GRAPHCORE

PopVision User Guide

Version latest

Graphcore Ltd

Aug 27, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

PopVision™ is a suite of graphical application-analysis tools. You can download PopVision from the Graphcore support portal.

The PopVision™ *Graph Analyser* helps you get a deep understanding of how your applications are performing and utilising the IPU resources. It shows data about the graph program, memory use, and the time spent executing code and communicating.

The *System Analyser* provides information about the behaviour of the host-side application code. It shows an interactive graphical view of the timeline of execution steps, helping you to identify any bottlenecks between the CPUs and IPUs.

The System Analyser is supported by the PopVision trace instrumentation library (libpvti). This enables you to control tracing of system events from within your Python or C++ application code.

The PopVision analysis library (libpva) can be used for programmatic analysis of Poplar profiling information.

The libpvti and libpva libraries are part of the Poplar SDK. See the Getting Started guide for your IPU system for information on installing and using the SDK.

CHAPTER

TWO

GRAPH ANALYSER

Version: 2.5.6

2.1 Overview

The PopVision™ Graph Analyser application is used to analyse the programs built for and executed on Graphcore's IPU systems. It can be used for analysing and optimising the memory use and performance of programs.

The PopVision Graph Analyser provides reports on the following:

	*Summary Report* of the IPU hardware, graph parameters, host configuration.
	*Memory Report*, which gives a detailed analysis of memory usage across all the tiles in your IPU system, showing graphs of total memory and liveness data, and details of variable types, placement and size.
	*Liveness Report*, which gives a detailed breakdown of the state of the variables at each step in your program.
	*Program Tree*, which shows a hierarchical view of the steps in the program.
	*Operations Summary*, which shows a summary of all the operations for a software layer in your model, displaying statistics about code size, execution cycles, debug data and FLOPs measurements.
	*Operations graph*, which displays the High Level Operations (HLO) graph for TensorFlow programs, allowing you to drill down through modules and see details of HLOs.
	*Execution Trace*, which shows how many cycles each step of your instrumented program consumes.

Each of these reports is described in further detail in the sections below.

You can search this documentation by entering text into the Search box at the top of the Table of Contents, on the left. To cycle through any search matches, press the Return key repeatedly.

2.1.1 About the IPU

An in-depth description of the IPU hardware is available on the online IPU Programmer's Guide. While we describe some of the relevant features of the IPU in this document, you should refer to the Poplar documents for a more in-depth understanding.

2.2 Capturing IPU reports

This section describes how to generate the files that the PopVision Graph Analyser can analyse. The PopVision Graph Analyser uses report files generated during compilation and execution by the Poplar SDK.

When you first open the application, there is a link on the opening page to a Getting Started with PopVision video.

The sections below describe the files supported by the PopVision Graph Analyser. These files can be created using the `POPLAR_ENGINE_OPTIONS` environment variable or the Poplar API. At a minimum you need either the `archive.a` or the `profile.pop` (or the legacy `graph.json/cbor` file) for the PopVision Graph Analyser to present reports.

With the release of Poplar SDK 1.2 a new entry in `POPLAR_ENGINE_OPTIONS` was added to make capturing reports easier. In order to capture the reports needed for the PopVision Graph Analyser you only need to set `POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true"}'` before you run a program. By default this will enable instrumentation and capture all the required reports to the current working directory. For more information, please read the description of the Poplar Engine options in the Poplar and PopLibs API Reference.

By default, report files are output to the current working directory. You can specify a different output directory by using, for example:

```
POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.directory":"./tommyFlowers"}'
```

If you have an application that has multiple Poplar programs (for example, if you build and run a training and validation model), then a subdirectory of the Engine name will be created in :ref:`autoReport.directory` in which the profile information will be written. This allows users of the Poplar API to make sure reports are written into different locations. (If no name is provided then the profile information will continue to be written in `autoReport.directory`). Further information can be found in the `Using TensorFlow <reports_from_tensor_flow>`, *Using PopART* and *Using PyTorch* sections.

2.2.1 Limiting report output

The `POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true"}'` outputs all report files. If you wish to capture only one file, or all but one file, you can use a combination of the following options:

- To capture just one report file, set the specific `autoReport` output, that is: `POPLAR_ENGINE_OPTIONS='{"autoReport.outputArchive":"true"}'`.
- To capture all but one report file, set the `all` option and disable the report you do not want, for example: `POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.outputArchive":"false"}'`

Execution reports can be very large if you are running many iterations of your programs. By default, only the first two runs of each of your Poplar programs are captured. This can be increased or decreased by setting the `executionProfileProgramRunCount` option as follows:

```
POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.executionProfileProgramRunCount":"10"}'
```

When running a replicated model, each replica executes the same application, so in order to reduce the size of the file you can select which replica you want to profile with `replicaToProfile` option as follows:

```
POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "profiler.replicaToProfile":"2"}'
```

Other options to consider to reduce the size of your profile are:

- Reduce the number of iterations per Poplar Engine::run
- Reduce the batches per step
- Reduce the epoch's profiled.
- Reduce the batch size

2.2.2 Reloading reports

The folder (or folders, if you're comparing reports) that contain the individual report files are monitored by the application in case any of the files changes, for example, if you've re-run your Poplar program and re-generated new version.

If the application detects that any of the files have changed, a dialog box appears telling you what files have changed, and prompting you to reload the report files.

Make sure that your Poplar program has finished executing (in particular that the `profile.pop`, or `execution.json/cbor` file has been completely written to disk before clicking on the Reload button), otherwise you may see inconsistent information displayed in the application.

2.2.3 Poplar report files

The PopVision Graph Analyser only supports fixed names for each of the files. If you save them with different names they will not be opened. When you are browsing directories to open, the PopVision Graph Analyser will highlight which of the following files are present in that directory.

Application and file versions

The `profile.pop` file, described below, is the new default format that combines the information in the graph profile file and the execution trace file into a smaller, faster-to-access SQLite 3 file. The `graph.json/cbor` and `execution.json/cbor` files should only be present in report files generated by older versions of the Graph Analyser that wrote files using the previous "V1" version format.

Binary archive (`archive.a`)

This is an archive of ELF executable files, one for each tile. With this you can see the total memory usage for each tile on the *Memory Report*.

Poplar Engine Options	`POPLAR_ENGINE_OPTIONS='{"autoReport.outputArchive":"true"}'`
Using Poplar API	Set the Poplar Engine option "autoReport.outputArchive" to true

Poplar Profile (`profile.pop`)

This file contains compile-time and execution information about the Poplar graph. This file is used to show *memory*, *liveness* and *program tree* views and also the *execution trace* view.

Poplar Engine Options	`POPLAR_ENGINE_OPTIONS='{"autoReport.outputGraphProfile":"true"}'` and/or `POPLAR_ENGINE_OPTIONS='{"autoReport.outputExecutionProfile":"true"}'`
Using Poplar API	Set the Poplar Engine options "autoReport.outputGraphProfile" to true and/or "autoReport.outputExecutionProfile" to true

Lowered Vars Information

Poplar can generate lowered vars information, which contains details about the allocation of variables on each tile, and is used to generate the variable layout in the *Memory Report*. IPU memory is statically allocated and this file contains the size, location, name and other details about every variable on every tile.

This information is not generated by default, as it can be quite large, and not useful to some users. However, there are engine options to collect the data and save it either into the `profile.pop` file, or as a stand- alone file.

Poplar Engine Options	When you use `POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true"}'` the lowered vars information will be captured in the `profile.pop` file. You can switch that functionality on separately with: `POPLAR_ENGINE_OPTIONS='{"autoReport.outputLoweredVars":"true"}'`
Using Poplar API	To capture the lowered vars data in a separate file (and not write it into the `profile.pop` file), use, for example,: `POPLAR_ENGINE_OPTIONS='{"debug.loweredVarDumpFile":"vars.capnp"}'`

Serialized Computation graph (`serialized_graph.capnp`)

This file contains a copy of the Poplar graph before compilation, including details on all of the compute sets, vertices, variables and edges (connections from vertices to variables). `autoReport.all` does not output this file by default, as it can be quite large for complex models, but you can enable it using the options below.

Poplar Engine Options	`POPLAR_ENGINE_OPTIONS='{"autoReport.outputSerializedGraph":"true"}'`
Using Poplar API	To save the serialised graph you need to use the Poplar API `Graph::serialize`.

Frameworks Information (`framework.json` & `app.json`)

You can use Poplar to create two more 'custom' files into which you can put your own data from frameworks or your application. See the *Framework and Application JSON files* section for more details

Debug Information (`debug.cbor`)

This file contains additional debug information collected from the Poplar software. From this information you can understand the source of variables, Poplar programs and compute sets. The debug information is viewable in the Liveness report and the Program Tree.

Poplar Engine Options	Automatically, when using `POPLAR_ENGINE_OPTIONS='{"autoReport.enable":"true"}'` and manually using `{"autoReport.outputDebugInfo":"true"}`
Using Poplar API	Automatically created

Collecting the enhanced debug information will not increase the memory footprint of your IPU application. The enhanced debug information is generated and streamed as the model is compiled.

See the two 'Debug information' sections in the *Liveness Report* and the *Program Tree* for details of what's included in the debug information, and where to find it in the Graph Analyser reports.

Graph Profile (`graph.json` or `graph.cbor`)

This now deprecated file type contains compile-time information about the Poplar graph in the old "V1" format. While the Graph Analyser can still read this file type, from old reports, it can no longer be generated. This file was used to show *memory*, *liveness* and *program tree* views and also compute set information on the *execution trace* view.

Execution Profile (`execution.json` or `execution.cbor`)

This now deprecated file type contains instrumentation data from the old "V1" format. While the Graph Analyser can still read this file type, from old reports, it can no longer be generated.

2.2.4 Using TensorFlow

If you use TensorFlow, the separate reports for each Poplar program compiled and executed will be placed in a subdirectory of `autoReport.directory` that contains the ISO date/time and process ID in its name.

The `debug.cbor` will be placed in the `autoReport.directory` and symbolic links are created in the subdirectories

The cluster name can now be found in details loaded from the `framework.json`.

For more details please see the guide Targeting the IPU from TensorFlow.

2.2.5 Using PopART

For PopART, the name of the Engine is by default set to inference or training depending on if you are using the InferenceSession or TrainingSession. You also have the option of providing your own Engine name when creating the session.

```
training_session = popart.TrainingSession(fnModel=builder.getModelProto(),
  ...
  deviceInfo=device,
  name="tommyFlowers")
```

The profile.pop will be written out to:

```
autoReport.directory/tommyFlowers
```

If your application has two inference sessions, by default the second will overwrite the first.

For more details please see the PopART User Guide.

2.2.6 Using PyTorch

For PyTorch which builds on top of PopART by default it will also name the engine inference or training depending on if you are using an InferenceModel or TrainingModel. You also have the option to name the Engine yourself using the SessionOptions:

```
opts = poptorch.Options()
  opts.modelName("tommyflowers")
  opts.enableProfiling(dirname)

  poptorch_model = poptorch.inferenceModel(model, opts)
```

The profile.pop will be written out to:

```
autoReport.directory/tommyFlowers
```

For more details please see the PyTorch User Guide.

2.3 Opening reports

In order to view reports, the PopVision Graph Analyser requires one or more of the files listed in the *Capturing Reports* section, above.

You can open report files on your local machine, or from a remote server over SSH.

2.3.1 Comparing reports

You can open two similar reports at once to compare them by clicking on the 'Compare reports…' link on the application's main page. This presents you with two file selection dialogs that work in exactly the same way as for opening a single report, as described below. The information from both reports is then combined on the report pages, allowing you to compare them.

- When opening a pair of reports to compare, you can click on the 'magnet' icon at the right-hand end of the directory textbox in either file selector. This copies the directory from the file selector on the other side.

2.3.2 Local reports

You can open report files stored on your local machine as described below.

Opening local reports

To open a local report on your machine:

1. On the Home screen of the PopVision Graph Analyser application, click on the 'Open a report…' button in the 'Open' panel. You'll be presented with a file selection dialog, and the 'local' tab at the top will be selected by default. You'll see listings of the directories and files on your local machine.
2. You can sort these files by name or modified date, in ascending or descending order, by clicking on the appropriate column header. Your sorting preference is saved.
3. Use this dialog box to navigate to the folder in which your report files have been saved. You'll notice that when the PopVision Graph Analyser identifies a directory in which any of the report files listed above are found, those files are listed on the right-hand side. Note that if the minimal file requirements are not present in a directory (see the table above), the 'Open' button will be disabled.
4. Once you've selected the directory with the necessary report files within it, click on the 'Open' button to load the report data from the files.

If you've used the application before, the Home screen also displays a list of recently opened report directories in the 'Recent' box. Click on one to open it again.

The *Summary Report* is displayed first, and the progress bar along the top of the screen shows the files being pre-processed by the application prior to being loaded and displayed.

Notice that the bottom of the Summary Report shows the *relevant files* that have been found, and their loading state. More details on these files, and which reports need which of them, can be found *here*.

2.3.3 Remote reports

If you are using an IPU system on a remote server, for example on a cloud service, any reports generated will be saved to that server, so you cannot open them 'locally'. You can, however, open them remotely by specifying the server address, and connecting to the machine over SSH. The report contents are them streamed back to the PopVision Graph Analyser application on your local machine, allowing you to view the reports.

When the PopVision Graph Analyser opens report files on a remote machine, it downloads a small binary app to it which pre-processes the report data and sends it back over SSH to the PopVision Graph Analyser application running on your local machine. If you're running other performance-critical processes on that remote machine, you should be aware of any effects this process may have on the capacity of the remote machine's hardware to run any other tasks. As server performance varies a great deal, the only way to know how much processor speed it takes is to try a small sample, and monitor the CPU usage.

Opening a remote report

To open a remote report on another machine:

1. On the Home screen of the PopVision Graph Analyser application, click on the 'Open a report…' link in the 'Open' panel. You'll be presented with a file selection dialog, and the 'local' tab at the top will be selected by default.
2. Click on the 'remote' tab at the top, and you'll see a login dialog that allows you to connect to a remote server. Enter your username, and the address of the remote machine.
3. If you just want to log in with a password for the remote machine, enter it in the Password field.
4. Alternatively, you can use your local machine's SSH key to authorise your connection. Enter its file path in the *Preferences dialog*.
5. Once you're logged in, you'll see a file dialog listing the directories and files on the server. You can sort these files by name or modified date, in ascending or descending order, by clicking on the appropriate column header. Your sorting preference is saved.
6. Navigate to the folder in which your Poplar report files have been saved. You'll notice that when you select a directory in which Poplar report files are found, the file window lists those files on the right-hand side. Note that if the :ref:`archive.a` file or the `archive_info.json` file are not present, the 'Open' button will be disabled, as one of these these files is the minimal requirement for generating a report in the PopVision Graph Analyser. See the `Report files <report_files>` section above for details of how to generate each file, and what it contains.
7. Once you've entered the directory with the necessary report files in it, click on the 'Open' button to load the report.

The SSH connection is constantly checked, and if, for any reason, it goes down, a warning dialog is displayed, letting you know.

The *Summary Report* is displayed first, and the progress bar along the top of the screen shows the files being loaded into the application, and the report data being analysed and prepared for display.

Notice that the bottom of the Summary Report shows the *relevant files* that have been found, and their loading state. More details on these files, and which reports need which of them, can be found *here*.

The Graph Analyser does not currently support encrypted SSH private keys, i.e. keys that are protected by a passphrase. However it does support SSH agents. If your key is passphrase protected you will need to make sure to add it to your SSH agent before the PopVision Graph Analyser can use it, by using the `ssh-add` command-line tool and ensure 'SSH Agent mode' is set correctly in the Preferences.

To configure ssh agent, from a terminal you can run the following.

```
# Start the ssh-agent in the background.
eval "$(ssh-agent -s)"
```

(continues on next page)

(continued from previous page)

```
# Add your SSH private key to the ssh-agent
ssh-add -K ~/.ssh/id_rsa
```

Then restart the Graph Analyser, click Preferences and remove the path pointing to your SSH private key path. Make sure that SSH agent mode is set to "Automatically obtain ssh-agent socket path from environment".

2.4 Viewing reports

The PopVision Graph Analyser displays interactive graphical and textual reports, and you can interact with these in a number of ways to get to the information you want. Each report has a few different options that are only relevant to that report, but they all share some features in common, as described below.

When you open a report, its file path is displayed in the title bar of the report window.

2.4.1 Using the side menu

When you're loading some report data, and the Summary Report is displayed, the side menu becomes visible on the left-hand side of the application window. This contains buttons at the top for viewing each of the main report types, and three buttons at the bottom:

	**Reload report** - if you need to reload a report, particularly if any of the files from which it was generated have been updated from a recent execution of your program, click this button to re-import all the report files. See *Reloading reports* for more information.
	**Close report** - once a report is loaded into the PopVision Graph Analyser, you can close it by clicking this option. This 'unloads' all the report data from the application and returns you to the opening page. If you want to view those reports again, you'll need to re-load the data.
	**Documentation** - this opens the documentation window (which you're now reading). If you were viewing one of the report pages, the Documentation window opens up on the relevant page.
	**Minimise/Expand** - this controls whether the side menu is expanded so that the menu option text is visible, or minimised, so that only the icons are visible.

2.4.2 Adjusting report size

There are several ways to change the size and scale of the report in the PopVision Graph Analyser window:

- You can increase and decrease the display size of the entire application (including the Help window) by using the Ctrl/Command keys with the + and — keys to magnify and shrink the display size, just as you would in a web browser. There are three Zoom options in the View menu that show the keys, and another option to reset the magnification level back to its default setting.
- To zoom in and out of a particular section of the graph, click and drag horizontally in the graph preview area above the main graph, and the display will change to show the graph that corresponds to that section of the data. A pair of *limiter* icons appear in the preview area to show the start and end of the data displayed in the main graph area, and these can be dragged left and right as well to change the amount of data in the main graph. Using the scroll-wheel on your mouse will also zoom in and out of the graph.
- You can also click and drag the main graph itself to view areas to the left and right of the currently viewed area. Note that clicking without dragging can sometimes select a specific tile (for example, in the Memory Report), but you can clear this selection from the input box above the graph.
- You can reset the zoom scale of the Memory and Liveness reports by clicking on the small button to the left of the preview area, top-left of the graph. This zooms out to the furthest level, showing the entire graph.

- To make a report larger, so that you can see more detail, you can drag the edges of the window to increase its size. This resizes the report images as you drag.
- To adjust the space that each half of a report takes up on the page, click the 'splitter' icon between the two halves of the report, and drag it up and down. The two report sections resize accordingly. Note that the Program Tree and Execution Trace reports also display a 'vertical splitter' when comparing two reports, so you can choose how much of each report fills the available screen space.

Navigating report graphs

If the report page contains a graph, and you have selected a datapoint on it by clicking it, you can use the arrow keys to move the selection to the previous or next datapoint. If you hold down the Shift key while using the arrow keys, the datapoint selection will move by ten datapoints at a time.

Similar to most web pages, you can use the Tab key to cycle through interactive elements on the report page, and this includes the graph itself, which displays the standard blue selection border when it has focus.

Each of the graph plots has an 'Options' button in the top left-hand corner which allows you to hide the graph keys.

Each of the graph plots also has a 'Reload' button in the top left-hand corner that allows you to reload the graph data manually.

Keyboard graph navigation

As well as using the mouse to navigate, you can use keyboard commands to pan and zoom around the report. This is particularly useful if you have no mouse present on your machine. The keyboard chortcuts are:

Pan Left	A
Pan Right	D
Zoom in	W
Zoom out	S

If you hold down the Shift key you will pan and zoom in further each step.

Graph data keys

Below a graph plot are one or more keys with a coloured block and the name of the dataset that's being plotted in that colour on the plot. You can click on a key to hide that dataset from the plot (and it appears with strike-through styling.

If you're *comparing two reports*, there will be one of each data key for the source and the target report. To see at a glance which is which, you can hover your mouse over a key, and the full path to that report is displayed.

2.4.3 Saving report images to disk

You can save report graphs to disk as image files or copy them to the clipboard, to avoid you having to make screen captures.

1. Click on the camera icon in the top right-hand corner of a report.
2. Select whether to save to a file or copy to the clipboard.
3. If saving to a file, select the directory on your computer where you want to save it.
4. If the image is saved (or copied) successfully, a confirmation appears.

Report images are saved as PNG files, with transparent backgrounds. Please be aware that some image-viewing applications on your machine may display the traditional 'checked background' to report images saved in this way, which identifies it as transparent. When imported into other applications (for example, presentation software) they will look best on a light background (for images saved while in *'light' mode*) or a black background (for images saved while in *'dark' mode*.

2.5 Viewing a Summary Report 

When you first open a report, the Summary view is shown. It consists of high-level information about the Poplar program, split into various sections, as described below.

Each summary section displays information in collapsible blocks (marked with a downward-pointing arrow, or, for the Engine Options section, clickable disclosure triangles), making it easy to show only sections of interest to you. Whether a section is collapsed or not is saved automatically across all reports.

2.5.1 Program Information

The top half of the report shows details of the IPU system the program was compiled for, and also details of the size of the graph that Poplar created.

Target

- **Type**: what kind of IPU the program was compiled for. This will either be IPU or IPUModel
- **Architecture**: what version of IPU was used to run the program. This will either be Mk1 or Mk2
- **Timestamp**: when the program compilation was started.
- **Tiles per IPU**: how many tiles are in each IPU
- **IPUs per Replica**: how many IPU are in each replica. (Only shown if more that one replica is used)
- **Replicas**: the number of replicas. (Only shown if more that one replica is used)
- **Total Tiles**: how many tiles in total (tiles per IPU * num IPUs)
- **Total IPU**: how many IPUs in total (IPUs per replica * num replicas)
- **Memory per Tile**: maximum memory on a tile.
- **Memory per IPU**: maximum memory on a IPU.
- **Memory per Replic**: maximum memory for a Replica.
- **Total Memory**: total memory on all IPUs the program was compiled for.
- **Compute Set Instrumentation**: the type of instrumentation compiled into the program to record compute set execution cycles. The method is controlled by the `debug.computeInstrumentationLevel` engine option and is enable or disabled via the `debug.instrumentCompute` option (or `debug.instrument` which enables all instrumentation).
  - **Off**: instrumentation disabled; estimates are used instead, if available.
  - **Vertex**: cycles are recorded for each vertex execution. This means vertex execution is serialised, and this only really works on very small graphs.
  - **Tile**: cycles recorded separately for each tile. If you find this uses too much memory try using the `Ipu` method.
  - **Ipu**: cycles recorded for the slowest tile on each IPU for each compute set. An internal sync is inserted before and after each compute set. The tile that is used to do the cycle recording is controlled by `debug.profilingTile`. If you have enough memory consider using the `Tile` method instead.

  - **Device**: cycles recorded for the slowest tile on the entire decide for each compute set. This mode is not recommended - use `Ipu` instead.

- **External Exchange Instrumentation**: the type of instrumentation compiled into the program to record extern exchange cycles (i.e. host exchange and global exchange). The is enabled or disabled using the `debug.instrumentExternalExchange` engine option (or `debug.instrument` which enables all instrumentation).
  - **Off**: instrumentation disabled; estimates are used instead.
  - **Tile**: cycles recorded separately for each tile.

Graph

- **Number of compute sets**: how many compute sets are in the graph. This is after compilation so it may be a larger number than the number of compute sets added through the Poplar API because some are created during compilation.
- **Number of edges**: the number of edges in the graph, again after compilation. An edge is a pointer from an `Input<>`, `Output<>` or `InOut<>` vertex field to a variable.
- **Number of variables**: the number of variables in the graph, after compilation. Post-compilation variables are called *lowered variables* to distinguish them from the variables you created by hand when defining the program.
- **Number of vertices**: how many compute *vertices* the graph contains after compilation. Some vertices are added to the graph during compilation (e.g. `memcpy` vertices) so this may be higher than the number of vertices added with the Poplar API.

2.5.2 Engine options

This section displays all of the engine options that were used to generate the reports, the values supplied for each, and whether they are the default values or otherwise. You can also choose to show the options for the three different phases: compilation, execution and target.

- If no engine options are displayed, click on the **Engine Options** heading to expand that section and show its contents.
- To view options by their phase (execution, compilation or target), select the phase from the drop-down menu. If a phase is not available it is disabled.
- Values which are different from the default are displayed in **bold**. Use the "View - All" selection box in the top right-hand corner to switch between a list of all options, or "View - Non-default" to view just those which are different from the default values.
- Click on the small book icon to follow a web link to the Poplar API Reference for a description of each of these options.

2.5.3 Framework and Application JSON files

If you have a `framework.json` or an `app.json` file that was created in your program, their contents are displayed here so that you can check any parameters that you recorded in them. This is useful when comparing two reports, allowing you to spot differences easily.

You can put whatever information you wish into these two files, and if they're found in the reports folder when the other report files are being loaded, their contents are displayed in a foldable tree. This assumes that the files are valid JSON.

2.5.4 Report files

This section of the Summary report shows the folder from which the report files were loaded (or both folders, if you're comparing reports). It also shows which individual files are being loaded into the PopVision Graph Analyser, as documented *here*.

- If no Report Files are displayed, click on the **Report Files** heading to expand the section.

For each file that is present:

- A set of three green dots indicates the file was found, and is being analysed and loaded.
- A green tick indicates that a file was found and has been loaded successfully.
- A greyed-out question mark indicates that the corresponding file was not found.
- A red cross indicates that the file could not be loaded. A warning message can be found in the Host Information section, directly above.

The folder from which the reports were loaded (or both folders, if you're comparing reports) is also displayed.

2.6 Viewing a Memory Report 

The Memory Report shows a graphical representation of memory usage across all the tiles in your IPU system, showing graphs of total memory and liveness data, and details of variable types, placement and size.

There are two main areas of the Memory Report:

- The Memory graph, in the top half of the window, which shows two different types of memory graph. Click on the 'Graph Type' drop-down menu at the top left-hand corner of the graph to select a graph type:
  - *Total Memory graph*, which shows the memory usage of your program across all the IPU tiles. You can view a breakdown of this data by region (whether to display interleaved and non-interleaved memory separately) or by category (what the memory is used for).
  - *Liveness graph*, which shows the memory usage of the two types of program variables: those that are always live, and those that are Not Always Live (note that this is a maximum value - see the *relevant FAQ*.)
  - *Variables graph*, which allows you to plot the memory usage of multiple individual variables.
  - *Tile Map*, which shows the memory usage of the tiles overlaid on a physical floor plan of the IPU.
- The *Tile Memory Usage* report, in the bottom half of the screen, which shows memory usage broken down by various categories, and memory maps of individual tiles.

You can choose various *view options* for each graph, and you can also click on the graph to view details for an individual tile.

2.6.1 Navigating memory reports

You can move around the main report graphs using your mouse, as described *above.*

Selecting individual tiles/IPUs

When you first open the Memory Report, the bottom half of the report shows the memory usage for all tiles or IPUs at once. You can view the memory usage for an individual tile/IPU in various ways, as follows:

- Move the mouse over the report graph, and you'll see a tooltip box which shows the Tile/IPU ID, together with how much memory that tile/IPU has used.
- Click on the main graph to select an individual tile/IPU. Its ID appears in the 'Select' input at the top of the screen, and also shown above the report in the bottom half of the screen.
- If you hold down the Shift key when you click, you can select multiple tiles.
- You can also type one or more tiles/IPU IDs in the 'Select' input box if you wish, separated by commas.

Memory Report view options

There are several options for viewing Memory Report graphs, which you can select from the 'Options' drop-down menu in the top right-hand corner of the report screen:

- **Tile Memory From Binary** - whether to use the tile memory data from the binary file (`archive.a`).
- **Include Gaps** - whether to include the memory gaps between variables ('on' by default). Some memory banks in IPU tiles are reserved for certain types of data. This leads to 'gaps' appearing in the tile memory.
- **Show Max Memory** - whether to add a line on the graph to indicate the maximum memory available. If your program uses too much memory, its memory graph will cross this line, and is highlighted.
- **Order By Physical Tiles** - whether to order the horizontal 'Tiles' axis by software tile ID (the default) or by the physical order of tiles.
- **Plot Data By IPU** - this allows you to view the memory usage by IPU rather than by individual tile. Memory usage is summed across all tiles on each IPU. Note that the Variable and Tile Map Graph Types, as well as the Variables Tab below the graph, are not available when viewing by IPU.

There are also a number of options in the preferences that you can use to view the memory report differently - see the *Preferences section*, below.

2.6.2 Total Memory graph

This memory report shows the total memory usage across all the tiles on all IPUs.

- On a Memory Report, select 'Total Memory' from the Graph Type menu.

The horizontal axis shows the tile number (which you can order by software or physical ID (see *above*), and the vertical axis shows the memory usage.

Memory Report breakdown

Breakdown by Region

IPU memory has two different types of memory regions which Poplar allocates to data depending on how that data needs to be accessed:

- **non-interleaved**: Consecutive words are stored in the same memory bank. Code must be stored here.
- **interleaved**: Consecutive words are stored in alternating memory banks. Some high bandwidth load/store instructions like `ld128` only work in interleaved memory, and therefore some codelets require variables they are connected to to be stored here. Code cannot be stored here.
- **overflowed**: Memory that exceeds the maximum amount available on a tile.

Breakdown by category

When you select “Breakdown - By Category” another dropdown box is displayed with all the available memory categories. This can be used to understand the overhead costs of instrumentation.

- The ‘Select Categories’ dropdown is multi-select so you can compare multiple categories simultaneously.
- The total memory can be toggled on and off by selecting the “All” option.
- Breakdown by category is also available when viewing memory by IPU.

2.6.3 Liveness Memory graph

This memory report shows the memory usage of the two types of program variables:

- On a Memory Report, select ‘Liveness’ from the Graph Type menu.
- Always Live variables - these variables must be accessible for the entire lifetime of graph execution. This means nothing else can ever use the memory allocated for these variables. Examples include code and constants.
- Max Not Always Live variables - “Not always live” variables are only needed for some program steps. As long as two variables are not live at the same time they can be allocated in the same location, thus saving memory. This option shows the *maximum* amount of live memory use on each tile. See the *relevant FAQ* for more details.

2.6.4 Variables Memory graph

This memory report allows you to select multiple variables and plot their memory usage across tiles.

- On a Memory Report, select ‘Variables’ from the Graph Type menu.
- A prompt appears, suggesting you enter a variable to search for. Type the variable name into the search box at the top, and the application will find matching variables and display them in a drop-down list.
- Select a variable from the list to plot on the graph.
- Remove variables from the graph by clicking the small ‘x’ icon in their names in the key legend, below the graph.

You can also access this feature from the ‘Total Memory’ report by selecting a variable from the Variables tab as described *below*.

The Variables Memory graph is not available when viewing memory by IPU.

2.6.5 Tile map Memory graph

This memory report displays a schematic of an IPU and overlays it with a coloured representation of the tile memory usage for every tile for the selected IPU. The colour key is displayed on the right, and its range can be changed, as described *below*.

The Tile Map Memory graph is not available when viewing memory by IPU.

- On a Memory Report, select ‘Tile Map’ from the Graph Type menu.
- Select an IPU to view using the input on the left, and the map updates to show the memory usage for that IPU.
- Hover the mouse over the tile map to see a popup of the details of a tile within the selected IPU. This shows physical and software tile ID, memory usage and rank (described below). While you hover, a black line within the colour key, to the right of the tile map, shows the memory usage of the hovered tile, according to the colour scale currently selected.

- Click on a tile on the map to select it and see its memory usage. Its details are shown in the tabs and tables below, as in other memory reports. You can select multiple tiles by holding down the shift key while clicking a tile. Details for each tile are displayed in the tables below, with a column for each tile. The selected tile numbers are displayed in the search box above the tile map, so can enter them by hand if you know which one you're looking for.
- The Breakdown menu at the top of the tile map allows you to break down memory usage by region (see here for an explanation of this). When breaking down by region, the 'Region' control to the left of the map allows you to choose to display interleaved or non-interleaved memory.
- The Options menu above the tile map allows you to view the memory usage *including or excluding gaps*.

Note that you can change the size of the tile map by dragging the split-screen control, and it will fill the space available in the top half of the screen.

Changing the colour scale

There are three methods of colouring the tiles on an IPU that show their memory usage in different ways. Use the 'Scale type' dropdown to the left of the tile map to select one of these scales:

- **Relative** (default) - The colour of a tile depends on the memory between the upper and lower memory values.
- **Absolute** - The colour of a tile depends on the memory between zero and max memory.
- **Rank** - The colour depends on a linear ordering of tiles based on their memory usage.

When your model is out or memory, the colours are scaled appropriately, not just to the max memory.

2.6.6 Tile memory usage

The bottom half of the Memory Report screen shows three tabs that contain an analysis of memory usage by several different categories.

- The default view shows memory usage for all tiles (or IPUs, if you are choosing to plot by IPU instead of tile), but you can select an individual tile/IPU as described *above*.

Tile memory usage: Details tab

The Details tab in the tile memory usage report displays a hierarchical list of memory usage by category on the selected tiles. This list is divided into three main sections:

- **Including Gaps** - this shows memory usage on the selected tiles which includes the gaps between variables.
- *Excluding Gaps* - this shows memory usage on the selected tiles which excludes the gaps between variables. It is split into interleaved/non-interleaved memory and also categorised by the type of data in that memory location.
- **Vertex Data** - this shows the memory used by variables in the graph vertices as the Poplar program executes, categorised by the types mentioned in the 'Excluding Gaps' section, below.

Excluding Gaps

Memory usage on the selected tiles is displayed here in two categories, with memory usage figures for each:

- **by Memory Region** - this show memory that is non-interleaved, memory that is interleaved, and any memory that has overflowed.
- **by Data Type** - this shows memory further categorised by the type of data that is stored there (either overlapping data or non- overlapping data). The meaning of each of these categories is explained in the table below.

Not Overlapped data	
This shows the memory usage for parts of the memory where only a single variable is allocated. This includes variables that *cannot* be overlapped with other variables (the Always Live variables), and also variables that just happen to be not overlapped with other variables, even though it isn't disallowed.	
Variables	These are the variables added using `Graph::addVariable()`.
Internal Exchange Message Buffers	During the exchange phase of program execution, it may not be possible to send data straight to its destination. For example sending a single byte directly is impossible because internal exchange has a granularity of four bytes. In cases like this Poplar will copy the data to and from temporary variables using on-tile copies (which can copy individual bytes) and then do the actual exchange from these buffers.
Constants	These are variables that created using `Graph::addConstant()`.
Host Exchange Packet Headers	Host exchange is performed using a packet-based communication protocol. Each packet starts with a header that contains the address that its payload should be written to. These addresses are determined at compile time and the packet headers are stored in these variables.
Stack	This is where the program stack lives. It is created automatically during compilation. There is a single `Stack` variable on every tile that contains the stacks for the supervisor and worker threads. The stack size is configurable at compile time via an engine option.
Vertex Instances	These store the state of each vertex instance. Each call to `addVertex()` adds a single vertex instance to the graph, whose size is equal to `sizeof(TheCodeletClass)`.
Copy Descriptors	During compilation, Poplar will add compute sets that perform copies. These contain copy vertices, and the copy vertices reference additional data called Copy Descriptors that describe how to perform the copy.
VectorList Descriptors	A vector of pointers that points to the values of a multi-dimensional vector. The data for `VectorList<T, DeltaN>` fields.
Vertex Field Data	Variable-sized fields, e.g. the data for `Vector<T>` fields.
Control Code	All code that is not vertex code, this includes all the code that is generated from you Poplar `Program` tree, and code for each compute set that calls the codelet `compute()` functions for each vertex. It does not include the `compute()` functions themselves, which fall under the Vertex Code category.
Vertex Code	This is where the assembled code from the codelets is stored. A *codelet* is a class written in C++ or assembly, whereas a *vertex* is an instance of that class. Adding multiple instances of a single vertex type does not increase the amount of Vertex Code memory required.
Internal Exchange Code	The code instructions used to move data between tiles on an IPU.
Host Exchange Code	The code instructions used to move data between an IPU and the host machine.
Instrumentation Results	If you set the `debug.instrument` option in the Poplar Engine, this is where the cycle counts for various Poplar functions are stored. You'll notice, therefore, that enabling instrumentation increases your memory usage. Different levels of instrumentation can be selected, which will use different amounts of memory. Note that the size of these variables is dependent on the level of dynamic branching in your program – if you're timing every instance of a function call, the compiler won't necessarily be able to tell in advance how much memory it will require to keep a cycle count for each of them.
Overlapped data	
This is data for variables that are Not Always Live, meaning that they are temporary and can be overlapped by other not-always-live variables if the two variables are not live at the same time. Reusing memory in this way reduces the amount that is required by Poplar programs. The sizes reported here count the memory used	

Tile memory usage: Vertices tab

This tab in the tile memory usage report lists the memory used by the graph vertices, together with the total memory size they occupy across the selected tiles (or all tiles, if none is selected). This list is ordered by decreasing memory usage.

- For each vertex, the Poplar namespace and function name are listed, together with any additional information about their types. Please refer to the Poplar API Reference for a description of each of these functions.
- The origin of each vertex, in a small blue box, is displayed after the vertex name, indicating whether the vertex was written in C++ or Assembler (ASM).
- You can filter the vertices by name, using the input box above the table, or by source (C++ or ASM (Assembler) using the dropdown list.

Tile memory usage: Exchanges tab

This tab in the tile memory usage report displays the internal exchange code size for all tiles/IPUs, or the currently selected tiles/IPUs. When comparing reports, there is an additional column that shows the difference between the source and target code size.

Tile memory usage: Variables tab

This tab in the tile memory usage report displays a memory map of the currently selected tile, showing code and variable usage across the memory locations. Note that this tab is not available when viewing memory by IPU.

Entries in this section of the report are only present if the :ref:`vars.capnp` file was present in the report files directory. See the `Report files <report_files>` section for details about how to generate this file when executing your program.

You can *toggle* between two different views (memory map and table view) on the Variables tab by clicking on the icon in the right-hand corner of the tab contents.

- Select an individual tile as described *above* to view its memory layout and variable usage.

There are several interactive features of the Variables view that can help you find the locations in which variables are stored:

- The selected tile's memory is displayed vertically in a scrollable area that is 1024 bytes wide. The tile memory is partitioned into *memory elements*, which are either Interleaved or Non-Interleaved (see *here* for more information). Any unused elements at the end of the IPU memory are not displayed.
- Variables are displayed as coloured bars which span the memory locations, and in places where two or more variables overlap, only the largest is shown.
- All variables in the memory layout are coloured according to their type. Click the colour key icon in the top right-hand corner to view the colour key for each type. The meaning of each of the categories displayed here is described in the table above. You can click on the checkboxes on the left-hand side of each variable to display it or hide it from the variable plot.
- Click on a variable to display its details, which appear on the right-hand side of the memory layout. This displays all variables which exist at any time at that memory location.
- Click on the 'Show' button at the bottom of the variable details, beneath the 'Interference' heading, to filter other variables that interfere with the selected variable in terms of memory placement. See the *Memory interference* section below.
- Search for a variable by entering search text into the input field above the memory layout. Variables with matching text in their names will be highlighted in the memory layout, with all others disappearing. You can clear any text you've entered here by hovering over the box and clicking the small x icon at the right-hand end.
- Plot one or more variables on the Memory graph, as described *below*.

You can expand the variables map to fill the report window by clicking on the arrow button in the top right-hand corner of the map. A corresponding arrow to shrink the map again appears in the top-right corner.

Memory interference

You can see other variables with which a selected variable 'interferes'. These are variables that are in contention in terms of their memory placement. There are three ways variables can interfere with each other:

- **Memory:** A variable cannot be occupy the same bytes as some other variables because it is live at the same time as them. Always-live variables interfere with every other variable in this way.
- **Element:** A variable cannot be in the same memory element as another one. This can occur in some case when two variables are connected to the same vertex, and it is reading from one and writing to another using certain instructions.
- **Region:** A variable cannot be in the same memory region as another one

To see which other variables interfere with a selected variable:

- Select a variable from the memory map by clicking on it. Several variables may occupy that memory location, and their details are displayed in a list on the right-hand side.
- Click on the 'Show' button at the bottom of a variable's details, beneath the 'Interference' heading, and the variables in the memory map will be filtered using that variable's name, showing only those that interfere with it.
- To re-display all variables, click the small cross in the filter box at the top of the variable memory map display.

Variable types

Variables in the memory layout diagram are categorised by colour as described below. Note that more detailed descriptions are available in the *Excluding Gaps* table, above.

- **User variables** - these are user-defined variables.
  - Variable - variables created using the addVariable() Graph function.
  - Constant - variables created using the addConstant() Graph function.
- **Code variables** - these are variables that are created by Poplar to execute the program code.
  - Control Table - experimental, empty by default.
  - Control Code - variables used by Poplar to run the program. Some specific control code variables are described in the section *Known variables*, below.
  - Vertex Code - the code within the vertex 'codelets'.
  - Internal Exchange Code - the compiled code used to move data between tiles on an IPU.
  - Host Exchange Code - the compiled code used to move data over PCI between an IPU and the host machine.
  - Global Exchange Code - the compiled code used to move data between multiple IPUs.
- **Vertex data variables** - these are variables associated with tensors.
  - Vertex Instance State - the internal state of each vertex instance.
  - Copy Descriptor - additional metadata used for copy vertices.
  - Vector List Descriptor - data for `VectorList<T, DeltaN>` fields.
  - Vertex Field Data - data for `Vector<T>` fields.
- **Temporary variables** - these are used to temporarily store information that Poplar uses while executing the program code. They are 'not always live', overlapping variables.

  - Message - temporary storage for internal exchange (between tiles on one IPU).
  - Host Message - temporary storage for host exchange (between an IPU and the host).
  - Global Message - temporary storage for global exchange (between IPUs).
  - Rearrangement - variables used to store intermediate values when an edge is connected to non-contiguous variables.
  - Output Edge - temporary output variable used when a vertex is connected to a variable on a different tile. The data is copied by internal exchange after the compute set has been executed. Note that this category is also used for Input Edges. It should really be named Input/Output Edge.
- **Miscellaneous variables** - other variables that don't fit into the categories above.
  - Multiple - sometimes variables are merged during lowering. If they came from two different categories the resultant variable is put in this category.
  - Control ID - the combination of program ID, sync IDs and software sync counter.
  - Host Exchange Packet Header - header information for PCI messages between the IPUs and the host machine.
  - Global Exchange Packet Header - header information for PCI messages between IPUs.
  - Stack - thread stacks for each tile.
  - Instrumentation Results - the cycle counts if instrumentation is enabled.

Known variables

Poplar uses some specific variables that you may encounter on various tiles. Their purpose is described below:

- **.text.poplar_start** – this is the entrypoint and main control code for your program. It's roughly equivalent to `main()` in a C program.
- **.text.supervisor.control__func[…]** – these are the control codes for the functions in your compiled program. These are the functions you can see on the *Program Tree* report.
- **.text.supervisor.control_initPrngSeed** - the control code to initialise the seed for the Psuedo-Random Number Generator (PRNG).

Plot multiple variables

When a variable is selected in the Variables tab, its details are displayed in the right-hand column next to the memory map for that tile. You can then plot that variable's position in memory on the main graph, as follows:

- With the variable selected, click on the 'Plot variable' button. The 'Graph Type' menu at the top changes to 'Variables'. The variable name is added to the variable list above the report, and you can see how it is placed across the tile memory.
- Click other variables in the memory map, and you can repeat the process above, adding them to the variable list at the top of the report, and displaying their memory placement together on the same graph. The default behaviour shows the size of the variable on each tile. If you select 'Plot variable by address' from the Options menu at the top of the screen, you can see how the variable is laid out in the memory space.
- To remove a variable from the graph, find its name in the list above the graph, then click the small 'x' button at the right-hand end.

Full-screen option

When viewing the content of the Variables tab it may be easier to view the data in full-screen mode. You can toggle this option on and off using the arrow button in the top right hand corner of the tab.

Toggle between table and graph view

The Variables tab has a table view, listing the variable names and their sizes, and also a graph view which provides more in-depth detail. You can toggle between these by using the chart/text button in the button group in the top right-hand corner of the tab.

Show differences between selected tiles

When multiple tiles are selected the difference between values is displayed in red or green text on the table view. You can remove variables that have the same value by enabling the "Show differences between selected tiles" option. This is accessible by clicking on the cog icon and checking the box in the drop-down menu. This filters out all variables that have the same value, and leaves only those that are different.

2.7 Viewing a Liveness Report 

The Liveness Report shows which of the 'Not Always Live' variables are allocated at certain points in your program, and gives a detailed breakdown of variable memory usage by the compute set that they're in. There are two main areas to the report:

- the top half of the reports shows the graph of *memory usage against compute set*.
- the bottom half of the report shows *details* about the variables that are used within the selected compute set.

For a standard machine learning model that you're training, for example a ResNET, you'll generally see a curve that ascends to a peak and descends again. The rising portion of this curve is the memory usage during the forward pass of the training algorithm, where many activations are created ('not always live variables'), and the peak represents the point where the maximum number of activations exist. As the curve descends again, which represents the backwards pass of the training algorithm, the activations are 'released' after being used to update the weights.

When you're inspecting a liveness graph, it's informative to look at the peak of this curve, and select the corresponding compute set to show the how the variables usage is contributing to the greatest memory utilisation.

2.7.1 Navigating Liveness reports

You can move around the main report graphs using your mouse, as described *above.*

2.7.2 The Liveness graph

- Hover your mouse pointer over the graph to see a callout containing the compute set ID, together with the amount of memory used by that compute set.
- Click on the graph to see that compute set's stack details below the graph, which displays its name and memory size. You can select multiple program steps by holding the Shift key and clicking other program steps. Each of these steps is then displayed in its own column on the *Not Always Live Variables* and *Vertices* tabs.
- You can choose to display memory usage statistics for the selected tiles - see the relevant *Preferences section*, below.

- You can plot the lifetime of Not Always Live variables directly on the graph - see *here* for more details. This is an experimental feature.

Selecting a source

You can choose whether you want to select an individual tile, or, if the report was generated using Poplar SDK 1.2 or later, select an individual IPU.

To select particular tiles or IPUs:

- Click on the 'Select Source' dropdown list at the top of the graph and select the source you want to use for the graph. Each source will have its own plot.
- **All tiles** - shows liveness data for all the tiles (the default)
- **Worst tiles** - shows the two tiles that have the highest memory usage during program execution, and you can select either of them to view.
- If the report was generated in Poplar 1.2 or later, you can also select one or more IPUs from this list.

There is a Poplar Engine setting which allows you to capture more than the default two worst tiles. Please refer to the Poplar API Reference for full details of these options.

Filtering steps

You can concentrate on the steps you're interested in by filtering on a particular search term. When you enter a term (and press the Return key, those steps that don't match in the execution graph are moved to a separate dataset in the graph, and 'greyed-out', leaving only the steps whose named match your search term.

To cancel the filtering, click on the small 'x' at the right-hand end of the search box.

Viewing options

There are several options for viewing Liveness Report graphs, which you can select from the 'Options' drop-down menu in the top right-hand corner of the report screen:

- **Include always live** - whether to include a second trace in the graph that shows variables that are always present during the entire execution of the program, and always have the same memory address (for example `stack`). Note that you can show and hide each of the traces on the graph by clicking on their colour key, just below the x-axis.
- **Include empty stacks** - whether to include stacks that contain no variables.
- **Stack IPUs** - whether to display any selected IPUs (see *above*) stacked or not.
- **Show Max Memory** - whether to display a line on the graph showing the maximum memory limit for total memory, a selected tile, or a selected IPU. Note that this requires the 'Include Always Live' option, above, to be enabled, and 'Stack IPUs' to be disabled.

2.7.3 Liveness stack details

In the bottom half of the Liveness report screen, you'll see a tabbed list of live variable details that show their memory usage. These details are described below.

Viewing enhanced debug information

If you have captured enhanced debug information when compiling your program, it is visible in the 'Always live' and 'Not always live' variables tabs. See the section on the *Debug Information* file, above, for details of how to capture this information.

If a variable has a small 'disclosure arrow' ( ) next to it, click it to show enhanced debug. You can see a list of software layers on the left (for example, Poplar, PopLibs, etc.) Selecting a layer shows the debug information that has been added by that layer.

For all variables you have information captured from the Poplar layer. This may include:

- whether it's a Variable, Cloned Variable or Constant,
- the shape of the variable as an array of dimensions,
- the type of variable (e.e. 'Half'),
- for cloned variables, which variable it was cloned from, and the method by which it was cloned,

If the variable was created as part of a PopLibs API call, you can also see the following information:

- the PopLibs API call
- the input tensors to the API call
- the arguments to the API call
- the output tensors to the API call

Note the variable may not be an output of the PopLibs API call. It could be an internal variable created for the operation. Depending on your application, you may see further debug information for the framework and/or application.

By default, the debug information for the first PopLibs and Poplar call is shown. You can choose to show all PopLibs/Poplar calls that are made internally by clicking the gear icon in the top right-hand corner of the debug information box and selecting "Show All Debug Infos". For instance, you may see the debug information for the PopLibs `matmul` API call. If you enable the option to "Show All Debug Infos", you will see the internal implementation PopLibs calls, which include the PopLibs convolution API call. (`matmul` is implemented as a convolution.)

Always Live Variables

This tab shows the variables that are always live. Their data must always be available and therefore they cannot share memory with any other variable.

Not Always Live Variables

This tab shows the variables that are not always live. At certain points in the program we do not need to store any data in them, and therefore other variables can be allocated at the same location. Variables are never "allocated" or "deallocated" at runtime. All variables are statically allocated at compile time and always have a fixed address.

You can select one or more of the Not Always Live variables to see their lifetime displayed on the graph above. Click on the small icon at the right-hand end of their listing, and they will be added to the plot, showing when they were created, and when they were destroyed. Each variable plotted also appears as a small blue box above the graph, and you can click the small x within them to remove them from the graph. This is an experimental feature.

Vertices

This tab shows the vertex functions that are contained in the selected compute set.

- The origin of each vertex, in a small blue box, is displayed after the vertex name, indicating whether the vertex was written in C++ or Assembler (ASM).
- The number of that vertex type created is also shown.

Cycle estimates

This tab shows the cycle estimates of each tile on a ipu. This is only available when using an IPUModel

This is an experimental feature, and can be enabled and disabled in the Preferences.

Show differences between selected tiles

When comparing reports the difference between values is displayed in red or green text on the table view. You can remove variables that have the same value by enabling the “Show differences between selected tiles” option. This is accessible by clicking on the cog icon and checking the box in the drop-down menu. This filters out all variables that have the same value, and leaves only those that are different.

2.8 Viewing a Program Tree 

The Program Tree report shows a hierarchical view of the steps in the program that is run on the IPU. The report has a menu on the left-hand side that lists the Control Programs and Functions. The program steps contained within the selected control program or function are displayed in the main report area, on the right.

- Click on one of the Control Program or Function numbers in the left-hand column to see the sequence of instructions that are contained within it.
- You can view any debug information that has been captured for the selected program step - see *below* for more details.

The program steps in the main report area are listed hierarchically, and you can collapse or open steps that have nested steps with them by clicking on them. A small grey triangle at the start identifies steps that have sub-steps: if it’s pointing to the right, then that step is ‘collapsed’, and all of its sub-steps are currently hidden. Click the step to open up that step and show all its sub-steps. All steps are initially shown opened up.

Each of the program steps is colour-coded, making it easier to ‘pick out’ particular steps that you’re interested in viewing. Where steps corresponding to those found in the *Execution Trace* report, they are coloured the same.

When you select a step that is associated with other program steps, they are all highlighted in yellow. Details of that vertex are then displayed in the *tabbed section* below the main program tree.

For a detailed explanation of what each of these program steps involve, please refer to the Poplar & PopLibs API Reference Guide on the Graphcore Developer Documentation website.

2.8.1 Searching for program steps

You can search for a particular step by entering some text in the Search box at the top of the report window, and then pressing the Return key. Program steps that match your search term are highlighted in yellow in the main part of the report. Control programs and Functions that contain matching steps are also highlighted in the left-hand window.

You can scroll through the results, either by pressing Return repeatedly, or by clicking the arrow buttons under the search box. Single arrows move the result selection back and forward by one, and double-arrows move by ten.

When comparing two reports, two sets of results are displayed, and you can step through them searately. The top set of results shows matches in the source report (on the left), and the bottom set of results shows matches in the target report (on the right). Match highlighting works independently in both reports.

2.8.2 Details tab

When a program step is selected, more details of that step are displayed in the Details tab, such as the ID, type and name of the step.

If there are any vertices associated with that step, they are listed in the table below, with the following details:

- The name of the vertex class, together with a small blue icon indicating whether the vertex was written in C++ or Assembler (ASM).
- The memory size occupied by each vertex instance of this type.
- How many instances of this vertex were created from the selected program step.

2.8.3 Viewing Debug Information

Clicking on a program step reveals debug information that was captured during program compilation. This reveals all the steps that are needed to execute the selected operation. In addition, when you select the the debug information, the program steps created for that API call are highlighted in the Program Tree.

The debug information is the same as that found on the *Liveness report*. See the display Options control above for showing and hiding the debug data.

2.8.4 Change Layout

When comparing the Program Tree of two reports a button in the top right hand corner allows you to arrange the Program Trees side by side or one on top of the other.

2.9 Viewing an Operations Summary 

The Operations Summary displays a *table* of all operations in your model, for a software layer, showing statistics about code size, cycle counts, FLOPs and memory usage. You can also select *which software layer's operations* you want to summarise.

Clicking on an operation in the table reveals further information about it in the *tabbed section* in the bottom half of the report, displaying graphs of code size, cycle counts, and various other measurements and estimates for the selected operation. You can *choose which columns* you want displayed in the table, and also apply *sorting and filtering* to it.

2.9.1 Operations table

The Operations summary table shows a list of all the operations within the selected software layer.

Because the column headings in this table are typically quite long, we've used abbreviated headings that match the full column names displayed in the **Columns** drop-down list. If you hover your mouse over the column headings, you'll see the full name displayed as a pop-up box.

Selecting a software layer

By default, the PopLibs software layer is displayed, as you can see from the drop-down **Layers** list in the top right-hand side of the table. You can select other software layers whose operations you wish to see by selecting from this list. Depending on your program, the following software layers may be available:

- Poplar and PopLibs
- PopART IR and PopART Builder
- ONNX
- TensorFlow Poplar Drivers
- TensorFlow HLO Instructions
- TensorFlow XLA Operations

Changing layers involves a sometimes lengthy re-calculation of the table metrics, so you may need to wait a short while for larger reports.

Selecting which columns to display

The 'Columns' control works differently depending on whether you're viewing a single report, or comparing two reports:

- When viewing a single report, you can select as many columns as your screen has room to display. Each column contains the values for that metric.
- When comparing two reports, the Operation Name is always displayed, and you can select one other column to display. As well as the Operation Name column, there are three other columns that show the value of the selected column metric for the source report and the target report, as well as a column that shows the difference between the source and target values.

By default, the operations table displays the following metrics for each operation (single view only):

- Operation Name
- Debug Name
- Code Size (Total)
- Measured Cycles (Total)
- FLOPs

FLOPs are not generated by default. Enable the `profiler.includeFlopEstimates` Poplar Engine option to generate FLOP estimates.

Many other operation metrics can be displayed or hidden in this table by checking or unchecking the data types in the drop-down **Columns** list in the top right-hand corner of the operations table. Your current column selection preferences are automatically saved.

The **Not Always Live Delta (NAL)** option (experimental) shows the difference that each operation makes to the variable memory as the program executes. This helps you identify which operations are the most 'expensive' in terms of memory. Note a operation may have multiple liveness values if it is repeated in the execution, the value show is the liveness delta from the first occurrence.

Columns showing a range of Cycle Estimates are experimental.

Sorting and filtering

You can sort the operations table by any of the column headings, as well as only showing operations that match a particular string:

- Click on a column heading to sort the table by that operation metric. Repeated clicking sorts in ascending, descending order, or removed the sort from that column. A small blue triangle (or none) indicates which order is currently being used.
- Enter some text in the **Filter operations** box above the table, and press Return to display only those operations whose operation name or debug name matches the text you enter. Remove filtering by emptying that text box.

2.9.2 Operations Summary tabs

The tabbed section in the bottom half of the Operations summary shows further information about the selected operation. When no operation is selected, only the Summary tab is visible, which shows some general statistics about all the operations in the currently selected software layer.

- Select an operation from the table by clicking on it. The selected operation is displayed at the top of the tabbed section, along with a small "x" icon that you can click to deselect it, and return to the Summary tab.

Summary tab

When no operation is selected from the table, this tab shows a breakdown of operations for the selected software layer, including:

- The total number of total operations for that layer
- The total code size for that layer
- The total number of cycles executed in that layer

When an operation is selected from the table, data from the default table columns is displayed.

Program Tree tab

When an operation is selected from the table, this tab shows the program steps involved in that operation. This is the same data displayed in the *Program Tree* report.

Code tab

When an operation is selected from the table, this tab shows a graph of the code size executed for that operation. Code size for OnTileExecute and DoExchange program steps is displayed against tile number for all IPUs. You can *zoom and pan* around this graph just like you can for other graphs.

Cycles tab

When an operation is selected from the table, this tab shows the number of cycles taken by the selected operation, plotted against all the IPU tiles. You can *zoom and pan* around this graph just like you can for other graphs.

By default, only cycle counts for the OnTileExecute and DoExchange program steps are displayed, but you can add other program step types to include on the graph by selecting them from the **Options** drop-down list in the top left-hand corner of the graph. Available options are:

- **Show Copies** - this separates steps for OnTileExecute programs that are just for copies.
- **Show Estimates** - this includes a number of estimated cycle counts:
  - DoExchange Estimated Cycles
  - OnTileExecute Estimated Cycles
  - OnTileExecuteCopy Estimated Cycles
  - StreamCopyMid Estimated Cycles
  - GlobalExchange Estimated Cycles

FLOPs tab

When an operation is selected from the table, this tab shows a graph of the total number of FLOPs (Floating Point Operations) executed for the selected operation, plotted against IPU tiles.

Debug tab

When an operation is selected from the table, this tab shows debug information from the currently selected software layer for that operation. This information is identical to that on the *Liveness Report* and the *Program Tree* Debug sections.

2.10 Viewing an Operations Graph 

The Operations Graph displays a graphical representation of TensorFlow models, showing High Level Operations (HLO) and enabling you to:

- drill down through the modules, expanding and collapsing the layers to get to the level you want;
- view details of operations, edges and layers;
- view the type and shape of the tensors between operations;
- view graphs of pipelined models to see what is in each pipeline stage, and what passes between them;
- colour items in the graph based on selected metrics (for example, code size or cycles used);
- configure the layout with a number of advanced options.

The Operations graph shows how the HLOs are connected to each other in your TensorFlow model in a number of nested layers. Layers and the operations they contain are displayed as boxes, and the tensors that those operations use are shown as arrows between operations.

The report is shown in split-screen, the left-hand side showing the graph at the selected level, and the right-hand side showing information about the selected object in the graph in a series of tabs.

- Pan around the operations graph by click and dragging your mouse anywhere on the graph.
- Zoom in and out of the operations graph using the mouse scroll wheel.

2.10.1 Graph entities

There are several types of entity displayed on the operations graph. You can select them, and expand the layers and calls to drill down into the model. You can click and double-click entities on the graph to expand them (layers and calls) or to display other information about them (operations and edges).

You can customise the appearance of the entities in the operations graph by adjusting the advanced view options: see *below*. The Options menu at the top of the report allows quick access to two of these layout options ('Show Backward Pass' and 'Show Edge Labels').

HLO layers

HLO layers are created when two or more operations have debug names which have the same prefix string, separated using the `/` character. They are displayed as boxes with solid borders and square corners:

HLO Layer

- Click on a layer to see its details displayed on the right-hand side (see below for details).
- Double-click on a layer to expand it. All of that layer's sub-layers are now displayed within the original layer, which is now displayed as a box around the sub-layer entities. Double-click that outer layer box to return to the original, enclosing layer.
- Other layers within the operations graph can be displayed by selecting them from the drop-down menu at the top of the graph.

Defining HLO layers

The operations graph works best if you name your TensorFlow operations using either Keras layers or `tf.variable_scope` (see the TensorFlow documentation for name_scope. Currently the view assumes the top level `tf.variable_scope` is called "all", which is a common convention in the public examples.

For best reults, all forward operations should be in the "all" layer, and all backward operation names should start with `gradents/all`.

It has been found that TensorFlow 2 `GradientTape` does not work well, as it does not record the scope names for the backward pass. For best results, use the TensorFlow 1 `compat` optimizers.

High Level Operations (HLOs)

HLOs are displayed as boxes with solid, black borders with rounded corners:

HLO Name

- Click on an HLO to show its details in the tabs on the right-hand side of the report.

Operations that are disconnected from the rest of the graph are displayed as boxes with dashed, grey borders with rounded corners:

Disconnected HLO

HLO Calls/fusion operations

HLO Calls or fusion operations are displayed as boxes with double borders and rounded corners:

HLO Call

- Click on an HLO Call to show its details in the tabs on the right-hand side of the report.
- Double-click an HLO Call to expand it and see the operations within it.
- Expanded HLO Calls show up as a list of breadcrumbs at the top of the graph, and you can click on those to retrace your steps.

Edge tensors

Tensors - data that is used as the input to, or output from, operations in the graph - are displayed as arrows. Their labels are either the tensor's shape and type (for single tensor), or how many of them there are (for multiple tensors).

- Click on a tensor to see its details in the tabs on the right-hand side.

There are several options to show or hide labels in the *Advanced Options* tab.

2.10.2 Selected entity information tabs

When you select an entity from the graph (a layer, operation, call, or edge), its name is displayed at the top of the right-hand side of the report, and some extra information associated with the entity is displayed in the tabs beneath. This includes:

- **Summary tab** - this displays different information depending on which type of graph entity is selected:
  - When a layer is selected, this displays some statistics about the layer in general, such as total estimated FLOPS.
  - When an operation or call is selected, the inputs and outputs to the operation are displayed, together with their associated tensor shape, as well as statistics about code size (OnTileExecute, DoExchange and Total) and estimated FLOPS.
  - When an edge tensor is selected, the source ('From') and destination ('To') operations are displayed, as well as tensor shape and size.
- **Program Tree tab** - any program tree steps associated with the selected operation.
- **Details tab** - graphs of various metrics plotted against tiles, including code size, cycles and FLOPS.
- **Debug Info tab** - any debug information associated with the selected operation.
- **Advanced Options tab** - see *below* for details of how to customise the appearance of the operations graph.

2.10.3 Highlighting operations by metric

The operations graph has a feature to colour graph entities based on various metrics. Entities that have a relatively high value for that metric are coloured with a 'hot' colour (red) to highlight those operations that are costly in terms of memory, cycles, etc., and entities with a relatively low value for the metric are coloured with a 'cool' colour (blue).

- From the 'Highlight' drop-down menu at the top of the report screen, select a metric that you want to use to highlight certain operations.

Metrics that are available to use for highlighting include:

- None - switch off highlighting
- Code size (Total)
- Code size (OnTileExecute)

- Code size (DoExchange)
- Estimated Cycles (Total)
- Measured Cycles (Total)
- FLOPS

The colour key for metric values is displayed in the top right-hand corner of the graph, showing the highest and lowest values present in the current view.

2.10.4 Advanced options

The right-most tab on the right hand side of the report shows a number of display options for laying out the items in the operations graph.

- Check and uncheck options to display or hide information on the graph.

The Options menu at the top of the report allows quick access to two of these layout options ('Show Backward Pass' and 'Show Edge Labels').

2.11 Viewing an Execution Trace 

The Execution Trace report shows the output of instrumenting a Poplar program, capturing the cycle count for each step in the program. Also displayed are statistics, tile balance, cycle proportions and compute-set details.

There are two halves to this report:

- the *Execution Trace graph*, in the top half of the screen, showing wither a flat compute set execution trace, or a flame graph of the stack,
- a *details report*, in the bottom half of the screen, showing statistics about the cycle proportions, tile balance and compute sets present in the portion of the graph currently displayed.

2.11.1 The Execution Trace graph

The top half of the Execution Trace report shows, by default, a 'flat graph', showing a set of consecutive blocks for each IPU, identifying what program steps were executed (as shown in the *Program Tree* report), and how many cycles they took. You can also view it as a flame graph, where the Poplar debug information is used to group compute sets together as part of the same operation, or application layer.

- Selecting features that you want in the graph from the *Graph View* drop-down, including Flame graph, BSP Trace, and displaying separate runs of the program.
- You can move around the main report graph using your mouse, as described *above.*
- Hover your mouse pointer over the graph to see a callout containing the compute set ID, together with the amount of memory used by that compute set.
- Click on the graph to see that compute set's stack details below the graph, which displays its name and memory size.
- Double-clicking on a layer when the flame graph option is selected expands that layer to the full width of the graph. You can select two layers at once by clicking on the first one, then Shift-clicking the second one. The graph then expands to contain just two layers. This makes it possible to inspect the cycle proportions of only the visible section of the execution trace, making it easier to understand the proportion of cycles spent in each type of process.

Execution View

Use the 'Execution view' dropdown list in the top left-hand corner of the Execution report to control what features you'd like to see on the graph. These include:

- **Runs** - this setting can be toggled to display or hide the inclusion of program run information on the graph. When enabled this displays, just below the mini-map, a set of dark grey markers that indicate when each program run starts and ends. On the graph itself, a bar at the top of each IPU 'lane' shows the names of each program that runs. See *Defining program names*, below.
- **Flat & Flame** - these settings toggle between a 'flat' view, where all the program steps are compressed into a single 'lane' on the graph, or a 'flame' view, where the call structure of all steps is displayed. Note that you can also control how overlapping steps are displayed with the **Separate overlapping steps** control, in the *View Options* control.
- **BSP** - whether to include a graphical depiction of the BSP activity, showing where the patterns of the IPUs' internal Sync, Compute and Exchange steps occur.

The current combination of settings is displayed on the drop-down button itself.

Defining program names

You can specify program names in your programs by using the Poplar or PopART APIs. The Execution trace graph can display this information by enabling the *Runs* option in the Execution View drop-down control above the graph.

- The Poplar `Engine::run` API now takes a debug string for the name of the run.
- The PopART `Session::run` API allows you to specify a string for the name of the run, as well as additional strings for internal program runs, for example: `WeightsToHost`.

If you enable the display of runs in the graph, but no run name was provided for a run, a sequentially numbered default name is generated, for example: `Engine::run #5`.

Selecting IPUs

The menubar above the graph contains a drop-down list named 'Select IPU' which allows you to select all IPUs or any individual IPU.

Filtering steps

You can concentrate on the steps you're interested in by filtering on a particular search term.

- Enter a term in the search box at the top, then press the Return key. Steps that don't match in the execution trace graph are 'greyed-out'.
- Cycle through the matching steps by pressing the Return key repeatedly, or clicking the arrow keys below the search box. The single arrows move backwards and forwards through the matching steps, one at a time, and the double arrows move ten at a time.
- To cancel the filtering, empty the search box by clicking on the small 'x' at its right-hand end.

This feature is currently only available on the 'flat' execution graph.

Viewing options

There are several options for viewing Execution trace graphs, which you can select from the 'Options' drop-down menu in the top right-hand corner of the report screen:

- **Show Tile Balance** - this shows what percentage of tiles are in use during that program step, visible as shading in each each step.
- **Show Terminals** - whether to include a line at the right- hand end of each process in the graph.
- **Group Executions** - whether to group executions together that are part of the same 'higher' process further up the call stack. Grouping is determined by the slash-delimited function calls that are logged to the execution trace profile output.
- **Group Syncs** - whether to group multiple successive `Sync` steps together in the graph.
- **Show Text** - whether to show the name of each step in the graph.
- **Separate Overlapping Steps** - whether to split up overlapping program steps into separate, non-overlapping 'lanes' in the graph so that they can all be seen at once.
- **Show External Syncs** - whether to display External Sync steps in the graph. There are often many fo these, and hiding them may make the execution graph plot easier to understand in some cases.
- **Show SyncAns** - whether to display SyncAns steps in the graph. As for External Syncs, above, hiding these steps may simplify the graph plot.

You can also view the colour key that the execution trace uses by clicking the key icon in the top right-hand corner of the graph.

2.11.2 Execution Trace details

The bottom half of the Execution Trace report shows more details about the execution trace. It includes:

- the *Summary tab*, which shows statistics, cycle proportions and tile balance (experimental).
- the *Details tab*, which shows details of a selected process from the execution trace graph.

Summary tab

This tab provides an overview of the portion of the execution trace currently displayed in the graph in the top half of the page.

Statistics

The statistics displayed are:

- **Cycles** - the number of cycles in the visible section of the execution trace.
- **Rx / Tx** - for StreamCopy and GlobalExchange steps, the amount of data transmitted and received during the operation.

Cycle proportions

A bar is displayed for each IPU that shows a graphical representation of the proportion of cycles that are taken executing each type of compute set. If you hover your mouse over these bars, you'll see a key that shows what process each colour represents, as follows:

- **Internal Sync** - the sync process that occurs between each tile on an IPU as part of the BSP process.
- **External Sync** - the sync process that occurs between each IPU as part of the BSP process. External syncs are also used in some host-device communication situations, where the IPUs all need to synchronise with an event outside their boundaries, for example a flow control step in the host program.
- **OnTileExecute** - the vertex compute code executed on each tile.
- **DoExchange** - the tile-to-tile data exchange within an IPU.
- **GlobalExchange** - an IPU-to-IPU data exchange.
- **StreamCopy** - a data exchange between an IPU and the host machine over PCI.

Tile balance

A bar is displayed for each IPU that shows a graphical representation of the percentage of tiles utilised by steps in the current viewport.

It is calculated by averaging the tile balance for all Steps (excluding Syncs) that were executed on the IPU. In previous versions of the Graph Analyser this value was derived from the percentage of cycles executed by a step, weighted by the number of tiles the step used.

Details tab

When an program step is selected in the flat graph, or a layer is selected in the flame graph, a list of program steps, with further details, is shown here. Many of the details are the same across the different types:

- **Cycles** - the number of cycles on active tiles that the program step used to execute,
- **Active Tiles** - the number of tiles involved in executing that program step,
- **All Cycles** - the number of cycles on all tiles, with additional statistics.
- **Tile Balance** - a measure of how efficiently the program step is spread across the tiles. See *View Options* for more details.
- **Active Tile Balance** - this is a recalculation of the tile balance measurement above, but excluding those tiles that do nothing.

Internal Sync

This is a sync process between tiles on an IPU.

External Sync

This is a sync process between IPUs.

SyncAns

This is an internal Automatic, Non-participatory Sync process. A tile can pre-acknowledge a number of internal/external syncs using the 'sans' instruction. The Sync ANS instruction will wait until all those pre-acknowledged syncs actually happen.

OnTileExecute

This is a piece of vertex code being executed in a tile. In addition to the common information listed above, the following is displayed:

- **By Vertex Type** - this shows what vertices are involved in the process execution.

Below these details, an interactive graph plot is displayed that shows how the selected program step makes use of cycles on each tile as it executes. For DoExchange programs, there is also a graph of the data received and transmitted by the program during its execution.

DoExchange

This is an exchange process, where data is exchanged between IPU tiles. In addition to the common information listed above, the following is displayed:

- **Total Data** - the total amount of data transferred during the exchange,
- **Data Transmitted** - the amount of data transmitted during the exchange,
- **Data Received** - the amount of data received during the exchange,
- **Data Balance** - the mean amount of data exchanged divided by the maximum amount of data exchanged,
- **Exchange Code** - how large the variable is that holds the code for performing the exchange,
- **Source Variables** - a truncated list of the variables from which data was sent in the exchange,
- **Destination Variables** - a truncated list of the variables to which data was sent in the exchange.

GlobalExchange operations

GlobalExchange is the process by which data is exchanged between IPUs. In addition to the common information listed above, the following is displayed:

- **Total Data** - the total amount of data transferred during the exchange,
- **Data Balance** - the mean amount of data exchanged divided by the maximum amount of data exchanged,
- **Source Variables** - a truncated list of the variables from which data was sent in the exchange (with temporary variables given basic integer names),
- **Destination Variables** - a truncated list of the variables to which data was sent in the exchange (with temporary variables given basic integer names).

A tile's physical location on an IPU, and how far it is away from the main exchange block, determines how quickly data can be moved between it and other tiles. Also, the highest-numbered tiles on an IPU are linked back directly to the lowest-number tiles in a ring-type topology. The combination of these two factors is what generates the typically triangular and curved shapes seen in these exchange graphs.

StreamCopy

This process copies data between tensors and streams, allowing data to be transferred between the IPUs and the host machine over PCI. The execution trace shows these program steps as three separate phases, StreamCopy-Begin, the Copy itself (StreamCopyMid), and StreamCopyEnd.

In addition to the common information listed above, the following is displayed:

- **Total Data** - the total amount of data transferred during the exchange,
- **Data Balance** - the mean amount of data exchanged divided by the maximum amount of data exchanged,
- **Copies from host** - how many copy instructions transferred data from the host machine,
- **Copies to host** - how many copy instructions transferred data to the host machine.

2.11.3 Change Layout

When comparing the Execution Trace of two reports a button in the top right hand corner allows you to arrange the Execution Traces side by side or one on top of the other.

2.12 Application preferences

To display the Preferences dialog, select 'Preferences' from the menu, or press the Ctrl / Cmd + , keys. As well as the settings displayed, the view options for the various reports are also saved.

You can reset your preferences at any time by selecting 'Reset Preferences' from the Help menu.

2.12.1 Setting the colour theme

The PopVision Graph Analyser supports light and dark colour themes, and you can select a preference here. There are three options:

- **Auto** - this is the default setting, and allows the application to follow your machine's system-wide theme setting for light or dark mode. If the PopVision Graph Analyser application detects a change in your operating system theme, it automatically switches to the corresponding mode in application.
- **Light** - this forces the PopVision Graph Analyser application into light mode, irrespective of your machine's theme settings.
- **Dark** - this forces the PopVision Graph Analyser application into dark mode, irrespective of your machine's theme settings.

2.12.2 SSH preferences

You can store your SSH preferences in the Preferences dialog to allow authorisation when opening reports on remote machines. There are two settings you can enter here:

- **SSH private key path** - enter the file path of your your machine's private SSH key here. This filepath will be used to authenticate you on remote machines during the connection process. The default path is `<home>/.ssh/id_rsa/`, where `<home>` denotes your home directory in your operating system.
- **SSH agent mode** - this dropdown-list allows you to choose whether you want to specify an ssh-agent socket path, and, if so, how you want to do so:
  - **Disabled** - do not use an ssh-agent socket (the default)
  - **Manually specify** - enter file path to the ssh-agent socket in the field that appears below this option.
  - **Automatically obtain from environment** - obtain the ssh-agent path from an environment variable.

2.12.3 Menu close delay

Drop-down menus allow you to select one (in the case of radio buttons) or several (in the case of checkboxes) values from a list which appears when you click a menubar button. This preferences defines how long (in milliseconds) you want the menu to be visible after you move your mouse away from it.

2.12.4 Help links

By default, 'Help links' are enabled for the application. These links appear in tooltips when you hover over various parts of the reports, including buttons, fields and tabs. Hovering your mouse over them for a second presents you with a tooltip, which contains a link to the section of the help documentation that corresponds to that part of the report. Click on the link to open the documentation at that place.

You can disable the help links, and prevent them appearing, by switching off this option here, in the Preferences.

2.12.5 Quit after last window is closed

This preferences control whether the Mac version of the application quits the program after the last window is closed.

2.12.6 Experimental features

Each version of the PopVision Graph Analyser contains some experimental features that are hidden by default. These features are not fully release-capable, and will have limited support and may change or be removed in future. You can enable them here, by toggling the button next to this option.

2.12.7 Graph stats

You can display (or hide) statistics for the Memory and Liveness reports. They appear in the top right-hand corner of the graph and show the Average, Minimum, Maximum and Standard Deviation of the memory usage across the selected tiles, for each data set plotted.

You can move this statistics box anywhere in your graph by dragging its title bar at the top.

2.12.8 Stack graph values

- When this option is turned on, the values shown in the tooltips on the memory graph are displayed 'stacked'.
- When turned off, the individual values are shown without stacking.

When this option is turned on, "(stacked)" is displayed in the tooltip.

2.13 FAQs

This section contains a set of frequently asked questions about capturing and understanding reports in the PopVision Graph Analyser.

2.13.1 Not Always Live memory discrepancy

Question: Why does the tile memory differ on the Memory and Liveness reports? If you open a Memory report, and select the 'Liveness' graph type from the drop-down menu, and then select a particular tile, you can see its memory consumption plotted. If you then find that same tile in the Liveness report, you may notice that its memory consumption is lower. Why does this happen?

Answer: The 'Not Always Live' plot on the Memory-Liveness report actually shows the maximum memory of the not-always-live variables, which can be lower than the actual tile memory required. Because memory is statically allocated on the tile, and the allocating algorithm isn't perfect, this could be less than the actual amount of memory required to store your program.

As an example, suppose you have two variables A and B, both 1 byte, but B needs to be stored in interleaved memory. If you have a program like this:

```
Write(A)
Read(A)
Write(B)
Read(B)
```

then the two variables are not live at the same time, so in theory could be overlapped, but because of the additional constraints they aren't. In this case the maximum not-always-live bytes is 1 byte, but they memory required (excluding gaps) is 2 bytes.

2.14 Release notes

For the latest release notes, please refer to Graphcore's Software Download site, where you can see what changes have occurred for each update of this software.

To see what's changed in the PopVision Graph Analyser application, select 'Release Notes' from the Help menu, or click the "What's new since the last release" link on the landing page.

2.15 Licensing information

Licensing information about the PopVision Graph Analyser is available to read by selecting 'License' from the Help menu. It contains an end-user agreement, copyright and trademark information, and license information about third-party software used in the application.

This information can also be found in the Installation README file, which you can find on the Graphcore Support site.

CHAPTER

THREE

SYSTEM ANALYSER

Version: 1.3.6

3.1 Overview

The Graphcore PopVision System Analyser is a desktop tool for analysing the execution of IPU-targeted software on your host system processors. It shows an interactive timeline visualisation of the execution steps involved, helping you to identify any bottlenecks between the CPUs and IPUs. This is particularly useful when you are scaling models to run on multiple CPUs and IPUs.

Used in combination with the PopVision Graph Analyser application, the System Analyser allows you to identify exactly how long execution events take to run, from the main program itself all the way down to individual IPU execution steps.

Poplar - and the machine learning frameworks it supports such as PopART, TensorFlow and PyTorch - use the Poplar `libpvti` library to capture profiling information from your code. This information is saved to a file which you can then open and analyse in the System Analyser application. User APIs in C++ and Python are available to instrument your own application code.

The System Analyser app requires Poplar SDK 1.4 or later.

3.2 Capturing execution information

To capture execution data from your program to a file, use the following options when executing your program. These are specified at the same time as the `POPLAR_ENGINE_OPTIONS`:

```
PVTI_OPTIONS='{"enable":"true"}'
```

Additional options include:

- `directory` - to specify where the `.pvti` file will be saved,
- `channels` - to specify a list of channels to capture from, as described Using the libpvti API, below.

`.pvti` files are streamed to disk as the program executes, so it should not have a significant effect on your host system's memory (although it may affect the speed of execution).

The `pvti.hpp` file, in the `libpvti` library in Poplar, gives more details.

3.2.1 Capturing function entry and exit

For most basic cases, there are `POPLAR_TRACEPOINT` macros that can be used inside your program to capture the timings of function entry and exit. For example:

```
void Engine::prepareForStreamAccess(StreamAndIndex stream) const {
  POPLAR_TRACEPOINT();
  const DataStream &streamInfo = getStreamInfo(stream.id);
  logging::engine::debug("prepareForStreamAccess {} \\"{}\\" (id {}, index {}))",
                         isHostToDevice(streamInfo.type) ? "host write of",
                                                         : "host read of",
                         streamInfo.handle, stream.id, stream.index);
}
```

This will capture the name of the method, and, using object construction and destruction, record the entry and exit time.

Similar macros are available for PopART (`POPART_TRACEPOINT`) and TensorFlow (`TENSORFLOW_TRACEPOINT`).

3.2.2 Using the libpvti API

The API allows you to 'mark' the begin and end of a 'section of code'. This is not limited to functions - markers can be placed anywhere in your code. More information on the instrumentation API can be found in the Poplar `libpvti.hpp` header file documentation & README.

To use the API to indicate the beginning and end of trace events, you must first create a channel, as the example below shows. In C++:

```
// Create channel
pvti::TraceChannel channel = {"MyChannel"};

// Functional implementation
void foo() {
  pvti::TracePoint::begin(channel, "foo");
  ...
  pvti::TracePoint::end(channel, "foo");
}

// Scoped tracepoint object
void bar() {
  pvti::Tracepoint tp(channel, "bar");
  ...
}
```

And in Python:

```
import libpvti as pvti
channel = pvti.createTraceChannel("MyChannel")

# Functional implementation
def foo ():
  pvti.Tracepoint.begin(channel, "foo")
  ...
  pvti.Tracepoint.end(channel, "foo")

# context manager implementation
def bar():
  with pvti.Tracepoint(channel, "bar"):
    ...

# wrapped object
class Bob:
  def somemethod(self):
    ...

bob = Bob()
pvti.instrument(bob ["somemethod"], channel)
```

(continues on next page)

(continued from previous page)

```
# decorator (later)
@pvti_instrument()
def cat():
  ...
```

3.2.3 Capturing scalar values

The API also allows you to capture scalar values over time. This can be used to log any scalar value, potential uses could be to capture host memory usage or CPU load. More information on the instrumentation API can be found in the Poplar `libpvti.hpp` header file documentation & README.

To use the API to capture scalar values, you must first create a graph (with units) and then a series against that graph, as the example below shows. In C++:

```
// Create graph
pvti::Graph graph("MyGraph", "%");

// Create series
auto series1 = graph.addSeries("series1");
auto series2 = graph.addSeries("series2");

// Capture values
series1.add(0.1);
series2.add(0.3);
```

And in Python:

```
import libpvti as pvti
# Create graph
graph = pvti.Graph("MyGraph", "%")

# Create series
series1 = graph.addSeries("series1")
series2 = graph.addSeries("series2")

# Capture values
series1.add(0.1)
series2.add(0.3)
```

You can capture default driver monitoring information by setting the environment variable `GCDA_MONITOR=1`.

3.3 Opening reports

After starting the System Analyser, you're presented with a 'landing page' from which you can open reports and view various topics within this online help. You can open report files on your local machine, or from a remote server over SSH.

3.3.1 Local reports

You can open report files stored on your local machine as described below.

To open a local report on your machine:

- Click on the **Open Report** link. You'll be presented with a file selection dialog, and the 'local' tab at the top will be selected by default. You'll see listings of the directories and files on your local machine.
- You can sort these files by name, modified date or size, in ascending or descending order, by clicking on the appropriate column header.

- The System Analyser application can open `.pvti` files and `.json` files that support the Chromium trace format. When the PopVision System Analyser identifies a directory in which any `.pvti` files are found, those files are listed on the right-hand side.
- After navigating to the desired directory, you can select a file by clicking on it. Multiple files can be selected at once.
- Once you've selected the file(s) you wish to open, click on the 'Open' button to load the report data from the file(s).

If you've previously opened a report, it will appear in the **Recent** list of report files. Click on one to open it again.

While the report file(s) load into the application, you'll see a 'Loading' progress bar, then the main view is displayed, as detailed below.

3.3.2 Remote reports

If you are using an IPU system on a remote server, for example on a cloud service, any reports generated will be saved to that server, so you cannot open them 'locally'. You can, however, open them remotely by specifying the server address, and connecting to the machine over SSH. The reports are analysed and the output is streamed back to the PopVision System Analyser application on your local machine, allowing you to view the reports.

When the PopVision System Analyser opens report files on a remote machine, it downloads a small binary app to it which pre-processes the report data and sends it back over SSH to the PopVision System Analyser application running on your local machine. If you're running other performance-critical processes on that remote machine, you should be aware of any effects this process may have on the capacity of the remote machine's hardware to run any other tasks. As server performance varies a great deal, the only way to know how much processor speed it takes is to try a small sample, and monitor the CPU usage.

To open a remote report on another machine:

- Click on the **Open Report** link. You'll be presented with a file selection dialog, and the 'local' tab at the top will be selected by default.
- Click on the 'remote' tab at the top, and you'll see a login dialog that allows you to connect to a remote server. Enter your username, and the address of the remote machine.
- If you just want to log in with a password for the remote machine, enter it in the Password field.
- Alternatively, you can use your local machine's SSH key to authorise your connection. Enter its file path in the Preferences dialog.
- Once logged in, you'll see listings of the directories and files on the remote machine. You can sort these files by name, modified date or size, in ascending or descending order, by clicking on the appropriate column header.
- The System Analyser application can open `.pvti` files and `.json` files that support the Chromium trace format. Multiple files can be opened together from this dialog by clicking to select all the files you want to open. You'll notice that when the PopVision System Analyser identifies a directory in which any `.pvti` files are found, those files are listed on the right-hand side.
- Once you've selected the file(s) you wish to open, click on the 'Open' button to load the report data from the file(s).

If you've previously opened a report, it will appear in the **Recent** list of report files. Click on one to open it again.

While the report file(s) load into the application, you'll see a 'Loading' progress bar, then the main view is displayed, as detailed below.

The System Analyser does not currently support encrypted SSH private keys, i.e. keys that are protected by a passphrase. However it does support SSH agents. If your key is passphrase protected you will need to make sure to add it to your SSH agent before the PopVision System Analyser can use it, by using the ssh-add command-line tool and ensure 'SSH Agent mode' is set correctly in the Preferences.

To configure SSH Agent, from a terminal you can run the following.

```
# Start the ssh-agent in the background.
eval "$(ssh-agent -s)"

# Add your SSH private key to the ssh-agent
ssh-add -K ~/.ssh/id_rsa
```

Then edit your Preferences to remove your SSH private key path. Make sure that SSH agent mode is set to "Automatically obtain ssh-agent socket path from environment".

3.4 Viewing reports

The application's main view displays the Timeline information of the execution events recorded in the files you opened, together with a scaled-down overview above, which shows the entire set of events irrespective of your current zoom and pan state.

In the main window, the following actions are available:

- Pan and zoom in and out of the timeline, viewing every event in the execution of the program.
- Select individual events to view their details, and view the durations of selected sections of the timeline.
- Save report images to your machine.

3.4.1 Using the sidebar buttons

Once a report has been opened, the application sidebar is displayed, which contains several buttons that allow you to perform the following actions:

- Reload the report (this option is also available from the application's **View** menu, or by pressing Control/Command and R).
- Close the current report(s),
- Open this documentation window,
- Expand or contract the sidebar button labels.

3.4.2 Timeline flamegraphs

Events in the timeline are grouped and layered according to the file where they originated, and beneath that to the process and thread in which they occurred. If there's room to display it, the event's name is shown within each event block. Hovering your mouse over an event block displays a pop-up that shows the details of the event, as described below.

The number of events within the currently displayed portion of the timeline is displayed in the top left-hand corner, as well as the duration of that portion. You'll see these numbers change as you pan and zoom around the timeline.

The time scale is displayed across the top of the timeline, showing elapsed time from the start of the first event. This is displayed in hours and minutes and seconds, and more significant digits of the seconds are displayed as you zoom in. You can choose to display the time scale in relative terms (starting at `0:00` at the beginning of the first captured event), or in absolute terms, where a real time is displayed. Choose the display you want by clicking the **Options** button at the top of the application window.

Once you've opened a report, you can open additional reports to display on the same timeline. Just click the **Add file** button at the top-left of the application window, and add files as above.

Events in the timeline are coloured as follows:

- **Poplar** - events triggered from the Poplar libraries are coloured red.

- **Framework** - events triggered from the PopART libraries, or any of the machine learning frameworks such as TensorFlow, are coloured orange.
- **Driver** - events triggered by the driver layer are coloured blue.
- **Others** - user-generated event categories are then assigned a different colour.

3.4.3 Timeline line graphs

If any scalar values have been captured, the timeline will also show a collection of line graphs grouped at the bottom of a file. These can be used to view these scalar values over time, and compare to the events shown in the flamegraphs above.

Hovering over a line graph will display a tooltip giving each separate series' value at that timestamp.

3.4.4 Timeline options

Above the timeline overview there are the following buttons:

- **Add file** - this reopens the file browsing dialog, allowing you to add more files to the current timeline as above.
- **Graph Type** - this allows you to switch between graph display types:
  - **Timeline** - displays a flamegraph showing the position and duration of each event in the call stack. Each event on the timeline view corresponds directly to an event logged in the PVTI file.
  - **Aggregated** - displays a flamegraph showing each event aggregated by its unique call stack. The starting timestamp at which a block appears in this view has no link to the timestamp that its events took place at within the trace. However, its duration does correspond to the total duration of the events that combine to make the block.
- **Options** - this shows the following options to apply to your timeline:
  - **Absolute timing** - checking this option will position files on your timeline using their absolute times rather than positioning all files to start at time 0.
  - **Collapse all charts** - checking this option will collapse all charts in your timeline, it will leave parent nodes in their current state. Unchecking this will similarly expand all charts in your timeline.
  - **Hide charts below threshold duration** - checking this option will hide all charts in your timeline with a total duration (sum of all top-level event durations) less than or equal to the threshold percentage of the total time-window of the file. You can specify the threshold percentage using the radio buttons below the option.
- **Key** - this allows you to see which channels the different colours represent on your timeline. Clicking one of these will reload the timeline without that channel shown.
- **Save (camera icon)** - this allows you to save the current timeline as a png. This can either save to a location on your machine or to your clipboard.

3.4.5 Panning and zooming

You can move around the timeline using the mouse to bring events into view:

- In the overview at the top of the timeline, click and drag a section to zoom into the corresponding are of the timeline. The section of the timeline you're currently viewing is highlighted in the overview.
- Drag the mouse left and right in the timeline to shift it left and right at the current zoom level.
- Use the mouse wheel to zoom in and out of the timeline. If the timeline is too deep to fit into the application window, a scrollbar is displayed to enable you to move the timeline up and down in the window. As the

mouse wheel is used for zooming in and out, you can hold down the Control key to scroll the timeline up and down using the mouse wheel.

You can switch between window and full-screen display by selecting **Toggle Full screen** from the **View** menu, or by pressing Control/Command and F.

3.4.6 Selecting events

You can select any individual event in the timeline by clicking on it. Tabs giving further event information are then displayed beneath the timeline.

- **Details:**
  - **Name** - for events dispatched through one of the libraries (Poplar, PopART, TensorFlow, etc.) this is a concatenated list of the namespaces from which the event originated. For user-created functions (for example, in your Python programs), it is the name of the function.
  - **Channel** - the channel is where this event has been called. This can be one of the predefined channels (Drivers, Poplar, Framework) or user-created.
  - **Timestamp** - this is the execution time at which the event occurred in the timeline, measured in hours, minutes and microseconds.
  - **Absolute Timestamp** - same as the timestamp but in absolute terms rather than relative to the start of the event.
  - **Duration** - the amount of time the event took to execute.
- **Call Tree:**

  The call tree shows the events that are descendants of the selected event in a tree structure. You can expand to see the children of an event by clicking the caret on the left. Percentages given in the call tree are a percentage of the total time of the selected event. Hovering on a node in the call tree will highlight the respective events in the timeline above.
  - **Self Time** - this is the total time of an event minus the time taken by its children.
  - **Total Time** - this is the total time that the event took.
  - **Activity** - this is the name of the event.

  The call tree also has the following options that apply to it:
  - **Aggregated** - when checked, this option will aggregate all nodes with the same event name at the same level. Summing their times and combining their children all under one node.

3.4.7 Expanding and collapsing regions

You will notice that your timeline has a tree-like structure of nodes on the left hand side. Each node can be individually expanded and collapsed by clicking the caret on the left of the node label. You will also notice nodes with an ellipsis button. This gives an options menu with the following options:

- **Collapse all other charts** - this will expand the node you have selected and then collapse all other charts on the timeline, allowing you to focus in on this one node's data.

Collapsed chart nodes will still show the top level events in that node against the collapsed line. This can be a useful way to gain an understanding of what is happening across multiple threads.

3.4.8 Viewing selected duration

You can view a duration on the timeline by holding down the Shift key and dragging the mouse from side to side anywhere in the timeline. This displays a timing duration marker at the top of the timeline, showing you the duration of the timeline you've selected. The selected duration is highlighted in pink in the overview.

3.4.9 Saving reports

To save the currently displayed portion of the timeline as a PNG image file:

1. Click on the **Save** button in the top, right-hand corner of the main screen.
2. Your system's file browser dialog appears. Select the directory in which you want the image file saved.
3. Click **Save**.

3.5 Application preferences

3.5.1 Colour theme

The PopVision System Analyser supports light and dark colour themes, and you can select a preference here. There are three options:

- Auto - this is the default setting, and allows the application to follow your machine's system-wide theme setting for light or dark mode.
- Light - this forces the application into light mode, irrespective of your machine's theme settings.
- Dark - this forces the application into dark mode, irrespective of your machine's theme settings.

3.5.2 SSH preferences

You can store your SSH preferences in the Preferences dialog to allow authorisation when opening reports on remote machines.

- **SSH private key path** - enter the file path of your your machine's private SSH key here. This filepath will be used to authenticate you on remote machines during the connection process. The default path is ~/.ssh/id_rsa/.
- **SSH agent mode** - this dropdown-list allows you to choose whether you want to specify an ssh-agent socket path, and, if so, how you want to do so:
  - **Disabled** - do not use an ssh-agent socket.
  - **Manually specify** - enter file path to the ssh-agent socket in the field that appears below this option.
  - **Automatically obtain from environment** - obtain the ssh-agent path from an environment variable.

3.5.3 Experimental features

Each version of the PopVision System Analyser contains some experimental features that are hidden by default. These features are not fully release-capable, and will have limited support and may change or be removed in future. You can enable them here, by toggling the button next to this option.

3.6 About System Analyser

To see the details of the System Analyser application, select **About PopVision System Analyser** from the application's main menu. A dialog window appears showing:

- **Version** - the version number of the application.
- **Commit** - the unique commit hash of this release version.
- **Date** - the data and time this version was released.
- **Component version numbers** - the version numbers of the main software components used by the application, including Electron, Node, Chrome and the V8 engine.

CHAPTER

FOUR

TRADEMARKS & COPYRIGHT