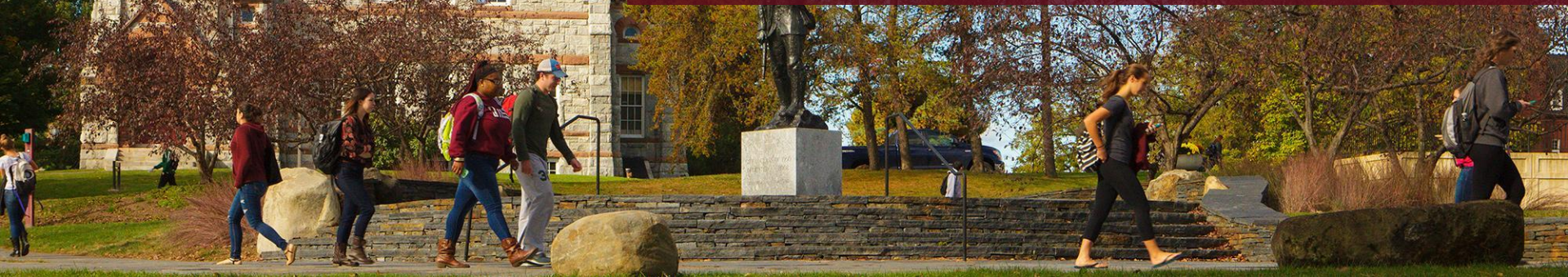
UMassAmherst
The Commonwealth's Flagship Campus

Efficient Inference (Multi-tasking)

Tian Wang & Cole Smith

10/07/2020



SALUS: FINE-GRAINED GPU SHARING PRIMITIVES FOR DEEP LEARNING APPLICATIONS

Peifeng Yu and Mosharaf Chowdhury

SALUS

Problem:

Modern GPUs do not natively support fine-grained sharing primitives.

Minimum granularity of GPU allocation today is often the entire GPU.

such exclusiveness in accessing a GPU simplifies the hardware design and makes it efficient in the first place.

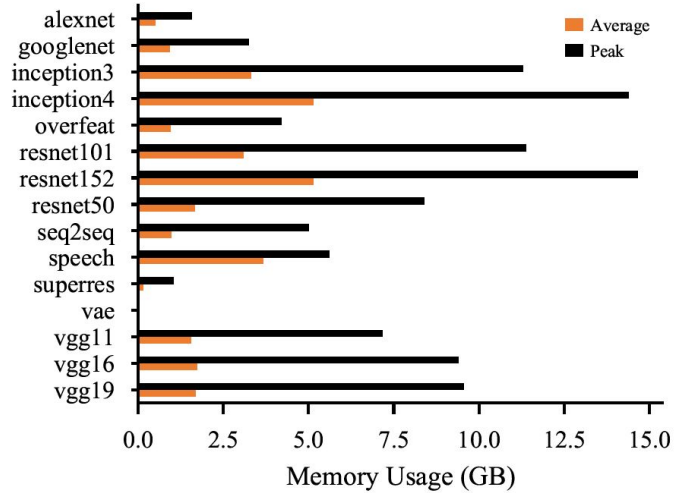
Problem

It leads 2 inefficiencies:

1.suspend/resume a jobs cost a lot of overhead. So, GPU cluster often employ non-preemptive scheduling. Like FIFO, but it cause HOL(head-of-line) block.

2. DL jobs can not fully use GPU memory

all the time.



Existing Techniques

1.disabling the exclusive access mode and statically partition the GPU memory among DL jobs.

2.NVIDIA’s Multi-Process Service(MPS): speed up the static partitioning approach. Limited support for DL frameworks.

Above 2 approaches can not provide performance isolation.

3.NVIDIA’s TensorRT Inference server: achieve simultaneous DL inference in parallel on a single GPU. But they lack support for DL training.

.

Observation and MOTIVATION

Heterogeneous Peak Memory Usage Across Jobs

Temporal Memory Usage Variations Within a Job

Low Persistent Memory Usage

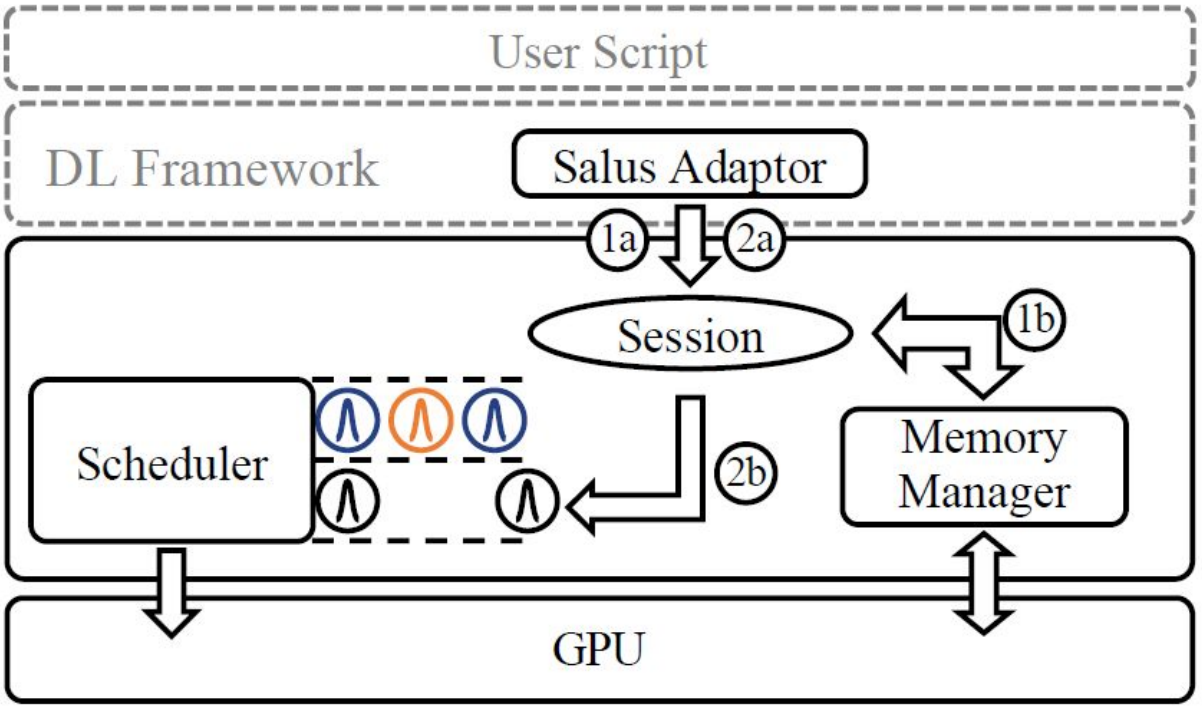
According these memory characteristics,define 3 types of memory:

1.Model(persistent, small)

2.Ephemeral(temporal, large)

3.Framework-internal(persistent, small)

Architecture:

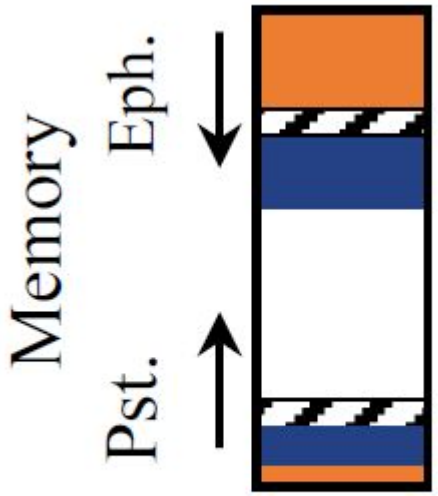


salus is user-agnostic
For framework: salus is like a computation device.

Job created by script, salus creates a session 1a. 1a process a lane request from memory manager.

After return a lane, user script generate iterations. Iterations forwarded to session(2a). Scheduler depend on associated lanes and send iterations to GPU.

Memory regions



(a) Memory regions

Divide GPU memory space into ephemeral and persistent regions.

DL job's ephemeral memory goes into Eph. region.

Others go into Pst. region.

Lane assignment

Safety condition:

$$\sum_{\text{job } i} P_i + \sum_{\text{lane } l} \max_{\text{job } j \text{ in } l} (E_j) \leq C$$

This condition make sure at least one job in the lane can proceed

Algorithm

Algorithm 1 Find GPU Lane for Job

Input: P: the job's persistent memory requirement
E: the job's ephemeral memory requirement
C: total memory capacity of the GPU
P_i: persistent memory usage of existing job i
L_j: lane size of existing lane j
$\mathbb{L}$: set of existing lanes
if $\sum_i P_i + P + \sum_j L_j + E \leq C$ **then**
  $lane \leftarrow$ new GPU lane with capacity E
  return $lane$
end if
for all $j \in \mathbb{L}$ **do**
  if $L_j \geq E$ and is the best match **then**
    return j
  end if
end for
for $r \in \mathbb{L}$ in L_r ascending order **do**
  if $\sum_i P_i + P + \sum_j L_j - L_r + E \leq C$ **then**
    $L_r \leftarrow E$
    return r
  end if
end for
return not found

How to organize lane allocation is an open question.

SCHEDULING POLICIES IN SALUS

PACK to Maximize Efficiency:attempts to pack as many jobs as possible in to the GPU. The goal is to minimize the makespan.

SRTF to Enable Prioritization:preemptive shortest-remaining-time-first scheduler

FAIR to Equalize Job Progress:uses time sharing to equally share the GPU time among many jobs.

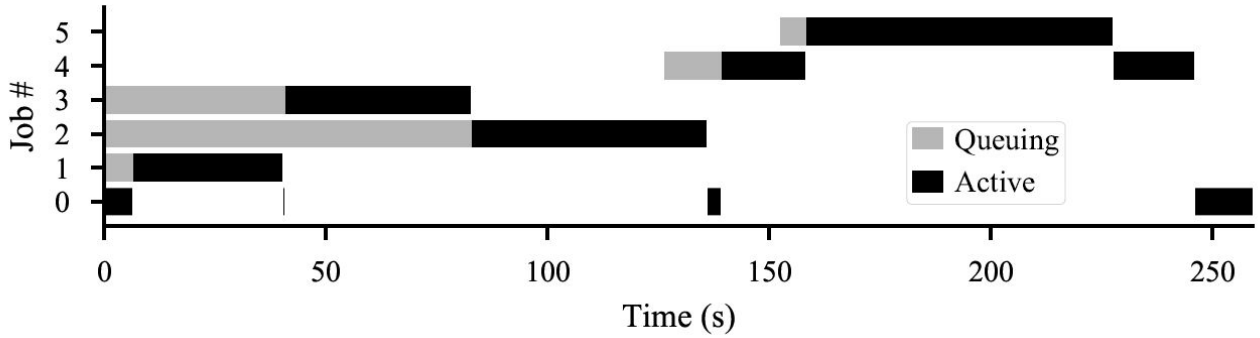
Evaluation

Baseline:FIFO scheduling commonly used in today's GPU clusters

Sched.	Makespan	Avg. Queuing	Avg. JCT	95% JCT
FIFO	303.4 min	167.6 min	170.6 min	251.1 min
SRTF	306.0 min	28.6 min	53.4 min	217.0 min
PACK	287.4 min	129.9 min	145.5 min	266.1 min
FAIR	301.6 min	58.5 min	96.6 min	281.2 min

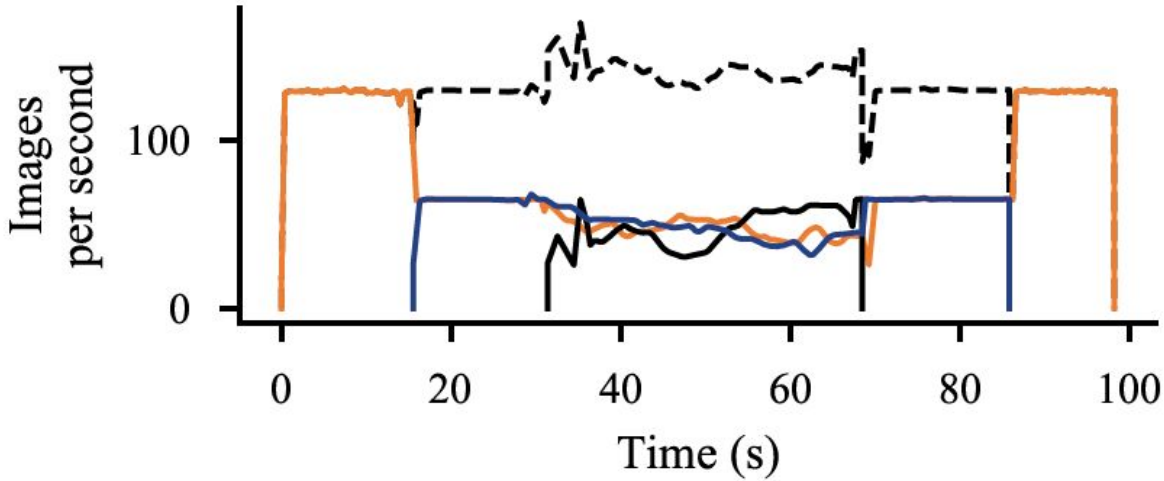
Job Switching

SRTF



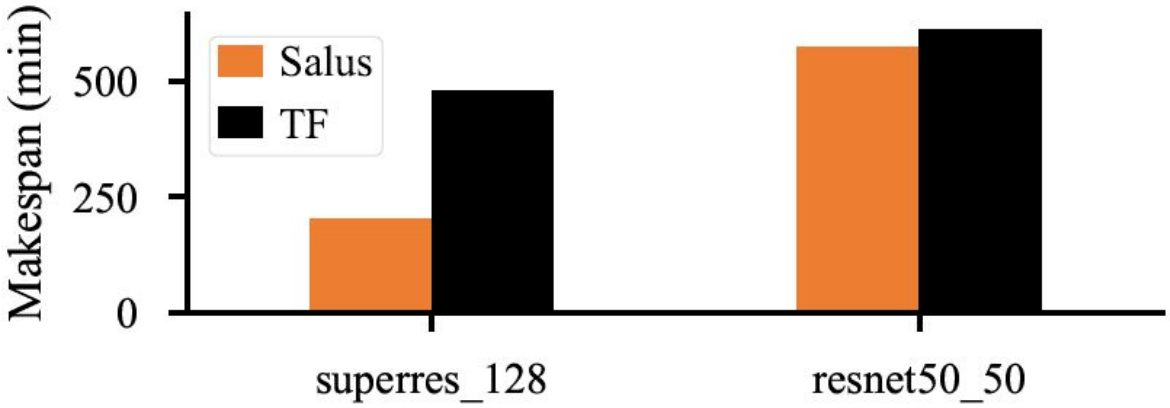
Job Switching

The throughput of the three jobs is almost the same



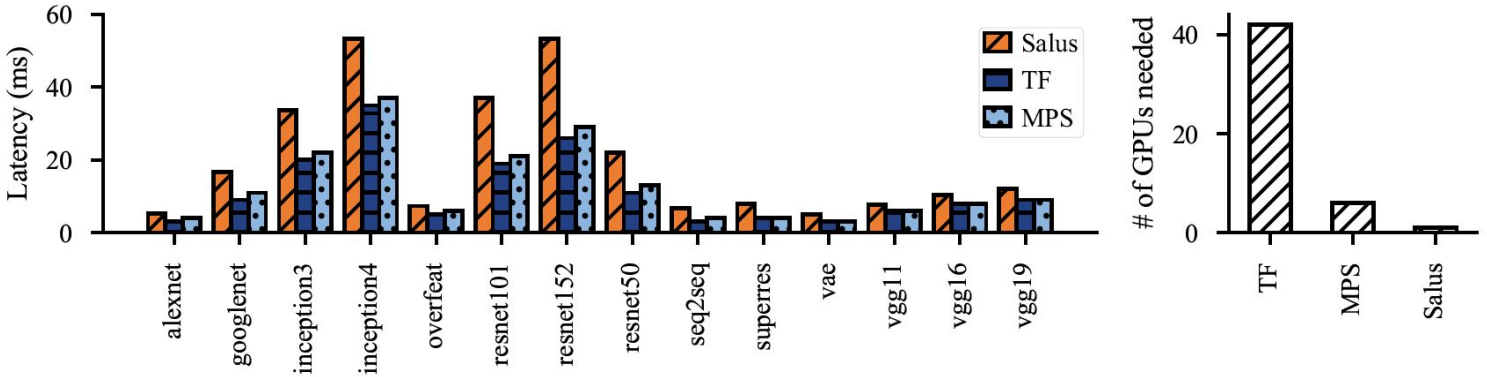
PACK

automatic hyper-parameter tuning case:



inference

latency overhead is less than 5ms



Fast and Scalable In-memory Deep Multitask Learning via Neural Weight Virtualization

Seulki Lee

University of North Carolina at Chapel Hill

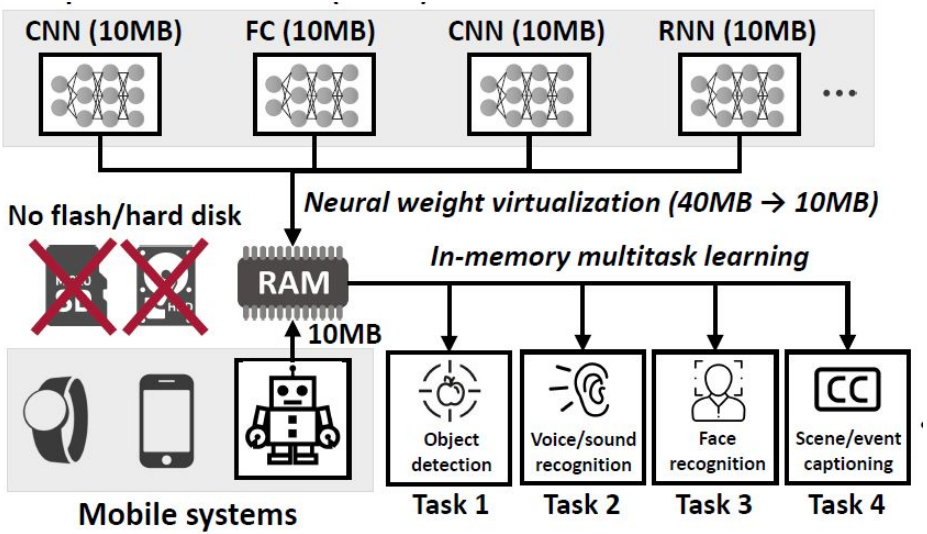
seulki@cs.unc.edu

Shahriar Nirjon

University of North Carolina at Chapel Hill

nirjon@cs.unc.edu

Problem



Multiple machine learning tasks running on the same embedded system. However, embedded system usually do not have powerful processor and sufficient memory to handle multiple DNN models running at the same time.

Existing solutions

Compression and pruning DNNs:

Weakness:

1. Still only can fit limited DNNs in main memory of embedded system.
2. DNNs can not benefit from knowledge transfer.
3. Need extra time and effort to compression and pruning the DNNs.

Existing solutions

Multitask learning:

DNN1 DNN2 DNN3 => a new single DNN

Weakness:

1. Only similar architecture and similar performing can be use multitask learning.
2. Limited compression

Existing solution

Use second level memory:

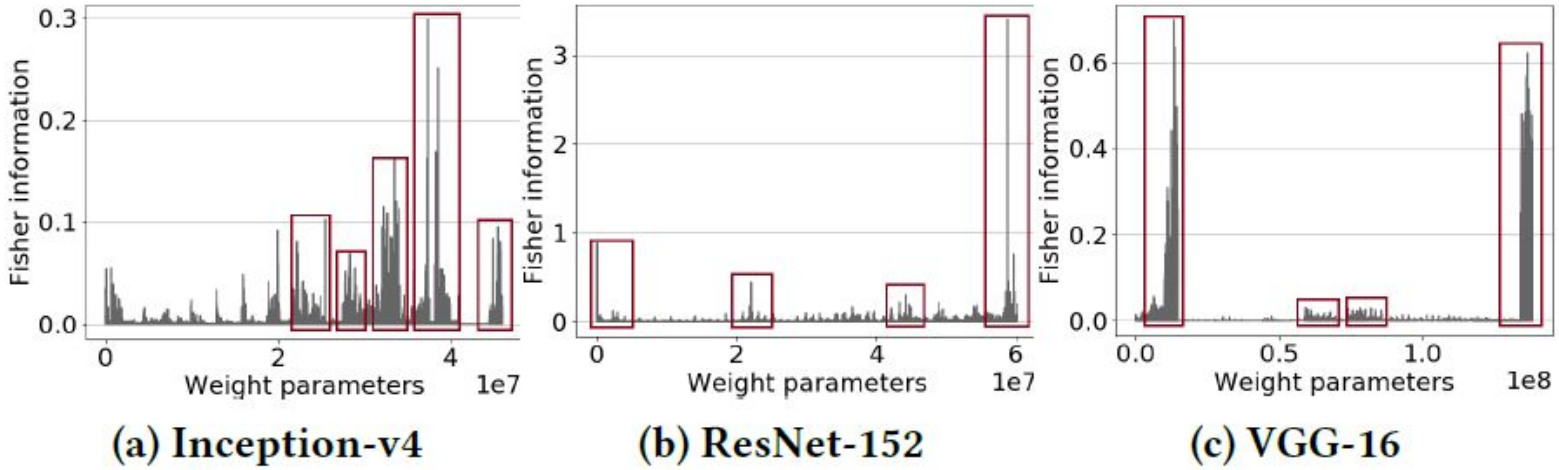
Some DNNs reside in the flash or hard disk.

Weakness:

Frequent swapping cost a lot of overhead.

Observation

1. Only a few weighs affect the final accuracy
2. Important weigh is more likely to be located nearly



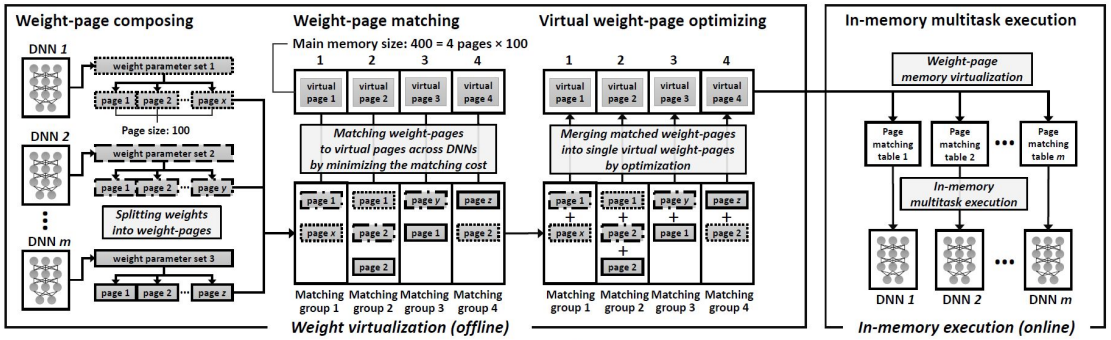
(a) Inception-v4 (b) ResNet-152 (c) VGG-16

Framework

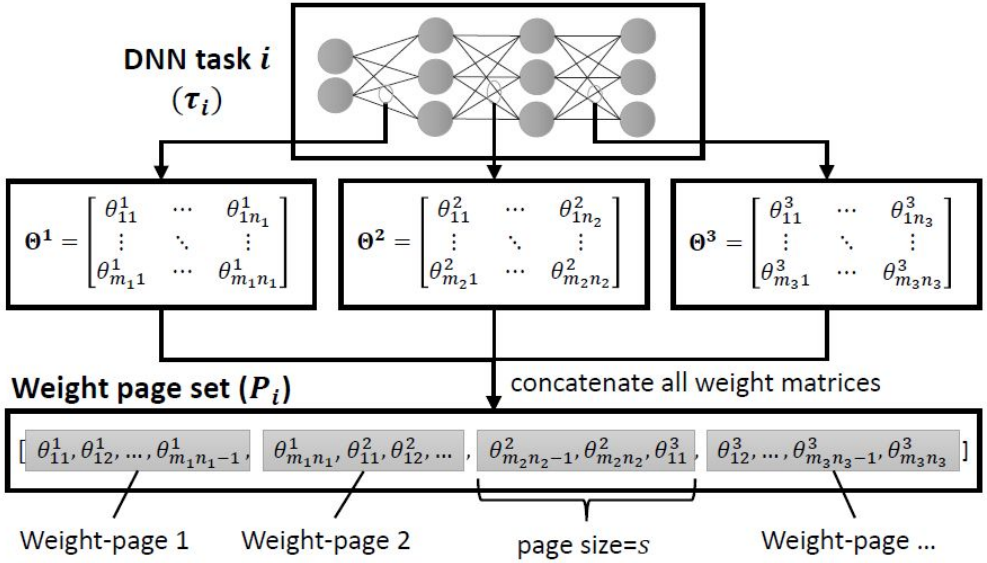
1.split weights into pages

2.match page to virtual pages

3.optimize virtual pages



1.split DNNs weights into pages



Weight-page matching

matching cost function:

$$C(p,q) = \kappa \sum_{(\theta_p \in p, \theta_q \in q)} (\theta_p - \theta_q)^2 (\tilde{F}(\theta_p|\tau_p) + \tilde{F}(\theta_q|\tau_q))$$

Matching the task-weights pages to virtual weight-pages by minimize the summation of cost between task-pages and virtual-pages.

($\theta p - \theta q$): difference between two weights, smaller is better
($\tilde{F}(\theta p\,|\tau p\,) + \tilde{F}(\theta q\,|\tau q)$: Quantify the importance of weights

Virtual pages optimization

$$\mathcal{L}(\tau_i) = \mathcal{L}_o(\tau_i) + \kappa \sum_{p \in P_i} \frac{1}{|Q_{i,p}|} \sum_{q \in Q_{i,p}} \sum_{\theta \in q} (\theta - \theta^*)^2 \tilde{F}(\theta | \tau_q) \quad (5)$$

L_o (τi): original loss without virtualization

$$Q_{i,p} = \{ q \in \bigcup_{j=1, j \neq i}^{m} P_j \mid \exists f_j^{-1}(f_i(p)) \}$$

Evaluation

- Evaluation was done on an RTX 2080 Ti (fancy!) and an i9-9900K (ew, intel)
- Variety of datasets, including:
  - ImageNet
  - VGGFace2
  - MS COCO
  - Google Speech Commands
  - and many more
- Different Architectures

Performance

Weight-page matching

- Weight page accuracy 1,140x random (doesn’t mean a whole ton on it’s own)
- 72% inference accuracy improvement on mobile robot (means much more)
- Demonstrates that proper weight-page matching is very important for performance reasons during inference

System

- Memory packing efficiency: 4.13x better packing ratio vs non-virtualized DNNs (about 80% compression).
- Nearly identical inference performance, maximum 3% drop among tested models.
  - Drop in accuracy caused by compression
  - Lack of farther drop offset by lack of overfitting due to compression.
- 36.9x faster successive execution (due to all models being stored in ram. File IO is expensive!)

Performance (IOT)

System

- Memory packing efficiency: 4.04x better packing ratio vs non-virtualized DNNs (about 80% compression).
- 7413x reduction in energy usage (No there isn’t supposed to be a decimal, file IO is very expensive)
- Nearly identical inference performance, maximum 3% drop among tested models (no difference from non-IOT).
    - Drop in accuracy caused by compression
    - Lack of farther drop offset by lack of overfitting due to compression.
- 1.76x faster successive execution

Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference

Yecheng Xiang and Hyoseung Kim University of California, Riverside

So what's this all about?

Problem:

- DNNs are very widely used
- Often used on heavy tasks
- Much focus has been put on making them smaller or take less space
- Not much effort has been put into speeding up inference
- Especially important when running multiple DNN inferences in parallel
- Problematic for tasks like autonomous driving

Solution:

- Implement the same scheduling algorithms originally used for process management on OS, but with a weight component wow, so cool!
- Use pipelines to implement basic scheduling algorithms
- Cluster CPU and GPU resources into heterogeneous nodes and distribute workload "intelligently".

Applications

- Self driving cars
- Cyber-physical systems
  - Automated targeting systems
  - Autonomous drone technology (like the BAE Taranis Jet)
  - LAWs (Lethal Autonomous Weapon)[1]
    - Robots that can decide who to kill, and kill them without human intervention
    - Illegal for offence as of 2018
    - Still legal for defence, see Israel's Samson automated turrets[2]
- “Other IOT devices, like vacuums”

1. https://en.wikipedia.org/wiki/Lethal_autonomous_weapon#Automatic_defensive_systems
2. https://www.wired.com/2008/12/israeli-auto-ki/

Restrictions

Key:
- Excusable
- Slightly problematic
- Very problematic

This paper makes some fairly strong assumptions about the target system, including:

- The system runs a standard ARM or x86 CPU system
- The system uses exclusively identical GPU and CPU cores
- The system's memory is shared among *all* CPUs
- All GPUs in the system must be Nvidia
- Each job runs a single inference, and each uses a single DNN
- The DNN has many stages (layers)
- Jobs can be classified into two priorities: Real Time or Best Effort

Example

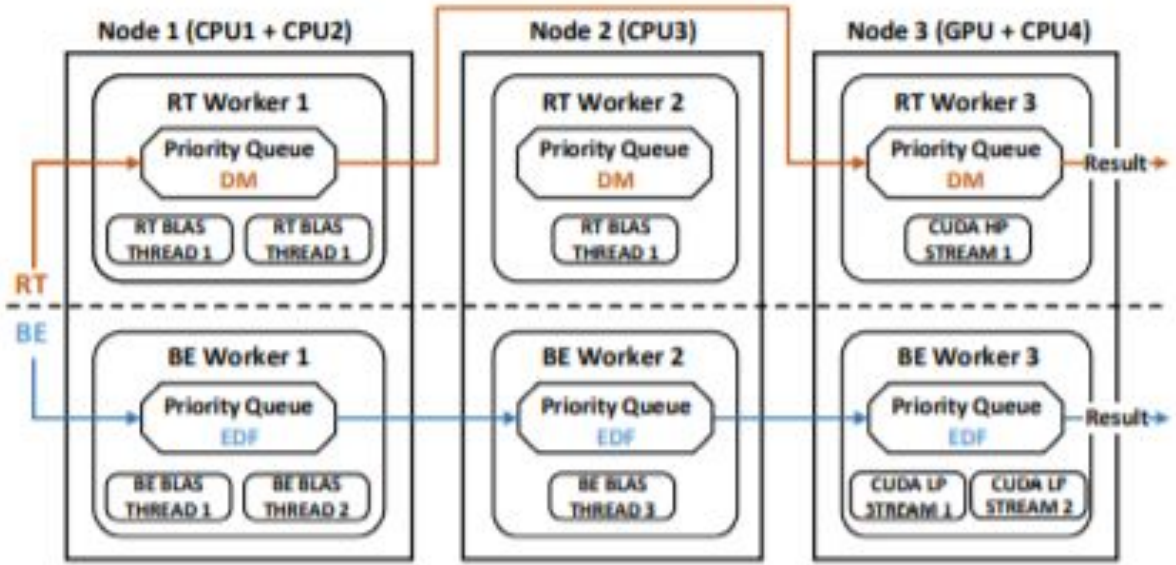
Here we see two jobs:

- RT: a “Real Time” job which must get done first.
- BE: A “Best Effort” job which can be done ASAP, but is distinctly lower than RT.

RT can be done in two stages, doesn’t use node 2

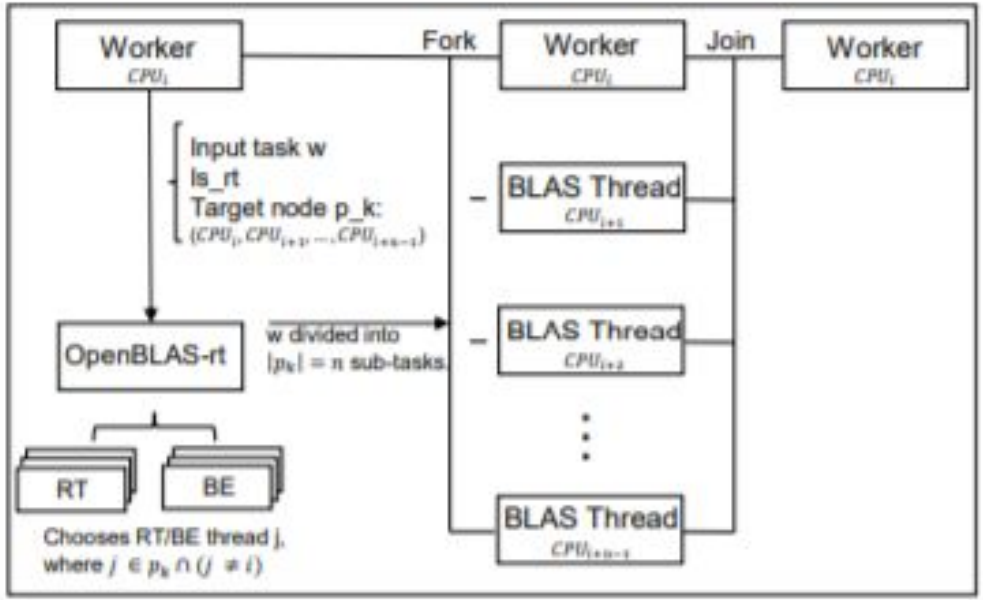
BE needs all three stages

Although both RT and BE use nodes 1 and 3, RT has the ability to pre-empt the BE task to get done first.



Note on BLAS

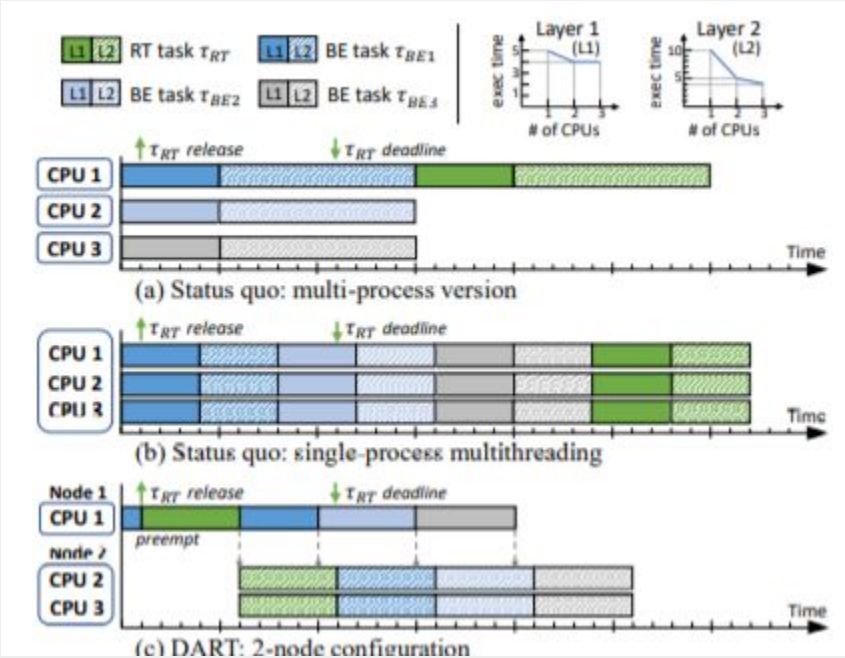
- BLAS is an idea for intra-node scheduling library for matrix operations.
- Allows for intra-node parallelization across CPU cores.
- Common library implementations include ATLAS and OpenBLAS
- This paper uses OpenBLAS
  - They don't like it
  - Limited priority levels[1]
- General multi-thread libs don't work because they don't support matrix operations



1. Sounds familiar, doesn't it?

Demo of why DART works

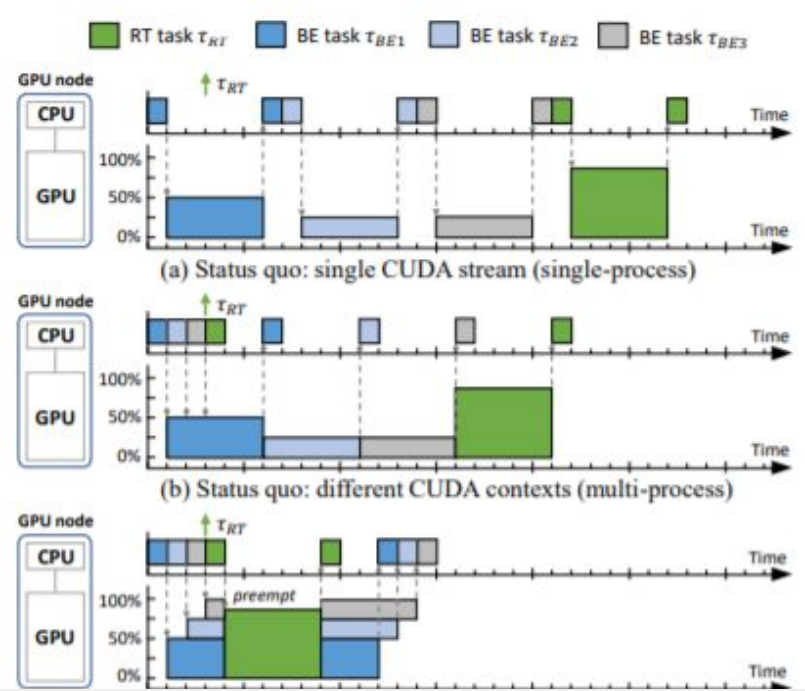
(DART is what they called their system). This is the CPU version



- In first example, we show how even on existing multithread solutions RT tasks can be delayed
- In second example we show how badly single-thread solutions botch things
- In third example we show how DART's proper dependency scheduling allows efficient task execution.

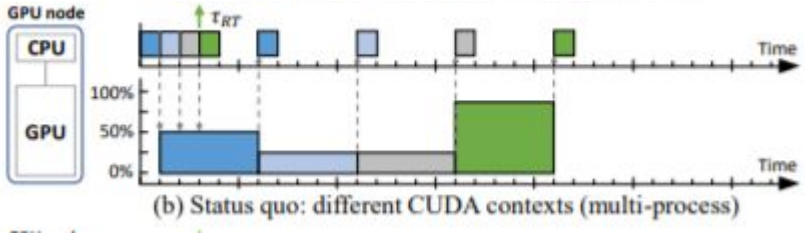
Demo of why DART works

(DART is what they called their system). This is the GPU version



- In first example, we show how single-thread solutions are bad
- In second example we that even multithread solutions don't deal well with CPU and split GPU workloads
- In third example we show how DART's proper dependency scheduling allows efficient task execution.

Now, I want to call something out on the previous slide:



(b) Status quo: different CUDA contexts (multi-process)

This was the “status quo” using multi-process GPU scheduling

This is wrong.

On modern GPUs, multiple processes can be run at the same time, up to the GPU’s capacity (100%). This doesn’t fix the issue with priority scheduling, but it’s worth noting the performance improvement here is greatly exaggerated.

They even reference it later in the paper, so they know it’s wrong…?

WARNING

The paper begins to incorrectly use the term “Response time” at this point. By “response”, they refer to the *completion* of a given job, not the start of it’s execution. I believe this is because they try to stick within the idea of a self-driving car

Node configuration algorithm

Algorithm 1 Find a Node Configuration for Tasks

Require: $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$: taskset
Require: $\mathbb{P}$: a set of candidate node configurations
Ensure: P^{sol}: a node config. found (solution); $P^{sol} = \emptyset$, if failed.

function FIND_NODE_CONFIGURATION(Γ, $\mathbb{P}$)
  $P^{sol} = \emptyset$ /* initialization */
  $W^{sol} = \infty$ /* weighted response time of RT tasks for P^{sol}
  for all $P \in \mathbb{P}$ **do**
    $N_P = |P|$
    Initialize $w[1...N_P]$
    for all $\tau_i \in \Gamma$ in descending order of U_i^{avg} **do**
      L_i = the number of layers of τ_i
      Compute $M[L_i, N_P]$ for τ_i by Eq. (7)
      Store the stage-to-node allocation of τ_i
      Update $w[1...N_P]$ with τ_i
    Γ^{RT} = a set of all RT tasks in Γ
    if $\forall \tau_i \in \Gamma^{RT}$ passes the schedulability test of Eq. (5) **then**
      $\forall \tau_i \in \Gamma^{RT}$, R_i = worse-case response time of τ_i
      $W = \sum_{\tau_i \in \Gamma^{RT}} (\pi_i / |\Gamma^{RT}|) * (R_i / D_i)$ /* π_i: priority */
      if $W < W^{sol}$ **then**
        $P^{sol} = P$
        $W^{sol} = W$
  return P^{sol}
end function

Algorithm 2 Generate Candidate Node Configurations

Require: $\mathcal{R} = \{c_1...c_n\}$: a set of available CPUs and GPUs
Require: N_P^{max}: the maximum number of nodes per config P
Ensure: $\mathbb{P}$: a set of node configurations

function GENERATE_CANDIDATE_CONFIGURATIONS(P)
  $\mathbb{P} = \emptyset$
  $\mathbb{V}$ = permutations of $\mathcal{R}$ with duplicate core types
  for all $V \in \mathbb{V}$ **do**
    for $k = 1$ to N_P^{max} **do** /* number of nodes per config */
      for all case Θ of $\binom{|V|}{k-1}$ **do** /* split V into k nodes */
        $\Theta = \{\theta_1, \theta_2, ..., \theta_{k-1}\}$
        /* each bracket [] in P indicates a node */
        $P = \{[c_1...c_{\theta_1 - 1}], [c_{\theta_1}...c_{\theta_2 - 1}], ..., [c_{\theta_{k-1}}...c_{|V|}]\}$
        if all nodes $\in P$ satisfies the system model **then**
          $\mathbb{P} = \mathbb{P} \cup P$
  return $\mathbb{P}$
end function

This is a lot of words. Good thing it's not actually that complex….

So what did all that mean?

“It computes the sum of the weighted worst-case response time of all RT tasks”

- Essentially a weighted implementation of Shortest Job First
  - None of the RT jobs will preempt another RT job
  - The goal here is to minimize lateness-- SJF is a *provably optimal algorithm* to do that
  - The difference here is that not all jobs are created equal
- The lack of preemption within RT is why this algorithm only supports two priorities. Otherwise, a higher weight would be the equivalent of a higher priority
- Note: if the algorithm cannot satisfy the time constraints, it *gives up*.

Relevant text:
“The algorithm uses this weight to better represent the relative importance of each RT task’s response time and to quantitatively compare different node configurations. Finally, the algorithm chooses the one with the minimum sum of weight response time and returns it.”

Worth noting:

This algorithm is called Weighted Shortest Job First (WSJF), and has been used in OS for decades.

Getting to the node creation

- Nodes are created based on the requirements of the jobs
- Prioritizes CPU+GPU pairs
- Splits nodes such that jobs can be shuffled around to avoid bottlenecks

Speaking of shuffling

The architecture, as it stands, can wind up being congested on a given node.

Take two nodes, one of CPU+GPU and one of 2xCPU. At first we have only one BE job, running on the GPU node. Along comes a RT job that needs GPU. What do?

We move the BE job to the CPU node, and run the RT on the GPU.

The shuffling algorithm is need based, moving the models along a “pipeline”. The order in which nodes are set up is defined at creation.

Performance testing

RT Only:

- Run all RT jobs, see which one minimizes weighted lateness.
- As this is the *exact* problem WSJF *optimally solves,* DART does significantly better, up to 98.5% better given a worst-case scenario with a single batch-fill bottlenecking the entire system. DART's interruption system beats that.

BE Jobs:

- To "make it fair", the industry leading solutions were given a priority queue to separate the two groups of jobs (RT and BE).
- DART achieves 17.5% higher throughput.
- This comparison is inherently flawed: the "priority queue" implemented by DART has exactly 2 priority levels, so new jobs can be inserted in O(1) time. A standard priority queue insertion takes O(n) time, where n is the *number of jobs*, which explains why they noticed higher runtime improvements on more jobs.

Thanks!