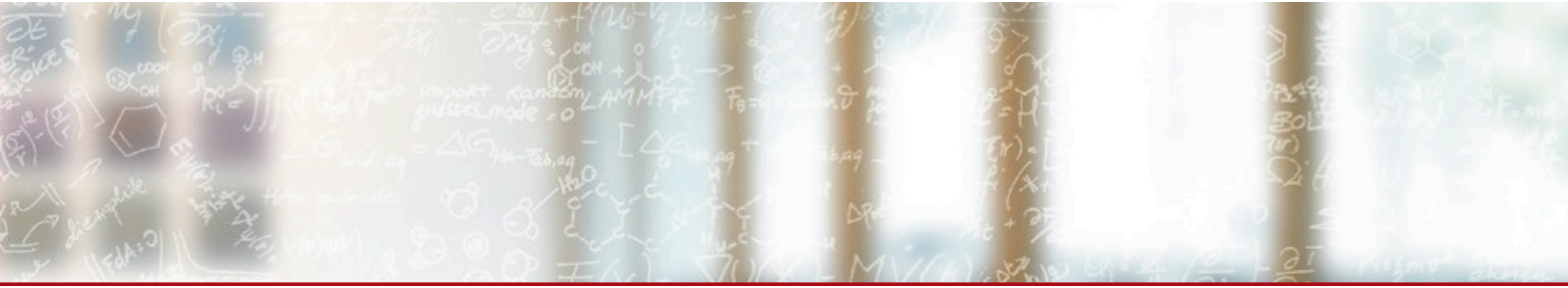
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Monitoring Nvidia GPU Resource Usage

Understanding performance metrics

Marcel Ferrari (CSCS)

February 15, 2024

Table Of Contents

1. Problem Overview
2. Existing Solutions
3. Nvidia DCGM
4. AGI - Alps GPU Insights
5. Performance metrics
6. Profiling workflow
7. Practical case & interpreting metrics

Problem

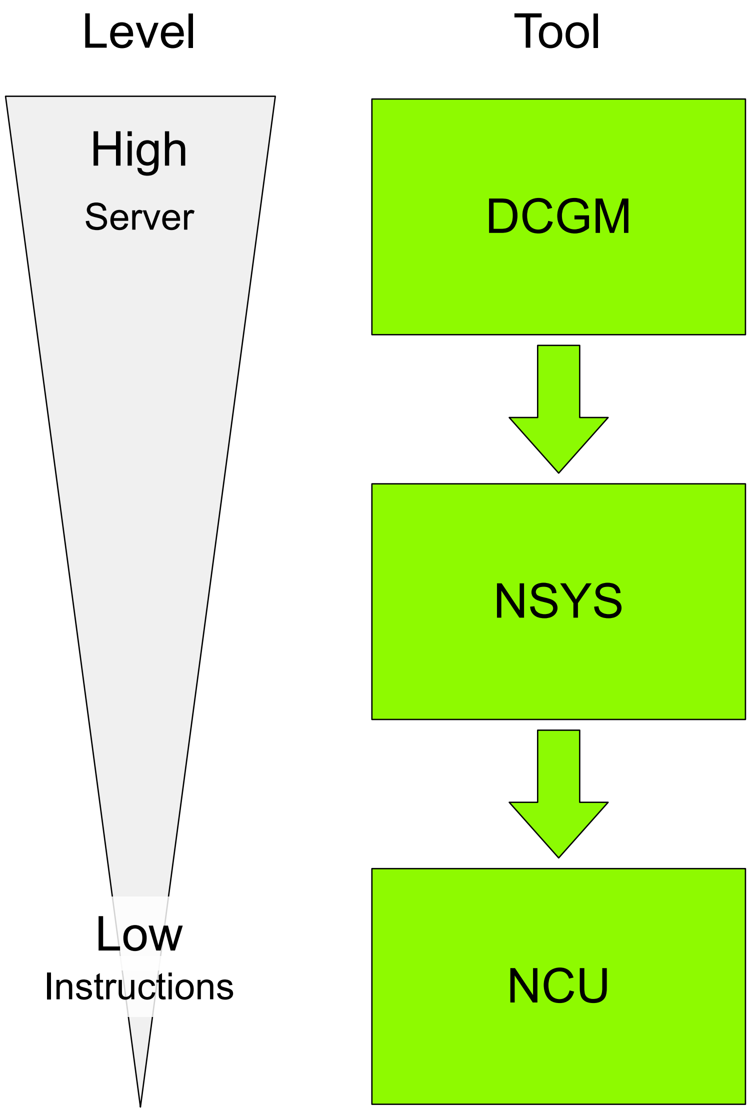
Problem Overview

- Number of workloads accelerated by GPUs is always increasing
- Need to understand how resources are used
- For CPUs ➜ look at core usage
- For GPUs ➜ look at GPU usage
    - This can be deceiving!
    - GPU usage = total work ≠ useful work!
    - Need a way to quantify **useful work**

Existing Solutions

Existing Solutions

- Nvidia offers different solution to profile its GPUs
- Nsight Compute
  - Very detailed kernel level information
  - High overhead and no MPI support
- Nsight Systems
  - Detailed function level information
  - Variable overhead ➜ depends on metrics
  - MPI support
- DCGM
  - Coarse grained information
  - System-wide monitoring support
  - Very little overhead



Source: ALCF Developer Session

DCGM

- **D**ata **C**enter **G**PU **M**anager
  - Open source project that enables easy GPU performance monitoring
- Nvidia GPU drivers offer access to hundreds of metrics
  - Eg.: temperature, power consumption, memory usage, …
  - DCGM offers APIs for easy access to “profiling” metrics
    - Handles communication with Nvidia drivers
- Why DCGM?
  - Introduces little overhead ➜ minimal code changes for users
  - Few simple to understand yet powerful profiling metrics
- Limitations:
  - Confusing and not user friendly ➜ designed for system administrators
  - No integration with SLURM

AGI - Alps GPU Insight

AGI - Alps GPU Insights

- **A**lps **G**PU **I**nsights
  - Goal: simplify and streamline the collection of GPU metrics
- Currently implemented key features:
  - Easy to run with minimal code changes
  - Simple options that are easy to understand
  - Very little overhead, can profile real workloads
  - SLURM and MPI integration
  - Automatic data analysis
    - Metrics overview summary
    - Process warmup and outlier detection
    - Time series averaging and visualisation
    - Load balancing visualisation

Profiling Metrics

Profiling Metrics

- DCGM offers hundreds of metrics
  - We are interested in the following **profiling** metrics:
    1. SM Activity
    2. SM Occupancy
    3. Tensor Activity
    4. FP64/32/16 Engine Activity
    5. DRAM activity
    6. PCIe Bandwidth
    7. (NVLink Bandwidth)

SM Activity

- The fraction of time **at least one warp** was **active** on a multiprocessor, averaged over all N multiprocessors:

$$SM_{\text{activity}} = \frac{1}{N}\sum_{i=1}^{N}\frac{t_{\text{active}}}{\Delta t}, \quad \Delta t = \text{sampling time}$$

- Active warps **include** those **waiting on memory**, not just computing
- A value ≥ 0.8 is necessary for effective GPU use, ≤ 0.5 suggests inefficiencies
- The value is not instantaneous!
  - SM = 1 ➜ 100% GPU activity over full time interval
  - SM = 0.2 could mean eg.:
    - 20% of SMs at 100% activity over full interval
    - 100% of SMs at 20% activity over full interval
    - 100% of SMs at 100% activity over 20% of interval
- Warp activity measurement **does not depend** on threads per block!

SM Occupancy

- Fraction of **resident warps** on a multiprocessor, **relative** to the **maximum** number of concurrent **warps supported** on a SM

$$SM_{\text{occupancy}} = \frac{1}{N \times \Delta t} \sum_{i=1}^{N} \int_{t_0}^{t_0+\Delta t} \frac{W_i^{\text{resident}}(t)}{W_i^{\text{max}}} dt$$

- Higher occupancy doesn't always mean better GPU usage
- For **memory bound** workloads, higher occupancy indicates more effective use
- For **compute bound** workloads, higher occupancy doesn't necessarily mean better usage
- Occupancy calculation **depends** on GPU properties, threads per block, registers per thread, and shared memory per block

Tensor Activity

- The fraction of cycles the tensor pipeline was active

$$TA = \frac{1}{N \times \Delta t} \sum_{i=1}^{N} \frac{1}{N_{\text{cycles}}} \sum_{j=1}^{N_{\text{cycles}}} \delta_j, \quad N_{\text{cycles}} = \frac{f \times \Delta t}{2}$$

$$f = \text{clock frequency}, \quad \delta_j = 1 \Leftrightarrow \text{instruction issued on cycle } j$$

- Higher values show greater Tensor Cores utilisation
  - 100% activity ➜ tensor instruction issued every other cycle throughout the interval
  - 20% activity could mean eg.:
    - 20% of SMs at 100% utilisation for 100% of the interval
    - 100% of SMs at 20% utilisation or 100% of the interval
    - 100% of SMs at 100% utilisation for 20% of the interval
- SM activity can help disambiguate these situations!

FP64/32/16 Engine Activity

- The fraction of cycles the FP64/32/16 pipeline was active

$$FP = \frac{1}{N \times \Delta t} \sum_{i=1}^{N} \frac{1}{N_{\text{cycles}}} \sum_{j=1}^{N_{\text{cycles}}} \delta_j, \quad N_{\text{cycles}} = \frac{f \times \Delta t}{k}$$

$f =$ clock frequency, $\delta_j = 1 \Leftrightarrow$ instruction issued on cycle j

- Higher values show greater FP engine utilisation
- On Volta arch. 100% utilisation means 1 instruction every…
  - … 4th cycle for FP64 calculations ➜ $k = 4$
  - … other cycle for FP32, Integer and FP16 calculations ➜ $k = 2$
- 20% can mean different scenarios ➜ see tensor core activity
- SM activity can help disambiguate these situations!

DRAM Activity

- The fraction of cycles where data was sent to or received from device memory

$$DRAM = \frac{1}{N \times \Delta t} \sum_{i=1}^{N} \frac{1}{N_{\text{cycles}}} \sum_{j=1}^{N_{\text{cycles}}} \delta_j, \quad N_{\text{cycles}} = f \times \Delta t$$

$f =$ clock frequency, $\delta_j = 1 \Leftrightarrow$ DRAM instruction on cycle j

- Higher values show greater device memory utilisation (not in terms of GB!)
  - Eg.: moving data from device memory to cache or registers
  - More likely that application is memory bound, although not always
- 100% utilisation means 1 DRAM instruction every cycle
  - Practical limit is 80%
- 20% activity indicates DRAM activity in 20% of the cycles ➜ no ambiguity!

PCIe Bandwidth

- The rate of data transmitted / received over the PCIe bus, including both protocol headers and data payloads, in **bytes per second**

$$PCIE_{\mathrm{BW}} = \frac{1}{\Delta t}\int_{t_0}^{t_0+\Delta t} TR(t)dt, \quad TR(t) = \text{transfer rate at time } t$$

- The value is an average rate over a time interval, not instantaneous
- The rate ignores whether data is transferred constantly or in bursts
  - Example: 1 GB transferred over 1 second equals a rate of 1 GB/s
- Theoretical maximum bandwidth for PCIe Gen3 is 985 MB/s per lane
- High PCIe bandwidth usage can be caused by
  - Heavy CPU-GPU communication
  - Heavy GPU-GPU communication for multi-GPU workloads (No NVLinks)

A Note On Sampling Frequency

- All metrics offered by DCGM are expressed either as time or cycle averages
  - **Shrinking sampling** time **interval** (i.e. increasing sampling frequency) **increases overhead**, but yields values closer to the true **instantaneous value**:

$$\frac{1}{\Delta t}\int_{t_0}^{t_0+\Delta t} f(t)dt \xrightarrow{\Delta t \to 0} f(t_0)$$

  - **Increasing sampling time interval** (i.e. decreasing sampling frequency) **decreases overhead**, but yields more aggregate values
- Need to find a balance between resolution and overhead!

Profiling Workflow

Profiling workflow

- 1) Profile workload using AGI
  - This generates an SQLite database containing the collected metrics
- 2) Use the analysis utility to generate a report
  - 2-dimensional data ➜ time and space (GPUs)
  - Three intuitive ways to aggregate data
    - Average over all dimensions ➜ point wise estimate
    - Average over space ➜ average time series
    - Average over time ➜ load balance heatmap

Profiling workflow

- Profile workload using AGI
    - This generates an SQLite database containing the collected metrics

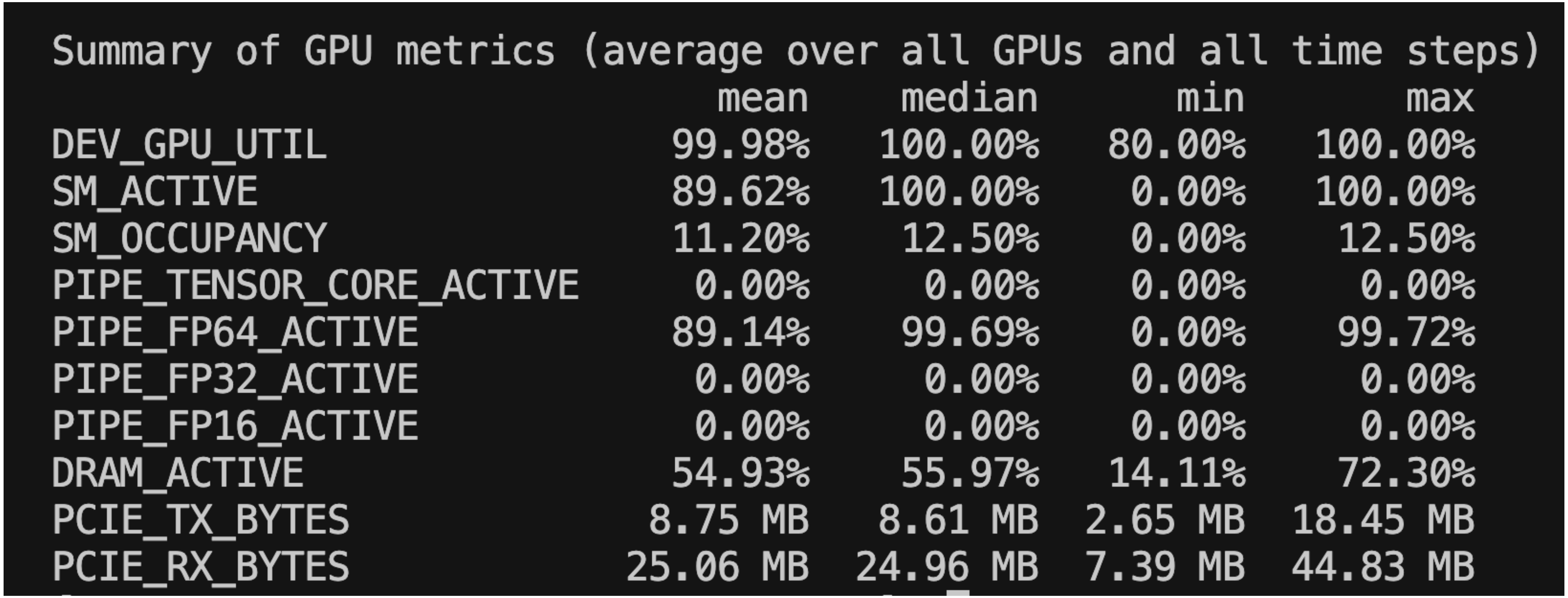
        -
          ```
          srun -N 2 -n8 --gpus-per-task=1 AGI --wrap=“python dgemm.py”
          ```
- Use the analysis utility to generate a report
    - 2-dimensional data ➜ time and space (GPUs)
    - Three intuitive ways to aggregate data
        - Average over all dimensions ➜ point wise estimate

            -
              ```
              AGI analyze --summary -i dgemm.sql
              ```
        - Average over space ➜ average time series

            -
              ```
              AGI analyze --plot-time-series -i dgemm.sql
              ```
        - Average over time ➜ load balance heatmap

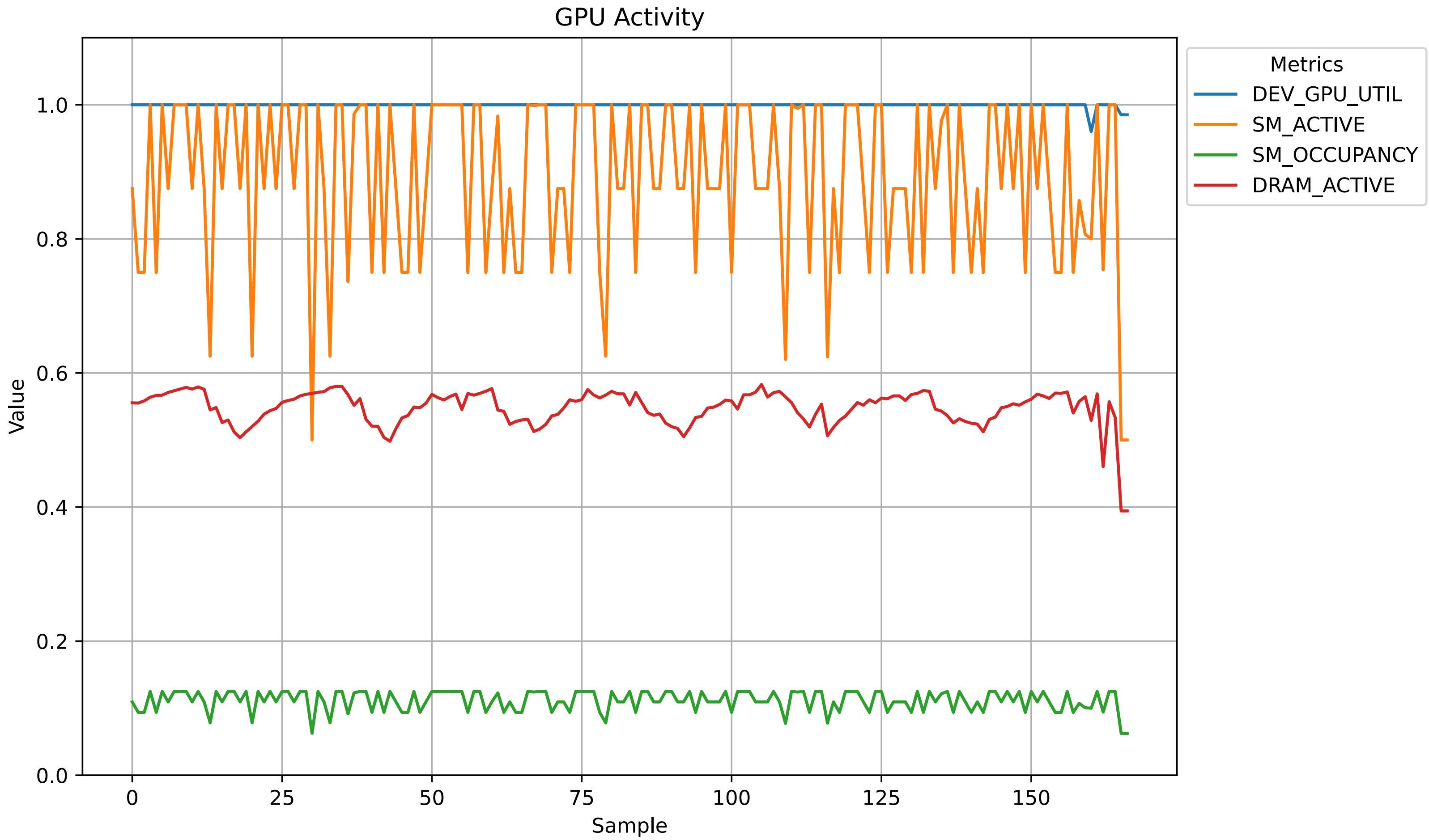
            -
              ```
              AGI analyze --plot-load-balancing -i dgemm.sql
              ```

Summary

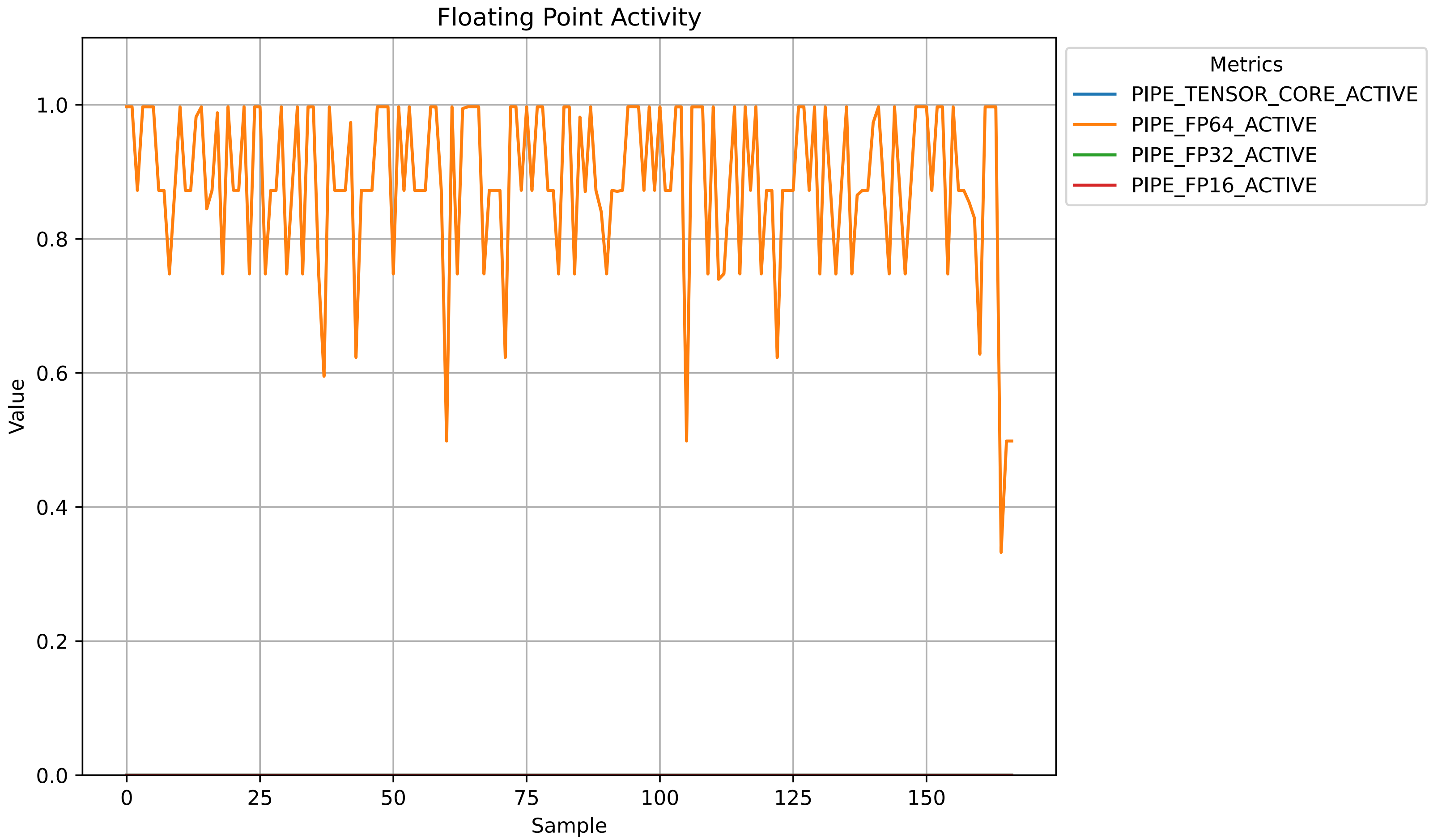


```
Summary of GPU metrics (average over all GPUs and all time steps)
                             mean      median        min         max
DEV_GPU_UTIL               99.98%     100.00%     80.00%     100.00%
SM_ACTIVE                  89.62%     100.00%      0.00%     100.00%
SM_OCCUPANCY               11.20%      12.50%      0.00%      12.50%
PIPE_TENSOR_CORE_ACTIVE     0.00%       0.00%      0.00%       0.00%
PIPE_FP64_ACTIVE           89.14%      99.69%      0.00%      99.72%
PIPE_FP32_ACTIVE            0.00%       0.00%      0.00%       0.00%
PIPE_FP16_ACTIVE            0.00%       0.00%      0.00%       0.00%
DRAM_ACTIVE                54.93%      55.97%     14.11%      72.30%
PCIE_TX_BYTES             8.75 MB     8.61 MB    2.65 MB    18.45 MB
PCIE_RX_BYTES            25.06 MB    24.96 MB    7.39 MB    44.83 MB
```

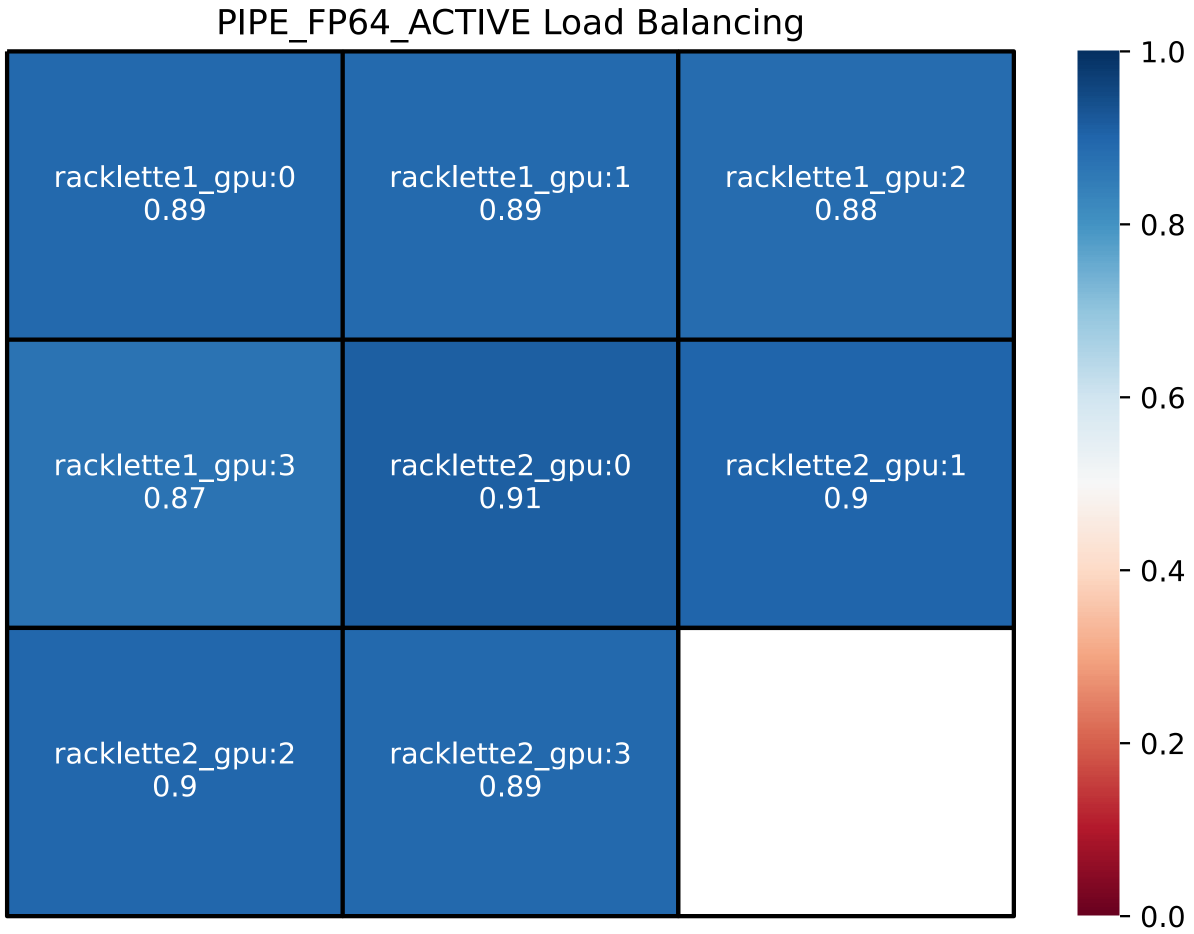
Average Time Series - Device Activity



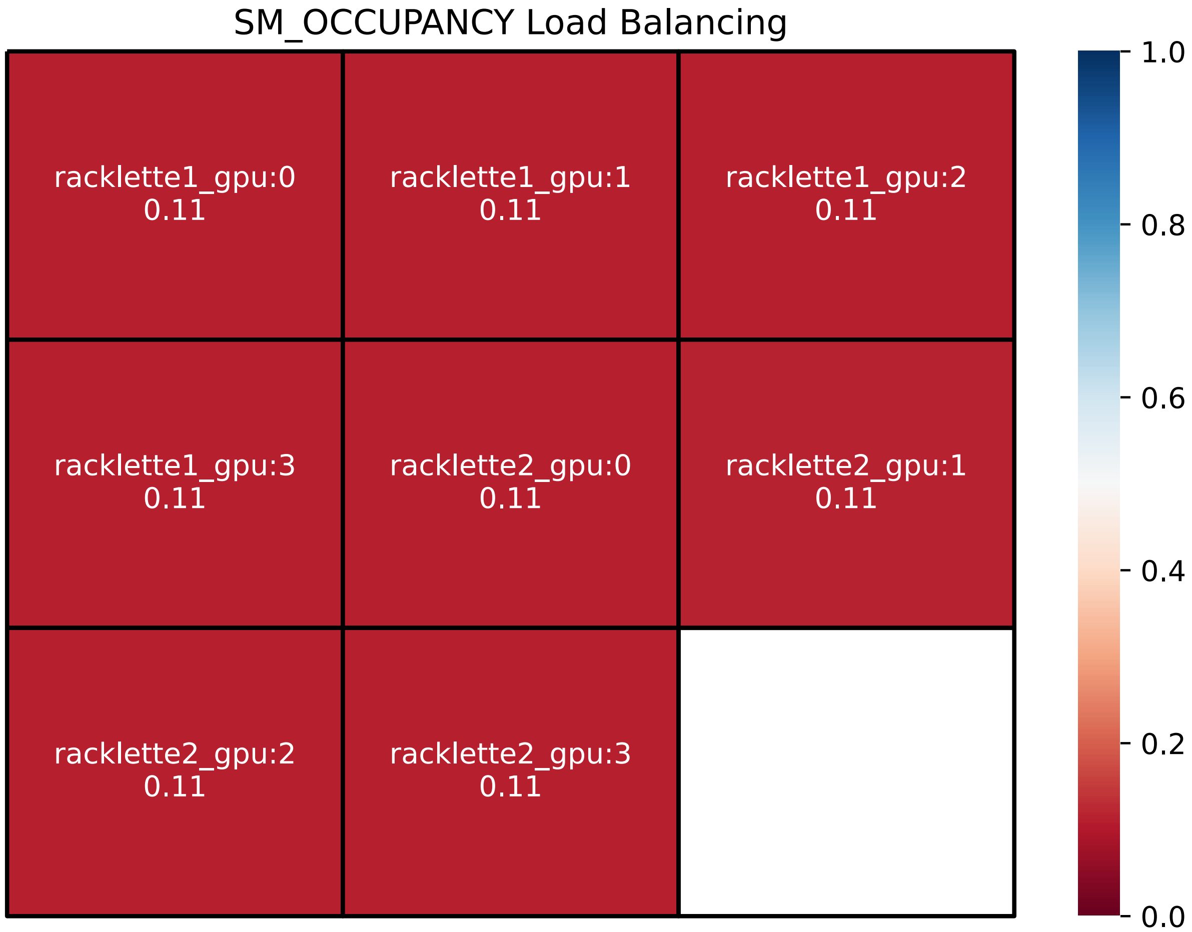
Average Time Series - FLOP Activity



Load Balancing - FP64 Engine Activity



Load Balancing - SM Occupancy



SM_OCCUPANCY Load Balancing

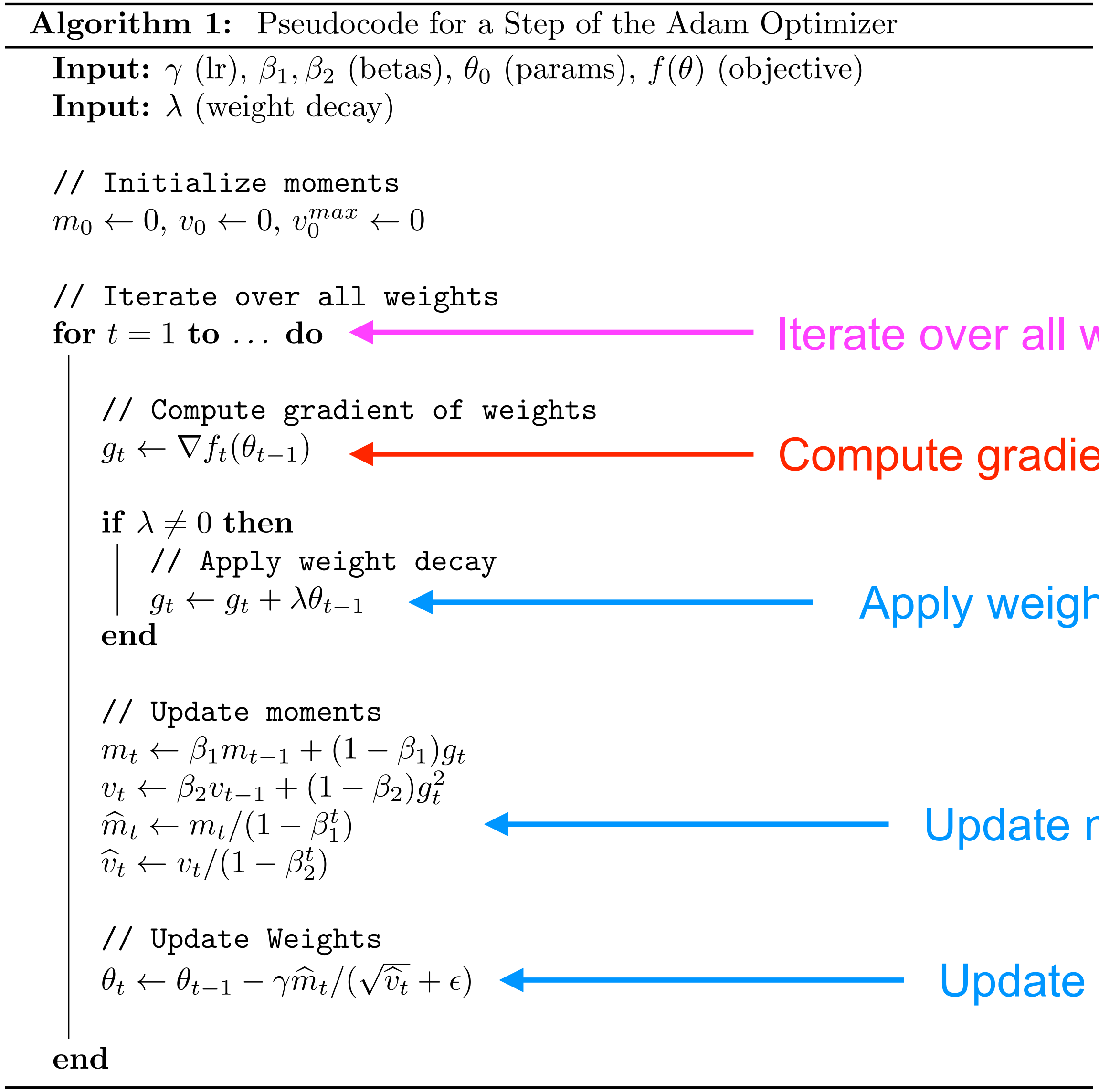
racklette1_gpu:0 0.11	racklette1_gpu:1 0.11	racklette1_gpu:2 0.11
racklette1_gpu:3 0.11	racklette2_gpu:0 0.11	racklette2_gpu:1 0.11
racklette2_gpu:2 0.11	racklette2_gpu:3 0.11	

Practical case: training a NN with different optimisers

A Practical Example

- Example data collected from a practical example workload
  - Goal: train a simple fully connected NN using different implementations of the Adam optimiser offered in PyTorch
    - Test three implementations of Adam
      - Standard Adam
      - Foreach Adam
      - Fused Adam

The Adam Optimiser



Algorithm 1: Pseudocode for a Step of the Adam Optimizer

```
Input: $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
Input: $\lambda$ (weight decay)

// Initialize moments
$m_0 \leftarrow 0,\ v_0 \leftarrow 0,\ v_0^{max} \leftarrow 0$

// Iterate over all weights
for $t = 1$ to $\dots$ do

    // Compute gradient of weights
    $g_t \leftarrow \nabla f_t(\theta_{t-1})$

    if $\lambda \neq 0$ then
        // Apply weight decay
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    end

    // Update moments
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1)g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$

    // Update Weights
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$

end
```

Iterate over all weights θ_t

Compute gradient via backpropagation

Apply weight decay

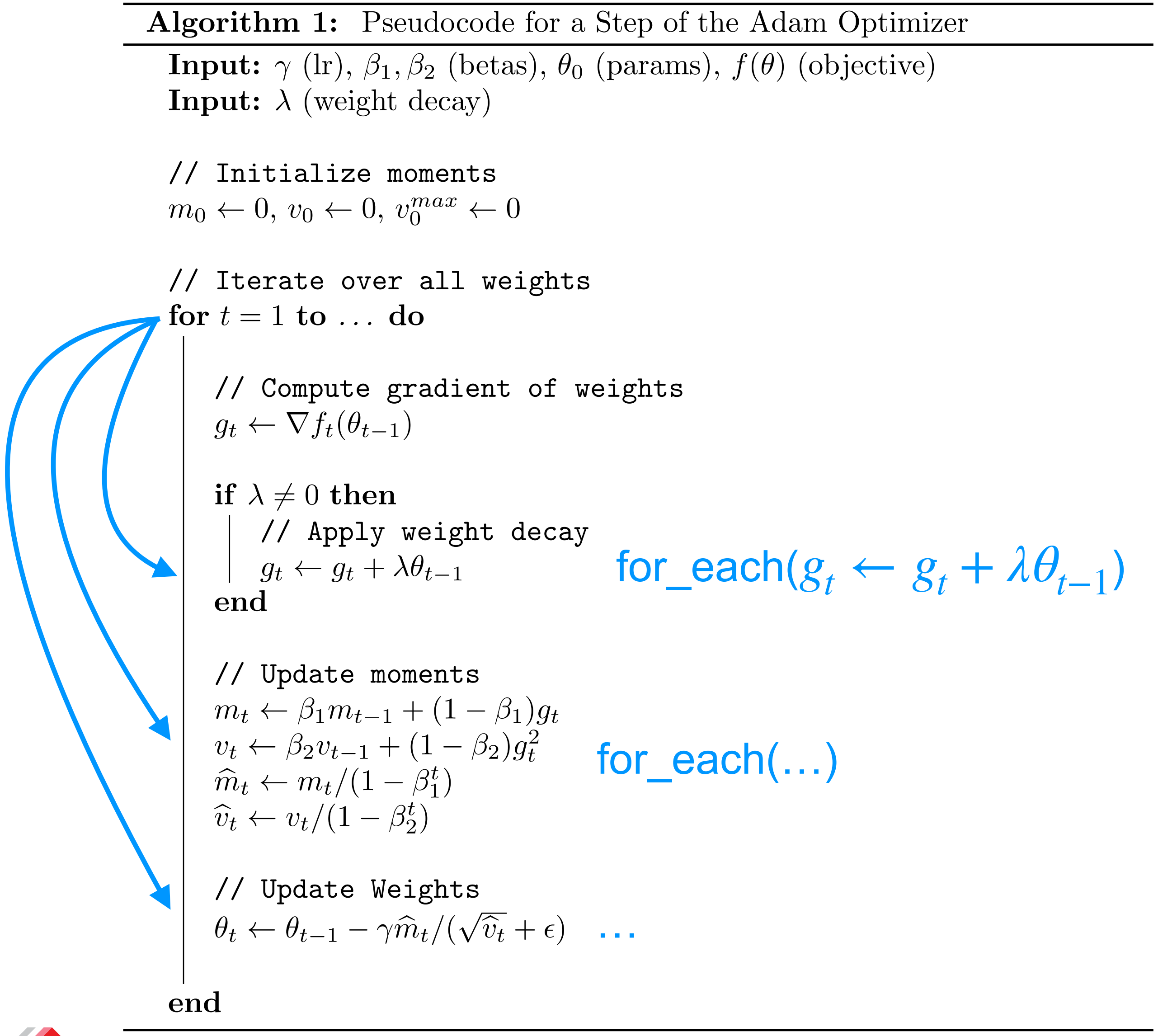
Update moments

Update weights

Computing $\nabla f_t\left(\theta_{t-1}\right)$ is the only loop dependency!

Idea: compute gradients a priori in order to parallelise the remaining the remaining operations

The Adam Optimiser - For Each Implementation



Algorithm 1: Pseudocode for a Step of the Adam Optimizer

```
Input: $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
Input: $\lambda$ (weight decay)

// Initialize moments
$m_0 \leftarrow 0,\ v_0 \leftarrow 0,\ v_0^{max} \leftarrow 0$

// Iterate over all weights
for $t = 1$ to $\dots$ do

    // Compute gradient of weights
    $g_t \leftarrow \nabla f_t(\theta_{t-1})$

    if $\lambda \neq 0$ then
        // Apply weight decay
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    end

    // Update moments
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
    $\widehat{m}_t \leftarrow m_t/(1-\beta_1^t)$
    $\widehat{v}_t \leftarrow v_t/(1-\beta_2^t)$

    // Update Weights
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$

end
```

for_each($g_t \leftarrow g_t + \lambda\theta_{t-1}$)

for_each(…)

…

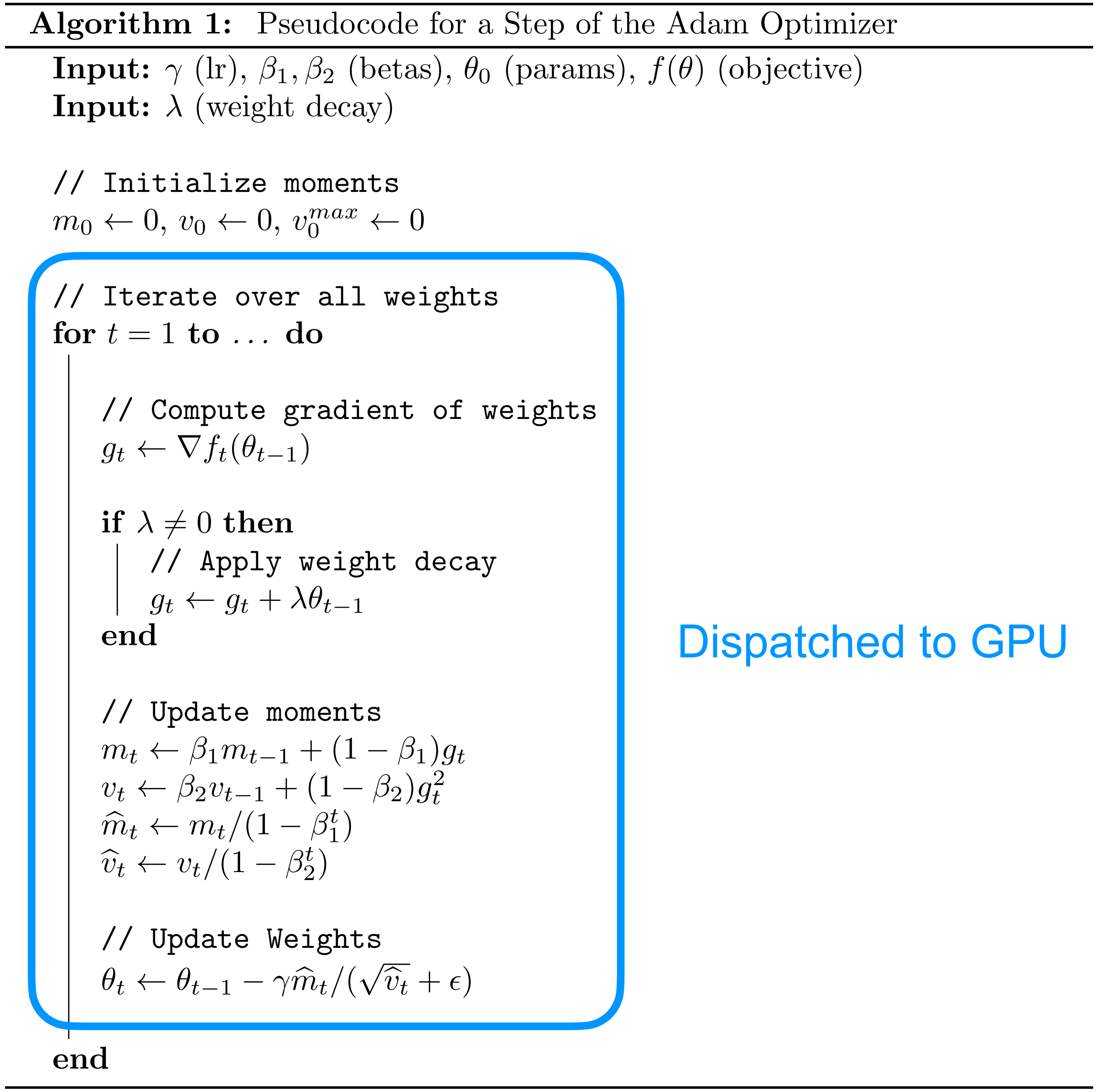
Option 1:

Distribute the outer for loop onto all internal operations. This allows to reduce the number of kernel calls by stacking tensors.

Eg.: $\begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} c \\ d \end{bmatrix}$ instead of $a + c, b + d$

This is known as the **for each** implementation.

The Adam Optimiser - Fused Implementation



Algorithm 1: Pseudocode for a Step of the Adam Optimizer

```
Input: $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
Input: $\lambda$ (weight decay)

// Initialize moments
$m_0 \leftarrow 0,\ v_0 \leftarrow 0,\ v_0^{max} \leftarrow 0$

// Iterate over all weights
for $t = 1$ to $\ldots$ do

    // Compute gradient of weights
    $g_t \leftarrow \nabla f_t(\theta_{t-1})$

    if $\lambda \neq 0$ then
        // Apply weight decay
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    end

    // Update moments
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
    $\widehat{m}_t \leftarrow m_t/(1-\beta_1^t)$
    $\widehat{v}_t \leftarrow v_t/(1-\beta_2^t)$

    // Update Weights
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$

end
```

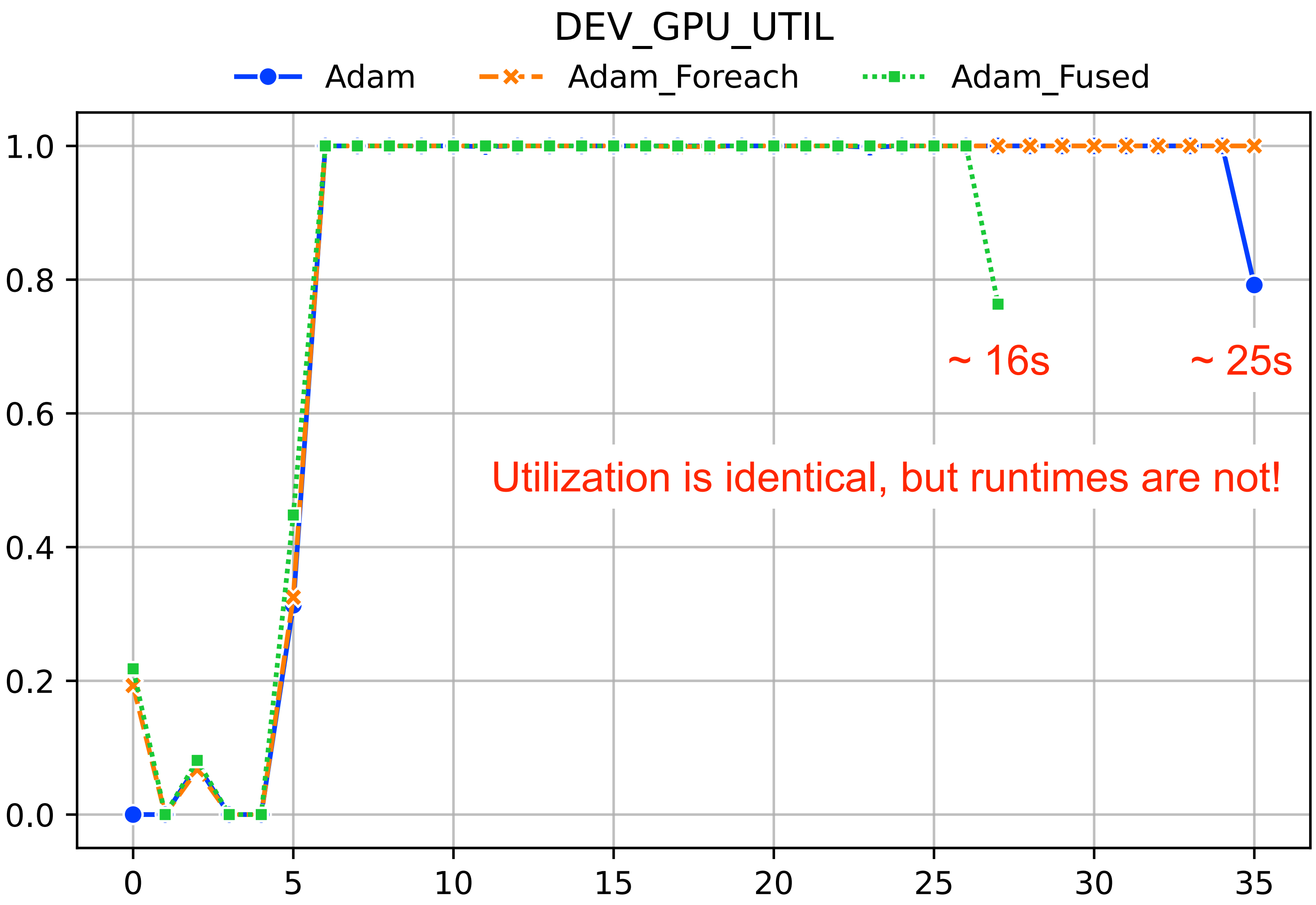
Dispatched to GPU

Option 2:

Dispatch full computation to GPU using a Kernel that implements the full optimisation step.

This is known as the “fused” implementation as everything is fused into a single kernel.

GPU Utilization

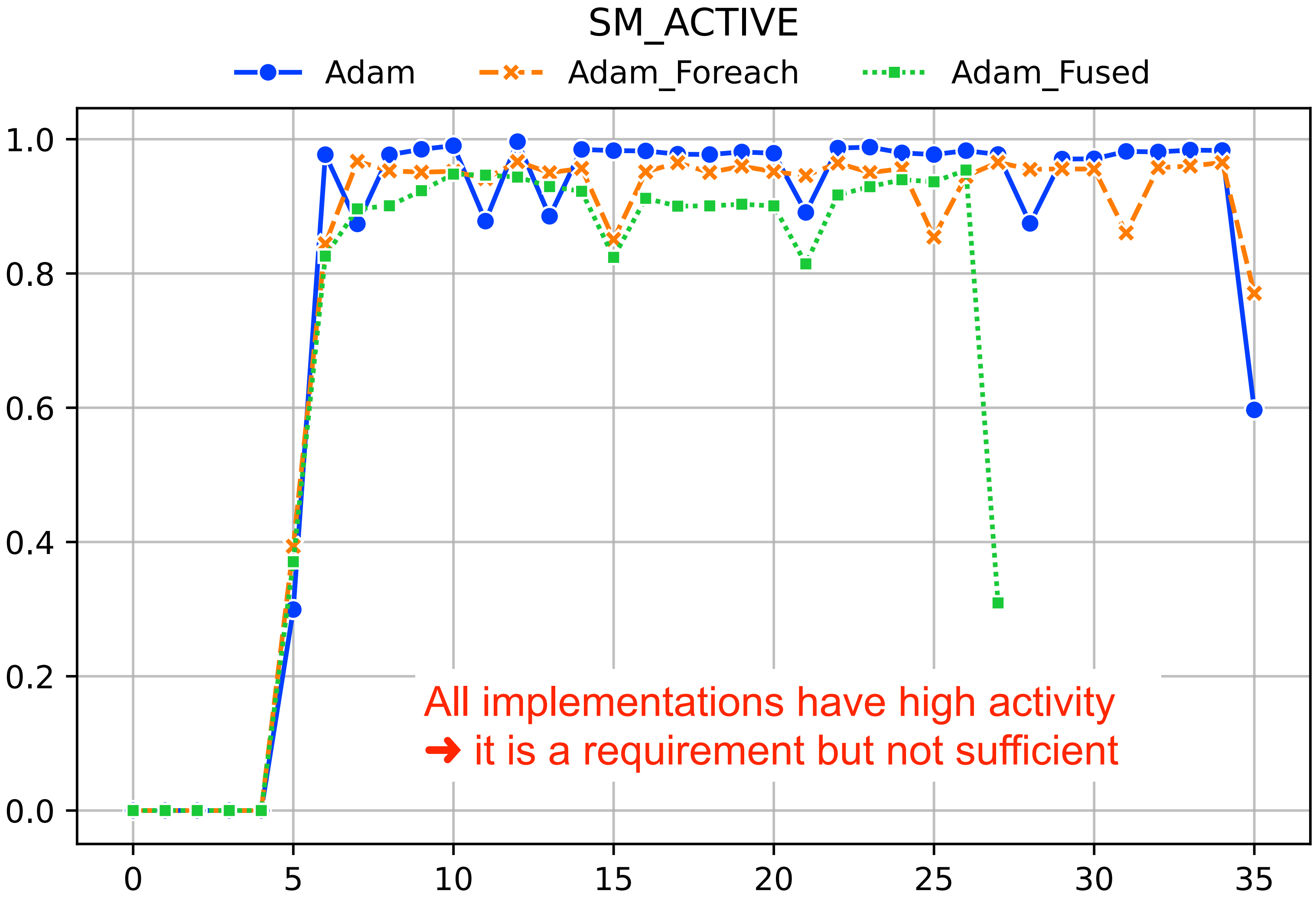


~ 16s

~ 25s

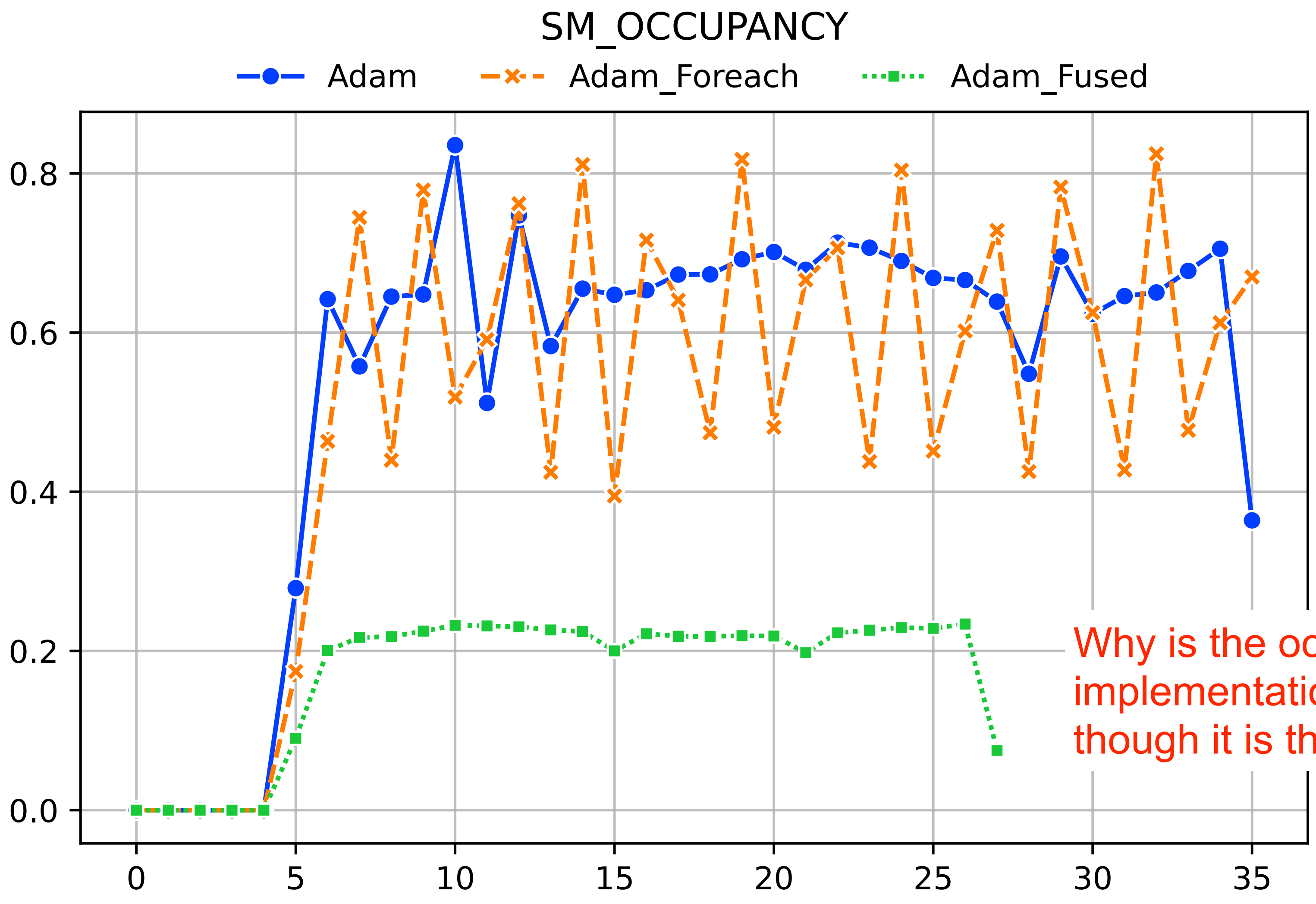
Utilization is identical, but runtimes are not!

SM Activity



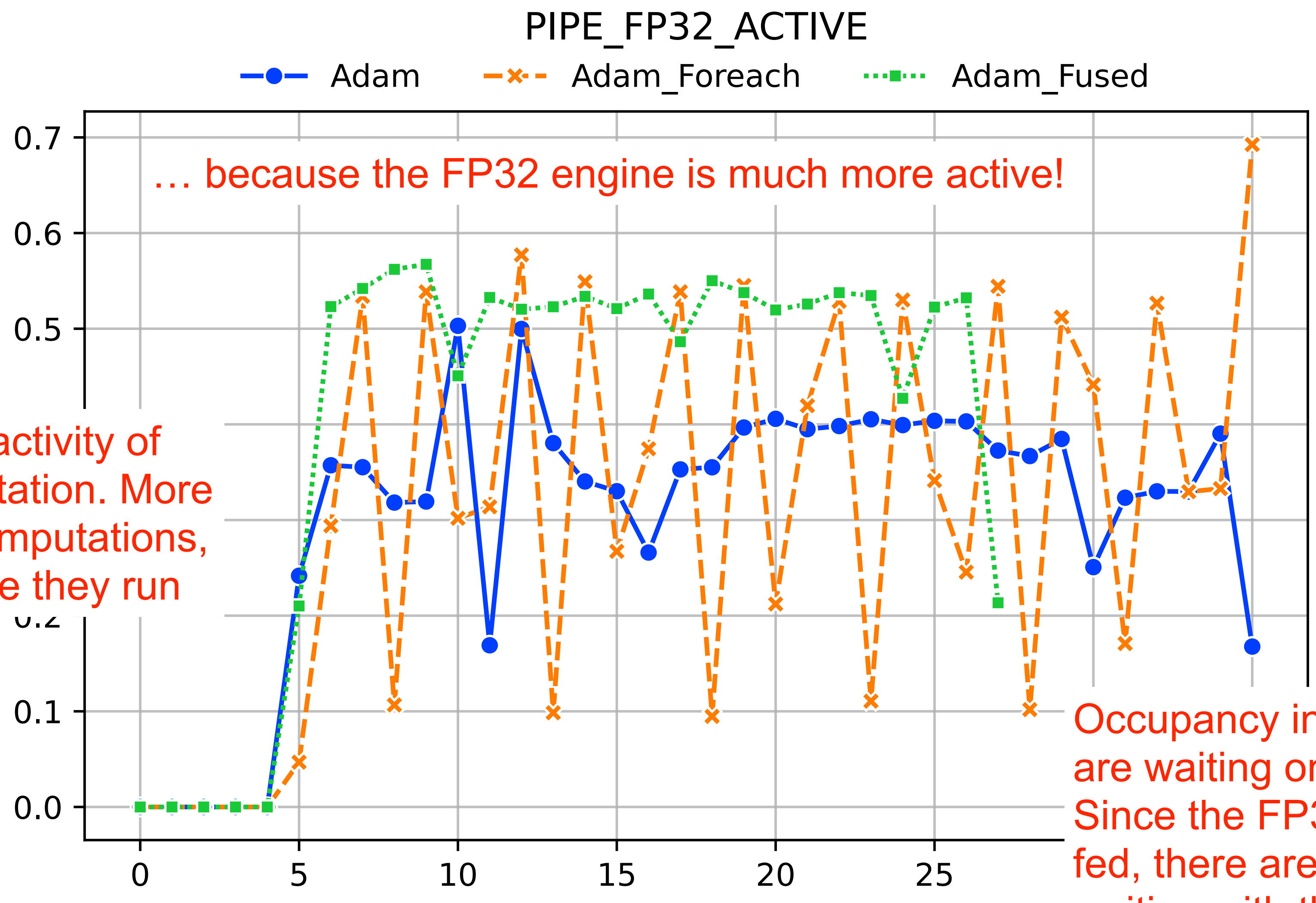
All implementations have high activity
➜ it is a requirement but not sufficient

SM Occupancy



Why is the occupancy of the fused implementation much lower even though it is the fastest one? …

FP32 Engine Activity

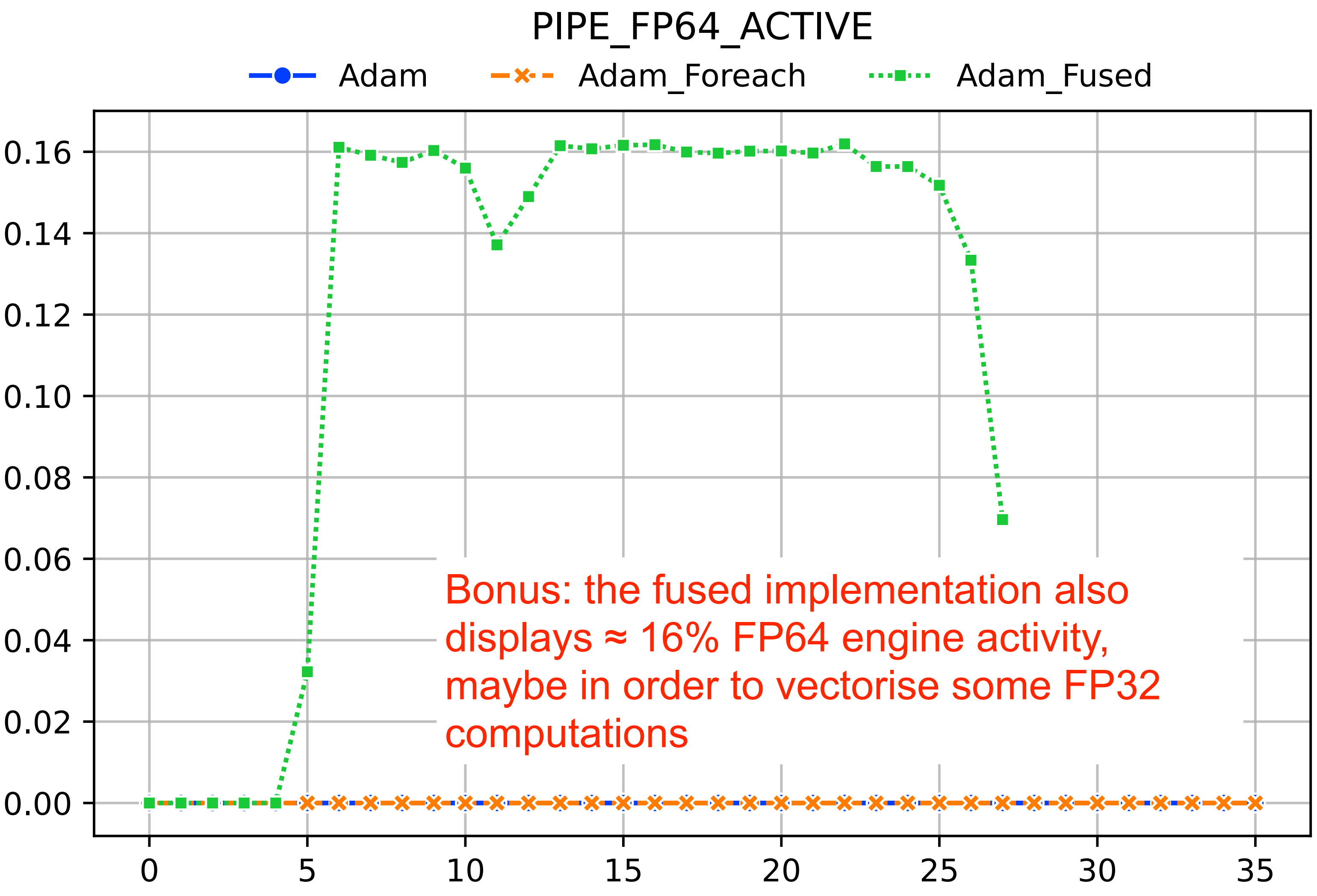


… because the FP32 engine is much more active!

Notice the oscillating activity of the foreach implementation. More overhead to set up computations, but higher activity once they run

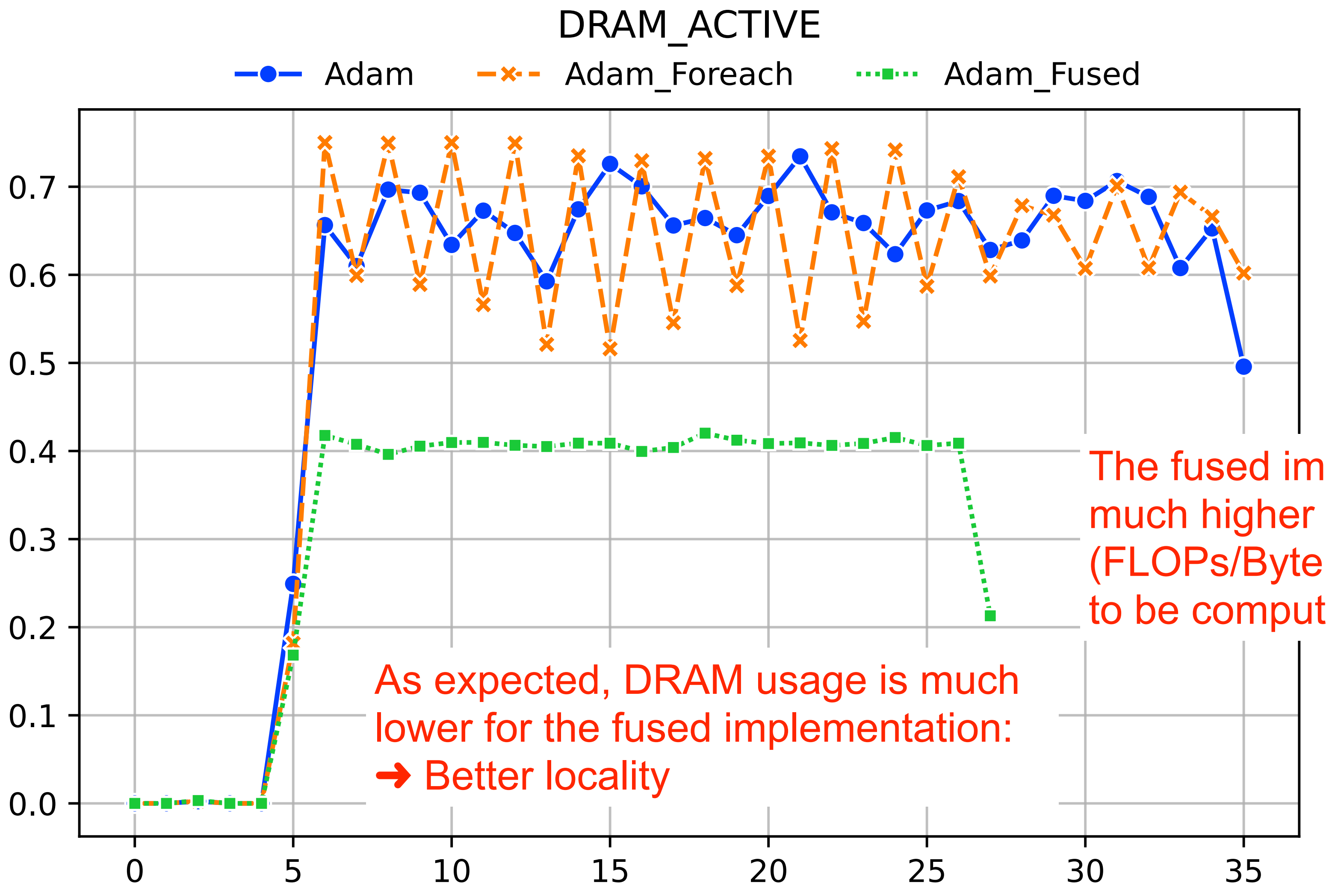
Occupancy includes warps that are waiting on resources. Since the FP32 pipeline is well fed, there are much less warps waiting with the fused implementation!

FP64 Engine Activity



Bonus: the fused implementation also displays ≈ 16% FP64 engine activity, maybe in order to vectorise some FP32 computations

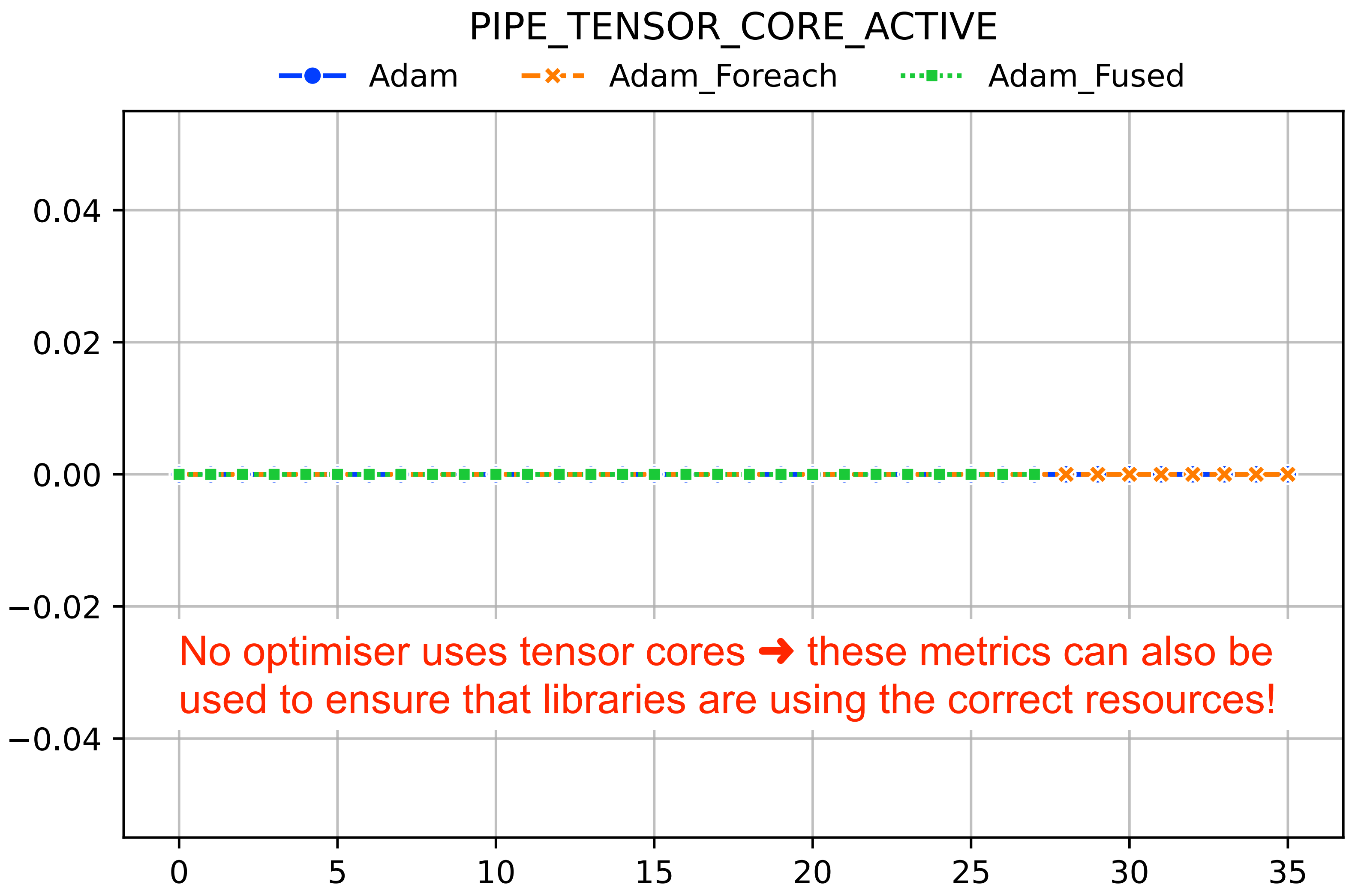
DRAM Activity



As expected, DRAM usage is much lower for the fused implementation:
➜ Better locality

The fused implementation has much higher operational intensity (FLOPs/Byte) and is more likely to be compute bound!

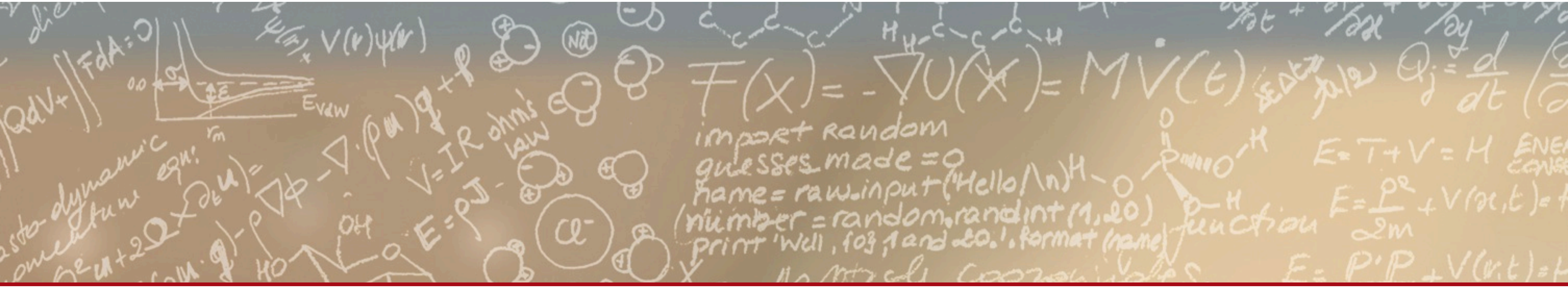
Tensor Core Activity



No optimiser uses tensor cores ➜ these metrics can also be used to ensure that libraries are using the correct resources!

Interpreting Metrics - How efficient is my workload?

- Initial idea ➜ use single “score” to describe efficiency of a workload
- What does efficiency mean?
  - Maximising FLOP/s?
    - Linear relationship between FP/Tensor activity and FLOP/s
  - Getting the most out of my GPU?
    - Need to consider if the application is compute or memory bound
    - For memory bound ➜ maximise occupancy
  - Increasing operational intensity?
    - Maximise FP/Tensor engine activity and decrease Occupancy/DRAM activity
- It is difficult to give a single universal value for efficiency!



Thank you for your attention.