HiveD:
Sharing a GPU Cluster for Deep Learning with Guarantees

Hanyu Zhao¹,³*, Zhenhua Han²,³*, Zhi Yang¹, Quanlu Zhang³, Fan Yang³, Lidong Zhou³, Mao Yang³, Francis C.M. Lau², Yuqi Wang³, Yifan Xiong³, Bin Wang³

¹ Peking University, ² The University of Hong Kong, ³ Microsoft

* Equal contributions
Multi-Tenant GPU Clusters Today

- Shared by multiple tenants
  - Built with budgets/hardware from tenants

- **Reservation** is necessary for *guaranteed* resource availability and user experiences

- Reserve *number of GPUs (quota)* for each tenant
  - A tenant expects to access at least its contributed share
Affinity Matters, but NOT Reserved

- Jobs usually have **affinity** requirements

- Quota reservation + global affinity optimization (defragmentation)

They are all equivalent in quota usage!
Anomaly: Sharing Leads to Suffering!

- “External” fragmentation across tenants
  - Fragmentation from other tenants, even everyone is within the quota
  - Quota cannot isolate fragmentation across tenants

- External fragmentation makes sharing harmful to tenants
  - Worse performance in the shared cluster than in private clusters
  - Global defrag might sometimes hurt job performance (a complex multi-objective optimization)

- Real users are reverting to private clusters!

![Diagram showing resource allocation between shared and private clusters]
HiveD

- Primary goal: **Sharing Safety**
  - Any sequence of GPU requests (possibly with affinity constraints) can be satisfied on the shared cluster if satisfied on the tenant’s private cluster

- Key idea
  - *Reserve GPU affinity explicitly*
  - Separate the concern of sharing safety from other scheduling goals
Two-Layer Architecture

- **Virtual Private Cluster**
  - New resource abstraction: *cell*, captures quota and affinity
  - Compatibility with SOTA deep learning schedulers in VCs

- **From Virtual to Physical**
  - Dynamic cell allocation with *proven sharing safety*
  - Natural support for low-priority jobs

---

**Separation of Concerns**

- Sharing Safety
- Scheduling Efficiency
- Utilization

---

**Virtual Private Clusters**
Third-party scheduler operates on VC view

**Physical Cluster**
Allocate and deallocate physical resources (cells) dynamically

---

Cell Binding  Free Cell  Allocated Cell  Cell Running Low-priority Job
Two-Layer Architecture

- **Virtual Private Cluster**
  - New resource abstraction: *cell*, captures quota and affinity
  - Compatibility with SOTA deep learning schedulers in VCs

- **From Virtual to Physical**
  - Dynamic cell allocation with *proven sharing safety*
  - Natural support for low-priority jobs

- **Best of both worlds**
  - A private cluster: guaranteed resource availability regardless of other tenants
  - A shared cluster: more resources when available

---

**Separation of Concerns**

- Sharing Safety
- Scheduling Efficiency
- Utilization
Hierarchical Cell Structures

- A cell is a set of GPUs at a certain level of affinity

Level-1 cell: GPU
Hierarchical Cell Structures

- A cell is a set of GPUs at a certain level of affinity

Level-2 cell: PCI-e switch 1 2
Hierarchical Cell Structures

- A cell is a set of GPUs at a certain level of affinity

[Diagram showing level-3 cell: CPU socket with numbered CPUs 1, 2, 3, 4]
Hierarchical Cell Structures

- A cell is a set of GPUs at a certain level of affinity

![Diagram of hierarchical cell structures]
Hierarchical Cell Structures

- A cell is a set of GPUs at a certain level of affinity

A cell can be split into multiple equivalent buddy cells
Virtual Private Clusters

- Cells at each level, modeling a tenant’s private cluster

- *Dynamic* cell binding
  - Reducing preemptions and fragmentation, handling faulty hardware
  - Handled by Buddy Cell Allocation algorithm

Dynamic binding via Buddy Cell Allocation

<table>
<thead>
<tr>
<th>Cell Level</th>
<th>Tenant A</th>
<th>Tenant B</th>
<th>Tenant C</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4 cell (8-GPU)</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>L3 cell (4-GPU)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L2 cell (2-GPU)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L1 cell (1-GPU)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Buddy Cell Allocation

- For a cell request at level-k:
  - Allocate a free level-k cell if any
  - Split a free level-(k+1) cell otherwise
Buddy Cell Allocation

- For a cell request at level-\( k \):
  - Allocate a free level-\( k \) cell if any
  - Split a free level-(\( k+1 \)) cell otherwise

Cell request for level-4 (node)
Buddy Cell Allocation

- For a cell request at level-\(k\):
  - Allocate a free level-\(k\) cell if any
  - Split a free level-(\(k+1\)) cell otherwise

Cell request for level-1 (GPU)
Buddy Cell Allocation

- For a cell request at level-$k$:
  - Allocate a free level-$k$ cell if any
  - Split a free level-$(k+1)$ cell otherwise
Buddy Cell Allocation

- For a cell request at level-k:
  - Allocate a free level-k cell if any
  - Split a free level-(k+1) cell otherwise

Cell request for level-1 (GPU)
Buddy Cell Allocation

- For a cell request at level-k:
  - Allocate a free level-k cell if any
  - Split a free level-(k+1) cell otherwise
Buddy Cell Allocation

- For a cell request at level-k:
  - Allocate a free level-k cell if any
  - Split a free level-\((k+1)\) cell otherwise

Another Cell request for level-1 (GPU)
Buddy Cell Allocation

- For a cell request at level-k:
  - Allocate a free level-k cell if any
  - Split a free level-(k+1) cell otherwise

- Cell release (and merge) works oppositely

- Keep as many higher-level cells as possible

- **Proven** safety guarantee
  - Satisfies any cell request within a VC, if the initial VC assignment is feasible

---

**Algorithm 1 Buddy Cell Allocation Algorithm**

1: // Initial state of free_cells: only top level has cells
2: procedure ALLOCATECELL(cell_level)
3:   if free_cells[cell_level].size() == 0 then
4:     c = AllocateCell(cell_level+1)
5:     cells = Split(c) \(\triangleright\) Split cells are buddies
6:     free_cells[cell_level].extend(cells)
7:   Return free_cells[cell_level].pop()
8: 
9: procedure RELEASECELL(cell)
10:   if cell.buddies ⊆ free_cells[cell.level] then
11:     higher_cell = Merge(cell, cell.buddies)
12:     free_cells[cell.level].remove(cell.buddies)
13:     ReleaseCell(higher_cell)
14: else
15:   free_cells[cell.level].add(cell)
Allocating Low-Priority Cells

- Two cell views
  - High-priority guaranteed jobs with VC safety
  - Low-priority opportunistic jobs to improve utilization
Allocating Low-Priority Cells

- Two cell views
  - High-priority guaranteed jobs with VC safety
  - Low-priority opportunistic jobs to improve utilization

Mark as used
Allocating Low-Priority Cells

- Two cell views
  - High-priority guaranteed jobs with VC safety
  - Low-priority opportunistic jobs to improve utilization

```
High-priority view

CPU

1 2 3 4

Low-priority view

CPU

1 2 3 4

Mark as dirty
```
Allocating Low-Priority Cells

- Two cell views
  - High-priority guaranteed jobs with VC safety
  - Low-priority opportunistic jobs to improve utilization

![Diagram showing two cell views: High-priority view and Low-priority view. The high-priority view has jobs 1, 2, 3, 4, and 5, whereas the low-priority view has jobs 1, 2, 3, 4, 5, 6, 7, and 8. Job 5 is preempted in the low-priority view.](image-url)
Open-Source Implementation

https://github.com/microsoft/hivedscheduler

- Implemented on Kubernetes

- Integrated with Microsoft OpenPAI
  https://github.com/microsoft/pai

- Deployed at Microsoft for 12+ months
  - Managing 1000+ heterogeneous GPUs
  - Serving research and production workloads at scale

- More implementation details and operation experiences in the paper
Evaluation: 96-GPU Cluster Experiment

- Schedulers
  - YARN-CS (Philly) [ATC 19]
  - Gandiva [OSDI 18]
  - Tiresias [NSDI 19]
  - A VC preserves the precise affinity structure, making STOA schedulers applicable

- HiveD achieves the best of both worlds
  - Sharing anomalies identified in all schedulers with quota; HiveD eliminates them all!
  - Significantly shorter queuing delay than in the private cluster
  - Similar overall JCT compared to applying the schedulers globally
Evaluation: Trace-Driven Simulation

- 2-month trace from a 2232-GPU cluster with 11 tenants
- More sharing anomalies under high load
  - Up to 8,000+ minutes of excessive queuing delay
  - 7x on average
  - Again HiveD eliminates them all
Evaluation: Trace-Driven Simulation

- 2-month trace from a 2232-GPU cluster with 11 tenants
- More sharing anomalies under high load
  - Up to 8,000+ minutes of excessive queuing delay
  - 7x on average
  - Again HiveD eliminates them all
- Sharing anomaly leads to diminishing benefits of sharing
  - Decommission a tenant with higher avg. queuing delay in the shared cluster with quota than in the private cluster
  - Tenants owning 37% quota (two large tenants) decommissioned!
  - Significantly longer queuing delay of the other 9 tenants in this smaller cluster
Evaluation: Buddy Cell Allocation

- Reducing preemptions
  - Avoid dirty cells with dynamic binding

- Reducing fragmentation
  - Pack cells across VCs
Conclusion

- HiveD addresses the challenge of sharing a GPU cluster with
  - *Sharing safety*: simple and practical guarantee easily appreciated by tenants
  - *Cell*: new resource abstraction for defining tenants’ affinity structures
  - *Buddy cell allocation*: proven safety and support for low-priority jobs
  - *Two-layer architecture* to incorporate other scheduling goals while guaranteeing sharing safety
Thank you!

Contact
Hanyu Zhao (zhaohanyu1994@gmail.com)
Zhenhua Han (Zhenhua.Han@microsoft.com)

Code released at
https://github.com/microsoft/hivedscheduler