to divide the pairs such that the total computation and data per node is balanced. In later work, Zhang et al. [15] reformulate the problem and find a data/work distribution using simulated annealing. Plimpton [9] considered distributed N-body simulations, and they propose a distribution scheme in which each node stores $\frac{2n}{\sqrt{p}}$ items. Kleinheksel and Somani [10] use cyclic quorums to lower this to $\frac{n}{\sqrt{p}}$, which appears to be the best-known lower bound.

Yeleswarapu et al. [12] propose a slightly different static approach having a memory footprint of just $\frac{3n}{p}$. Each node initially loads $\frac{n}{p}$ inputs in memory and the pair computations are performed in p *rounds*. Every round involves exchanging items between nodes and processing the pairs that result from combining the local and the received items.

Unfortunately, the above methods all utilize a static workload distribution, which suffers from load-imbalance if the computation is irregular or the platform is heterogeneous. There is also a limit to the size of the problems that can be solved since each node must have sufficient memory to store the assigned items. Additionally, none of these authors consider systems containing both CPUs and GPUs.

Other researchers have looked into dynamic scheduling of the workload. For instance, Zhang et al. [15] extended their static data distribution scheme to support a limited form of dynamic scheduling. Their solution is based on the observation that, since data is partially replicated, one pair can sometimes be processed by more than one node in the cluster. However, while this solution allows some flexibility in scheduling of the work, load imbalance is still a possibility.

Moretti et al. [13] investigate full dynamic scheduling, and they propose *All-pairs*: a framework for all-pairs computation on grids consisting of loosely coupled computers. Their framework replicates all inputs across all nodes and uses a centralized master to dynamically dispatch batches of jobs. Li et al. [14] present a similar framework intended for performing pairwise Needleman-Wunsch computations on distributed heterogeneous platforms. Again, data is replicated across all devices, but they use a two-level scheduling solution: a centralized dispatcher distributes batches of jobs to nodes, and a node-level dispatcher distributes these jobs across the CPUs/GPUs. However, both solutions require full replication, which is expensive and infeasible if the input data exceeds memory capacity.

Overall, we observe that all-pairs computation problems show friction between two aspects: workload distribution and data distribution. Dynamically scheduling the work avoids the problem of workload imbalance, but requires all data to be available at all nodes. Statically scheduling the work avoids full replication, but could lead to workload imbalance.

In the next sections, we discuss our approach that allows dynamic work distribution while avoiding full replication. The main differences with related work are that our solution (1) uses dynamic work scheduling by means of work-stealing without the need for a centralized master, (2) supports partial replication by means of caches on multiple levels of the memory hierarchy, and (3) fully supports heterogeneous GPU platforms and irregular workloads.

III. Design

In this section, we explain the design of our framework. The essence of an all-pairs compute problem is straightforward: calculate the result of $f(\ell(i), \ell(j))$ for each pair (i, j) where $1 \leq i < j \leq n$. Given are a deterministic function ℓ that loads the required data for the i-th item into memory and a binary function f applied to the two items. For this work, we assume the inputs are coarse-grained static files (e.g., images, sensor data, serialized objects) and $\ell(i)$ reads the i-th file and, if needed, parses its content and performs some pre-processing (e.g., filters, transformations, or feature extraction). The results of $x = \ell(i)$ and $y = \ell(j)$ are passed to $f(x, y)$ which computes an application-specific answer, such as a correlation score or distance metric. Although ℓ and f are presented here as simple functions, they could be processing pipelines that consist of multiple stages executed on CPUs and GPUs.

For Rocket, we assume the user's processing pipelines for ℓ and f follow the pattern shown in Fig. 2. This design is simple and elegant, making it easy to understand and offering a clear separation of concerns between what the user needs to implement and what is handled by Rocket. Nevertheless, the model is sufficiently flexible to implement several real-world applications, as we shall demonstrate in Section V.

The $\ell(i)$ pipeline is performed in four stages: (1) load the required file from (possibly remote) storage into local memory;

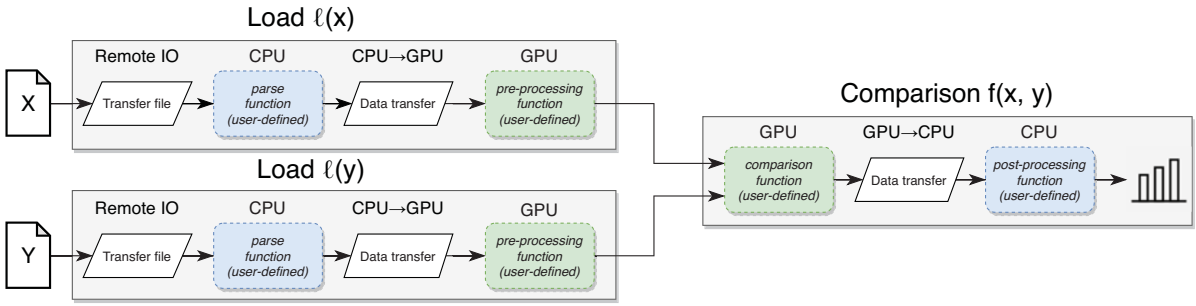

Fig. 2. Rocket's pipeline for performing one pair-wise comparison.

```
interface Application<Key, Result> {
    Path getFilePathForKey(Key key);
    void parseFile(Key key,
        HostBuffer fileContents, HostBuffer result);
    void preprocessGPU(Key key,
        DeviceBuffer input, DeviceBuffer result);
    void compareGPU(
        Key leftKey, DeviceBuffer leftItem,
        Key rightKey, DeviceBuffer rightItem,
        DeviceBuffer result);
    Result postprocess(HostBuffer result);
}
```

Fig. 3. The interface that must be implemented by the user for the application.

(2) *parse* the file's raw content into the appropriate format on the CPU; (3) transfer the data from CPU memory to GPU memory; and, (4) *pre-process* the data on the GPU.

Furthermore, $f(x, y)$ is performed in three stages: (1) perform the *comparison* on the GPU; (2) transfer the result from GPU memory to CPU memory; and, (3) *post-process* the result on the CPU.

As an example, consider an algorithm that calculates the pair-wise similarity between photos such as shown in Fig. 1. Such an application consists of the following tasks: decoding of the image format on the CPU (*parsing*), applying filters to the image on the GPU (*pre-processing*), calculating a correlation score on the GPU (*comparison*), and applying a threshold to the score on the CPU (*post-processing*).

With this design the user defines four application-specific functions: parsing (CPU), pre-processing (GPU), comparison (GPU), and post-processing (CPU) adhering to the interface defined in Fig. 3. Launching an all-pairs application on the cluster can then be achieved by simply calling Rocket's main class with an input array of `Key` elements. Rocket will automatically take care of network communication, data transfers, memory management, scheduling, exploiting data reuse, load balancing, and overlapping computation with I/O.

IV. Implementation

This section dives into the implementation of our framework. Three aspects lay the foundation for Rocket. First, a naive approach to processing would be to ignore any form of data reuse and execute both $\ell(i)$ and $\ell(j)$ when processing one pair (i, j). However, this is expensive since the cost of loading the items, often considerably outweighs the cost of actually comparing the two items. Fortunately, the functions $\ell(i)$ and $\ell(j)$ are deterministic meaning that, after they have been executed once, their results can reused for future jobs that require item i or j. Exploiting this data reuse will significantly improve performance. We use a software-based cache to store these results at different levels of the distributed memory hierarchy. This design is described in Section IV-A.

Second, as discussed in Section I, dynamically scheduling the workload is necessary to avoid the problem of load imbalance, either due to irregular completion time of ℓ or f, or because of heterogeneity of the platform. A straightforward solution would be to have a single master dynamically distributing the pairs (i, j) to worker nodes. However, this would not take data locality into account which is crucial to maximize cache effectiveness. Moreover, scalability would be limited due to having a central point. Instead, we have chosen to perform load-balancing by a divide-and-conquer approach with hierarchical work-stealing, since it shows excellent scalability and data locality in practice. This allows for dynamic load balancing while also exploiting data reuse. This solution is described further in Section IV-B.

Third, to maximize resource utilization, Rocket keeps a large number of comparison jobs in progress at all times and relies on asynchronous processing to make progress. This approach maximizes throughput (i.e., number of comparisons performed per second) and thus minimizes the runtime of the all-pairs application. Section IV-C describes this design.

We have implemented Rocket using Ibis [17] as communication library, Constellation [18] for distributed work-stealing, CUDA for GPU programming, and the Xenon library [19] to access remote storage resources.

A. Multi-Level Caching

As explained before, exploiting data reuse is essential since re-executing the entire pipelines of $\ell(i)$ and $\ell(j)$ for each pair (i, j) is expensive. For instance, our evaluation presents an application from digital forensics (Section VI) where the average GPU compute time is 1 ms for one comparison, but parsing one input file takes 130 ms. By storing the results of each execution of ℓ in a cache, the number of times items need to be loaded is reduced and the system can be fully dedicated to performing comparisons.

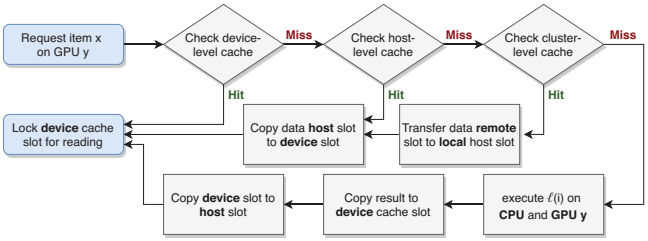

Fig. 4. Flow diagram describing the cache policy.

Rocket uses a three-level software-based cache to maximize the available memory capacity by storing results at different levels of the memory hierarchy. At the first level is a per-device cache that stores the results in GPU memory since both the last stage of ℓ and first stage of f are performed on the GPU. At the second level is a per-node host memory cache that extends the first-level device cache with the usually much larger memory capacity ($\sim$10-100 GB) of the host compared to that of an average GPU ($\sim$5-10 GB). At the third level is a cluster-wide communication scheme that allows nodes to query remote caches, essentially establishing one large distributed memory cache. Note that these caches are managed in *software* and should not be mistaken with *hardware* caches such as disk caches, L1/L2/L3 cache on CPUs, or shared memory on GPUs.

Below we describe each level in detail. Figure 4 visualizes how these different levels interact.

1) First-level (Device): At the first level is a per-device cache that manages a fixed number of fixed-sized slots. This cache resides in GPU device memory. Each slot contains a memory buffer and a status flag which can be `WRITE` (a writer is active) or `READ` (n readers are active).

When a job (i, j) is submitted, this cache is checked for items i and j. On cache miss for item i (or j), the least-recently-used slot is evicted (discarding its content) and assigned to item i after which the slot is set to `WRITE`. Now the result of $\ell(i)$ needs to be copied into the slot from the next level cache after which the flag is set to `READ`.

On cache hit, the status flag is checked. For `READ`, the comparison f can be performed immediately (assuming j is also available) while the number of readers is temporarily incremented. For `WRITE`, another job is busy writing to the slot so the current job is put on hold until the data becomes available. Note that the cache thus introduces synchronization between jobs: while one job is writing item i, other jobs that depend on item i are stalled until the slot becomes available. In practice, this does not lead to performance issues, because Rocket ensures that a sufficient number of concurrent jobs are in progress at all times (see Section IV-C).

2) Second-level (Host): At the second level is a per-node cache that stores the results in page-locked main memory. The implementation is similar to that of the device cache, only buffers reside in main memory instead of device memory. This cache is thus shared by all GPUs in one node.

On a first-level device cache miss, the second-level host cache is checked for the item. On a hit, the contents of the host slot is transferred to the device slot. As we show later (see Section IV-C), the overhead of copying data between host and device caches is negligible since Rocket overlaps data transfers and computation. On a miss, an item is evicted and the empty slot is assigned to item i for which the data needs to be obtained from the next level cache. Note that data is thus always written to both the device and host cache. We chose this solution since it is important for the third-level cache that allows nodes to query remote host caches.

3) Third-level (Distributed): At the third level we use a communication scheme that, after a local cache miss, allows nodes to query the host cache of remote peers. While the previous two levels reduce the number of loads per *node*, the third level reduces the loads for the *cluster as a whole*.

An important consideration is how to locate a node that has the required data in its local cache, which is non-trivial since we use dynamic scheduling. A centralized registry that keeps track of the data on nodes would be a poor solution since it introduces a central bottleneck and requires excessive coordination and bookkeeping. Alternatives we considered were broadcasting the request to all peers (not a scalable solution) or to one (or several) randomly chosen peers (but the probability that a random node has one specific item is slim).

Instead, we use a simple communication scheme that allows the system to form a distributed hash table. Our scheme is based on the observation that a node that requested an item in the past, will eventually find the data and keep it for some time into the future. Each node is assigned the responsibility to serve as *point-of-contact* for some subset of the items, where item i is assigned to node $i \ mod \ p$ and p is the number of nodes. These nodes do not necessarily store these items themselves. Instead, each node keeps track of a local bookkeeping array `candidates` where `candidates[i]` stores the list of the h nodes that most recently requested item i in the past and are considered the most likely *candidates* for future requests.

For node A to request an item, the following steps are taken:

- Node A sends a request for item i to node $B = i \ mod \ p$.
- Node B retrieves from `candidates[i]` the nodes $C_1, ..., C_h$ and prepends A to `candidates[i]`. The request and the list $C_2, ..., C_h$ are forwarded to node C_1.
- Each node C_x checks its local host cache for item i:
  - On a hit, it sends the data directly to node A.
  - Otherwise, if $x < h$, it forwards the request to C_{x+1}.
  - Otherwise, it sends a failure directly to node A.

If this best-effort mechanism does not provide the item, node A is forced to execute $\ell(i)$ locally. In essence, node B acts as a mediator helping node A (searching the data) to find node C (offering the data). The parameter h determines the maximal number of *hops* to check, and we evaluate this parameter in Section VI. This scheme is scalable, contains no central component, requires a small amount of bookkeeping (only the `candidates` array) and communication (just $h+2$ messages per request). Note that in some scenarios, node B or node C_x could be node A itself, but this does not affect the correctness of the scheme.

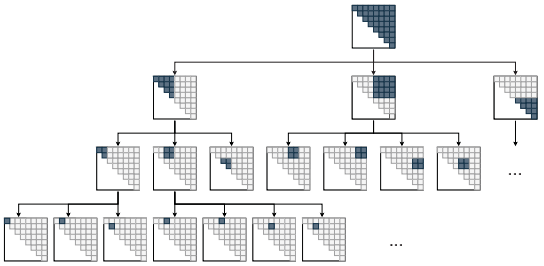

Fig. 5. Hierarchical splitting an all-pairs workload of 8×8 items.

B. Locality-Aware Work Scheduling

To dynamically schedule the pairs (i, j), Rocket uses a divide-and-conquer approach together with work-stealing inspired by frameworks such as Cilk [20] and Satin [21]. Divide-and-conquer is a common technique in which a larger problem is recursively divided into smaller sub-problems until they become small enough to compute directly. It is known that this approach naturally offers excellent data locality while allowing for dynamic workload balancing [22].

Recall that the total workload consists of processing each pair (i, j) where $1 \leq i < j \leq n$. This workload can be seen as an upper triangular matrix. This larger matrix can be split into four sub-matrices, one for each quadrant, and each sub-matrix can recursively be split into smaller quadrants until eventually reaching individual entries (i, j). Figure 5 shows this process for a small 8×8 matrix. Note that quadrants may sometimes contain no work; these can be ignored.

Rocket performs distributed work-stealing using *Constellation* [18]: a software platform for distributed, heterogeneous, hierarchical environments. During initialization, each node launches one Constellation worker thread per GPU. The master node then spawns a single root task representing the entire matrix to be computed. This task spawns four new tasks (one for each quadrant of the matrix) and each sub-task recursively spawns four new tasks, one for each sub-quadrant. The tasks at the lowest level represent a single (i, j) entry and these leaf tasks submit the actual job (i, j) to the Rocket runtime system. Worker threads always prioritize local tasks at the lowest level in the tree since these provide the best data locality.

Load balancing is performed by *random work-stealing*: Workers that become idle will repeatedly attempt to steal a task from a randomly selected peer. This technique has been shown to be a suitable solution to balance workload in distributed environments [21]. The task stolen is always at the 'highest' level (i.e., the largest task available) since it results in the most work per steal request. Stealing is performed hierarchically: workers first attempt to steal from a worker on the same node before selecting a remote node. The advantage of work-stealing over master-worker is that it exhibits good data locality while also balancing the workload: if there are no idle nodes, work is not stolen and thus executed locally on the node that generated it. It is well-known that divide-and-conquer leads to excellent exploitation of locality, both for hierarchical memory architectures [20] and in distributed systems [21].

Rocket's runtime system operates asynchronously and submitting a job does not block the caller. Without any form of back-pressure, one node could rapidly claim all available work meaning others become idle. To prevent this, Rocket has a *concurrent job limit* parameter that limits the number of concurrent jobs that can be simultaneously submitted to Rocket. Once this limit is reached, worker threads will stop submitting new jobs until an older job completes.

C. Asynchronous Processing

A naive implementation to process submitted pairs is to have one thread (or a small pool of threads) processing the submitted pairs synchronously one-by-one. However, this would lead to inefficient resource usage since it can result in moments of resource contention (e.g., all threads could perform I/O simultaneously) while under-utilizing other resources (e.g., the GPU is idle in the meantime).

To maximize resource utilization, Rocket keeps many jobs in progress (*concurrent job limit* as described in Section IV-B) and relies heavily on asynchronous processing to make progress on these jobs. Different threads are launched during initialization with each thread responsible for one type of resource, meaning that tasks executed by different threads do not interfere with each other. For our current implementation, Rocket launches the following types of threads:

CPU	A thread pool performs CPU computations.
GPU	One thread per GPU to launch device kernels.
CPU→GPU	One thread per GPU for performing data transfers from host to device memory.
GPU→CPU	One thread per GPU for performing data transfers from device to host memory.
I/O	One thread for I/O on the (remote) file system.

With this scheme, CPU processing, GPU processing, data transfers, and I/O operations are all overlapped. For example, multiple *parsing* tasks can be executed simultaneously on CPUs together with a *comparison* task running on the GPU, while also transferring data to and from GPUs and performing I/O operations in parallel. An optional profiling flag can be enabled to trace the tasks executed by different threads, which can be useful for debugging purposes and performance analysis. Figure 6 shows an example of such a trace visualized on a timeline.

Asynchronous processing is essential for Rocket to achieve good performance since it means that resources are fully utilized. For instance, for each job (i, j), a first-level cache hit on items i and j is cheap (since the comparison pipeline can be executed immediately), but a cache miss is expensive since it involves many steps (e.g., data transfers, I/O, parsing, preprocessing, etc.). It is thus important to keep a larger number of jobs active so that the system can 'anticipate' first-level cache misses and acquire the necessary data before running out of work. Figure 6 demonstrates this well: the GPU remains fully utilized since sufficient work is available, even though slow I/O and CPU tasks are performed in the background.

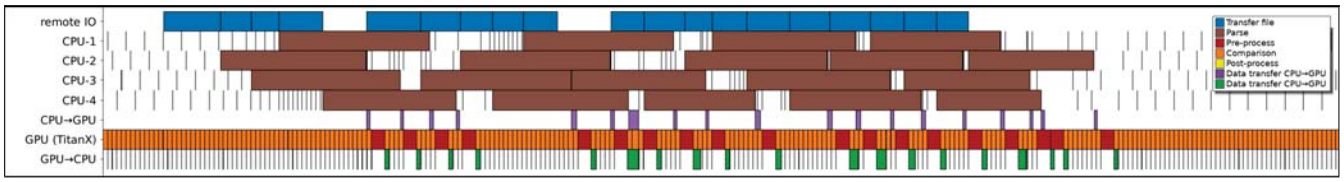

Fig. 6. Small section of a trace from the forensics application (Section V) visualized on a timeline. Rows represents threads and boxes represent executed tasks.

V. Applications

To demonstrate the generality of our framework, we use three scientific applications that are used by researchers in digital forensics, localization microscopy and evolutionary biology. These are realistic applications, not simplified benchmarks, and include all required pre-processing, I/O, and application logic. The computational kernels are taken as black boxes and are not analyzed in this work. The applications have different compute and data characteristics, thus demonstrating the generality of our approach.

A. Common-Source Identification (Forensics)

Common-source identification is a digital forensics application that takes a set of images and identifies which images were made by the same camera based on sensor noise patterns. These noise patterns, called *Photo Response Non-Uniformity* (PRNU) patterns [23], originate from small deficiencies in the imaging sensor, resulting in small differences in the responsivity. In the resulting images, this leads to pixels being brighter or darker while having received equal saturation. To find the images that have been acquired by the same sensor, the noise patterns of all images have to be extracted and compared with each other.

Our Rocket-based implementation is based on the application by Van Werkhoven et al. [4] developed for the Netherlands Forensics Institute. We reuse their GPU kernels for extracting the PRNU patterns from images and for computing the similarity between the PRNU patterns of different images. The metric that is used to measure the similarity of two PRNU patterns is the *Normalized Cross Correlation*. The decoding of the JPEG format is done on the CPU using `libjpeg`. The application compares images that have equal dimensions and as such, computations are highly regular.

Our data set consists of images having dimensions 3648×2736 from the Dresden image database [16], which is developed specifically for the goal of researching PRNU-based algorithms.

B. Phylogeny Tree Construction (Bioinformatics)

Phylogenetic tree reconstruction is the problem of reconstructing how species descend from common ancestors given their genetic material. The popular alignment-free method by Qi et al. [3] is a fast algorithm to achieve this by hierarchical clustering of the distance matrix between all species. The distance between two species is calculated based on the distance of their *composition vectors* (CVs). The CV of a species is derived from the frequency of substrings, of a chosen length k, in their protein sequences. These CVs are represented as sparse vectors and have between 100.000 and 1.800.000 entries. Extracting these CVs is expensive since it requires scanning the entire genome, but comparing two CVs is cheap, essentially being the dot product between two sparse vectors. Computation is irregular since the vectors are sparse.

We have implemented this algorithm in CUDA based on the description and formulas by Qi et al. [3]. The input data set consists of 2500 randomly chosen reference bacteria proteomes from Uniprot Proteomes database [24] and files are stored in compressed FASTA format. Pre-processing consist of decompressing the file (on CPU) and constructing the CV (on GPU), while comparing two CVs is done entirely on the GPU.

C. Localization Microscopy Particle Fusion (Microscopy)

Localization microscopy is an optical super-resolution microscopy method that operates on the localization of individual fluorophores, rather than pixelated images, obtained by a fluorescence microscope. To achieve a resolution well beyond the diffraction limit, multiple images of the same structure are fused to improve the signal-to-noise ratio and the resolution. The method by Heydarian et al. [2] uses *all-to-all registration* of particles to achieve robustness against individual misregistrations and under labeling.

Our Rocket-based implementation reuses the GPU kernels from the application by Heydarian et al. [2]. These kernels implement two different methods to score the similarity of two particles, which in turn consist of thousands of localizations. The first method is a quadratic L2-distance metric between two Gaussian Mixture Models [25] and the second method is known as the Bhattacharya distance function [2]. An optimizer calls these two methods many times and therefore the registration process is very compute-intensive, even on a small number of localizations, and heavily data-dependent, making the execution time highly irregular.

Our benchmark data set was generated using the simulator by Heydarian et al. and contains 256 particles stored in JSON format. Each particle consists of between 1000 and 2000 localizations. Since the application works directly on the localizations, there is no pre-processing required other than reading and parsing the particle files.

VI. Experimental Evaluation

In this section, we evaluate our framework's performance. After discussing a basic performance model and our experimental setup, we analyze results for one node, a homogeneous cluster, a heterogeneous cluster, and Cartesius (the Dutch National supercomputer).

A. Performance Model

To establish a baseline for the performance of Rocket, we present a performance model which determines a lower bound on the run time using a hypothetical computing system.

Given n items, the *comparison* pipeline (Fig. 2) must be executed $\binom{n}{2} = \frac{n^2-n}{2}$ times (once for each pair). The *load* pipeline must be executed at least n times (once for each item), but it may be executed more than n times (i.e., items were evicted from cache). We assume Rn loads are performed in total, where R indicates the number of loads relative to the data set size. For instance, $R = 4.3$ indicates that each item was, on average, loaded 4.3 times.

R serves as a basic metric for data reuse since a lower value indicates fewer loads and thus better reuse of previously loaded items. With perfect data reuse, each item is loaded once (thus $R = 1$). In practice $R > 1$ due to two reasons: (1) insufficient local cache capacity means that items are evicted that are later loaded again and (2) different nodes in a distributed environment load the same item independently of each other.

For simplicity, we assume our system contains one CPU and one GPU. The total GPU processing time (T_{GPU}) is determined by executing pre-processing Rn times and comparison $\binom{n}{2}$ times (where t_x is the average execution time of stage x):

$$T_{\text{GPU}} = Rn\ t_{\text{pre-process}} + \binom{n}{2} t_{\text{comparison}} \tag{1}$$

The total CPU processing time (T_{CPU}) is determined by executing parsing Rn times and post-processing $\binom{n}{2}$ times.

$$T_{\text{CPU}} = Rn\ t_{\text{parse}} + \binom{n}{2} t_{\text{post-process}} \tag{2}$$

The time spent on I/O can be estimated based on the file sizes and the average I/O bandwidth. However, the actual bandwidth depends heavily on the load on the storage system.

$$T_{\text{IO}} \approx Rn\ \frac{\text{average file size in MB}}{\text{IO bandwidth in MB/sec}} \tag{3}$$

Overhead of CPU-GPU data transfers is negligible since it is easily overlapped. Perfectly overlapping CPU time, GPU time, and I/O means the total run time will be the maximum of T_{CPU}, T_{GPU}, and T_{IO}. This motivates why it is important to maximize data reuse: R appears in all three equations and thus minimizing R means maximizing performance.

To establish a baseline, we assume our system has infinite memory and thus perfect data reuse (i.e., $R = 1$), I/O has infinite bandwidth (i.e., $T_{IO} \approx 0$), and most processing is performed on the GPU ($T_{\text{GPU}} > T_{\text{CPU}}$). In this scenario, the lower bound on the runtime T_{min} equals T_{GPU} for $R = 1$.

$$T_{\text{min}} = n\ t_{\text{pre-process}} + \binom{n}{2} t_{\text{comparison}} \tag{4}$$

Our system would show optimal performance on p nodes if the measured run time is T_{min}/p. We therefore define *system efficiency* as the ratio between this modeled lower bound on the runtime and the actual measured run time T on p nodes.

TABLE I
CHARACTERISTICS OF APPLICATIONS FOR NVIDIA TITANX MAXWELL. TIME IS REPORTED AS AVERAGE ± STANDARD DEVIATION.

Application	Forensics	Bioinformatics	Microscopy
No. of input files (n)	4980	2500	256
Size of raw data on disk	19.4 GB	1.8 GB	150 MB
Size of preprocessed data in memory	189.7 GB	110.0 GB	0.7 MB
No. of pairs	12,397,710	3,123,750	130,816
Total data pair-wise processed	944.7 TB	275.0 TB	179.2 MB
Cache Slot Size	38.1 MB	145.8 MB	6.0 KB
No. Device Cache Slots	291	81	256
No. Host Cache Slots	1050	280	256
Time Parse (CPU)	130.8±14.11 ms	36.9±14.79 ms	27.4±1.56 ms
Time Pre-process (GPU)	20.5±0.02 ms	27.0±4.90 ms	N/A
Time Comparison (GPU)	1.1±0.01 ms	2.1±0.79 ms	564.3±348 ms
Time Post-process (CPU)	0 ms	0 ms	0 ms

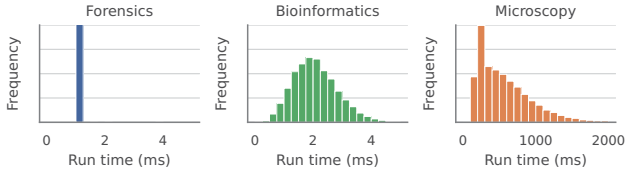

Fig. 7. Histogram of the run times for the comparison kernel (i.e., $t_{\text{comparison}}$) from the three applications. Note the different scales on the horizontal axis.

$$\text{system efficiency} = \frac{T_{\text{min}}/p}{T}. \tag{5}$$

B. Experimental Setup

Experiments were performed on DAS-5 [26], the distributed platform for experimental computer science research in the Netherlands. We used the VU site; each node has two Intel Xeon E5-2630 CPUs (16 cores total), offers 64 GB of memory (40 GB allocated to host cache), runs CentOS Linux 7, and nodes are connected by 56 Gb/s InfiniBand FDR. The site offers a variety of GPUs across the different nodes. We use MinIO over InfiniBand to serve as a central file storage server.

The last section discusses experiments performed on the Dutch national supercomputer (*Cartesius*[1]). Each node has two E5-2450 v2 CPUs (16 cores total), two NVIDIA Tesla K40m GPUs, offers 96 GB of main memory (80 GB allocated to host cache), and two Mellanox ConnectX-3 InfiniBand adapters (each providing 56 Gb/s inter-node bandwidth).

C. Single Node

Table I shows information on the data set size for the three applications including the size on disk (i.e., the total of the compressed input files) and the size in memory (i.e., the total after parsing and pre-processing each file). For the forensics application and the bioinformatics application, the data set increases considerably in size after pre-processing and does not fit into memory of a single node. For the microscopy application, the data set size is small and actually decreases during pre-processing due to the conversion from a text-based to a binary format. The table also shows the total amount of data that needs to be combined to perform all pair comparisons (i.e., each of the n items is retrieved n times), highlighting the

[1]https://www.surf.nl/en/dutch-national-supercomputer-cartesius

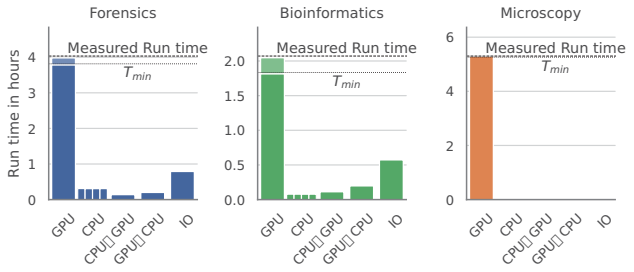

Fig. 8. Processing time per thread for each application using one node (TitanX Maxwell). The GPU bar is divided into *pre-processing* (top) and *comparison* (bottom). The dashed line indicates the start-to-end run time of the framework while the dotted line indicates the value of T_{min}.

quadratic nature of all-pairs compute problems. For instance, for the forensics application, this total reaches almost 1 PB.

To establish a baseline, we ran the three applications on one node equipped with one NVIDIA TitanX Maxwell. Table I shows the timing results and cache configuration for these runs. The table clearly shows that the three applications have different compute- and data-characteristics: The microscopy application is compute-intensive (comparisons are slow and a small amount of data is processed) while the other two applications are data-intensive (comparisons are fast and a large amount of data is processed). The forensics application is slightly more data-intensive than the bioinformatics application since more data is processed and comparisons are faster. For all three applications, the post-processing stage on the CPU is negligible since it only interprets the GPU's result.

Figure 7 shows the distribution of run times of one comparison and confirms that the forensics application is a regular problem, while the other two are highly irregular. Dynamic load balancing for those applications is thus necessary since static scheduling could lead to load-imbalance.

Figure 8 shows, for each application, the overall run time of Rocket together with the total active time of each thread. The data per thread was extracted from a profile trace by taking the total time of tasks executed by each thread. The figure shows that all three applications are GPU-intensive since the GPU processing time is dominant. Additionally, the results show that the overall run time of the framework equals the GPU processing time, indicating that the asynchronous processing excellently overlaps GPU processing with other activities in the system. For instance, the bioinformatics application spent more than 30 minutes on I/O operations, but this had no impact on the overall run time since it is overlapped with GPU processing.

The system efficiencies are high: 94.6% (forensics), 88.5% (bioinformatics), and 99.2% (microscopy). They would increase even further if more memory were available which would allow better data reuse and would lower the overhead of loading items multiple times. Table I indicates that, for two applications, only a fraction of the inputs can be cached in host memory slots (21.1% for forensics and 11.2% for bioinformatics).

To simulate a desktop computer with fewer available resources, we artificially further limit the number of local cache slots and study the effect on performance. Figure 9

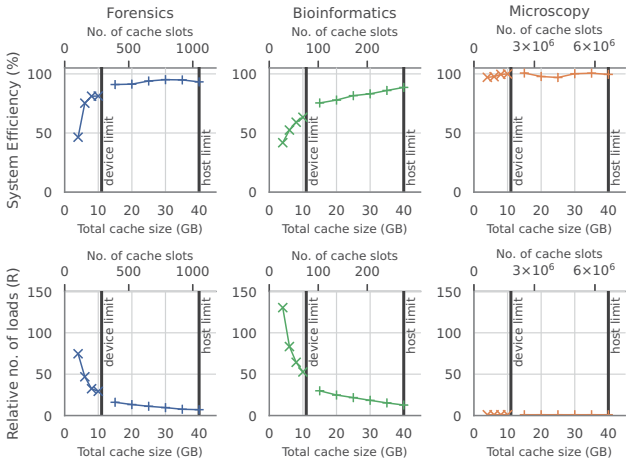

Fig. 9. System efficiency and factor R versus cache size on one node (NVIDIA TitanX Maxwell).

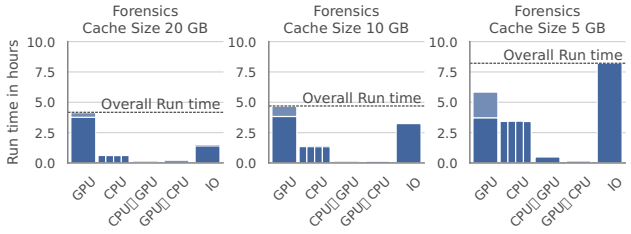

Fig. 10. Processing time per thread on one node (NVIDIA TitanX Maxwell) for different host cache sizes. Results shown are for the forensics application.

shows performance when varying the number of cache slots (thus the maximum cache size S) of host and device cache. For $S < 11$ GB, the host cache was disabled and the device cache was set to S. For $S > 11$ GB, the host cache was set to S and the device cache to 11 GB (GPU memory capacity).

The microscopy application is not affected by the local cache size since its input easily fits into memory. For the other two applications, the number of loads is inversely proportional to the local cache size and system efficiency gradually degrades when shrinking the cache. For example, for the bioinformatics application, at 6 GB only 1.7% of the inputs can be cached at any moment in time, but system efficiency is still 52.5% compared to a hypothetical system having infinite memory. Overall, Rockets continues to deliver decent performance even when limited to a tiny memory footprint. This is due to the hierarchical processing approach which provides excellent data locality, even for a single node.

Figure 10 shows the processing time per thread when varying the local cache size for the forensics application. The figure shows that decreasing the cache size results in increasing values T_{CPU}, T_{GPU}, and T_{IO} since items are re-loaded more frequently. On the other hand, increasing total cache capacity will thus lead to better performance and would be be possible by using more that one node.

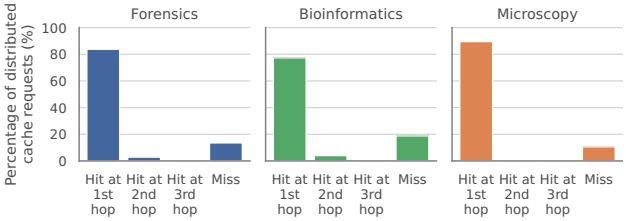

Fig. 11. Percentage of cache hits/misses of the distributed cache ($h = 3$) for each of the three applications on 16 nodes (NVIDIA TitanX Maxwell).

D. Scalability

In this section, we evaluate performance on 16 nodes, each having one NVIDIA TitanX Maxwell GPU. First, we investigate parameter h which specifies the maximum number of hops to check for each distributed cache request (see Section IV-A3). Figure 11 shows the percentage of cache hits and misses for $h = 3$. The figure indicates that the vast majority of requests either results in a hit at the first hop (between $75-88\%$) or a miss (between $11-19\%$). Subsequent hops after the first one thus contribute little to the number of cache hits. The remaining experiments in this paper are performed for $h = 1$ since this already provides an excellent cache hit ratio while generating the least amount of network traffic.

We now consider scaling to 16 nodes for two scenarios: one with the distributed cache and one without. Figure 12 shows speedup, system efficiency, data reuse, and average I/O usage versus the number of nodes for these two scenarios. For the microscopy application, we see an excellent speedup of $15.8\times$ on 16 nodes; this application is expected to scale well since it is compute-intensive. For the other two applications, we even see *super*-linear speedup on 16 nodes when enabling the distributed cache ($16.9\times$ for bioinformatics and $16.1\times$ for forensics), but *sub*-linear speedup when disabling the distributed cache ($14.6\times$ for bioinformatics and $14.7\times$ for forensics). The applications show much better scalability when the distributed cache is enabled.

The super-linear speedup can be understood when considering the total number of loads. The distributed cache exploits the larger combined memory capacity which results in better data reuse and thus higher efficiency. For instance, for the forensics application, going from 1 to 16 nodes with the distributed cache means factor R lowers from 6.7 to 1.7 and system efficiency increases from 95.8% to 97.6%. Without, R raises to 14.3 and system efficiency decreases to 87.5%.

Besides run time, it is also important to consider the impact on I/O when scaling to large platforms: shorter run times combined with more nodes result in increased pressure on the storage system. Most production platforms run more than one application simultaneously. Therefore, it is important to reduce the I/O pressure on the storage system, even if it does not directly improve the performance of our application. Less I/O pressure will improve the overall system performance.

Figure 12 shows that the average I/O usage (i.e., total bytes transferred by all nodes divided by total run time) is negligible

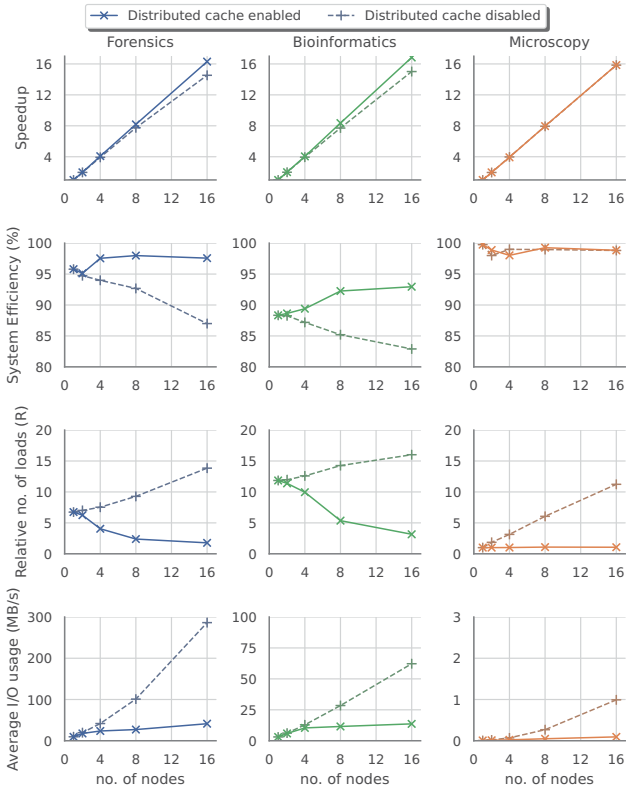

Fig. 12. Speedup, system efficiency, factor R, and average I/O usage when scaling cluster size from 1 to 16 nodes (NVIDIA TitanX Maxwell).

for the microscopy application. For the other two applications, I/O usage scales linearly with the number of nodes, but at a much slower rate when the distributed cache is enabled. For instance, for the forensics application, the average I/O usage is 9.6 MB/s when using one node (137 GB over $\sim$4 hours). Using 16 nodes with the distributed cache leads in I/O usage of only 39.9 MB/s, an increase of just $4.1\times$. Disabling the distributed cache results in I/O usage of 294.7 MB/s (289 GB over 16.3 minutes), an increase of almost $31\times$ over one node.

E. Heterogeneity

Dynamic load-balancing means that Rocket can exploit heterogeneous systems efficiently, even if the applications are irregular or the system is shared with multiple users. Nodes might also contain different GPUs from different generations, which is common in production environments since these platforms often replace GPUs in several phases throughout the lifetime of the system due to the fast evolution of GPUs.

To demonstrate how our framework handles such a scenario, we execute the applications on four nodes equipped with different (combinations of) NVIDIA GPUs from different generations: node I (Kepler K20m), node II (Maxwell GTX980 + Pascal TitanX), node III ($2\times$Turing RTX2080Ti), and node IV (Kepler Titan + Pascal TitanX). Figure 13 shows the performance for each node individually and when using all four nodes together. Performance is measured in average throughput

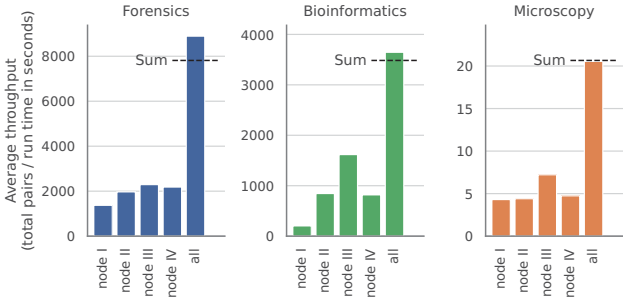

Fig. 13. Results for heterogeneous runs of applications, see Section VI-E.

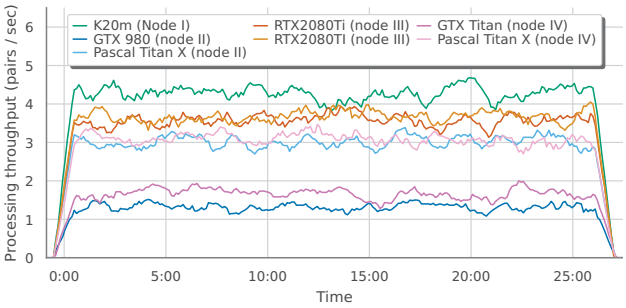

Fig. 14. Heterogeneous run for the microscopy application, see Section VI-E. Throughput is measured using a rolling average of one minute. Fluctuations are due to the irregular run times of the pair computations, see Fig. 7.

(i.e, total pairs divided by total run time) to ease comparison of the performance.

The results show good performance for each application on each of the four nodes individually, where some nodes inherently provide better performance (e.g., node III) than other nodes (e.g., node I) due to the performance differences of the GPUs. Combining the four nodes should ideally provide performance equal to the sum of the individual nodes and the figure shows that the actual performance often even outperforms this sum due to the distributed cache. Overall, Rocket delivers high performance even when on a diverse platform consisting of 7 GPUs from 4 different generations across 4 nodes.

Figure 14 shows the processing throughput over time (i.e., pairs processed per second) of the combined run for the microscopy application. We make the following observations: First, all nodes finish at roughly the same time, indicating that the workload is well-balanced. Second, the processing rate is fairly consistent across the run for each GPU, although fluctuations are present due to the irregularity of this application (i.e., some pairs take longer to process than others, see Fig. 7). Rocket is designed to always acquire more jobs well before the GPUs become idle, meaning the GPUs are always fully utilized. Third, the processing rate differs for the different devices, with more powerful GPUs (e.g., RTX2080Ti) delivering a higher processing rate than others (e.g., GTX980).

F. Large-scale Experiment

Large-scale experiments were performed on *Cartesius* for the bioinformatics application since it requires the largest cache slot

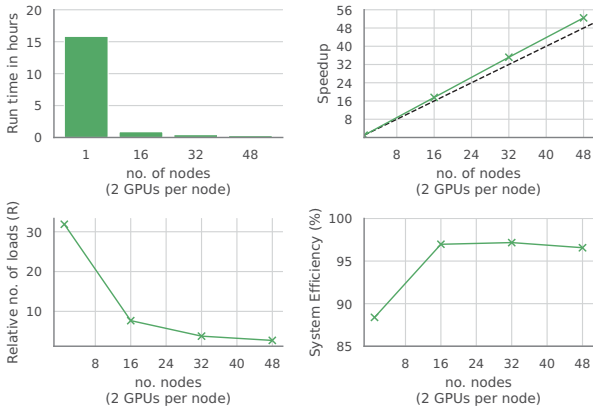

Fig. 15. Results for the large-scale experiment on 96 GPUs (see Section VI-F).

size, thus making it the most difficult application to maximize data reuse. The input data set consists of all available reference bacteria proteomes (6818 files) from the Uniprot Proteomes database [24] as of March 2020.

Figure 15 shows the run time, speedup, R, and system efficiency when scaling from 1 node (2 GPUs) to 48 nodes (96 GPUs). Run times decrease from 16 hours on one node to just under 20 minutes with 96 GPUs. Super-linear speedup is present even on 96 GPUs which, as explained previously, is due to the distributed cache that exploits the larger combined memory capacity of the nodes. The number of loads decreases by a factor $11.8\times$ from $R = 31.9$ for 1 node to just $R = 2.7$ for 48 nodes.

VII. Future Work

In this section we discuss several directions for future work. We are working on extending the generality of our solution and cover more types of parallel applications that involve data reuse. For example, applications with more complex workloads, such as processing triples (or any n-tuple) or using user-defined heuristics to reduce the number of pairs. We are also considering applications that have more complex pipelines consisting of many phases and using different accelerators (e.g., FPGAs, APUs, co-processors). An exciting direction is to extend the work-stealing algorithm with some form of cache-awareness such that remote tasks are chosen based on locally available data, thus enabling more reuse.

Furthermore, extending the caching design would also present many opportunities. For example the ability to cache different items at different levels; persistent caches that reuse data from previous runs for the next execution; or including novel memory technologies (e.g., NVM, flash storage). We are also working on solutions that enable variable-sized cache slots instead of fixed-sized ones.

Finally, other interesting system aspects we did not consider in this paper are fault-tolerance, energy consumption, elasticity, cloud environments, or multi-cluster computing.

VIII. Conclusions

In this paper, we have studied the problem of all-pairs compute problems. Our solution combines multi-level caches to exploit data reuse; random work-stealing to allow dynamic workload balancing; a divide-and-conquer approach to exploit data locality; and asynchronous processing to overlap computation and data movement at all levels. The implementation of our framework, *Rocket* [27], is available online.

We performed a detailed evaluation with three different real-world scientific applications on different platforms: from a single node, to a medium-scale heterogeneous cluster, and finally to a large-scale supercomputer containing 96 GPUs. Results show that we achieve excellent scalability to multiple nodes, often showing super-linear speedup thanks to the distributed cache. Moreover, we demonstrate perfect load balancing even on a highly heterogeneous platform. Our results demonstrate, for example, that with Rocket we can reconstruct the evolutionary tree of all reference bacteria proteomes on Uniprot in under 20 minutes using a supercomputer. We conclude that our Rocket framework is easy to use, and enables extremely efficient execution of all-pairs applications on large-scale systems, often achieving super-linear speedups.

Acknowledgment

This project has received funding from the Netherlands eScience Center under file number 027.016.G06 (*A methodology and ecosystem for many-core programming*) and the European Unions Horizon 2020 research and innovation programme under Grant Agreement 777533 (*PROCESS*).

References

[1] R. V. van Nieuwpoort and J. W. Romein, "Correlating Radio Astronomy Signals with Many-Core Hardware," *Int J Parallel Prog*, vol. 39, no. 1, pp. 88–114, Feb. 2011.

[2] H. Heydarian, F. Schueder, M. T. Strauss, B. van Werkhoven, M. Fazel, K. A. Lidke, R. Jungmann, S. Stallinga, and B. Rieger, "Template-free 2D particle fusion in localization microscopy," *Nat Methods*, vol. 15, no. 10, pp. 781–784, Oct. 2018.

[3] J. Qi, B. Wang, and B.-I. Hao, "Whole Proteome Prokaryote Phylogeny Without Sequence Alignment: A K -String Composition Approach," *Journal of Molecular Evolution*, vol. 58, no. 1, pp. 1–11, Jan. 2004.

[4] B. van Werkhoven, P. Hijma, C. J. H. Jacobs, J. Maassen, Z. J. M. H. Geradts, and H. E. Bal, "A Jungle Computing approach to common image source identification in large collections of images," *Digital Investigation*, vol. 27, pp. 3–16, Dec. 2018.

[5] P. Phillips, P. Flynn, T. Scruggs, K. Bowyer, J. Chang, K. Hoffman, J. Marques, J. Min, and W. Worek, "Overview of the face recognition grand challenge," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, Jun. 2005, pp. 947–954 vol. 1.

[6] O. Maimon and L. Rokach, *Data mining and knowledge discovery handbook*. Springer, 2005.

[7] S. Zhu, A. Potapova, M. Alabduljalil, X. Liu, and T. Yang, "Clustering and load balancing optimization for redundant content removal," in *Proceedings of the 21st International Conference on World Wide Web*, 2012, pp. 103–112.

[8] C. Liu, B. Petroski, G. Cordone, G. Torres, and S. Schuckers, "Iris matching algorithm on many-core platforms," in *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, 2015, pp. 1–6.

[9] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, Mar. 1995.

[10] C. J. Kleinheksel and A. K. Somani, "Efficient Distributed All-Pairs Algorithms: Management Using Optimal Cyclic Quorums," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 391–404, Feb. 2018, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[11] Y.-F. Zhang, Y.-C. Tian, W. Kelly, and C. Fidge, "Distributed computing of all-to-all comparison problems in heterogeneous systems," in *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society*, Nov. 2015, pp. 002 053–002 058.

[12] V. K. V. Yeleswarapu and A. K. Somani, "A Memory Efficient Parallel All-Pairs Computation Framework: Computation – Communication Overlap," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer International Publishing, 2018, pp. 443–458.

[13] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 1, pp. 33–46, Jan. 2010.

[14] D. Li, K. Sajjapongse, H. Truong, G. Conant, and M. Becchi, "A distributed CPU-GPU framework for pairwise alignments on large-scale sequence datasets," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, Jun. 2013, pp. 329–338.

[15] Y.-F. Zhang, Y.-C. Tian, C. Fidge, and W. Kelly, "Data-aware task scheduling for all-to-all comparison problems in heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 93-94, pp. 87–101, Jul. 2016.

[16] T. Gloe and R. Böhme, "The 'Dresden Image Database' for benchmarking digital image forensics," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. Sierre, Switzerland: Association for Computing Machinery, Mar. 2010, pp. 1584–1590.

[17] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal, "Ibis: An efficient Java-based grid programming environment," in *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, ser. JGI '02. Seattle, Washington, USA: Association for Computing Machinery, Nov. 2002, pp. 18–27.

[18] J. Maassen, N. Drost, H. E. Bal, and F. J. Seinstra, "Towards jungle computing with Ibis/Constellation," in *Proceedings of the 2011 Workshop on Dynamic Distributed Data-Intensive Applications, Programming Abstractions, and Systems - 3DAPAS '11*. San Jose, California, USA: ACM Press, 2011, p. 7.

[19] J. Maassen, S. Verhoeven, J. Borgdorff, J. H. Spaaks, N. Drost, C. Meijer, A. van der Ploeg, P. T. de Boer, R. van Nieuwpoort, B. van Werkhoven, and A. Kuzniar, "Xenon," Zenodo, Mar. 2020.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[21] R. V. van Nieuwpoort, G. Wrzesinska, C. J. H. Jacobs, and H. E. Bal, "Satin: A high-level and efficient grid programming model," *ACM Trans Program Lang Syst*, vol. 32, no. 3, pp. 1–39, 2010.

[22] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 501–510, 2008.

[23] J. Fridrich, "Sensor Defects in Digital Image Forensic," in *Digital Image Forensics: There Is More to a Picture than Meets the Eye*, H. T. Sencar and N. Memon, Eds. New York, NY: Springer, 2013, pp. 179–218.

[24] U. Consortium, "UniProt: A worldwide hub of protein knowledge," *Nucleic Acids Res*, vol. 47, no. D1, pp. D506–D515, Jan. 2019.

[25] B. Jian and B. C. Vemuri, "Robust point set registration using gaussian mixture models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 8, pp. 1633–1645, 2010.

[26] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

[27] S. Heldens, P. Hijma, B. van Werkhoven, J. Maassen, and R. van Nieuwpoort, "Rocket: Runtime system for all-pairs computations," Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3878159

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Experiments were performed using Rocket software artifact on two systems: DAS5 cluster at VU University and the Cartesisus Supercomputer. The paper presents results for four scenarios: - 1 node with 1 NVIDIA TitanX GPU from DAS5 cluster - 16 homogeneous nodes with each 1 NVIDIA TitanX GPU from DAS5 cluster - 4 heterogeneous nodes with different GPUs: node026, node027, node028, node029 from DAS5 cluster. - 48 homogeneous nodes, each having 2x NVIDIA K40m GPU, from Cartesius supercomputer.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪ https://github.com/JungleComputing/rocket
Artifact name: Rocket software

Persistent ID: 10.5281/zenodo.3878159
Artifact name: Rocket software (Permanent archive)
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: DAS5 cluster at VU University and Cartesisus Supercomputer

Operating systems and versions: DAS5: CentOS Linux release 7.4.1708; Cartesius: Red Hat Enterprise Linux Server release 7.7

Compilers and versions: DAS5: OpenJDK Runtime Environment 18.9 (build 11.0.3+7-LTS), CUDA release 10.0 V10.0.130; Cartesius: OpenJDK Runtime Environment 18.9 (build 11.0.2+9), CUDA release 10.0 V10.0.130

Libraries and versions: Constellation 2.0.1, Ibis-IPL 2.3.3, libjpeg 62.1.0

Key algorithms: common-source identification, localization microscopy particle fusion, phylogeny tree construction

Input datasets and versions: common-source identification: images 3648x2736 from Dresden image database, localization microscopy particle fusion: 256 randomly generated particles, phylogeny tree construction: 2500 randomly chosen reference bacteria from Uniprot Proteomes Database