AlloX: Compute Allocation in Hybrid Clusters

Tan N. Le[1,2], Xiao Sun[1], Mosharaf Chowdhury[3], and Zhenhua Liu[1]
[1]Stony Brook University [2]SUNY Korea [3]University of Michigan

Abstract

Modern deep learning frameworks support a variety of hardware, including CPU, GPU, and other accelerators, to perform computation. In this paper, we study how to schedule jobs over such *interchangeable* resources – each with a different rate of computation – to optimize performance while providing fairness among users in a shared cluster. We demonstrate theoretically and empirically that existing solutions and their straightforward modifications perform poorly in the presence of interchangeable resources, which motivates the design and implementation of AlloX. At its core, AlloX transforms the scheduling problem into a min-cost bipartite matching problem and provides dynamic fair allocation over time. We theoretically prove its optimality in an ideal, offline setting and show empirically that it works well in the online scenario by incorporating with Kubernetes. Evaluations on a small-scale CPU-GPU hybrid cluster and large-scale simulations highlight that AlloX can reduce the average job completion time significantly (by up to 95% when the system load is high) while providing fairness and preventing starvation.

ACM Reference Format:
Tan N. Le[1,2], Xiao Sun[1], Mosharaf Chowdhury[3], and Zhenhua Liu[1]. 2020. AlloX: Compute Allocation in Hybrid Clusters. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387547

1 Introduction

In recent years, deep learning (DL) is gaining rapid popularity in various application domains, including computer vision, speech recognition, etc. Deep learning training is typically compute-intensive. As a result, GPUs are more popular in deep learning clusters. Nonetheless, CPUs can still be used to perform such computation, albeit at a slower speed. At the same time, Google has developed Tensor Processing Units (TPUs) for its AI systems [39], while Microsoft is relying on FPGAs [24]. Indeed, most modern frameworks support heterogeneous computation devices [9, 12, 45, 56].

Given that they are all computation hardware, it is possible to use them in an *interchangeable* manner by maintaining multiple configurations in modern cluster managers such as Kubernetes [6]. For example, if a job typically has a CPU configuration (1 CPU, 12 GB Memory), the same job can have another GPU configuration such as (1 GPU, 2GB Memory).

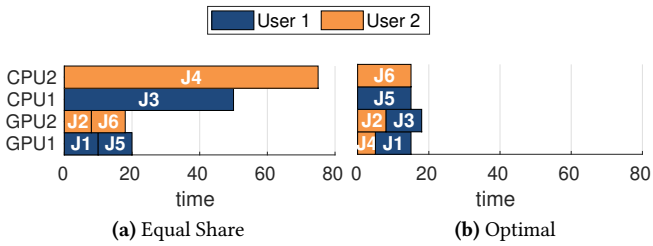
The central problem we address in this paper is *how to pick the configuration for each job and order the jobs* to optimize performance objectives such as the average job completion time while providing fairness among multiple users. Furthermore, this needs to be done in an online manner with minimal user effort.

Let us consider a simple example in Table 1, where two users share a small cluster of two CPUs and two GPUs, to illustrate the crux of the problem. Each user has 3 jobs queued up at the beginning that can be processed on either CPUs or GPUs with the processing times (PT).

Figure 1 compares an existing solution and the optimal solution for this example. Equal Share in Figure (1a) represents a typical fair sharing solution used in modern cluster managers [6, 34] that is unaware of resource interchangeability and picks resources for jobs in a fair manner across all resources. Essentially, it divides CPUs and GPUs equally between the two users and schedules the jobs in a First-Come-First-Served (FCFS) manner. When it is the turn for a

Table 1. A motivating example. PT stands for processing time in minutes.

User	Job ID	PT on GPU	PT on CPU
User 1	J1	10	15
User 2	J2	8	10
User 1	J3	10	50
User 2	J4	5	75
User 1	J5	10	15
User 2	J6	10	15



(a) Equal Share **(b)** Optimal

Figure 1. Inefficiency from ignoring interchangeability.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387547

particular job to run, this job picks its "favorite" resource (the one resulting in a shorter processing time) if that resource is available, otherwise it picks the other resource. Overall, the average job completion time using this approach is $\frac{181}{6}$ minutes. In contrast, Figure (1b) illustrates the optimal solution. By judicious scheduling, the average job completion time is reduced to $\frac{76}{6}$ minutes, a 59% improvement. The makespan is also reduced by 76%. Similar poor behavior is demonstrated by extensions of existing solutions as well (§2.2).

The key challenge, however, is achieving such benefits in practice. Indeed, there are significant algorithmic and systems challenges in the presence of multiple job configurations. From the algorithmic perspective, while minimizing the average job completion time is relatively easy when jobs have only one configuration [42], we prove that the problem is APX-hard in this context.[1] At the same time, while the Dominant Resource Fairness (DRF) allocation and its variants [17, 27–29, 51, 63] provide desirable properties, there exists a hard trade-off among the fundamental properties with multiple job configurations; we show that DRF fails to maintain most of its properties in the presence of interchangeable resources. Finally, we note that job scheduling over interchangeable resources more challenging than existing work on heterogeneous resources [29, 57, 68], where jobs are assumed to have the same speedup when running on different resources.

From a systems design perspective, existing systems heavily rely on users to provide key information such as which configuration to use, even though many users may not have the expertise or system-level insights to do so. Even if they do, the best configuration for a given job still depends on the presence of other jobs in the cluster. To this end, we want to design a solution that can automatically pick the best configuration for a given job at a given point in time.

Overall, in this paper, we make the following contributions in tackling these challenges by designing AlloX.

- Motivated by experimental results on a real system, we identify a new job scheduling and resource allocation problem and analyze the inefficiencies of existing solutions (§2). Specifically, we show that most existing solutions may lead to *arbitrarily* poor performance when jobs have multiple configurations.
- We design AlloX to optimize performance and provide dynamic fair allocation (§3). Our key idea is to transform the multi-configuration job scheduling problem into a min-cost bipartite matching, which can be solved in polynomial time. It provides the optimal solution in simplified settings and outperforms all baselines significantly in general settings. AlloX dynamically schedules jobs from the users that are furthest from their fair share.
- We implement AlloX on Kubernetes (§4). Besides the scheduling algorithm, AlloX profiles jobs in an online manner to automatically decide job configurations and estimate job processing times. Both are necessary inputs for the scheduling algorithm.
- We conduct experimental and numerical evaluations to show AlloX's performance improvements using TensorFlow workloads (§5). Results highlight that AlloX reduces the average job completion time significantly, provides fairness among users, and prevents starvation.

[1] An APX-hard problem is an NP-hard problem that does not have any efficient approximation solution.

2 Background & Motivation

2.1 Interchangeable Resources

Interchangeability at the application level. Frameworks like Tensorflow [9], PyTorch [52] and Caffe [2] are capable of leveraging both CPUs and GPUs. To support interchangeability at the application level, the frameworks need to have a simple configuration. For example, they need to indicate all variables and operations on GPUs (or CPUs) in Tensorflow. Since accelerators like GPUs are becoming popular for data-intensive and compute-intensive applications, we believe that there will be more frameworks that support the interchangeability in the near future. Furthermore, heterogeneous resources have interchangeability and do not need any modification at the application level. Table 2 summarizes the interchangeability on various resources.

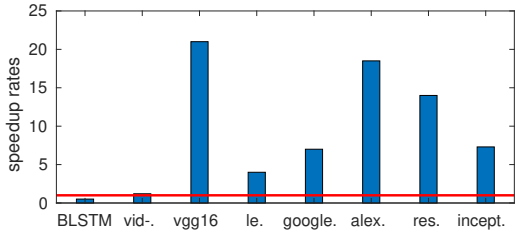
Table 2. Interchangeability support in frameworks for different computation resources.

Resources	Frameworks
CPU & GPU	Tensorflow, Caffe, PyTorch, Matlab, Chainer [58], TVM [16]
FPGA & GPU	CNNLab [70], PaddlePaddle [8], TVM [16]
TPU, CPU, & GPU	Tensorflow, TVM [16]

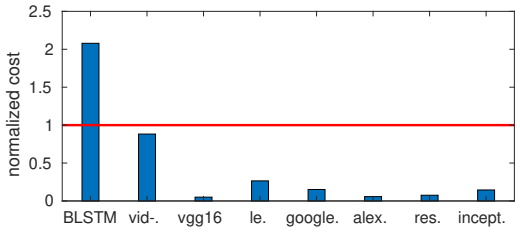
Distinct speedup rates for different applications. Although GPUs are in wide use for different deep learning applications [18, 19, 21, 22, 30], different jobs obtain distinct speedups by using GPUs w.r.t. CPUs (Figure (2a)). The speedup rate is how much GPU can reduce the job processing time, i.e., job processing time on a particular CPU divided by that on GPU.

While GPUs are generally more efficient for machine learning jobs (with a speedup rate larger than 1), they are more expensive. Figure (2) shows that the normalized costs (the cost ratio between GPUs and CPUs divided by the speedup) vary a lot. When the normalized cost is greater than 1, using GPU is not cost-effective.

The ineffectiveness of using GPUs in some jobs is due to several reasons. First, modern frameworks such as TensowFlow are not good at speeding up memory networks like



(a) Speedups from Nvidia K80 GPU versus Intel Xeon E5 2.4 GHz 20-core CPU.



(b) Normalized costs using GPU node (p2.xlarge) versus CPUs (c5.large) on EC2.

Figure 2. GPUs provide distinct speedups and costs. When GPUs are overloaded, we should move workload with low speedup rates to CPUs.

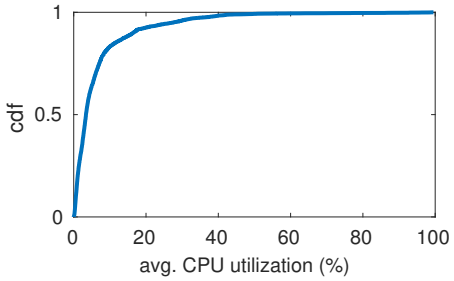


Figure 3. Most of Microsoft Azure users (92.6% of 5,958 users) have average CPU utilization under 20%.

Bidirectional LSTM (Bi-LSTM) [4]. RNN models are often updated for each training example for the dependency between two timeframes, which creates difficulty for parallel computing [37]. Second, for the large data input like video-analytics (vid.) [7], GPUs are not effective when the bottleneck is in memory and I/O for processing the large data.

GPUs can be fast but they are expensive, while CPUs are available and abundant. In addition to being cost-effective for some jobs, CPUs are often under-utilized in large clusters. To this end, we analyzed the Azure Public Dataset [1, 23], which recorded CPU utilization of 5,958 users over 30 days. We found that more than 90% users use less than 20% of their allocated CPU (Figure 3). Because jobs can be executed on CPUs when GPUs are busy, we could utilize the available CPUs before spending a lot to add more, expensive GPUs.

2.2 Inefficiencies of Existing Solutions

While existing cluster schedulers do not support interchangeable resources (Table 3), it is not trivial to extend existing schedulers to handle both performance and fairness in the presence of multiple configurations. Usually, jobs are pre-configured to run on either CPUs or GPUs, therefore systems do not have the flexibility to determine whether to place a job on CPU or GPU.

Table 3. Popular resource managers with GPU support.

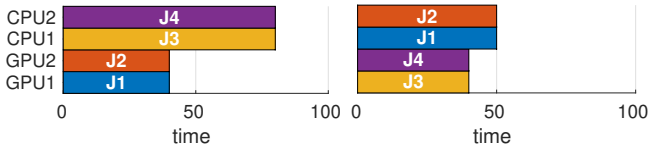
Systems	Algorithm	Interchangeability
YARN [60]	Fair, DRF	No
Kubernetes [6]	Best effort	No
Mesos [34]	DRF	No
Spark [67]	Fair	No

Consider a simple setup where n users share a system consisting of interchangeable resources and each user submits their jobs over time. Each job has up to k configurations to run. For simplicity of presentation, we restrict our attention to two configurations: CPU and GPU; however, our algorithm and analysis can be readily extended to more configurations, and even additional scenarios such as networking interfaces or storage devices.

A job scheduler for interchangeable resources needs to decide (i) the configuration to use for each job and (ii) the order of jobs to optimize performance and fairness objectives. While the latter has been studied extensively in recent works [17, 27, 28, 31, 32], the former is a new challenge. In this section, we revisit existing algorithms and their straightforward extensions for this new problem to illustrate their inefficiencies and demonstrate why we need new algorithms.

Best Fit (BF). As performance is often the main focus, it is natural to pick the configuration (CPU or GPU) for each job that gives the best performance, also known the Best Fit (BF) algorithm. The problem is that the load on CPU and GPU can be largely unbalanced. For example, GPUs can be overloaded resulting in huge waiting time, while CPUs are little used. The imbalance has a profound impact on job completion time, especially when the system load is high. Therefore, *picking the most effective configuration for each job may result in low utilization and high job waiting time.* To deal with this problem, the interchangeability scheduler must enable fall-back from overloaded GPUs to CPUs and vice versa. The challenge is how to do it efficiently.

Join the Shortest Queue (JSQ+). The inefficiency of BF comes from the lack of the consideration of the system load in the configuration selection. Therefore, a better approach could be a modified Join the Shortest Queue (JSQ+). Assume users know the current waiting time on each resource in real time. On the arrival of a job, the scheduler picks the resource that has the shortest completion time, i.e., the sum of waiting



(a) JSQ+ **(b)** Optimal

Figure 4. Inefficiency of JSQ+.

(a) SJF+ **(b)** Optimal

Figure 5. Inefficiency of SJF+.

time and the processing time of the job on the corresponding resource. In this way, loads can be more balanced because as the load on some resource increases, its longer waiting time would force new jobs to be placed on other resources.[2]

The key drawback of JSQ+ is that it is short-sighted: each job attempts to minimize its own completion time without considering its impact on later jobs. Consider an example with 2 CPUs and 2 GPUs. Assume there are 4 jobs, all arrive at the beginning but in the order of Job 1, 2, 3, 4. The processing time can be shown in a matrix:

$$P = \begin{bmatrix} 40 & 50 \\ 40 & 50 \\ 40 & 160 \\ 40 & 160 \end{bmatrix}$$

In this matrix, the i-th row consists of job i's processing times on GPUs and CPUs, respectively. Under JSQ+, Job 1 first picks a GPU, and then Job 2 picks the other GPU. After that, Job 3 and Job 4 have no choices but to pick CPUs, as shown in Figure 4a with an average completion time (AJCT) of 100 minutes. Clearly, the optimal solution is to put Job 3 and 4 on GPUs and Jobs 1 and 2 on CPUs, which can reduce the AJCT from 100 minutes to 45 minutes as Figure 4b.

Shortest Job First (SJF). The sub-optimality of BF and JSQ+ implies that we cannot just let each user pick her own job configurations. Therefore, the scheduler needs to coordinate the decisions, where Shortest Job First (SJF) [20] is widely used.

When there are multiple configurations for each job, we can extend SJF to SJF+ to handle jobs with multiple configurations: for each type of resource, maintain a queue of all available jobs. The jobs are sorted based on the processing time on this resource in an increasing order. Whenever a resource becomes available, schedule the first job in the corresponding queue and remove the job from all queues. When multiple resources become available simultaneously, first schedule the job with the shortest processing time.

While SJF is optimal for only 1 configuration [20], its performance can be *arbitrarily bad* for multiple configurations. Consider the following processing time matrix:

[2]JSQ+ is different than the vanilla JSQ because the processing time of the same job on different resources varies.

$$P = \begin{bmatrix} 10 & 20 \\ 10 & 20 \\ 20 & 90 \\ 20 & 90 \end{bmatrix}$$

Under SJF+, there are two queues for GPU and CPU, respectively. The order in both queues is Job 1, Job 2, Job 3, Job 4. Therefore, Job 1 and 2 are placed on GPUs first. Then Job 3 and 4 are scheduled on CPUs. This is shown in Figure 5, resulting in an AJCT of 50 minutes. In contrast, the optimal solution places Jobs 3 and 4 on GPUs and Jobs 1 and 2 on CPUs, reducing the AJCT to 20 minutes. The root cause is while Jobs 3 and 4 are longer (disadvantage in SJF+), the processing time reduction of using GPU is much larger (overlooked by SJF+).

Summary. When jobs have multiple configurations, even if jobs arrive at the same time, the problem is more challenging than that with single configuration because we need to consider the processing time reduction among configurations, which may contradict with other factors, e.g., the length of the job. Therefore, algorithms that perform well for single configuration job scheduling may result in arbitrarily bad performance in the new problem.

3 Algorithm Design

In this section, we design a scheduler that works with applications that can run on interchangeable resources, e.g., CPUs and GPUs. For applications that only can run either on CPUs or GPUs, we can assume that they take infinite time to complete on the non-executable resource.

Minimizing the average job completion time with multiple configurations is APX-hard by a reduction from a maximum bounded 3-dimensional matching problem. The details of proof can be found in Section 3 of [35]. In other words, unless $P = NP$, there is no polynomial-time approximation algorithm with the approximation ratio bounded by a constant. Therefore, we start with a simpler case where we can design a polynomial-time optimal algorithm, and then extend the algorithmic idea to the general case.

3.1 Optimal Approach for Queued Up Jobs

Assume all jobs arrive at the beginning and each job has one configuration for CPU and one for GPU. If a job can only run on GPU or CPU, we can set the processing time on the

other resource to be a very large number. We assume that each job uses either an entire GPU or an entire CPU, which is further discussed in Section 4.

For this simplified problem, we can transform the scheduling and placement problem to a min-cost bipartite matching problem, which can be solved efficiently [13, 25, 36]. Specifically, our algorithm consists of three steps: (i) generate input for the min-cost bipartite matching problem based on job information; (ii) solve the matching problem to obtain a solution; (iii) convert the solution to a feasible scheduling and placement.

i. Generate input for the matching problem. Our observation is that for each resource (a CPU or a GPU), a job scheduled as the k-th last one contributes k times its processing time to the total job completion time. Assume we have three jobs of sizes 3, 4, and 5 to be scheduled on one resource. If we schedule in the order, their completion times are 3, 7, and 12, resulting in a total completion time of 22. There is another way to calculate the total completion time based on the waiting times. As the Job 1 is scheduled to be the first one, i.e., there are two jobs waiting, it contributes three times (two from waiting times of Jobs 2 and 3, and one from Job 1's own processing time), which is 9. Similarly, Job 2 contributes twice its processing time, which is 8. Job 3 is the last job and contributes only its processing time 5. The sum is also 22.

This observation allows us to obtain the contribution of a job placed on machine i as the k-th last job to the total completion time. Consider a simple example with 2 machines (1 CPU and 1 GPU) shared by 3 jobs. The job processing time can be represented by the following processing time matrix P where each row contains processing time on CPU and GPU. In this matrix, the first row represents that Job 1 takes 3 minutes on GPU or 4 minutes on CPU. The size of P is $n \times m$ for n jobs over m machines.

$$P = \begin{bmatrix} 3 & 4 \\ 4 & 6 \\ 5 & 10 \end{bmatrix}$$

Based on the processing time matrix P, we can generate the following cost matrix Q of size $n \times (nm)$:

$$Q = \begin{bmatrix} P & 2P & \cdots & nP \end{bmatrix}.$$

For our example,

$$Q = \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{matrix} (G,1) & (C,1) & (G,2) & (C,2) & (G,3) & (C,3) \end{matrix} \\ \begin{pmatrix} 3 & 4 & 6 & 8 & 9 & 12 \\ 4 & 6 & 8 & 12 & 12 & 18 \\ 5 & 10 & 10 & 20 & 15 & 30 \end{pmatrix} \end{array}$$

The element (j, i, k), corresponding to $\big(j, m*(k-1)+i\big)$ in the matrix, represents the cost of scheduling job j at machine i (1 stands for GPU, and 2 stands for CPU) as k-th last job. For example, the entry $(2, 2, 3)$ represents Job 2 contributes 12 minutes processing time if it is placed as the third last job on GPU.

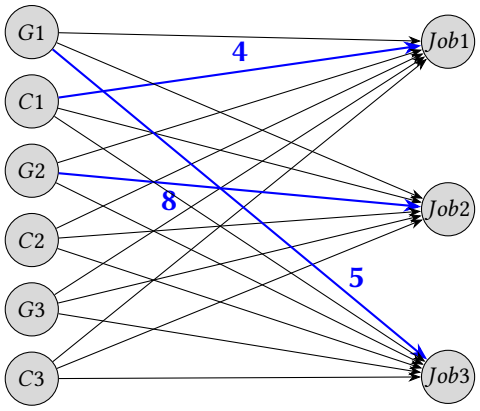


Figure 6. The corresponding min-cost bipartite matching.

ii. Solve the matching problem. Given the matrix Q, we can formulate the problem into a min-cost bipartite matching problem in Figure 6 as follows.

On the right side of the bipartite graph, each node represents a job. For our example, there are three jobs. Each job has a demand of 1 unit. On the left side, each node represents a position on a machine. As we have two machines (one CPU and one GPU) and 3 jobs, we need at most three positions for each machine. For instance, G2 represents the second last position on GPU. Because each position on a machine can serve one job at most, it has a supply of 1 unit.

Each edge has a capacity of 1 and there is a one-to-one correspondence between the cost of using that edge and the entry from the matrix Q we generated. For example, the cost from $G2$ to node $Job2$ is the entry at the second row and $(G, 2)$ column, which is 8.

This min-cost bipartite matching problem can be solved in polynomial time with standard network flow algorithms or Hungarian method [41, 50]. In the example, the optimal cost is 17 and the matching is shown in a matrix M. Three highlighted edges in Figure 6 are active to feed the demand.

$$M = \begin{array}{c} \\ \\ \\ \end{array} \begin{array}{c} \begin{matrix} (G,1) & (C,1) & (G,2) & (C,2) & (G,3) & (C,3) \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

iii. Convert the matching solution to job scheduling. The solution from the matching problem is converted as follows. Each edge picked by the matching algorithm correspond to the scheduling and placement of a job. For instance, the edge between $Job2$ and $G2$ is picked, meaning Job 2 is scheduled on GPU as the second last job. Similarly, $Job3$ is connected to $G1$, meaning it is scheduled on GPU as the last job. $Job1$ is connected to $C1$, so it is scheduled on CPU as the last job. Combined the information, our algorithm places Job2 and Job 3 on GPU and Job 2 is scheduled before Job 3,

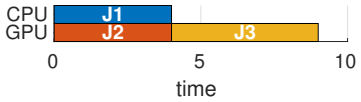


Figure 7. The scheduling from the corresponding min-cost bipartite matching problem. The total completion time is 17.

while Job 1 is placed on CPU as the only job. This scheduling and placement are shown in Figure 7.

Optimality. We have the following formal result of the performance of this algorithm.

Lemma 3.1. *When all jobs arrive at the same time, the optimal solution can be found in polynomial time by solving the corresponding min-cost bipartite matching problem.*

The proof sketch is described below. A formal proof can be found in the appendix. For the problem transformation, each scheduling and placement solution corresponds to a feasible solution of the matching problem. To see this, each job has been placed on one and only one machine, meaning the demand of each job node in the matching problem has been met. In addition, no jobs have the same order at the same machine, so the supply of each (machine, order) node in the matching problem is at most 1.

Conversely, any feasible solution to the matching problem corresponds to an extended scheduling problem with possibly dummy jobs. To do this, we first place jobs based on the edges picked by the matching problem. Then we fill each gap in the positions by a dummy job. Clearly, the insertion of dummy jobs always increases the total completion time, so there is no dummy job in the optimal solution. Therefore, the optimal solution of the matching problem corresponds to the optimal scheduling and placement solution.

Algorithm 1 Primitive AlloX without online arrivals

```
Generate the cost matrix Q
Solve the min-cost matching problem defined by Q to get the matching matrix M
for j = 1 : n do                                   ▷ n is the total # of jobs
    for i = 1 : m do                               ▷ m is the total # of machines
        for k = n : 1 do
            if M(j, m(k − 1) + i) = 1 then         ▷ Job j is scheduled on machine i as the k-th last job
                Add job j to the queue from machine i
            end if
        end for
    end for
end for
```

3.2 Handling Online Arrivals

The algorithm described above requires all jobs to arrive at the beginning. Here, we extend our idea in the previous section to incorporate arrivals by updating the scheduling and placement over time. This can be done whenever a resource becomes available or periodically.

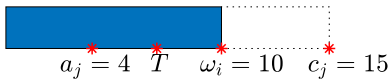


Figure 8. An illustration example that shows the impacts of online arrivals and available time of a machine.

We focus on the case where preemption is not allowed, because preemption is not well supported in many systems such as Kubernetes. Even if it is supported, the overhead of migration is often very high. We assume that the scheduler has no information about future arrivals. The major difference with arrivals over time is that when we generate a new schedule, some machines are occupied so new jobs need to wait until current jobs finish. We use both the arrival time of new jobs and the available time of machines (defined shortly) to adjust the cost matrix Q for the matching problem.

Consider the example in Figure 8. Assume we generated a schedule at time 0 and a job was placed on the machine i, which is expected to finish at time 10 based on our estimation detailed in Section 4.1. In other words, the available time of machine i is $\omega_i = 10$. Job j arrives at time 4 but the scheduler was not triggered at that time. At time $T = 6$, we want to update the schedule, while machine i is still busy. For Job j, if it is scheduled as the next job on machine i,its (expected) completion time comes from two parts: a waiting time of $(\omega_i - a_j) = 6$ and the processing time $p_{ji} = 5$. This gives an expected completion time of 11.

Motivated by this, we define delay matrix $D(j, i) = \omega(i) - a(j)$, where $a(j)$ is the arrival time of job j, and $\omega(i)$ is the earliest available time of machine i when it finishes its currently allocated job(s). If machine i is idle, then $\omega(i) = T$, where T is the current time. The cost matrix Q is calculated by the following: $Q = \begin{bmatrix} P & 2P & \cdots & nP \end{bmatrix} + \begin{bmatrix} D & D & \cdots & D \end{bmatrix}$, where P is processing time matrix. The processing time can be estimated with more details in Section 4.1 and we evaluated the impacts of estimation errors in Figure 17. Delay matrix D is used to handle the non-preemption constraint.

3.3 Incorporating Fairness

3.3.1 Existing Fair Allocation Algorithms are Insufficient

When multiple users share the same cluster, fairness is often important in order to provide performance isolation and to avoid starvation. In traditional multi-resource allocation problem, where each job has only one configuration, there are four main properties [28]:

- *Efficiency (PE)*: No user can increase her performance without hurting the performance of at least another user.
- *Sharing Incentive (SI)*: Each user is no worse by sharing than using $\frac{1}{n}$ of the system resources exclusively.
- *Envy-Freeness (EF)*: No user prefers the allocation of another user for better throughput.

- *Strategy-proofness (SP)*: No user is able to benefit by lying.

DRF [28] satisfies all four properties when each job has only one configuration. Intuitively, one might expect to extend DRF to the allocation of interchangeable resources. One straightforward extension is to pick the resource configuration with the shortest processing time for each job and then use DRF to allocate the resources by ignoring the interchangeability among resources, e.g., taking CPUs and GPUs as different resource types. Unfortunately, this does not work. Formally, we have the following lemma regarding this DRF extension.

Lemma 3.2. *If each job picks the configuration with shorter processing time, there exist cases where DRF fails to provide PE, SI or SP under multiple configurations.*

Surprisingly, there is a hard tradeoff among these basic properties. More details are in our preliminary theoretical work [55]. In particular, we can show for any allocation, if it provides sharing incentive and Pareto efficiency, it cannot be strategyproof. Conversely, if it is strategyproof, sharing incentive and Pareto efficiency cannot be provided simultaneously. There is one exception: if all jobs have the same speedup by using GPUs, then the problem degenerates to a traditional multi-resource allocation, where DRF can be applied. However, it is not true in practice as shown in Figure 2a. Formally, we have the following impossibility result.

Lemma 3.3. *No multi-configuration allocation can satisfy (i) PE and SI, and (ii) SP simultaneously unless the relative efficiency of CPU and GPU is the same for all jobs.*

The proofs of Lemmas 3.2 and 3.3 are included in the appendix.

Now we briefly discuss the intuition behind this lemma. Consider the environment with CPU, GPU and memory defined above. Two users i and j have the true relative efficiency of GPU compared to CPU g_i and g_j, we assume $g_i < g_j$. Let the reported relative efficiency be $\tilde{g}$. PE implies that user i should utilize CPU first while user j utilizes GPU first. SI requires that both users get at least $\frac{1}{2}(1+g_i)$ and $\frac{1}{2}(1+g_j)$ computation, respectively. As a consequence, user i can report a $\tilde{g}_i$ that $\tilde{g}_i = \tilde{g}_g - \delta_1$ for small $\delta_1 > 0$ to get more resources, which is ensured by SI. User j can also manipulate its demand to counter i's movement by lowering its $\tilde{g}_j$ to $\tilde{g}_j = \tilde{g}_i + \delta_2$ for small $\delta_2 > 0$. As a consequence, there does not exists a point $(\bar{g}_i, \bar{g}_j)$ where both users are satisfied.

3.3.2 Our Idea

AlloX maintains a *progress* for each user over time, which is defined in the following way. For job i, denote its CPU configuration by (c, m_c, p_c), where c is the CPU demand, m_c is the memory demand, and p_c is the processing time on CPU, and its GPU configuration by (g, m_g, p_g), where g is the GPU demand, m_g is the memory demand, and p_g is the processing time on GPU. If p_c is smaller than p_g (CPU is more effective), we use the dominant share of the CPU configuration $d_i = \max\{c/C, m_c/M\}$, where C and M are the CPU and memory capacity of the cluster. The dominant share of CPU configuration is the maximum of normalized CPU usage and normalized memory usage. Otherwise, we use the dominant share of the GPU configuration $d_i = \max\{g/G, m_c/M\}$, where G is the GPU capacity of the cluster.

The value of the job i is the dominant share d_i at the time of scheduling discounted if the job is not placed on the more effective resource. For instance, if job i is more effective on GPU but is placed on CPU, its value $v_i = \frac{p_g}{p_c} d_i = \frac{p_g}{p_c} \max\{g/G, m_g/M\}$, where $\frac{p_g}{p_c}$ is the discount factor. A user's progress is the sum of values of all her jobs that are currently running.

Consider a simple example where the job has CPU configuration $(1, 4, 20)$ and GPU configuration $(1, 2, 5)$. The system capacity is $(8, 2, 64)$ for CPU, GPU, and memory. Because GPU provides a shorter processing time, the dominant share of the job is $\frac{1}{2}$. If the job is actually scheduled on GPU, its value is $\frac{1}{2}$. Otherwise, it is discounted by $\frac{1}{4}$ (to $\frac{1}{8}$) because running on CPU is 4 times slower than GPU. Over the execution of the job, either on CPU or GPU, its aggregated value is the same. Actually, the progress of a user can be viewed as her instantaneous throughput.

3.3.3 Incorporating Fairness into AlloX

AlloX provides a fairness knob and the system operator can adjust its value α in $[0, 1]$, which affects how many users are taken into consideration when a scheduling is triggered. For instance, with 20 users and $\alpha = 0.3$, only jobs from 6 users with lowest progress are considered for scheduling. When $\alpha = 1$, there is no fairness constraint and all users are considered. The AlloX algorithm is based on Algorithm 1 and incorporates online arrivals and fairness considerations.

Lines 2-6 of Algorithm 2 prepares inputs for the matching problem. In particular, it only considers jobs from $\lceil \alpha n \rceil$ users with the lowest progress. After solving the min-cost matching problem in Line 7, the algorithm simply searches for the first job scheduled on machine i. Specifically, the scheduler checks all entries affiliated with the available machine i and find a valid entry with largest k, which implies that the corresponding job w is scheduled first according on machine i.

Occasionally, the scheduler cannot find a valid job. It occurs when no job is scheduled on the available machine based on current jobs and system load. In this case, the algorithm returns no job and the system waits until the next event such as new job arrivals or a new machine becomes available.

Lemma 3.4. *For static allocation, AlloX is Pareto-efficient. If all jobs are identical and divisible within each individual user, AlloX is envy-free and sharing-incentive under $\alpha = 0$.*

Algorithm 2 AlloX Scheduler

```
function ScheduleNextJob(available machine i)
    Update users' progress and get the set of users A_α consisted of ⌈αn⌉ users with the lowest progress.
    for all job j in the waiting queue from A_α do
        Add processing time of job j to matrix P
    end for
    Generate the delay matrix D and further the cost matrix Q;
    Solve the min-cost matching problem defined by Q to get the matching matrix M
    for k = J : 1 do                          ▷ J is the total # of jobs in A_α
        for w = 1 : J do
            if M(w, m(k − 1) + i) = 1 then    ▷ w is first job scheduled on machine i
                schedule job w to machine i
                Update available time ω_i and users' progress
                return job w
            end if
        end for
    end for
    return null
end function
```

The proof of this lemma is included in the appendix.

4 AlloX Implementation

We build AlloX based on Kubernetes using roughly 3000 lines of Go code for the resource manager and 1500 lines of Python for its online job estimation tool. We pick Kubernetes because it well supports clusters consisted of heterogeneous resources such as CPU and GPU.

AlloX has three main components: *Estimator*, *Scheduler*, and *Placer* (Figure 9). AlloX first uses the estimator to obtain job characteristics. The scheduler then uses the algorithm described in the previous section to decide which job to be scheduled next and whether to place it on CPU or GPU. Finally, the placer executes the schedule in the system.

4.1 Estimator

We propose an estimator that works for training jobs where we know the number of iterations. The estimator predicts jobs' resource demands and their processing times on CPU and GPU in an online manner. The completion time on each resource is linearly estimated based the two small samples of the job. Totally, there are four samples for each job on CPU and GPU. In our experiment, the length of the sample jobs is 3% of the real jobs. The estimation of completion times is relatively accurate, especially for most machine learning jobs that are iterative [61, 69]. Figure 10 shows the CDF of estimation errors of 40 jobs through real experiments. The mean absolute error is 8% and the standard deviation is 11%.

Similar to Gandiva [64], the estimator determines the memory demands of jobs by monitoring the memory usage of their corresponding samples. Currently, GPUs in most

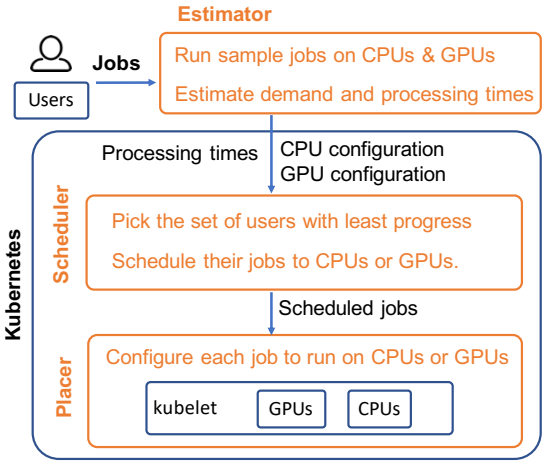


Figure 9. The AlloX system has three main components: Estimator, Scheduler, and Placer.

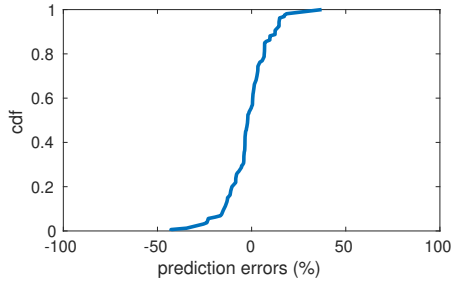


Figure 10. CDF of the estimation errors from experiments.

clusters do not support fine-grained sharing among multiple jobs [64], except for research efforts such as [66]. Therefore, we consider a job using a whole GPU in AlloX. For CPU, the estimator picks the maximum number of cores in a CPU to optimize the performance of the job on CPU. Because Kubernetes does not allow a container to have more than 1 CPU, AlloX does not give more CPU cores than a single physical CPU has either.

4.2 Scheduler

Kubernetes does not support fair allocation or job scheduling. As a result, we cannot simply modify some existing scheduler for our algorithm. Instead, we implement our scheduler from scratch using the `kube-scheduler` API.

Jobs arrive in a single queue in `kube-scheduler`. Given the set of available jobs, the scheduler decides which job to run. `kube-scheduler` receives the estimated processing times of the CPU and GPU configurations from the estimator and passes them to AlloX scheduler via `kubectl`. AlloX scheduling procedure is activated prior to *Pod Admission*. If the job is not admitted, it is sent back to the waiting queue. In addition to our scheduling algorithm, we implement other methods described in Section 5.1.1.

We add fairness support to `kube-scheduler` by updating the progress of all users over time (§3.3.2). `schedulerCache`

in `kube-scheduler` captures the snapshot of the whole system. When there is an update from the system, `schedulerCache` is notified and AlloX checks if a job gets resources or finishes. If a job receives allocated resources, the progress of that user is increased accordingly. Recall that if a job is running with the unfavorable configuration (with longer processing time), its value is discounted. If a job finishes, we deduct its value from the progress of the corresponding user.

4.3 Placer

The placer *dynamically* configures proper containers and executes jobs within these containers. In the current Kubernetes system, jobs are configured to run on CPU or GPU ahead of time; hence, they do not need the placer. This is another new component added. To enable the resource placement, it is also required to have a change at application level. In our experiments, we added a configuration function that allows the scheduler to control their runtime resource.

4.4 Operational Issues

Scalability. The network flow problem for job scheduling can be solved in polynomial time. Using the Hungarian algorithm, the computation complexity is $O((mn)^3)$, where m is the number of nodes and n is the number of jobs. If the scheduling interval has to be very short, this approach may be too slow. There are several solutions to this problem. We suggest using one of the three solutions, i.e., parallel programming, divide-and-conquer, or a heuristic. However, the latter two may not retain the optimality of AlloX.

Using multiple cores of GPU to solve the Hungarian algorithm is one of the best ways to speed up the solver and retain the optimal solution [46]. We evaluated the GPU version of Hungarian algorithm and find out that it can solve a problem with 100 nodes and 10 queued up jobs at 0.2 seconds using NVIDIA Quadro P4000.

Kubernetes supports multiple schedulers in the same cluster; therefore, we can divide a large cluster into multiple small ones [6] to conquer. However, although both the two aforementioned solutions can speed up the solvers and retain the optimality, they just solve the scalability issue up to some certain level. Therefore, we suggest using a heuristic solution. The idea here is to get performance close to AlloX but do not starve the jobs.

In the heuristic solution, AlloX picks the shortest jobs across CPU and GPU first. Additionally, if a job is waiting beyond a timeout, it will be prioritized. We call this algorithm AlloX+. The computational complexity of AlloX+ is $O(mnlog(n))$. We tested the heuristic algorithm of AlloX+ on 1000 jobs and 10000 nodes, showing that it can schedule these jobs in sub-second level (0.7 seconds). We show that performance of AlloX+ can be close to AlloX in the Section 5.

Generality of estimator. Our estimator in Section 4.1 is not general enough to handle all type of applications. In fact, there is already a large body of research on this problem. Optimus proposes using an inverse linear function to estimate the complete times of deep learning jobs [53].

For distributed jobs with large data input, we can use Ernest [61] or Cherrypick [10]. For pipelining jobs like SQL queries, an estimator like progress indicators are applied [14, 47]. For jobs that have multiple similar tasks, we can estimate resource demand and completions using ParaTimer [48] and Paralax [49].

Minimum CPU per job. When developing our scheduler, we realized all jobs running on GPUs also require (a small amount of) CPU for proper execution. AlloX addresses this by reserving a small number (one by default) of CPU cores for GPU jobs. As the number of CPU cores in a cluster is often large, this change has little impacts on the performance.

Job profiling overhead. Before a job is scheduled, its sample jobs must be completed first. To this end, AlloX prioritizes all sample jobs. Because these sample jobs are relatively small, the overhead is minimal. AlloX also sets a limit on resources for sample jobs to reserve enough resources for the real jobs. In addition, if a sample job is significantly longer than others, we do not need to complete it as it already indicates the original job is very long. We can adjust the sample job size to balance the overheads and estimation accuracy.

Low utilization with small α. The fairness parameter, α, allows the explicit tradeoff between fairness and performance. When α is small, the small set of users may not have jobs or want to wait for better resources, e.g., there are available CPUs but they prefer GPUs. This results in low resource utilization. To deal with this, AlloX temporally increases α to include more users who need the available resources.

Resource availability. While it is sufficient to use estimated processing times in the scheduling algorithm, resource availability needs to be obtained separately over time because there may be significant estimation errors. By default, Kubernetes periodically checks the health and updates from each node. Therefore, resource availability can be collected together with the current health check without additional overheads. If we need to inquire the availability information from all nodes very frequently, it might still lead to significant communication overheads for large-scale clusters. In this case, the information of nodes and jobs (`schedulercache.NodeInfo`) are cached in `kube-scheduler` and can be updated via events.

5 Evaluation

We evaluate AlloX through both experiments on real systems and numerical simulations. Our key takeaways are:

- AlloX reduces the average job completion time by as much as 95% compared to existing methods and baselines in various settings.
- AlloX achieves comparable performance of SRPT (an unrealistic lower bound) under a wide range of settings.
- AlloX provides fairness among users comparable to existing fair allocation and prevents starvation.

5.1 Experimental Methodology

Cluster. We setup Kubernetes with GPU support on a cluster of one master node, 8 CPU workers, and 4 GPU workers. The master node coordinates the workers and runs the job estimation tool. Each CPU worker is a *xl170r* server from Cloudlab [3] with 20 virtual CPU cores and 64GB RAM. Each GPU node is a *p2.xlarge* instance from Amazon EC2 with 1 K80 GPU, 4 CPU cores and 61GB RAM. This cluster setup mimics a hybrid cloud that has traditional CPU nodes local and expensive GPU nodes on public clouds. Furthermore, We do not run distributed jobs on the cluster so that there is no strict requirement on network bandwidth. There are 4 users, which is increased in the simulations.

Workload. Each user has 10 popular Tensorflow jobs, e.g., Googlenet, Lenet, and Alexnet. The job configurations such as batch sizes and batch numbers are different, resulting in the speedup rates of using one GPU versus one CPU ranging from 1.8 to 10. For jobs on CPU, the number of threads is set at 19 to best utilize the virtual cores while leaving one core for other services on each node. We run a small sampling job for each real job to obtain the parameters for both CPU and GPU configurations, as we discussed in Section 4.1. The total overhead of sampling jobs is 3% of the real jobs. We vary the settings in Section 5.3 to evaluate the impacts.

Simulator. To evaluate AlloX at a larger scale, we implement a Java-based cluster simulator, which emulates the cluster with multiple resources, e.g., CPU, GPU, and memory. We validate the accuracy of the simulator by comparing its results to those from real experiments over the cluster (Figure 11). There are 20 GPUs, 20 CPUs with 20 cores each, and 1280 GB RAM. Since GPU memory is small, RAM is not the bottleneck when we run the same jobs on CPUs.

For numerical simulations, we use the workload trace from the Google cluster [5] to generate arrival times for Tensorflow jobs. There are 10 users and over 1000 jobs for each user. By default, the fairness level α is set at 0.1, meaning, we schedule jobs from the 1/10 of all users who have the least progress whenever a node becomes available. The estimation errors are around 10% and their profiling overheads are 3% of the corresponding real jobs as we discussed in Section 4.1. The impacts of the fairness level, estimation errors, and overheads are studied in Sections 5.2.4, 5.3.1, and 5.3.2, respectively.

Metrics. To evaluate the performance, we measure the *average completion time* of all jobs under AlloX and baseline algorithms. We use standard deviation of *progresses* across users to evaluate fairness. For starvation, we focus on the progress of users with longer jobs.

5.1.1 Baselines

We compare AlloX to the following methods.

ES (equal share with shortest job first): ES divides all resources equally among users statically. For a particular user, whenever a resource becomes available, ES picks the job with the shortest processing time on this resource. For instance, if all jobs prefer GPUs, ES first fills up all available GPUs with shortest jobs based on their processing time on GPU, and then fills available CPUs with the shortest jobs using CPU configurations. ES needs the estimator to predict the processing time in different configurations.

DRFF (online DRF with FCFS): Whenever a resource becomes available, DRFF schedules the first job of the user with the least dominant share. Jobs are processed in a First-Come-First-Served manner within every user. For job configuration, we assume users have some preference. If all jobs prefer GPUs, DRFF always picks the GPU configuration. DRFF does not need the estimator to pick the configuration.

DRFS (DRF with shortest job first): DRFS is similar to DRFF, but within each user, jobs are scheduled in a shortest-job-first manner. Therefore, the estimator is needed. Each user relies on the estimation to pick whether CPU or GPU for each job configuration.

DRFA: DRFA uses some average speedup rate to convert GPU resources to the corresponding CPU ones, e.g., if the speedup rate is 10, 1 GPU is considered 10 CPU. Then the problem is simplified to the original multi-resource allocation without interchangeable resources, and online DRF is applied. Within each user, jobs are processed in a shortest-job-first manner, and therefore the estimator is needed.

SRPT: At any time, the job with the shortest *remaining* processing time is executed, which requires preemption. This approach is *unrealistic* in many real systems such as Kubernetes because jobs cannot be paused and moved from one resource to another, or even a different resource, without large overheads. However, SRPT is good at minimizing the average job completion time, and therefore serves as a goal for AlloX to achieve. Note that AlloX used in this section does not use preemption for conservative evaluations of the improvements.

AlloX+: A heuristic version of AlloX. AlloX+ first priotizes the jobs with waiting times beyond a time-out. If jobs are not timed out, AlloX+ picks the shortest job first.

5.2 AlloX Performance

We first evaluate AlloX through real experiments and validate the accuracy of the simulator in Section 5.2.1. Results from the simulator are discussed in Section 5.2.2.

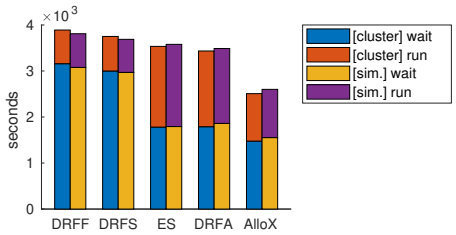


Figure 11. [Cluster] AlloX reduces the average job completion time. For each algorithm, the first bar shows results from experiments, and the second bar is from our simulator.

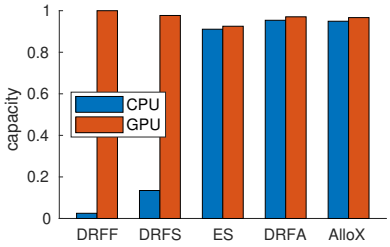
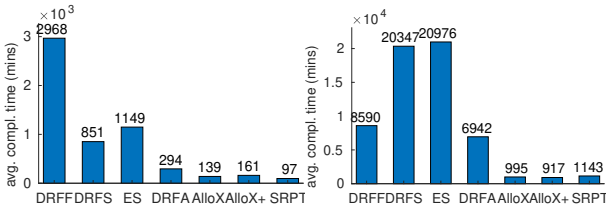


Figure 12. [Cluster] CPU and GPU utilization.



(a) all jobs **(b)** 1% longest jobs

Figure 13. [Simulation] AlloX and AlloX+ outperform others and are not far from SRPT in large-scale simulations. They even outperform SRPT for the longest jobs.

5.2.1 Experiments on a Cluster

Figure 11 illustrates the average job completion time under AlloX and other baselines through experiments and the simulator we developed. First, Allox reduces the average completion time significantly compared to other baselines. In particular, DRFF and DRFS do not fully utilize the CPU resources as shown in Figure 12, therefore incur longer waiting time. ES and DRFA reduce the waiting time by increasing the CPU utilization (Figure 12). Although AlloX has similar CPU and GPU utilization compared to DRFA and ES, AlloX outperforms DRFA and ES by better job scheduling and configuration selection. This is highlighted by the significant reduction in job processing time.

Figure 11 validates that the completion times in simulations and experiments are consistently similar. This allows us to perform larger-scale evaluations using the simulator.

5.2.2 Simulation Results

Figure 13 shows our simulation results. With a larger scale and more jobs, AlloX ($\alpha = 0.1$) consistently outperforms

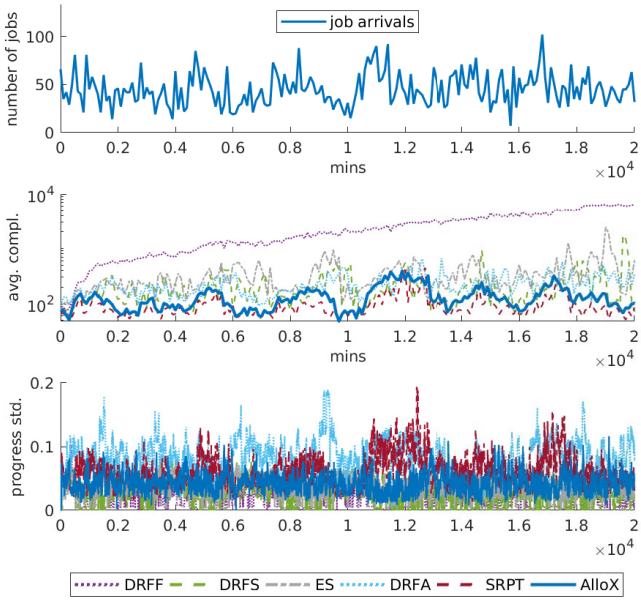


Figure 14. [Simulation] AlloX consistently maintains the best performance and small disparity across users' progress.

DRFF, DRFS, ES, and DRFA even more, reducing the completion time by 95%, 84%, 88%, and 53%, respectively. Impressively, AlloX is not far (30%) from SRPT that only minimizes the average completion time without considering fairness, and with preemption allowed. When we focus on the longest 1% jobs, AlloX has even larger improvements and beats SRPT. This is not surprising because SRPT prioritizes shorter jobs. The heuristic-based algorithm (AlloX+) with low scheduling latency is also much better than other baselines.

To provide more details regarding the comparison, we show the job arrivals, average job completion time, and the standard deviation of progresses across users over time in Figure 14. The average completion times of AlloX and SRPT are consistently better over time compared to other baselines. Note the completion time is shown on a logarithmic scale. When the arrival rates are high, the completion time of other baselines are much higher than that of AlloX. Not surprisingly, DRFF cannot process the jobs fast enough so queues are built up. This highlights by effective scheduling and configuration selection, AlloX processes jobs faster and therefore allows high arrival rates.

The figure also shows the disparity across users' progress over time. In this case, SRPT and DRFA are much worse than AlloX, while DRFF, DRFS, and ES are a little better than AlloX (all users progress at similar, but much slower rates compared to AlloX). While the disparity in SRPT is intuitive, DRFA fails to provide fairness because it ignores the different speedups of users, and instead use some averaged value to allocate resources.

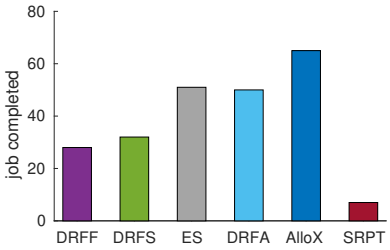
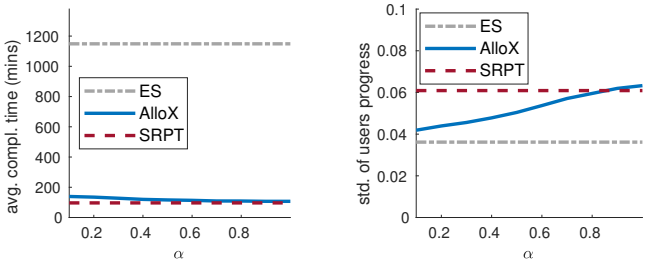


Figure 15. AlloX provides the best progress for users with longer jobs and prevents starvation.



(a) Performance **(b)** Fairness

Figure 16. [Simulation] Performance and fairness trade-off. A larger α reduces completion times at the cost of fairness.

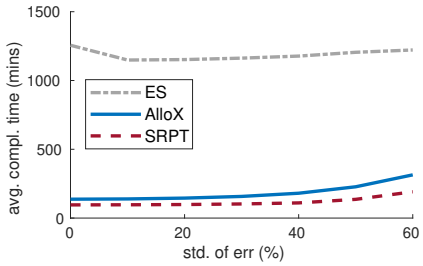


Figure 17. AlloX is robust to estimation errors.

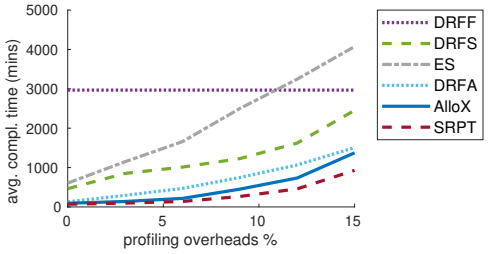


Figure 18. Impacts of profiling overheads.

5.2.3 Starvation

In extreme cases, some users may starve under schedulers like SRPT. Figure 15 shows the number of completed jobs of the user with longer jobs than others. As expected, SRPT performs the worst with the least number of jobs finished. In contrast, AlloX provides the best progress of this user because it maintains fairness and is more effective than other baselines.

5.2.4 Performance and Fairness Trade-offs

Figure 16 shows the trade-offs between performance (average completion time) and fairness. We vary the parameter α from 0.1 to 1 (smaller means fairer) and compare the performance with ES and SRPT. This shows with larger α, AlloX approaches SRPT in performance. The small gap (9%) between AlloX and SRPT when $\alpha = 1$ is due to the fact SRPT is preemptive at no cost while AlloX does not allow preemption. However, the unfairness in terms of standard deviation of users' progress also increases with larger α. Normally, a small α around 0.2 is good at providing large performance improvements at little cost of fairness.

5.3 Sensitivity Analysis

5.3.1 Estimation Errors

Figure 17 evaluates the impacts of misestimations on the performance. Though DRFF does not need the estimation, its performance is so poor that we compare AlloX with a stronger baseline ES. We do not compare with DRFA because it is not practical to know the average speedup rate. In spite of large estimation errors, AlloX still provides large improvements that are similar to SRPT. This highlights the value of incorporating (even noisy) estimation.

5.3.2 Profiling Overhead

Estimations require running profiling jobs with some overhead. Figure 18 evaluates the impacts of profiling overheads on performance. Recall in our experiments have profiling overheads at 3%. As DRFF does not require estimation, its performance is unchanged. With large overheads, the performance of all other solutions degrades. However, AlloX provides consistent, significant improvements compared to other baselines. In practice, for long and iterative machine learning jobs, it is reasonable to use small sampling jobs. We can also obtain the runtime estimation from users [29, 59], which may lead to further improvements.

6 Related Work

Resource configurations. Currently, developers or (data) scientists select job configurations based on their own experience and/or some recommendations. Recently, there have been some works on selecting cloud virtual machine (VM) configurations such as Paris [65], Ernest [61], and CherryPick [10]. While these focus on picking the number of VMs of different types, the estimation tool in AlloX decides the configuration over interchangeable resources automatically.

Resource managers. Given users' resource demands, YARN [60] and other allocation tools [54, 62] fit the submitted jobs when there are available resources. In contrast, AlloX does not ask users to submit their resource demands ahead. Instead, AlloX configures the jobs for users and submits the

jobs for scheduling automatically without users' involvement. Finally, none of the existing GPU cluster managers [33, 53, 64] consider interchangeable resources.

Multi-resource job scheduling. While job schedulers traditionally deal with a single resource [11, 38, 68], modern cluster resource managers, e.g., Mesos [34], YARN [60], and others [54, 62], employ multi-resource schedulers [26, 28, 31, 32] to handle multiple types of resources and optimize diverse objectives. These objectives can be fairness (e.g., DRF [28]), efficiency (e.g., Tetris [31]), performance (e.g., [26]), or combinations of different objectives (e.g., BoPF [44], Carbyne [32] and Quincy [38]). However, *none* of these focus on interchangeable resources. To the best of our knowledge, AlloX is the first multi-resource job scheduler over interchangeable resources (CPUs and GPUs) for both performance and fairness.

Heterogeneous resources. Recent schedulers also deal with jobs with placement constraints, e.g., Kubernetes [6] handling resource constraints in a best-effort manner and Choosy [29] in a fair way. In addition, Phoenix [57] focuses on minimizing the job response time, and Late [68] improves MapReduce job response time in heterogeneous environments. They mainly focus on the resource constraints and implicitly assume that the speed-up rates among nodes are identical. Hence, none of them deal with interchangeable resources.

There are a few works on interchangeable resources like μLayer [40] and TetriSched [59]. μLayer does resource placement for each layer of artificial neural networks, while AlloX performs both inter-job scheduling and resource placement. The most related work is probably TetriSched [59]. Compared to TetriSched which focuses on deadlines only, AlloX considers both performance and fairness. TetriSched formulates the problem as a Mixed Integer Linear Programming (MILP), which cannot be solved in polynomial time so far. In contrast, AlloX solves a linear programming with low complexity. In addition, TetriSched does not provide fairness and may lead to starvation, while AlloX explicitly balances performance and fairness, and prevents starvation. Finally, TetriSched simply assumes estimations needed are known beforehand, while AlloX obtains the information in an online and automatic manner. Beyond GPU/CPU systems, heterogeneous resources have been studied, e.g., heterogeneous computing, power and cooling resources for sustainable data centers [43], heterogeneous energy storage systems for electricity market [15].

7 Concluding Remarks

In this paper, we design and implement AlloX, a system that minimizes the average job completion time in CPU-GPU hybrid clusters while providing fairness among users and preventing starvation. AlloX profiles and schedules jobs in an automatic and online manner. Our algorithm solves a min-cost bipartite matching problem and obtains the corresponding placement and scheduling decisions. It provides an optimal solution in simplified settings and outperforms all baselines significantly in general settings. Evaluations highlight that AlloX can significantly improve system performance while maintaining fairness among users. The problem studied in this paper is a generalization of the traditional job scheduling and fair resource allocation problems, and can be applied beyond computational resources, e.g., to different network interfaces and storage devices. Extending the algorithmic study and system design to these resources is our ongoing work.

Future directions. Having more than two interchangeable resources such as multiple GPU types raises challenges at both the estimator and scheduler. It increases the overheads for the estimator and adds more computing dimensions to the scheduler. We leave these challenges as future work. Another extension to AlloX would be working with distributed jobs that require the combination of intra-job and inter-job scheduling. Furthermore, if jobs can be preempted and switched from this resource to another, it would be interesting to find an algorithm that beats shortest remaining time first (SRPT).

8 Acknowledgments

This research is supported by NSF grants CNS-1617773, 1629397, 1909067, 1730128, 1919752, 1617698, 1717588 and was partially funded by MSIT, Korea, IITP-2019-2011-1-00783. We would like to thank you Adhita Selvaraj and Manideep Attanti for their helpful contributions in the paper.

A Appendix

A.1 Proof of Lemma 3.1

Let the optimal solution be $OPT(\cdot)$. The proof follows two steps: first we show that any scheduling instance $\mathcal{S}$ can be transformed into a corresponding bipartite matching problem $\mathcal{M}^{\mathcal{S}}$; then we prove their optimal solutions are indeed equivalent.

Let g_i^k and c_i^k be the machine-position variables, where g and c represent the GPU and CPU machines, i is the machine id and k is the position index counted backward from the end. Let J_j represent job j. Define graph $M = (N, E)$, where N consists of nodes g_i^k,c_i^k and J_j for all i, j, k and E consists of all pairs in (g_i^k, J_j) and (c_i^k, J_j) for all i, j, k. Define the cost for edge (g_i^k, J_j) $((c_i^k, J_j))$ as kp_g^j (kp_c^j), where p_g^j (kp_c^j) is the processing time of job j on GPU (CPU). By definition, M is a bipartite graph.

Now we show that any feasible scheduling for problem $\mathcal{S}$ has a valid matching in the corresponding $\mathcal{M}^{\mathcal{S}}$. To see this, notice that for any feasible scheduling, each job is scheduled only once and there is a unique ordering on each machine. For any job j that is placed on g_i (c_i), let its order be r (counted backward from the end of g_i), then in graph M from $\mathcal{M}^{\mathcal{S}}$,

consider the edge set P from (g_i^r, J_j) $((c_i^r, J_j))$ for all j and its corresponding i and r. Clearly, P is matching as all nodes in J_j are connected and each of them has a unique destination. Now we show that the the total completion time in $\mathcal{S}$ is equivalent to the total cost in $\mathcal{M}^{\mathcal{S}}$. To see this, consider any job j that is placed on g_i (c_i) with order r counted backward. Then obviously, there are exactly r jobs will be consuming $p_g(p_c)$ computation time on machine i based on the placement of job j, so the total completion time of this placement is exactly the cost of the corresponding edge in $\mathcal{M}^{\mathcal{S}}$. So the total cost of the two instance will also be the same. We have $OPT(\mathcal{S}) \geq OPT(\mathcal{M}^{\mathcal{S}})$.

Conversely, we show that any valid matching in graph G corresponds to a feasible extended scheduling problem $\mathcal{S}'$ by allowing adding 'dummy jobs' with 0 processing times. Firstly, we add n dummy jobs on each machine. Then for all edges (g_i^r, J_j) $((c_i^r, J_j))$ picked in $\mathcal{M}^{\mathcal{S}}$ problem, simply replace the r_{th} last dummy job on g_i (c_i) with job J_j. Since the cost of using the edge is equivalent to the computation time of the corresponding placement. $OPT(\mathcal{M}^{\mathcal{S}}) \geq OPT(\mathcal{S}')$.

However, with extra dummy jobs, the total completion time will be nondecreasing, $OPT(\mathcal{S}') \geq OPT(\mathcal{S})$. So we have $OPT(\mathcal{S}) = OPT(\mathcal{M}^{\mathcal{S}})$.

A.2 Proof of Lemma 3.2

Consider a simple case with n users, and each user has identical jobs. The speedup of using a GPU versus a CPU is $(1+\epsilon)$ for all jobs. Clearly, users would choose the configuration that runs faster, which are the GPU configurations in this case. By the allocation with DRF, all GPUs are shared equally among users while all CPUs remain idle. Assuming other resources such as the memory is not the bottleneck, the allocation is not Pareto efficient because CPUs can be utilized to improve the progress of all users. This allocation does not provide sharing incentive because every user is worse than equal sharing, where each user has some CPUs in addition to the same amount of GPUs allocated. Finally, if some user lies that she prefers CPUs, she will get all the CPU nodes and progress faster than others. This violates strategy-proofness.

A.3 Proof of Lemma 3.3

Consider two users A and B. Both have identical jobs. The speedup of user A's jobs is $\beta_A = 2$, while the speedup of user B's jobs is $\beta_B = 4$. Assume computation is the bottleneck for both users. The system has the same amount of CPUs and GPUs, normalized to 1.

We first consider the sharing incentive (SI) and Pareto efficiency (PE) properties. SI requires user A gets at least $\frac{1}{2}(1+2) = \frac{3}{2}$ computational resources (CPUs and GPUs combined), while user B gets at least $\frac{1}{2}(1+4) = \frac{5}{2}$ computational resources. From PE, we know that user A should use CPUs first while user B should use GPUs first because $\beta_A < \beta_B$. Because GPUs are more effective, user A should get all the CPUs and some fraction of GPUs.

Note PE also requires that there should not be any leftover CPUs or GPUs if computation is the system bottleneck. Let A's share on GPU be x. Then B's share on GPU is $1-x$. By the sharing incentive property, for user A, we have $2x+1 \geq \frac{3}{2}$, i.e., $x \geq \frac{1}{4}$; for user B, we have $4(1-x) \geq \frac{5}{2}$, where we have $x \leq \frac{3}{8}$. Therefore SI and PE requires $\frac{1}{4} \leq x \leq \frac{3}{8}$.

If both users report truthfully, assuming at the final allocation, $\exists \delta > 0$ s.t. $x+\delta < \frac{3}{8}$, we show it is not strategyproof for user A. Specifically, we show that by lying about her speedup ratio, user A can always get at least $(\frac{3}{8} - \sigma)$ fraction of GPU for any small $\sigma > 0$.

To see this, let user A report $\beta'_A = 4 - \epsilon$ for some small $\epsilon > 0$ instead of the true value 2. By the SI property, user A needs to get at least $\frac{1}{2}(1+4-\epsilon)$ computational resources. As she has a lower speedup ratio than user B, she will get all CPU, therefore the computational resources from GPUs are $\frac{1}{2}(1+4-\epsilon) - 1 = 1.5 - 0.5\epsilon$. This implies that user A needs to get at least $\frac{1.5-0.5\epsilon}{4-\epsilon}$ fraction of GPU, which approaches arbitrary close to $\frac{3}{8}$ with decreasing ϵ. Therefore, there exists an ϵ that user A can use to get at least $(\frac{3}{8} - \sigma)$ fraction of GPU. Thus, to make sure user A has no incentive to lie, the allocation has to provide at least $\frac{3}{8}$ fraction of GPU to user A.

Similarly, user B can report $\beta'_B = 2 + \epsilon$ to increase her allocation on GPUs. If she reports $\beta'_B = 3$, B can get at least $\frac{2}{3}$ GPU. Clearly, there is not enough GPU to share as $\frac{3}{8} + \frac{2}{3} > 1$. So no allocation can be strategyproof.

A.4 Proof of Lemma 3.4

Pareto-efficiency comes from the fact that AlloX is optimal for static scheduling. So clearly no one can increase her completion time without hurting others' performance.

With assumption of divisibility and $\alpha = 0$. AlloX will maintain a strict fairness allocation, where the progress of all users are equal all the time. To show envy-freeness, consider arbitrary two users and compare their dominant resources. If the type of their dominant resources is the same, then from the equality of progress, their allocation of that resource is also the same, so there won't be envyness. If they have different dominant resources, then by switching their resource, both users will have less resource in their dominant resource, which in return make their progress worse.

By contradiction, if sharing-incentive is not satisfied, then there exists a user whose progress is worse than equal sharing. But by the equality of progress and subsequently equality of dominant share, this means all users will be worse than equal sharing. However, this is not possible as the allocation is Pareto-efficient.

References

[1] [n. d.]. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset.

[2] [n. d.]. Caffe 2. https://caffe2.ai. Accessed: 2017-04-21.

[3] [n. d.]. Cloudlab. http://www.cloudlab.us/.

[4] [n. d.]. deep-learning-cpu-gpu-benchmark. https://github.com/minimaxir/deep-learning-cpu-gpu-benchmark.

[5] [n. d.]. Google Cluster Traces. https://github.com/google/cluster-data.

[6] [n. d.]. Google Container Engine. http://kubernetes.io.

[7] [n. d.]. keras-video-classifier-web-api. https://github.com/chen0040/keras-video-classifier.

[8] [n. d.]. PaddlePaddle. https://github.com/PaddlePaddle/Paddle.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.

[10] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In *NSDI*.

[11] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in MapReduce Clusters using Mantri. In *OSDI*.

[12] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *FPL*.

[13] James Bruno, Edward G Coffman Jr, and Ravi Sethi. 1974. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM* 17, 7 (1974), 382–387.

[14] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In *SIGMOD*. ACM, 803–814.

[15] Hao Chen, Zhenhua Liu, Ayse K Coskun, and Adam Wierman. 2015. Optimizing energy storage participation in emerging power markets. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 1–6.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. 578–594.

[17] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*.

[18] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745* (2012).

[19] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. 2011. Flexible, high performance convolutional neural networks for image classification. In *IJCAI*, Vol. 22. Barcelona, Spain, 1237.

[20] Alan Cobham. 1954. Priority assignment in waiting line problems. *Journal of the Operations Research Society of America* 2, 1 (1954), 70–76.

[21] Aleksandar Colic, Hari Kalva, and Borko Furht. 2010. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM, 13–22.

[22] Alexis Conneau, Holger Schwenk, Loıc Barrault, and Yann Lecun. 2016. Very deep convolutional networks for natural language processing. *arXiv preprint* (2016).

[23] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*. 153–167.

[24] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *ISCA*. 1–14.

[25] Zvi Galil and Éva Tardos. 1988. An O (n 2 (m+ n log n) log n) min-cost flow algorithm. *Journal of the ACM (JACM)* 35, 2 (1988), 374–386.

[26] Michael R Garey, David S Johnson, and Ravi Sethi. 1976. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research* 1, 2 (1976), 117–129.

[27] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-resource fair queueing for packet processing. *SIGCOMM* (2012).

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*.

[29] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*.

[30] Yoav Goldberg. 2016. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research* 57 (2016), 345–420.

[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*.

[32] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*.

[33] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*.

[34] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*.

[35] Han Hoogeveen, Petra Schuurman, and Gerhard J Woeginger. 1998. Non-approximability results for scheduling problems with minsum criteria. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 353–366.

[36] WA Horn. 1973. Minimizing average flow time with parallel machines. *Operations Research* 21, 3 (1973), 846–847.

[37] Zhiheng Huang, Geoffrey Zweig, Michael Levit, Benoit Dumoulin, Barlas Oguz, and Shawn Chang. 2013. Accelerating recurrent neural network training via two stage classes and parallelization. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE, 326–331.

[38] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*.

[39] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*.

[40] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *EuroSys*. 1–15.

[41] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[42] Jacques Labetoulle, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. 1984. Preemptive scheduling of uniform machines subject to release dates. In *Progress in combinatorial optimization*. Elsevier, 245–261.

[43] Tan N Le, Zhenhua Liu, Yuan Chen, and Cullen Bash. 2016. Joint capacity planning and operational management for sustainable data centers and demand response. In *Proceedings of the Seventh International Conference on Future Energy Systems*. ACM, 16.

[44] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2019. BoPF: Mitigating the Burstiness-Fairness Tradeoff in Multi-Resource Clusters. *ACM SIGMETRICS Performance Evaluation Review* 46, 2 (2019), 77–78.

[45] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*.

[46] Paulo AC Lopes, Satyendra Singh Yadav, Aleksandar Ilic, and Sarat Kumar Patra. 2019. Fast block distributed CUDA implementation of the Hungarian algorithm. *J. Parallel and Distrib. Comput.* 130 (2019), 50–62.

[47] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. 2004. Toward a progress indicator for database queries. In *SIGMOD*. ACM, 791–802.

[48] Kristi Morton, Magdalena Balazinska, and Dan Grossman. 2010. ParaTimer: A progress indicator for MapReduce DAGs. In *SIGMOD*.

[49] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. 2010. Estimating the progress of MapReduce pipelines. In *ICDE*. IEEE, 681–684.

[50] James B Orlin. 1993. A faster strongly polynomial minimum cost flow algorithm. *Operations research* 41, 2 (1993), 338–350.

[51] David C Parkes, Ariel D Procaccia, and Nisarg Shah. 2015. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation (TEAC)* 3, 1 (2015), 3.

[52] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration.(2017). *URL https://github.com/pytorch/pytorch* (2017).

[53] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*.

[54] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*.

[55] Xiao Sun, Tan N Le, Mosharaf Chowdhury, and Zhenhua Liu. 2018. Fair Allocation of Heterogeneous and Interchangeable Resources. *Performance evaluation review* (2018).

[56] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin N Gurcan, Wagner Meira, Umit Catalyurek, and Renato Ferreira. 2009. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *CLUSTER*.

[57] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2017. Phoenix: a constraint-aware scheduler for heterogeneous datacenters. In *ICDCS*. IEEE, 977–987.

[58] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, Vol. 5. 1–6.

[59] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*. ACM, 35.

[60] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*.

[61] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics.. In *NSDI*. 363–378.

[62] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.

[63] Wei Wang, Baochun Li, and Ben Liang. 2014. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *INFOCOM, 2014 Proceedings IEEE*. IEEE, 583–591.

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*. USENIX Association, 595–610.

[65] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *SoCC*. ACM, 452–465.

[66] Peifeng Yu and Mosharaf Chowdhury. 2020. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *MLSys*.

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.

[68] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*.

[69] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*. ACM, 390–404.

[70] Maohua Zhu, Liu Liu, Chao Wang, and Yuan Xie. 2016. CNNLab: A novel parallel framework for neural networks using GPU and FPGA–A practical study with trade-off analysis. *arXiv preprint arXiv:1606.06234* (2016).