Tools to profile JAX and jaxsnn

Internship report

Lasse Mauersberger

Supervisors: Elias Arnold and Eric Müller
Department of Physics and Astronomy
Heidelberg University

July 2025

Abstract

JAX was recently introduced as an accelerator-oriented Python library combining a NumPy-style API, just-in-time compiling, and various parallelization features with the XLA compiler. The package sacrifices some of the convenience provided by other machine learning libraries, but offers functionalities that can significantly improve both training and inference performance.
The JAX-based library `jaxsnn` [1] is designed for event-based training of neuromorphic hardware using the EventProp-algorithm [2], specifically targeting the BrainScaleS-2 system. The forward pass is executed on the hardware, and using spike times and current information, the EventProp-algorithm can compute exact updates for the neuron weights without relying on surrogate gradients.
This internship report aims to outline tools to benchmark JAX. Because of JAX's reliance on XLA and jitted functions, benchmarking performance is not always trivial. Python modules like `time.time()` or `.costanalysis()` fail to work as intended due to XLA compilation and lazy evaluation. We will use XProf, a TensorBoard plugin, to obtain results and gain access to the inner workings of XLA and JAX.

The functionalities are demonstrated using a simple batched matrix multiplication example. We find that a correct choice of batch dimension can improve performance by up to 30%, and that memory usage can be reduced by up to one third by identifying unnecessary steps in the XLA optimization and addressing them at the source code level. These results show, that although XLA and JAX are powerful tools and perform a lot of optimization out of the box, they still need manual optimization and profiling to achieve the best performance. Understanding XProf and the tools it offers will thus be crucial for effectively profiling and optimizing JAX applications such as `jaxsnn` and achieving peak performance in machine learning tasks.

Contents

1 Introduction

1.1 Jaxprs, XLA, HLO and lazy evaluation

Profiling in JAX can be problematic. The code one writes may differ extensively from the code executed by the device. This is because of two main concepts JAX implements:

- **Lazy evaluation**:
  JAX makes use of lazy evaluation. This means instead of so called "eager" evaluation which is the standard across most python packages like numpy, code is not executed on the device and values are not obtained in a "chronological" order, but only when the results are requested. If you have a line that for example computes $x = 1 + 1$ JAX will ignore this line until x is called.
- **XLA: High Level Optimization/HLO**:
  The way that JAX is able to use methods like lazy evaluation and other optimization techniques like just in time compiling is through the google developed and since turned open-source compiler XLA (Accelerated Linear Algebra). JAX first traces the computations and turns them into high-level-optimized functions, so called Jaxprs. These can be accessed and printed in python through `.make_jaxpr`. This step includes accounting for functions such as `vmap`, `jit` and `grad`. It then uses XLA to compile this already optimized code into HLO-instructions. At this point the code is highly optimized and depending on the circumstances may vary greatly from what we see in the actual python script. At last these HLO-instructions generate code optimized for different hardware (GPU, TPU etc.).

These optimization techniques are powerful and the combination of Jaxprs and XLA is the reason why JAX is able to significantly outperform other frameworks, even ones that also utilize XLA such as Tensorflow. However, these quirks can also lead to a host of issues and make traditional profiling hard (other Python built-in features like `.costanalysis()` are also often faulty).

1.2 XProf

JAX has a built-in profiler, the `jax.profiler`. There are several ways to call it, the easiest one being `jax.profiler.start_trace('<path-to-dir>')` followed by the code to be traced, followed by `jax.profiler.stop_trace`. This generates a .json file which can theoretically be interpreted by conventional trace viewers such as Perfetto. However, because of the intricacies of JAX/XLA these viewers will often have problems actually interpreting the file and calculations will simply not show up. This is where XProf comes in. It is a TensorBoard plugin and part of the OpenXLA project specifically for profiling programs compiled by XLA[3]. It has a suite of functions, the most important one being a trace viewer which explicitly gives access to the different levels of code (Source, HLO, Device, see screenshot below). It also gives high-level overviews of the different functions, information on FLOPS and FLOPS utilization, and memory usage.

2 Capabilities and Use of XProf

2.1 Installation and Use

XProf is a Tensorboard plugin, so to use it, install TensorBoard and XProf in e.g. a venv. Then files can be read as with any other trace. To avoid the hassle of setting it up on a remote machine, it can be run locally with the downloaded files without any problems. Once TensorBoard recognizes the file, one can access the desired functionalities and metrics through a dropdown menu. For a guide on installation visit the JAX documentation [4].

2.2 Trace Viewer

XProf's trace viewer gives access to activity both on the host and device. We will focus mostly on the capabilities for tracing on the device, as this is the most relevant. The CPU traces mostly become relevant if there is a lot of idle time on the device to understand why this is happening. The device trace is split into the following levels:

- **Compute Streams**: Actual activity on device e.g. the Kernel for matrix multiplications.
    - **Framework Name Scope**: Metadata on the Compute Stream, annotates functions, their frameworks etc.
- **Framework Ops**: Gives information on the operations and their frameworks, e.g. Matmul is annotated as dot_general.
- **XLA Modules**: This refers to the top function identified by XLA. If one e.g. were to define a function to perform matrix multiplication called matmul and then jit it, this would be jit_matmul.
- **XLA Ops**: This is possibly the most important level, as this shows the actual functions passed on to the hardware after XLA creates the HLO-instructions. It also gives access to HLO-graphs, which are representations of the computing stream and are important as these are not necessarily trivial. More on this in the next section on the graph viewer.
- **Source Code**: If possible, this will include references to the source code, i.e., the code written in the actual file. Especially useful for debugging purposes, as it can be hard to identify functions after they have been lowered to HLO by XLA.

The trace viewer is a starting point. It gives a rough overview of whether resources are utilized efficiently. If device activity is low, that is an indicator the code is inefficient. Through the drag and select tool one can also find function calls in certain areas and find the most expensive functions. It also gives timing info on the operations, which can help to find the actual time it takes the device to execute certain code and as the time captures are done by XLA itself on the device these timing estimates are much more accurate than e.g. `time.time()` as these modules usually use the computer's clock. Conventional methods would also require changes in the code such as introducing `.block_until_ready()` calls to interrupt asynchronous dispatch and lazy execution. However this in turn may possibly affect how XLA optimizes code.

2.3 Graph Viewer

As explained one can access HLO-Graphs through XProf. This is best done through the trace viewer by selecting the XLA Op in question. It shows the node of the operation itself and the nodes of the operations it depends on, as well as the functions its passed to. As XLA optimized Code can differ significantly from the code written in the python script, this is a very useful tool to understand what is actually happening on the device and debug code. It also specifies input shapes and data types, so if functions experience rejitting, one can use these graphs to figure out which datatypes or shapes are inconsistent.

2.4 HLO Op Profile

HLO Op Profile is a high-level overview of the different functions executed on the device, including device FLOP and memory bandwidth utilization. It gives a list of the most expensive functions in terms of time spent on the device, including time spent Idle. It also lists the number of function calls, time spent on computing flops, wasted time and time transferring data and the same info on their sub functions. When clicking on the functions you find a text representation of the HLO-graph, which further helps to understand the computation.

It thus gives a good overview of the performance and makes it quite easy to identify obvious bottlenecks in the code and debug it accordingly. This is aided by the fact that clicking on the subfunctions provides you with a text version of the HLO-graph. However, one has to be slightly careful about some of the metrics. As an example, the FLOPS utilization is implemented in such a way that it takes a presumed number of peak FLOPS and calculates the utilization based on that. So for an FP16 calculation, the FLOP utilization will be doubled compared to an FP32 calculation, even though the actual utilization of the device is the same; the throughput for FP32 is just obviously lower. Which reference is used is also obscure, for the case of an RTX3090 it seemed to have been FP16 with FP16 accumulate, which is odd, as using the 3090 in this mode is unusual, it usually uses FP32 accumulate, which halves peak FLOPS. So even for an FP16 matmul the utilization only reached about 52 percent.

2.5 Framework-/HLO-Op-Stats and Roofline Model

These are all sort of complimentary to the HLO Op Profile. Each of the tools provides a high level overview of the different functions executed on the device. Their valyue lies less in direct profiling and more in identifying bottlenecks and performance issues in complex code.

Framework Op Stats gives a high-level overview of the different framework operations and includes the time spent on the host.

- It lists the most expensive operations in terms of time spent on the device and can thus be used to identify bottlenecks that affect the device.
- It lists the time spent on the host, which is important for debugging, as it can help identify if rejitting is causing problems.
- It gives a rough overview of the FLOPS/s and total time, which can help estimate FLOP counts.

- It also gives a rough overview of the memory bandwidth usage, which can help identify if memory is a bottleneck.

HLO Op Stats is similar, but focuses on the HLO Ops and their subfunctions.

- It provides plots comparing the time spent by the different HLO Ops, which can help identify bottlenecks.
- It also provides a table with the most expensive or rather longest total self time HLO Ops, which can help identify performance issues.
- It also provides a bar chart to compare the GFLOPS/s of the most expensive HLO Ops, which can quickly make it apparent which parts of the code need optimization.

I also want to touch on the Roofline Model, which is again similar to the other two in helping to identify bottlenecks, although this is very focused on identifying if HBM is a bottleneck, as it compares the FLOPS/s to the FLOPS/byte and thus makes it easy to identify if the program overall and if the individual functions are memory bound or compute bound. However, personally I find the combination of Framework Op Stats and HLO Op Profile more useful, as it gives a more detailed overview of the individual functions and their performance. The Roofline Model summarizes both sections and visualizes them but provides slightly less info; it for example does not provide easy access to HLO-graphs. However, it is useful as a quick overview as it makes it instantly apparent which functions are bound by memory and which are bound by compute, by their position on the graph. If the point is in the range of the x-axis where the roofline model is on a slope, it is memory bound; if it is in the flat range of the x-axis, it is compute bound.

2.6 Memory Usage

XProf has two tools that give information on memory usage: the Memory Profile and the Memory Viewer.

- The Memory Profile gives an overview of the memory usage on the device, provides a time graph of memory usage broken down into free, stack, heap and also lists fragmentation. A table provides the largest memory allocations and the functions that allocated them.
- The Memory Viewer gives a more detailed breakdown of memory usage listed by XLA Op and specifies the memory allocated by the functions or, more specifically, their parameters and intermediary results.
  - Though total memory is not shown in this graph, it is possible to identify the peak memory usage and the functions that contribute. It takes a baseline when the HLO Module is started into account but not other modules that get executed in parallel after it has been compiled.
  - It also makes clear which parts of the function get allocated at which point, e.g., when the output is allocated and if these preallocations are contributing to OOM failures.

3 Example: Matmul

3.1 Code

In the following section we will take a look at a simple matmul that was batched through vmap and take a look at the different metrics and how they can be used to identify performance issues. The code is shown in the picture. We will trace the matrix multiplication for multiple different batch sizes. The code is roughly as follows:

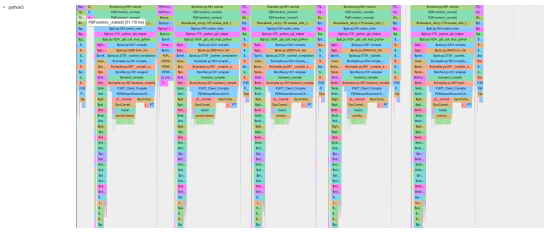
Algorithm 1 Profiling a simple batched matmul

1: Define function `matmul`$(A, B) \leftarrow$ `jnp.dot`(A, B)
2: Define `batched_jitted_matmul` $\leftarrow$ `jit(vmap(matmul, in_axes=(None, 0)))`
3: Define `batch_sizes` $\leftarrow [1, 10, 50, 300, 1000, 1000]$
4: Generate random matrix $A \in \mathbb{R}^{1000\times1000}$ with key `key_A`
5: Start JAX profiler trace to folder `'traces_vmap_example'`
6: Initialize empty list `times[]`
7: **for** each b in `batch_sizes` **do**
8: Generate random tensor $B \in \mathbb{R}^{b\times1000\times1000}$ with key `key_B`
9: Record start time: $t_{\text{start}} \leftarrow$ `time.time()`
10: Compute: `result` $\leftarrow$ `batched_jitted_matmul`(A, B)
11: Append elapsed time: `times.append(time.time()` $- t_{\text{start}}$`)`
12: **end for**
13: Stop JAX profiler trace

3.2 Profiling of the Example Matmul

3.2.1 Profiling and Improving the Computation

We will begin by looking at the trace viewer, to get a rough overview of the performance. We can see what is happening on the host, which is mostly the compilation of the functions. What quickly becomes apparent is that functions will be recompiled for different batch sizes. However, jitting for the first batch size was significantly longer than for subsequent batch sizes, starting at around 95 ms and then dropping to around 44 ms. This shows that XLA keeps the previous compilation in mind and is, at least in some cases, able to reuse some of the compiled code, even for different data shapes. Simultaneously, as we can see by the device traces, the execution of the compiled function is very fast.



(a) Trace on the host for the vmapped matmul

(b) Trace on the device for the vmapped matmul

Figure 1: Trace Viewer for the vmapped matmul

Moving on to the HLO Op Profile, we can see that the matmul is indeed the most expensive operation, which is expected. The reason it is listed multiple times is because for each batch size a different function is created and dispatched. However, FLOP utilization is considerably below the roughly 25 percent one would expect for such a simple matrix operation. We can see that the matmul itself is made up of different subfunctions and that the subfunctions of loop_fusion are considerably expensive, without utilizing the computation power of the device.

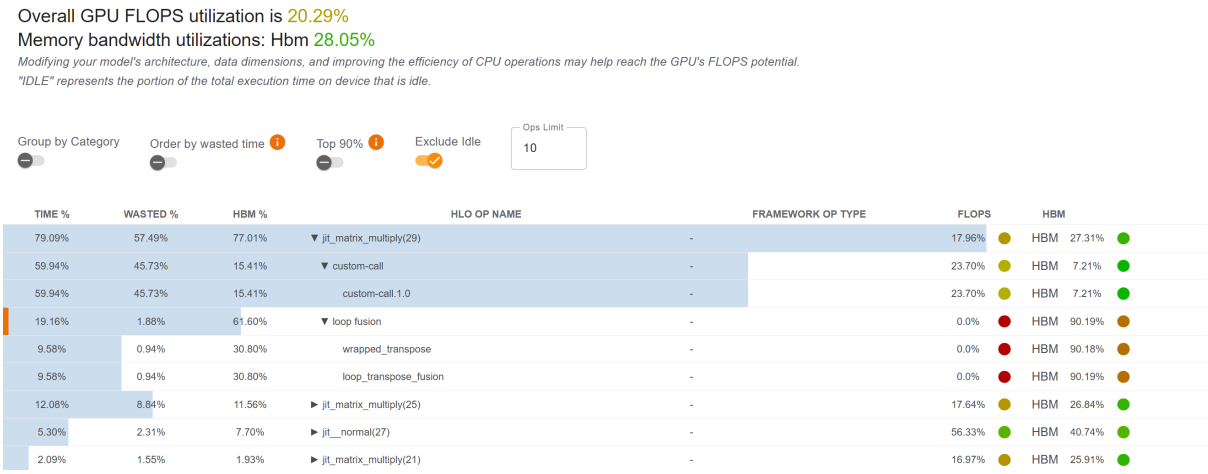


Figure 2: HLO Op Profile Page

We will now take a look at the Graphs of the vmapped matmul functions to understand especially what is happening in the two subfunctions of loop fusion. As the graph shows, the optimization of the vmapped matmul consists of stacking the matrices and then performing a single matmul on the stacked matrices and then outputting them as (batch, n, n). For the matmul the datatype of the formerly (1000, 1000, dtype=FP32) A matrix gets changed to a column vector of 2_{22} elements of signed 8 bit integers. This is probably faster to compute the device and will later be fused back into FP32 as there are approximately 4 times the number of elements. We can see that about 30 percent of the execution is spent on the transposition of the inputs. As this seemed odd and the process seemingly only consist of moving the first to the second dimension and later moving it back, I moved the batch dimension to axis 1. This led to the transposition being removed (see figure 4) and thus a speedup of about 30 percent of the computation, increasing total FLOPS utilization to about 28 percent. This seems to happen as the bitcast requires the tensor to be in a format, where it can simply be reinterpreted as a different shape, and $(\text{batch\_size}, n, n)$ was not such a format as a bitcast to $(\text{batch\_size} \times n, n)$ would lead to a dimension mismatch.

3.2.2 Memory Usage

We can also take a look at the memory usage of the vmapped matmul and will again compare the original and improved versions. As mentioned before, the Memory Profile gives a breakdown of memory usage over time. As expected, we observe an increase in memory usage with increasing batch size. However the memory has short but sharp peaks in the original graph, which do not appear in the improved code. Figuring out why this

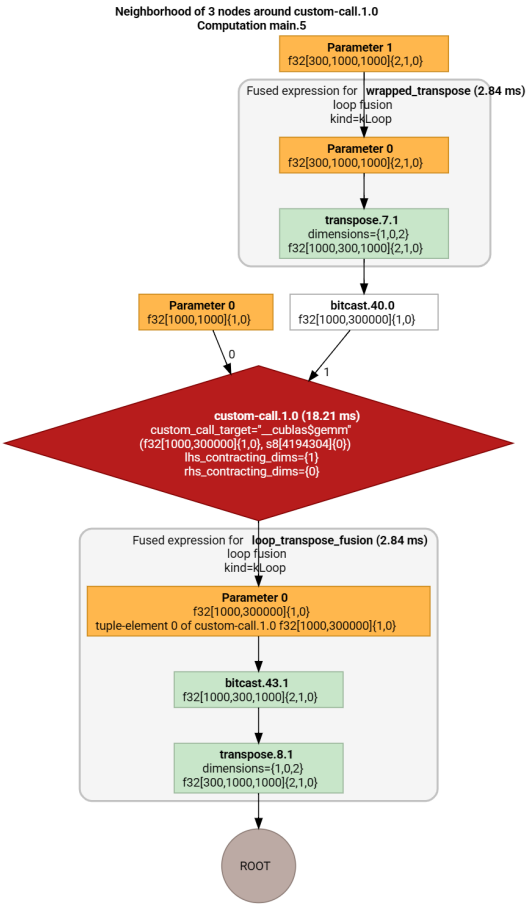


Figure 3: HLO Graph of the vmapped matmul function

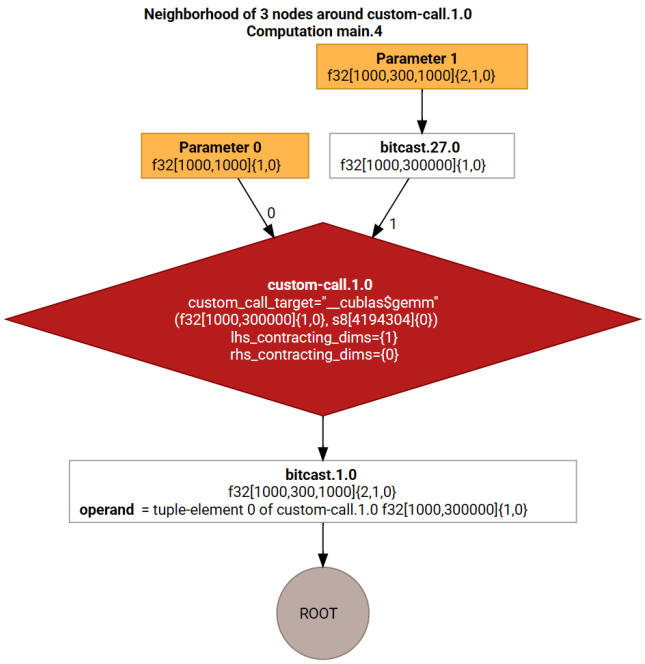


Figure 4: HLO Graph of the vmapped matmul function without transposition

is the case can be important to avoid OOM errors.

We will look at the memory viewer to better understand these peaks. To do this we select

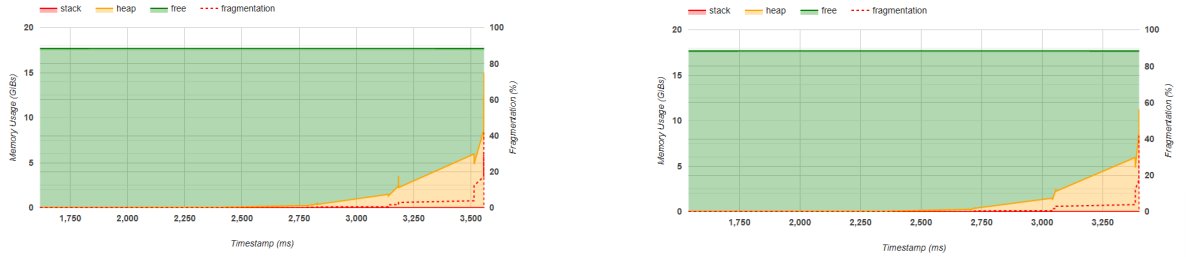


Figure 5: Comparison of the memory profiles. In this case fragmentation was defined as $<$ fragmentation $>= 1 - (<$ largest chunk of free memory $> / <$ total free memory $>)$.

the HLO module of one of the batched matmuls (for purposes of visibility batch size 10) and the program creates a memory graph. On the y-axis it shows the memory allocated in MiB and on the x-axis the program order. This means we no longer have a time axis; instead we can see when in the steps of the function memory is allocated. The baseline is the memory allocated at the beginning of the function; it then calculates additional memory. However, changes in memory allocation are not shown, so it can sometimes be hard to actually identify why OOM errors are happening if many different functions are compiled and executed in parallel. However, for our simple example this problem does not occur, as the matrix multiplications are executed sequentially. Looking at the memory graph, we can see that we no longer have a rise in memory in the middle of the program in the improved version, which is because the bitcast does not allocate additional memory, whereas the transposition did add an extra matrix. It simply reinterprets the memory already allocated. The original version, on the other hand, allocates additional memory for the transposition, which is why we see the peaks in the memory profile, as this is only an intermediary and is removed afterwards.

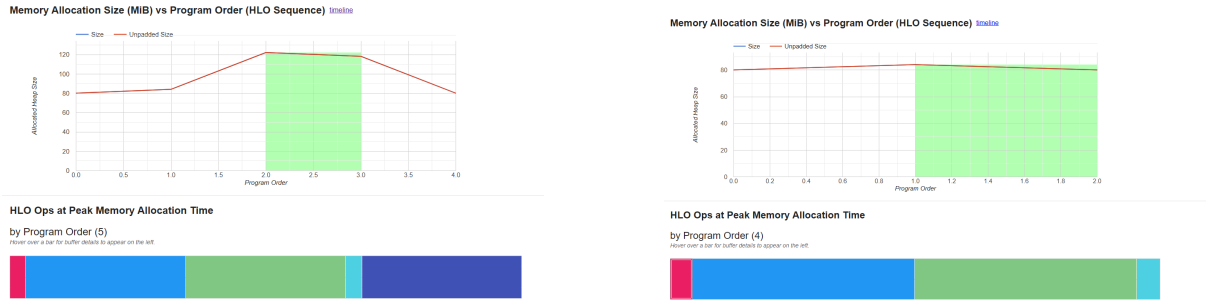


Figure 6: The sections in the bar below represent the memory allocated by different parts of the function. In both cases you have the A matrix and B matrix as parameters which are kept throughout the function (A red for both, B blue for both). The dark blue section on the left is the transposed B, calculated in the first step and no longer necessary for the improved version. The light blue section represents the intermediary representation of the A matrix in signed 8-bit integers.

4 Discussion

XProf is a powerful tool to profile JAX code and can be used to identify performance issues and bottlenecks. Through figuring out how the code is actually executed on the device, the performance of the code was able to be improved by about 30 percent, both in terms of FLOPS utilization and memory usage. Although the example was derived from a simple batched matmul, it is important, for instance in the context of MLPs, and the principles of addressing data types to optimize memory and accesses can be useful in many different circumstances.

When it comes to jitting, we can conclude that it is important to keep data types consistent, as the jitting process is fairly expensive, often taking orders of magnitude longer compared to the actual computation. However, for some functions, using different data types can be beneficial, as the rejitting process might be cheaper and it might lead to better performance by using smaller arrays.

5 Outlook

We will use the knowledge and tools gained from this report to profile `jaxsnn` and identify performance issues, especially in the EventProp-algorithm [2]. The goal is to improve the performance of the library and make it faster and more efficient for training of BrainScaleS-2. For this purpose we will design a benchmark experiment that produces a controllable amount of spikes in each layer. Following this we will use XProf to identify performance issues and bottlenecks in the code.

Additionally, it is of great interest to see how the performance of the EventProp algorithm scales with the number of events, which is why the controllable firing rate is crucial. In theory, EventProp is very useful because it takes advantage of the sparsity of spiking neural networks and only updates the weights at actual spike times, whereas an approach that works with surrogate gradients has to make calculations at each time step. However, as event incidents increase, EventProp will have to update the weights more often, diminishing the performance advantage gained by utilizing sparsity. By profiling the code, we will be able to identify the point at which the calculation changes in favor of a time-continuous/surrogate-gradient-based approach, and why.

References

[1] E. Müller, M. Althaus, E. Arnold, P. Spilger, C. Pehle, and J. Schemmel, *jaxsnn: Event-driven Gradient Estimation for Analog Neuromorphic Hardware*, arXiv preprint arXiv:2401.16841 [cs.NE], submitted 30 Jan. 2024, `https://arxiv.org/abs/2401.16841`

[2] C. Pehle, L. Blessing, E. Arnold, E. Müller, and J. Schemmel, *Event-based Backpropagation for Analog Neuromorphic Hardware*, arXiv preprint arXiv:2302.07141, submitted 13 Feb. 2023, `https://arxiv.org/abs/2302.07141`

[3] OpenXLA Project, *XProf: Accelerator Performance Analysis*, `https://openxla.org/xprof`, Accessed: July 30, 2025.

[4] JAX Developers, *Profiling JAX Programs*, JAX Documentation, `https://docs.jax.dev/en/latest/profiling.html`, Accessed: July 30, 2025.