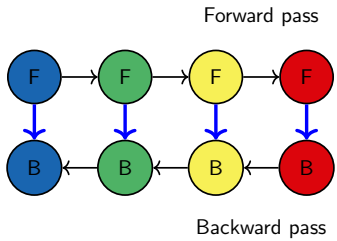
Optimal Steady-State Schedules for Memory-Efficient Pipeline Parallelism

Adrien Aguila–Multner, Olivier Beaumont,
Lionel Eyraud-Dubois, Julia Gusak

TOPAL team, Inria Centre at the University of Bordeaux

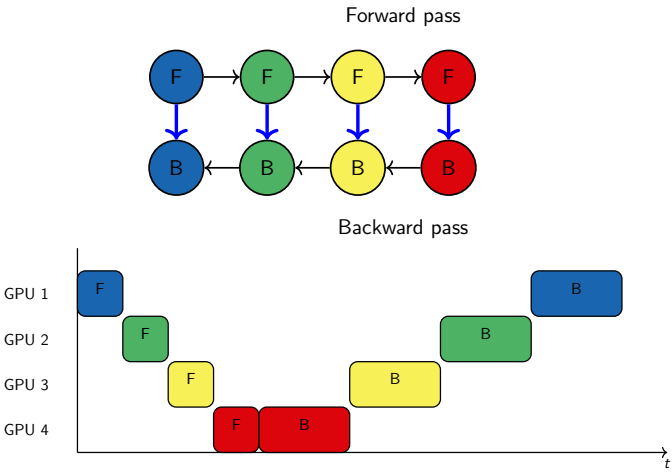
Pipeline Parallelism for LLMs



Training: same DAG many times

- Node: computations
- Edges: data dependencies

Pipeline Parallelism for LLMs



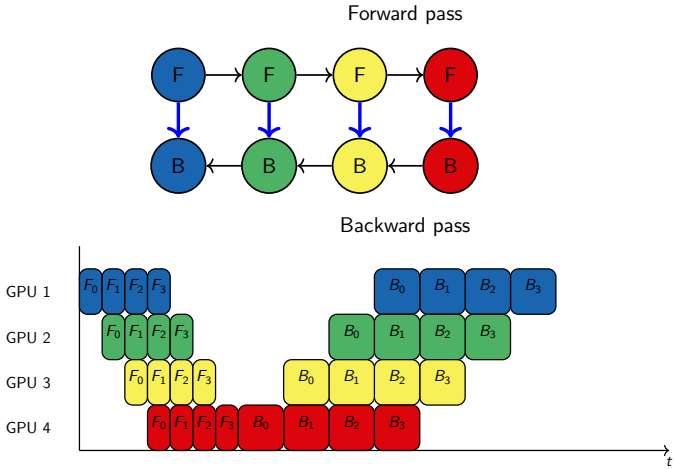
Training: same DAG many times

- Node: computations
- Edges: data dependencies

Pipeline Parallelism:

- split model into *stages*
- assign each stage to a different GPU
- assume identical stages

Pipeline Parallelism for LLMs



Training: same DAG many times

- Node: computations
- Edges: data dependencies

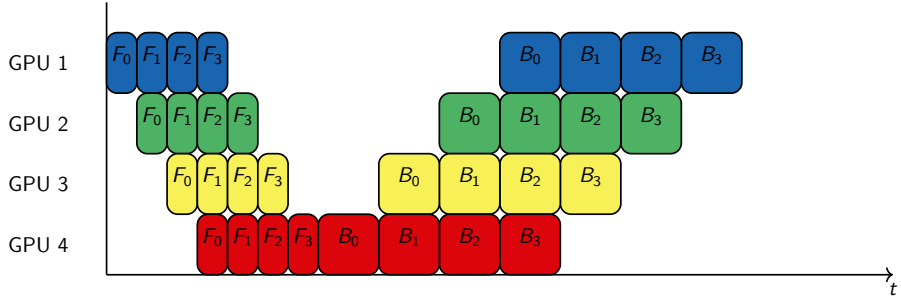
Pipeline Parallelism:

- split model into *stages*
- assign each stage to a different GPU
- assume identical stages

Split data into micro-batches

- number of micro-batches m
- F costs 1, B costs 2

Pipeline Parallelism



Two main issues

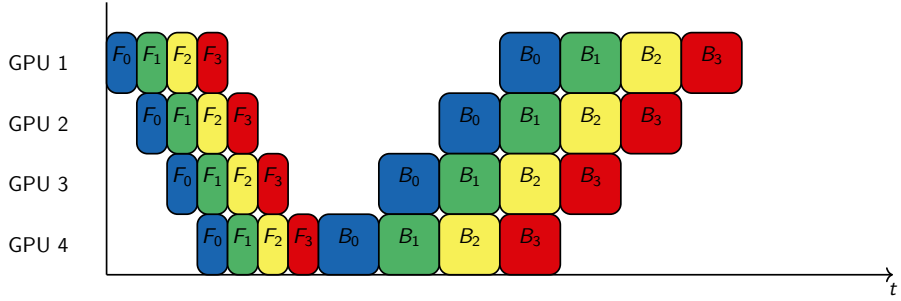
Scheduling:

- respect the ordering for each micro-batch
- but the relative ordering of different micro-batch is free

Memory occupation:

- increasing number m of micro-batches increases efficiency
- but also increases the memory pressure!

Pipeline Parallelism



Two main issues

Scheduling:

- respect the ordering for each micro-batch
- but the relative ordering of different micro-batch is free

Memory occupation:

- increasing number m of micro-batches increases efficiency
- but also increases the memory pressure!

Pipeline Parallelism when memory is too small

Solution to reduce memory usage: *re-materialization*

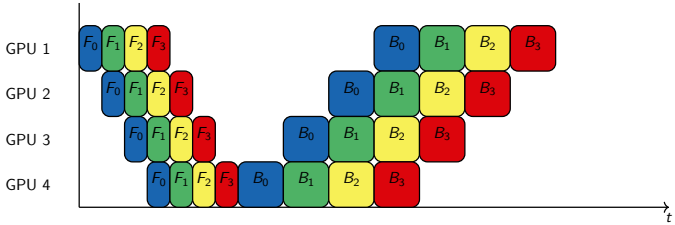
Delete intermediate results (*activations*), recompute them later

Rules of the game

Consider a fixed memory constraint: can store at most K activations in memory.
You can:

- Re-order the execution
- Re-compute (some of) the activations

in order to minimize makespan



Pipeline Parallelism when memory is too small

Solution to reduce memory usage: *re-materialization*

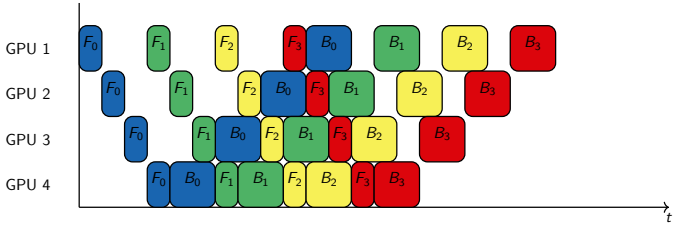
Delete intermediate results (*activations*), recompute them later

Rules of the game

Consider a fixed memory constraint: can store at most K activations in memory.
You can:

- Re-order the execution
- Re-compute (some of) the activations

in order to minimize makespan



Pipeline Parallelism when memory is too small

Solution to reduce memory usage: *re-materialization*

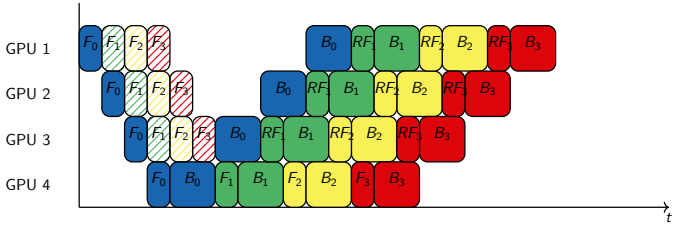
Delete intermediate results (*activations*), recompute them later

Rules of the game

Consider a fixed memory constraint: can store at most K activations in memory.
You can:

- Re-order the execution
- Re-compute (some of) the activations

in order to minimize makespan



Pipeline Parallelism when memory is too small

Rules of the game

Consider a fixed memory constraint: can store at most K activations in memory.
Setting: N GPUs, m micro-batches, K memory slots, durations t_F and t_B

Two ways of processing a micro-batch on a GPU: save or do not store and recompute

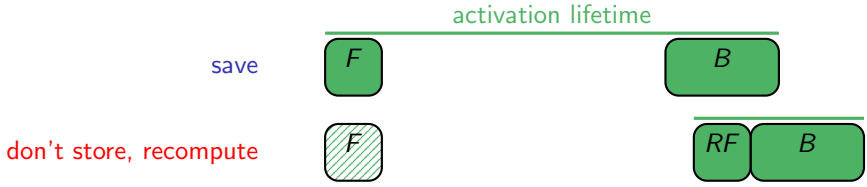


Table of Contents

How to obtain a lower bound on makespan

Setting: N GPUs, m micro-batches, K memory slots, durations t_F and t_B

Bound on computation time

If GPU i keeps a activations in total, its computation time C_i satisfies

$$C_i \geq \underbrace{m(t_F + t_B)}_{\text{base computation}} + \underbrace{(m - a)t_F}_{\textit{recomputations}}$$

How to obtain a lower bound on makespan

Setting: N GPUs, m micro-batches, K memory slots, durations t_F and t_B

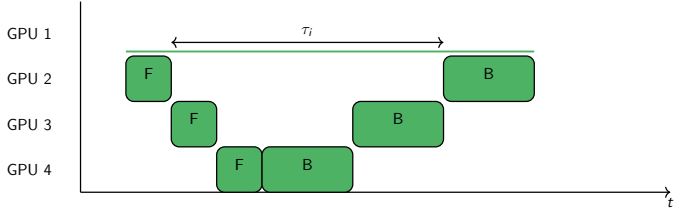
Bound on computation time

If GPU i keeps a activations in total, its computation time C_i satisfies

$$C_i \geq \underbrace{m(t_F + t_B)}_{\text{base computation}} + \underbrace{(m-a)t_F}_{\textit{recomputations}}$$

Bound on lifetime

An activation kept on GPU i stays in memory for at least an extra $\tau_i = (N-i)(t_F + t_B)$



How to obtain a lower bound on makespan

Setting: N GPUs, m micro-batches, K memory slots, durations t_F and t_B

Bound on computation time

If GPU i keeps a activations in total, its computation time C_i satisfies

$$C_i \geq \underbrace{m(t_F + t_B)}_{\text{base computation}} + \underbrace{(m-a)t_F}_{\textit{recomputations}}$$

Bound on lifetime

An activation kept on GPU i stays in memory for at least an extra $\tau_i = (N-i)(t_F + t_B)$

If GPU i keeps a activations in total, its global memory usage M satisfies

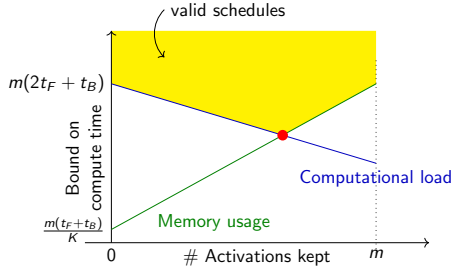
$$K \cdot C_i \geq M \geq m(t_F + t_B) + a \cdot \tau_i$$

How to obtain a lower bound on makespan

Two valid bounds:

$$C_i \geq m(2t_F + t_B) - a \cdot t_F \quad \text{compute}$$
$$K \cdot C_i \geq m(t_F + t_B) + a \cdot \tau_i \quad \text{memory}$$

Best tradeoff:
$C_i^* = \min_a \max(\text{compute}, \text{memory})$

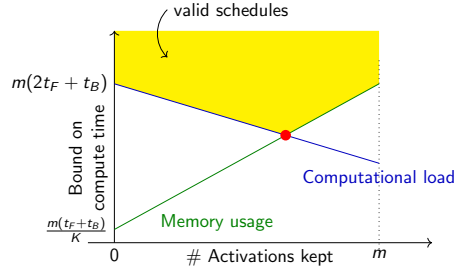


How to obtain a lower bound on makespan

Two valid bounds:

$$C_i \geq m(2t_F + t_B) - a \cdot t_F \quad \text{compute}$$
$$K \cdot C_i \geq m(t_F + t_B) + a \cdot \tau_i \quad \text{memory}$$

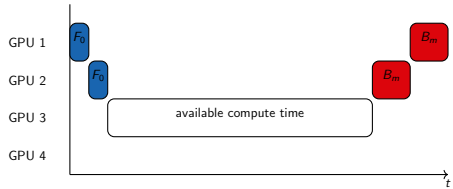
Best tradeoff:
$C_i^* = \min_a \max(\text{compute}, \text{memory})$



Critical GPU

Dependencies before/after computing.
Final bound:

$$\text{Makespan} \geq \max_i C_i^* + (i-1)(t_F + t_B)$$

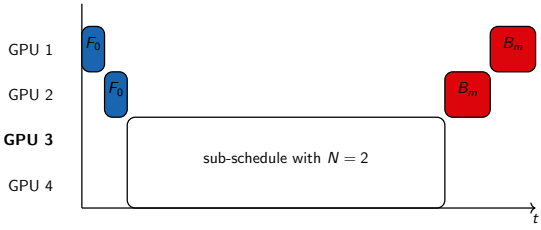


How tight is the bound? Can we reach it?

Sub-schedule

If GPU i^* is critical, an optimal schedule contains an optimal sub-schedule on $N - i^* + 1$ GPUs.

When $m = N$, critical GPU close to the end
$\Rightarrow$ we look at $m \gg N$



When $m \gg N$

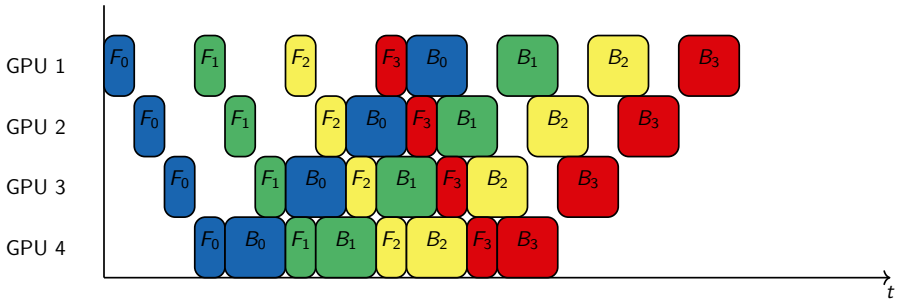
- GPU 1 is always critical
- The lower bound is linear in m, so we can compute

$\frac{a^*}{m}$ the proportion of activations saved (not recomputed)

$\frac{C^*}{m}$ the average time for processing one micro-batch

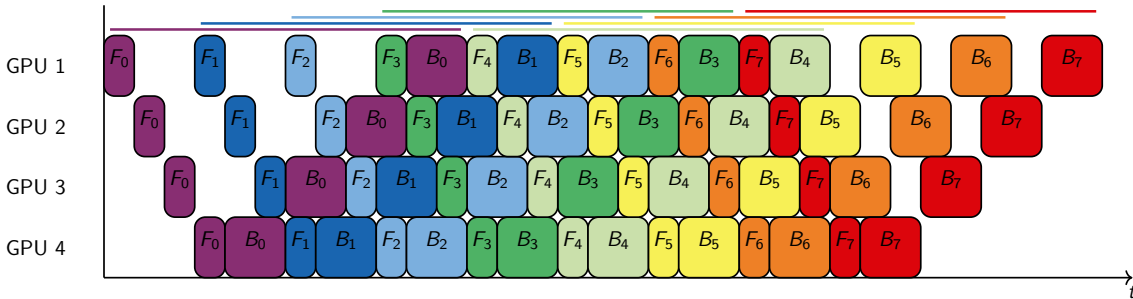
Easy case: optimal schedules for $N \leq K$

Just alternate F and B, no need for rematerialization



Easy case: optimal schedules for $N \leq K$

Just alternate F and B, no need for rematerialization



Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$

Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$

Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

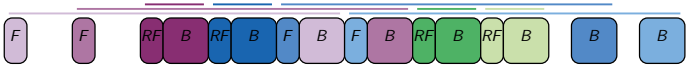
Example for $N = 5$ $K = 3$, $\tau_1 = 12$

Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$

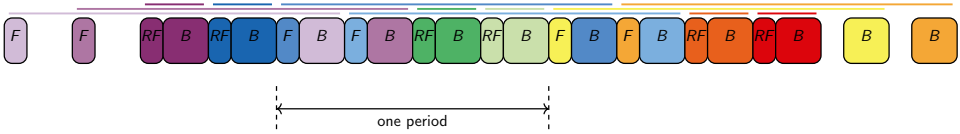


Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$

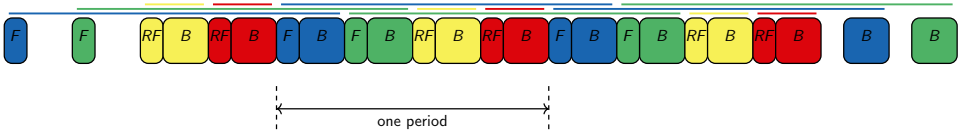


Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$

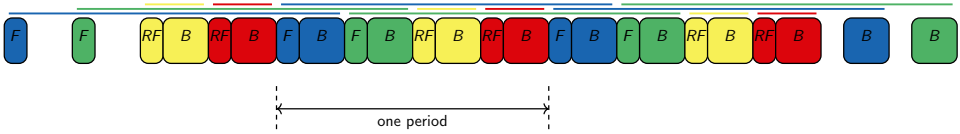


Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$



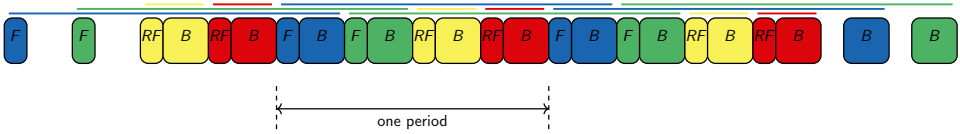
Very efficient, but unsustainable: when are the pre-computations performed?

Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

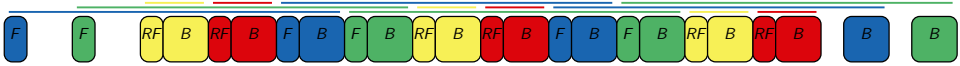
Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$



Very efficient, but unsustainable: when are the pre-computations performed?

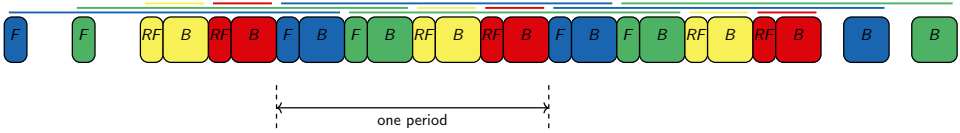


Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$



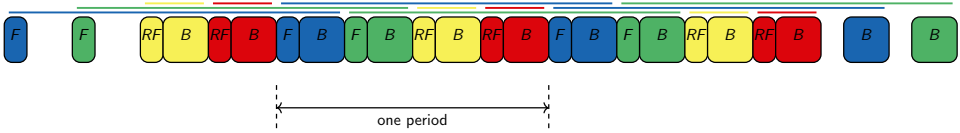
Very efficient, but unsustainable: when are the pre-computations performed?

Building optimal steady-state schedules $(t_F = 1, t_B = 2)$

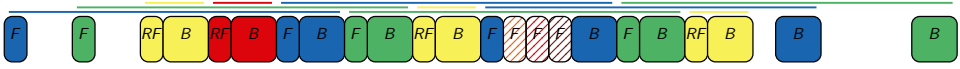
Focus first on the critical GPU. Matching the bound requires no idle time and always full memory

Main constraint: correct delay between F and B

Example for $N = 5$ $K = 3$, $\tau_1 = 12$



Very efficient, but unsustainable: when are the pre-computations performed?



- Need to perform the correct amount of precomputations
- Need to make sure that the operations "line up" correctly

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

- 7 

transitions (type 1), and
- 2 

transitions (type 2)

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

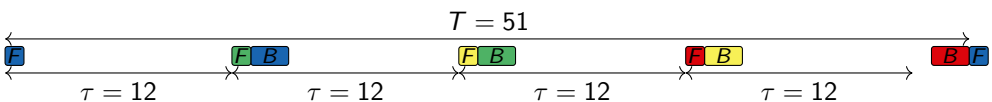
- 7 

transitions (type 1), and
- 2 

transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

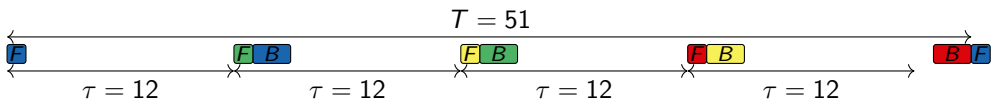
- 7 

transitions (type 1), and
- 2 

transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

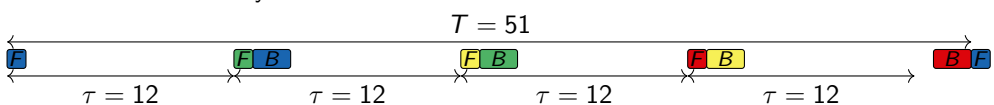
For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

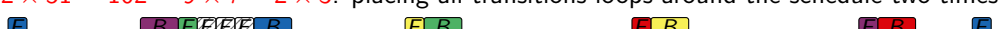
- 7  transitions (type 1), and
- 2  transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.



Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

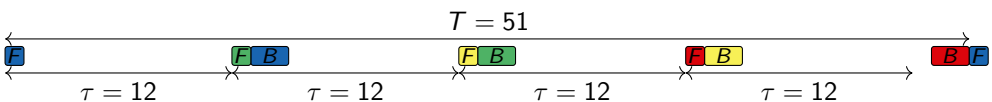
For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

- 7  transitions (type 1), and
- 2  transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

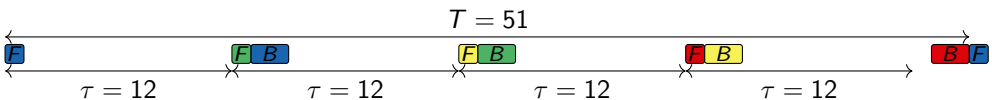
- 7 

transitions (type 1), and
- 2 

transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

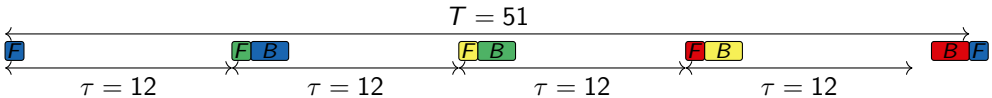
- 7 

transitions (type 1), and
- 2 

transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

- This is general: $\underbrace{(K-1)}_{\#\text{ loops}} \times \underbrace{3(4N-3)}_{\text{schedule length}} = \underbrace{(4K-3)}_{\#\text{ saved activations}} \times \underbrace{3(N-1)}_{\tau} - \underbrace{3(N-K)}_{\#\text{ recomputations}}$

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

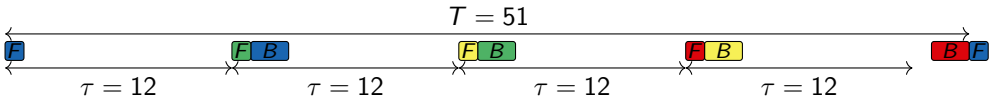
For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

- 7  transitions (type 1), and
- 2  transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.



- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

- This is general: $\underbrace{(K-1)}_{\#\text{ loops}} \times \underbrace{3(4N-3)}_{\text{schedule length}} = \underbrace{(4K-3)}_{\#\text{ saved activations}} \times \underbrace{3(N-1)}_{\tau} - \underbrace{3(N-K)}_{\#\text{ recomputations}}$
- Memory usage: 3 during transitions, 2 otherwise

Building optimal steady-state periodic schedules – example $N = 5$, $K = 3$

For $N = 5$ $K = 3$, the bound is $C^*/m = 51/15$. Target: 15 micro-batches in 51 time steps

$15 \cdot (t_F + t_B) = 15 \cdot 3 = 45$. We have time for 6 recomputations, so we must place 9 saved activations:

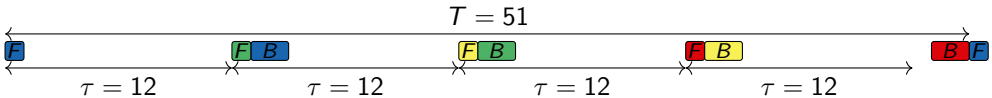
- 7 

transitions (type 1), and
- 2 

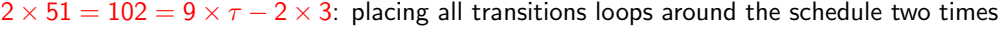
transitions (type 2)

We can do it because:

- $51 = 3 \times 17$ and $\tau = 12 = 3 \times 4$. So that $4\tau = 3 \times 16 = 51 - 3$, so 4 successive type 1 transitions finish exactly 3 time units before the first F.

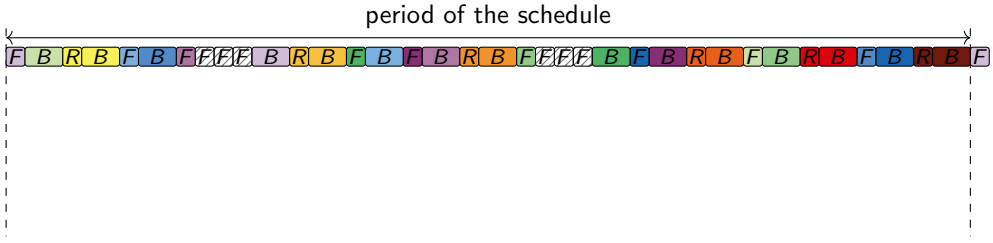


- $2 \times 51 = 102 = 9 \times \tau - 2 \times 3$: placing all transitions loops around the schedule two times.

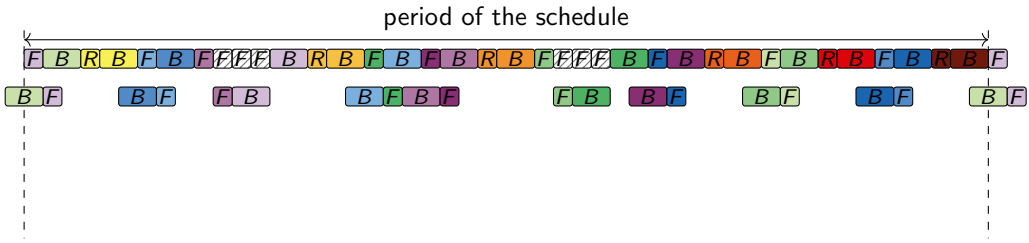


- This is general: $\underbrace{(K-1)}_{\#\text{ loops}} \times \underbrace{3(4N-3)}_{\text{schedule length}} = \underbrace{(4K-3)}_{\#\text{ saved activations}} \times \underbrace{3(N-1)}_{\tau} - \underbrace{3(N-K)}_{\#\text{ recomputations}}$
- Memory usage: 3 during transitions, 2 otherwise

From this schedule on the critical GPU, how to obtain a schedule for the others?

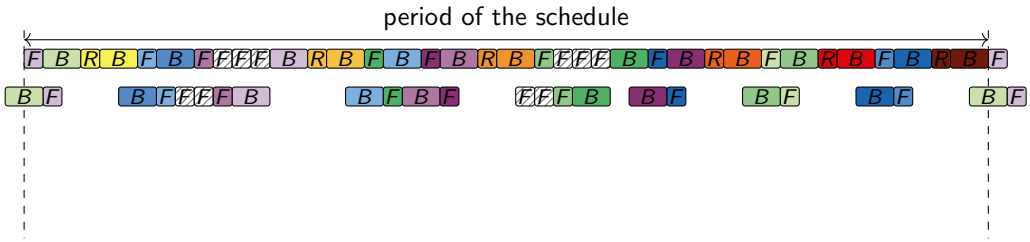


From this schedule on the critical GPU, how to obtain a schedule for the others?



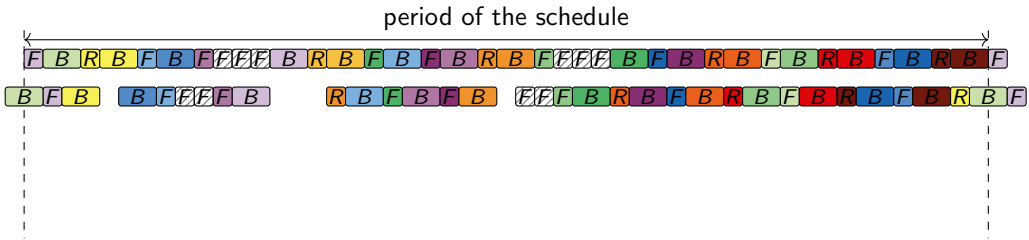
1. Match the F B to ensure dependencies (required)

From this schedule on the critical GPU, how to obtain a schedule for the others?



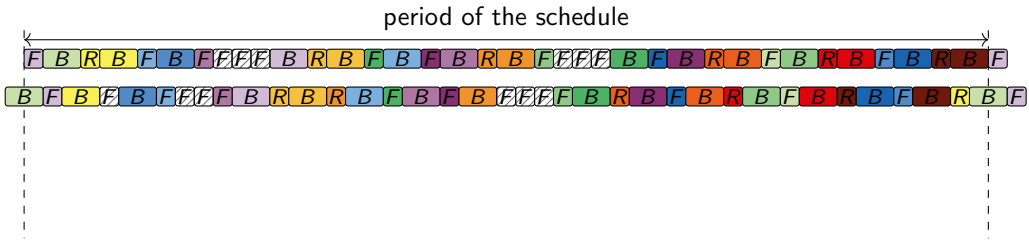
1. Match the F to B to ensure dependencies (required)
2. Match the F F which appear in B slots (choice)

From this schedule on the critical GPU, how to obtain a schedule for the others?



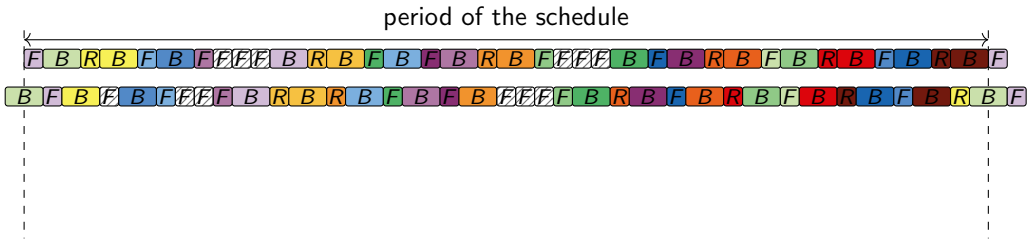
1. Match the F B to ensure dependencies (required)
2. Match the $F\,F$ which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)

From this schedule on the critical GPU, how to obtain a schedule for the others?



1. Match the F B to ensure dependencies (required)
2. Match the F F which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)
4. Add R B s and F s in all remaining slots

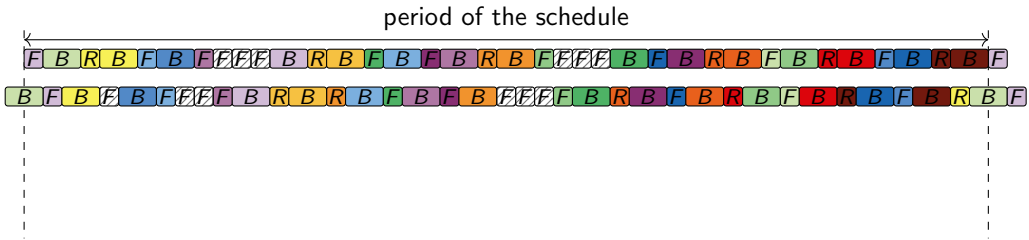
From this schedule on the critical GPU, how to obtain a schedule for the others?



1. Match the F with B to ensure dependencies (required)
2. Match the FF which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)
4. Add RB s and F s in all remaining slots

Step 1 reduces memory usage (smaller lifetime of activations), step 3 increases memory usage. The result remains below K.

From this schedule on the critical GPU, how to obtain a schedule for the others?

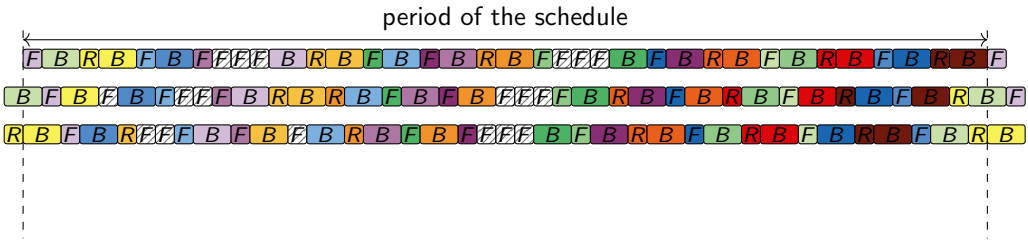


1. Match the F with B to ensure dependencies (required)
2. Match the F F which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)
4. Add R B s and F s in all remaining slots

Step 1 reduces memory usage (smaller lifetime of activations), step 3 increases memory usage. The result remains below K.

Iteratively applying this until the last GPU: valid schedule whose throughput matches the

From this schedule on the critical GPU, how to obtain a schedule for the others?

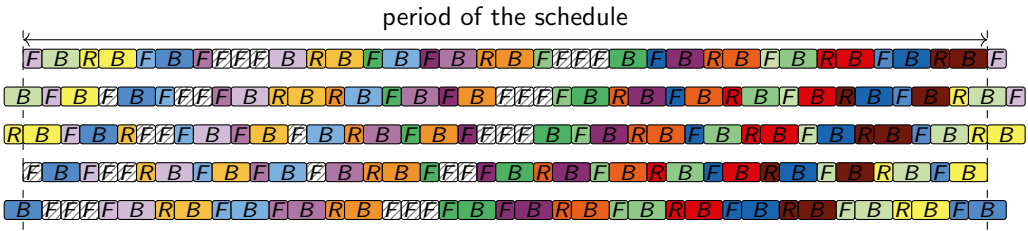


1. Match the F to ensure dependencies (required)
2. Match the F F which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)
4. Add R B s and F s in all remaining slots

Step 1 reduces memory usage (smaller lifetime of activations), step 3 increases memory usage. The result remains below K.

Iteratively applying this until the last GPU: valid schedule whose throughput matches the

From this schedule on the critical GPU, how to obtain a schedule for the others?



1. Match the F B to ensure dependencies (required)
2. Match the F F which appear in B slots (choice)
3. For each idle time after an F, add a B and a R in the first available slot before (trick)
4. Add R B s and F s in all remaining slots

Step 1 reduces memory usage (smaller lifetime of activations), step 3 increases memory usage. The result remains below K.

Iteratively applying this until the last GPU: valid schedule whose throughput matches the

Summary

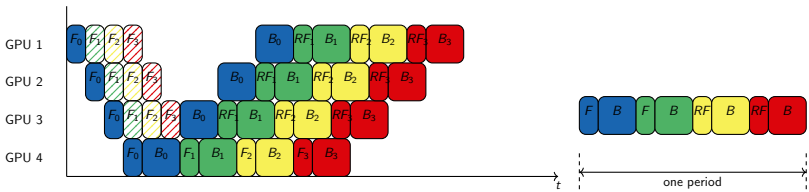
The bound is tight, except maybe for startup and ending.

For $m = \infty$

- Optimal periodic schedules (match the lower bound)
- Several schedules possible depending on the initial placement choice for $\overline{F}\overline{F}\overline{F}$ tasks

Valid schedules for fixed m

- Work in Progress
- Difficult part: transition between startup (with a stock of pre-computed $\overline{F}$ tasks) and steady-state – unsustainable high-throughput schedules have different periods!



What now?

Design and evaluate other algorithms

- Constraint Programming formulation
- Greedy algorithms
- Also consider multi-wave approaches – then the stage assignment matters too

Analyze decoupled backward – change the rules

- The B tasks can be decomposed into B and W, the W carry no dependencies
- But they still need activations as input

Consider more general task graphs

- Beyond the sequential dependencies: some NNs *do* contain graph parallelism

Evaluating lower bound for small cases (with a Constraint Programming formulation)