

Getting Started with EASYPAP



The EasyPAP Team

March 18, 2025

EASYPAP aims at providing students with an easy-to-use programming environment to learn parallel programming. The idea is to parallelize sequential computations on 2D grids (which are images, most of the time) or 3D meshes over multicore and GPU platforms. At each iteration, the current data can be visualized, allowing to visually check the correctness of the computation method. Multiple variants can easily be developed (e.g. sequential, OpenMP, MPI, OpenCL, CUDA, etc.) and compared.

Most of the parameters can be specified as command line arguments, which facilitates automation of experiments through scripts:

- size of input data, image/mesh file to be loaded;
- kernel to use (e.g. blur, pixelize, invert, ...);
- variant to use (e.g. seq, omp, omp_task, pthread, mpi, ocl, ...);
- code optimization to use (e.g. default, AVX, MIPP, ...);
- interactive mode / performance mode;
- monitoring / tracing mode;
- and much more!

The goal of this document is to provide a *Quick Start Guide* to the EASYPAP user, using a step-by-step discovery of the main features and most useful options of the EASYPAP utilities.

Contents

1	Before you start	4
1.1	Compiling EASYPAP	4
1.2	Enabling bash completion (strongly recommended)	4
2	Running EASYPAP	5
2.1	Our first kernel	5
2.2	Changing the size of images	6
2.3	Implementing multiple variants	7
2.4	Interactive mode	8
2.5	Setting the refresh rate	8
2.6	Performance mode	8
2.7	Loading images	9
2.8	Drawing images	9
2.9	Switching between two images	10
2.10	Tiling	11
2.10.1	Tile parametrization	11
2.10.2	Tile variants	12
3	Monitoring	13
3.1	Real-time monitoring	13
3.2	Post-mortem trace analysis	15
3.2.1	Trace generation	15
3.2.2	Visualizing traces	15
3.2.3	Thumbnails generation	17
3.2.4	Trace comparison	18
3.2.5	Comparison between different granularities	19
4	Distributed Computing with MPI	19
4.1	A simple example	20
4.2	Running MPI variants	20
4.3	Debug mode	21
4.4	Traces	22
5	Exploiting GPU accelerators	22
5.1	OpenCL	22
5.2	Checking the OpenCL configuration	22
5.2.1	Writing and executing 2D kernels	23
5.3	NVIDIA Compute Unified Device architecture	23
6	Advanced topics	23
6.1	Initialization hooks	23
6.1.1	Initialization	24
6.1.2	First-touch data allocation	24
6.1.3	Drawing hook	25

6.2	Using your own data structures	25
7	Plotting performance graphs with EASYPLOT	27
7.1	Introduction	27
7.2	Production of experimental data	27
7.3	Production of graphs	27
7.3.1	Data selection	27
7.3.2	Data presentation	28
7.3.3	About speedup computations	31
7.3.4	Cosmetic options	31
7.3.5	Plotting multiple attributes	31
8	Installing EASYPAP	31
8.1	Prerequisites	31
8.2	Required packages	31
8.2.1	SDL2	31
8.2.2	Hwloc	33
8.2.3	Lex	33
8.2.4	Cglm	33
8.3	Optional packages	33
8.3.1	FxT (recommended)	33
8.3.2	Scotch (recommended)	34
8.3.3	OpenCL	34
8.3.4	CUDA (Linux only)	34
8.3.5	MPI	34
8.3.6	OpenSSL	34
8.3.7	MIPP	34
8.3.8	PAPI (Linux only)	35
8.4	Troubleshooting	35
8.5	Customizing EASYPAP	35
Index of options		37

1 Before you start

This section describes how to compile the EASYPAP programming environment. For instructions about installing EASYPAP and the packages it depends on, please refer to Section 8. In particular, Section 8.5 shows how to customize the `Makefile` by enabling/disabling various functionalities.

1.1 Compiling EASYPAP

Go into your EASYPAP main directory (e.g. `$(HOME)/Devel/easypap`) and simply type:

```
./script/build-all -j
```

This will compile the `ezv` 2D/3D rendering library, the `ezm` monitoring library (including the trace visualization utility) (see Section 3.2) and finally the EASYPAP programming environment itself. It is equivalent to running:

```
make -C lib/ezv -j
make -C lib/ezm -j
make -C lib/ezm/view -j
make -C . -j
```

Note that all arguments will be passed to the `make` command. For instance, to clean all previously compiled files before recompiling, you can run:

```
./script/build-all clean
```

1.2 Enabling bash completion (strongly recommended)

If you are using the Bourne Again Shell (`bash`), you will probably want to enable automatic bash completion to save time when typing commands. To do so, just source the following file in your terminal:

```
. script/easypap-completion.bash
```

You should now be able to trigger auto-completion of the EASYPAP run script arguments by pressing the `→` key. For instance:

```
[my-machine] ./run --kernel → →
mandel none spin
```



Tip

To avoid repeating this process each time you open a new terminal, you may add the following lines to your `$(HOME)/.bashrc` file:

```
EASYPAPDIR=$(HOME)/Devel/easypap
. ${EASYPAPDIR}/script/easypap-completion.bash
```

2 Running EASYPAP

To check if EASYPAP was correctly installed and built, you can invoke the “run” script without passing any argument:

```
./run
```

A window should pop up, displaying a black uniform picture. This is OK: by default, EASYPAP allocates an image of size 1024×1024 where each pixel has the black color (i.e. `(unsigned) 0`). We talk about colors in more details later. Just press `esc` to quit.

2.1 Our first kernel

In EASYPAP, functions performing computations on images are called *kernels*. EASYPAP comes with a set of predefined kernels, and new kernels can easily be added to the pool. CPU implementations of kernels are stored in the `kernel/c` subdirectory.

Open the `spin.c` source file and observe the implementation of the `spin_compute_seq` function. The code is showed in Figure 1. Its name reveals that it implements the “seq” *variant* of the “spin” *kernel*.

```
1 // Simple sequential version (seq)
2 // Suggested cmdline: ./run --size 1024 --kernel spin --variant seq --debug u
3 //
4 unsigned spin_compute_seq (unsigned nb_iter)
5 {
6     for (unsigned it = 1; it <= nb_iter; it++) {
7
8         for (int i = 0; i < DIM; i++)
9             for (int j = 0; j < DIM; j++)
10                cur_img (i, j) = compute_color (i, j);
11
12        rotate (); // Slightly increase the base angle
13    }
14
15    return 0;
16 }
```

Figure 1: Sequential version of kernel spin (from `kernel/c/spin.c`)

The outer loop (line 6) performs `nb_iter` iterations in a row. In interactive mode¹, this variable is assigned the value 1 by default, so that the screen is refreshed after each iteration. In performance mode², no display is involved and the `nb_iter` variable is set to the total number of iterations requested by the user.

Lines 8–10 illustrate how the contents of the image are accessed during an iteration. For the sake of simplicity, images are squares shape of size $DIM \times DIM$. The pixels of the image can be accessed through the `cur_img (row, column)` macro.

¹Interactive mode is further explored in Section 2.4, page 8

²Performance mode is further discussed in Section 2.6, page 8

In this example, the `compute_color` function computes the color of each pixel using its polar coordinate in the image. To execute the variant illustrated in Figure 1, simply run:

```
./run --kernel spin --variant seq
```

Or use the abbreviated³ version:

```
./run -k spin -v seq
```

If no variant is specified, the `seq` one is used by default. So we could even just run:

```
./run -k spin
```

Figure 2 illustrates the output of EASYPAP when running the `spin` kernel. Note that both the kernel name and the variant name appear in the window's title bar.

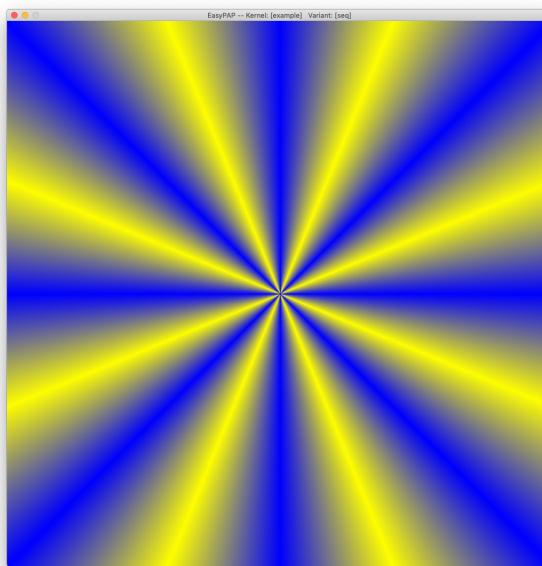


Figure 2: Snapshot captured during the execution of the `spin` kernel.

2.2 Changing the size of images

By default, when no specific image size is specified and no image is loaded (see 2.7), images are defined as matrices of 1024×1024 pixels (i.e. `DIM = 1024`).

Dimensions of images can be changed using the `--size` option. For instance, here is how to run the `spin` kernel over an image of size 2048×2048 :

```
./run --size 2048 --kernel spin
```

³The comprehensive list of options can be obtained by running `./run --help`



Warning

Even though the image is now much bigger, it is resized when displayed on the screen to fit the dimensions of the graphical window. As a consequence, it will not necessary make any difference on the screen. However, the computation time will surely be significantly different!

2.3 Implementing multiple variants

Multiple variants of a given kernel are usually coded in the same file. As a follow-up of our previous example, `kernel/c/spin.c` contains a second variant of our `spin` kernel. It is a straightforward OpenMP version designed as an incremental evolution of the sequential variant: a single `#pragma omp parallel for` clause was inserted before the `for` loop iterating over lines (see Figure 3, line 9). This variant is named “omp”, so the full function name is `spin_compute_omp`.

```

1 ////////////////////////////////////////////////////////////////// Simple OpenMP version (omp)
2 // Suggested cmdline:
3 // OMP_NUM_THREADS=4 ./run --kernel spin --variant omp
4 //
5 unsigned spin_compute_omp (unsigned nb_iter)
6 {
7     for (unsigned it = 1; it <= nb_iter; it++) {
8
9     #pragma omp parallel for
10    for (int i = 0; i < DIM; i++)
11        for (int j = 0; j < DIM; j++)
12            cur_img (i, j) = compute_color (i, j);
13
14    rotate (); // Slightly increase the base angle
15 }
16
17 return 0;
18 }
```

Figure 3: OpenMP parallel version of kernel spin

Let us try this new variant:

```
./run --kernel spin --variant omp
```

Did you feel it ran faster than the `seq` one? In the general case, it may not be obvious to observe a difference, because the graphical display adds a significant overhead to the kernel computation. This overhead comes from the transfer of all pixels to the graphical card, from the screen vertical sync, etc. In some situations, this overhead may be larger than the duration of one single iteration.

In Section 2.5, we discuss how to decrease this overhead. In Section 2.6, we detail how to get entirely rid of this overhead and achieve accurate performance measurements.

Key	Action
Space	enter/leave the <i>pause</i> state
<input type="button" value="s"/>	enter <i>pause</i> /perform one iteration step
<input button"="" type="button" value="p"/>	zoom in
<input button"="" type="button" value="m"/>	zoom out
mouse drag	scroll image/rotate mesh
<input type="button" value="r"/>	reset view
<input button"="" type="button" value="↓"/>	change the refresh rate (detailed in Section 2.5)
<input type="button" value="i"/>	toggle display of iteration number On/Off
<input type="button" value="h"/>	toggle “heat map” mode (only in monitoring mode, see Section 3.1)
<input type="button" value="esc"/> or <input type="button" value="q"/>	quit EASYPAP

Table 1: Interacting with the EASYPAP main window.

2.4 Interactive mode

By default, EASYPAP runs in interactive mode and updates the main graphical window to reflect the contents of the current image after each iteration. In this mode, the user can interact with EASYPAP through the keyboard and mouse to trigger various actions such as pausing the application, displaying the current iteration number, or exiting the program. All available controls are listed in Table 1, page 8.

2.5 Setting the refresh rate

As previously mentioned, updates the graphical window after each iteration by default. You can use the `--refresh-rate` option to specify the number of iterations between two screen updates, that is, the number of iterations in a row performed by kernel variants. Hence, the following command line will refresh the graphical window every five iterations:

```
./run --kernel spin --variant omp --refresh-rate 5
```

The perceived speed of the computation should be noticeably faster.

Note that the refresh rate can also be interactively adjusted by pressing keys during execution. Beware that the refresh rate increases exponentially in this case, so do not press the

2.6 Performance mode

The interactive mode is useful to visually check if a given kernel behaves correctly. When it comes to benchmarking and comparing multiple variants however, we need a way to completely eliminate the overhead of graphical refresh. This is precisely what the `--no-display` (or `-n`) option is intended for: EASYPAP runs silently and reports the overall wall clock time after completion of the requested number of iterations. The number of iterations can be specified with the `--iterations` (or `-i`) option.

Here is how to perform 100 iterations within the spin kernel:

```
[my-machine] ./run -k spin -v seq --no-display --iterations 100
Using kernel [spin], variant [seq]
Computation completed after 100 iterations
3087.406
```

The last line of output displays the completion time in milliseconds⁴.
Let us see what happens with the omp variant:

```
[my-machine] ./run -k spin -v omp --no-display --iterations 100
Using kernel [spin], variant [omp]
Computation completed after 100 iterations
402.701
```

In this case, the OpenMP variant achieves a speedup of $\frac{3087}{403} \approx 7.68$.

2.7 Loading images

The spin kernel belongs to a family of kernels which have no input data: they generate an output image out of nothing. The mandel kernel, which draws the **Mandelbrot set**, is another member of the same family (see `kernels/c/mandel.c`).

Other kernels, such as invert, expect an image as an input. One simple way to provide a kernel with an image is to load it from a file, using the `--load-image` (or `-l`) option:

```
./run --load-image data/img/shibuya.png -k invert -i 1
```

Notice that we perform only one iteration to avoid the inconvenience of screen blinking, which would inevitably occur with the invert kernel that continuously switches from positive to negative images...

Another way of avoiding screen blinking is to force the program to *pause* between iterations:

```
./run -l data/img/shibuya.png -k invert --pause
```

Simply press **Space** to step over the next iteration. Pauses always take place between iterations. Note that any execution (i.e. even when the `--pause` flag was not set) can be paused or resumed at any time by pressing **Space**.

2.8 Drawing images

Another way to start with an existing image is to draw it using a preamble function. Before executing a kernel, EASYPAP checks if a “draw” function has been defined. If so, it is called before the first iteration. The function must either be named `<kernel>_draw`, or `<kernel>_draw_<variant>`. EASYPAP first looks for the variant-specific one and, if not found, looks for the general one.

To illustrate the use of such a drawing hook, let us add a function in `invert.c` that will draw a pink diagonal line before any kernel starts:

⁴By default, completion time is also reported in the `data/perf/data.csv` file, together with all the run parameters.

```

1 // If defined, the draw function is called before the computation starts.
2 // This is typically the place where a kernel can modify the pixels of the
3 // original image.
4 void invert_draw (char *param)
5 {
6     // draw a pink diagonal line
7     for (int i = 0; i < DIM; i++)
8         cur_img (i, i) = 0xFF00FFFF;
9 }
```

Note that the draw function is called **after** an image has been potentially loaded from the disk (cf Section 2.7), enabling the draw function to modify the loaded image.

If not `NULL`, `param` contains the argument specified on the command line using the `--arg` option. Please explore the source file of the `life` kernel (i.e. `kernel/c/life.c`) to observe how this parameter can be used to select a specific drawing function among a set of predefined ones:

```
./run --kernel life --arg "random"
```

2.9 Switching between two images

In many simulations (e.g. stencil-based schemes), computations must be synchronous with respect to iterations: data cannot be safely written until all read operations have completed. Using two images (instead of just one) is a popular method to cope with such cases: during iteration 1, the first image is read and the second is modified, then the roles are swapped before iteration 2 starts, and so on... Stencil codes are typically programmed this way (see for instance the `blur` kernel).

Actually, EASYPAP always allocates two images at initialization time, even though many kernels do only use one. The `next_image (row, col)` macro allows to access the second image. The `transpose` kernel is a simple example to illustrate how to use two images and swap them between iterations:

```

1 unsigned transpose_compute_seq (unsigned nb_iter)
2 {
3     for (unsigned it = 1; it <= nb_iter; it++) {
4
5         for (int i = 0; i < DIM; i++)
6             for (int j = 0; j < DIM; j++)
7                 next_img (i, j) = cur_img (j, i);
8
9         swap_images ();
10    }
11
12    return 0;
13 }
```

Note that the `swap_image` is not a costly operation: it only swaps the value of two pointers. When parallelizing such a kernel, make sure the `swap_image ()` call is performed only once per iteration!

For a more general discussion about kernels managing their own data structures, please refer to Section 6.2.

2.10 Tiling

Parallelization of many kernels relies on a tiling approach, where data is divided into rectangular tiles that can be assigned to different computing units.

2.10.1 Tile parametrization

The tiled variant of the spin kernel shows how to divide computations using tiles in a straightforward manner (Figure 4, page 11).

```
1 // Tile computation
2 static void my_tile (int x, int y, int width, int height)
3 {
4     for (int i = y; i < y + height; i++)
5         for (int j = x; j < x + width; j++)
6             cur_img (i, j) = compute_color (i, j);
7 }
8
9 /////////////////////////////// Simple tiled version (seq)
10 // Suggested cmdline:
11 // ./run --kernel spin --variant tiled --tile-width 64 --tile-height 32
12 //
13 unsigned spin_compute_tiled (unsigned nb_iter)
14 {
15     for (unsigned it = 1; it <= nb_iter; it++) {
16
17         for (int y = 0; y < DIM; y += TILE_H)
18             for (int x = 0; x < DIM; x += TILE_W)
19                 my_tile (x, y, TILE_W, TILE_H);
20
21         rotate ();
22     }
23
24     return 0;
25 }
```

Figure 4: Example of a tiled computation

In addition to the `DIM` variable which contains the image dimension, EASYPAP provides a couple of global variables to ease the implementation of tiling:

- `TILE_W` and `TILE_H` store the dimensions (width and height) of tiles. Consequently, each tile contains $TILE_W \times TILE_H$ pixels.
- `NB_TILES_X` and `NB_TILES_Y` indicate the number of tiles along the x and y dimensions. Thus, the image can be seen as grid of $NB_TILES_X \times NB_TILES_Y$ tiles.

Dimensions of tiles can be set on the command line using the `--tile-width` (or `-tw`) and `--tile-height` (or `-th`) options:

```
./run --kernel spin --variant tiled --tile-width 64 --tile-height 32
```

To define square tiles, one can use the `--tile-size` shortcut:

```
./run --kernel spin --variant tiled --tile-size 32
```

If no tile dimension is specified, then tiles are defined by default as squares of 32×32 pixels. If only one dimension is specified (e.g. `width`), then tiles are defined as squares using the given value for each dimension. Variables `NB_TILES_X` and `NB_TILES_Y` are deduced from both image and tiles dimensions. For instance, `NB_TILES_X` is obtained by calculating $\frac{\text{DIM}}{\text{TILE_W}}$.

2.10.2 Tile variants

In many cases, programmers want to experiment with multiple implementations of the tiling function, to explore different optimization strategies, or to enforce explicit vectorization through the use of *intrinsics*⁵.

EASYPAP allows to define multiple *tiling variants* for a given kernel, and provide the `do_tile` generic wrapper to invoke the appropriate variant at execution time. Figure 5 illustrates how multiple tiling variants can be defined for the `spin` kernel. Note that the `do_tile` wrapper accepts an optional fifth parameter to indicate which thread is computing the tile for monitoring purposes (see Section 3). If not provided, `do_tile` assumes that OpenMP is used by default and calls `omp_get_thread_num()` to get the ID of the current thread.

```
1 // Tile computation
2 int spin_do_tile_default (int x, int y, int width, int height);
3 int spin_do_tile_sse (int x, int y, int width, int height);
4 int spin_do_tile_avx (int x, int y, int width, int height);
5
6 unsigned spin_compute_tiled (unsigned nb_iter)
7 {
8     for (unsigned it = 1; it <= nb_iter; it++) {
9
10         for (int y = 0; y < DIM; y += TILE_H)
11             for (int x = 0; x < DIM; x += TILE_W)
12                 do_tile (x, y, TILE_W, TILE_H);
13
14         rotate ();
15     }
16     return 0;
17 }
```

Figure 5: Example of multiple variants for the tiling function (cf `kernel/c/spin.c`)

The *tiling variant* can be specified on the command line using the `--with-tile` option:

```
./run --kernel spin --variant tiled --with-tile avx
```

⁵See for instance the [Intel® Intrinsics Guide](#)



Tip

You can tell EASYPAP your preferred tiling variants (in decreasing order of priority) via the `EASYPAP_TILEPREF` environment variable. For instance:

```
export EASYPAP_TILEPREF="avx:sse:opt"
```

When no tiling variant is specified on the command line, EASYPAP tries to select the most relevant implementation by scanning the `EASYPAP_TILEPREF` variable, and falls back to the default one if no matching is found.

3 Monitoring

EASYPAP comes with powerful tools to help students understanding the behavior of their code, in terms of both correctness and efficiency. These tools allow to visually check parameters such as the number of threads, the tile dimensions or even the scheduling policy.

If you are using EASYPAP's `do_tile` tile wrapper, your code is already instrumented and ready to be monitored. Please go to Subsection 3.1!

In case you are not using the `do_tile` wrapper, you can still enable monitoring in your code by just adding two function calls, respectively at the beginning/end of your tile computation code. Figure 6 shows how to instrument the tiled OpenMP version of the `spin` kernel to enable monitoring.

```
1 // Tile inner computation
2 static void my_specific_tile (int x, int y, int width, int height, int thread_id)
3 {
4     monitoring_start_tile (thread_id);
5
6     for (int i = y; i < y + height; i++)
7         for (int j = x; j < x + width; j++)
8             cur_img (i, j) = compute_color (i, j);
9
10    monitoring_end_tile (x, y, width, height, thread_id);
11 }
```

Figure 6: Note the calls to `monitoring_start_tile` and `monitoring_end_tile`

3.1 Real-time monitoring

Once the code has been instrumented, real-time monitoring can simply be activated using the `--monitoring` option. The following command line shows how to monitor the execution of the `mandel` kernel.

```
./run --kernel mandel --variant omp --monitoring
```

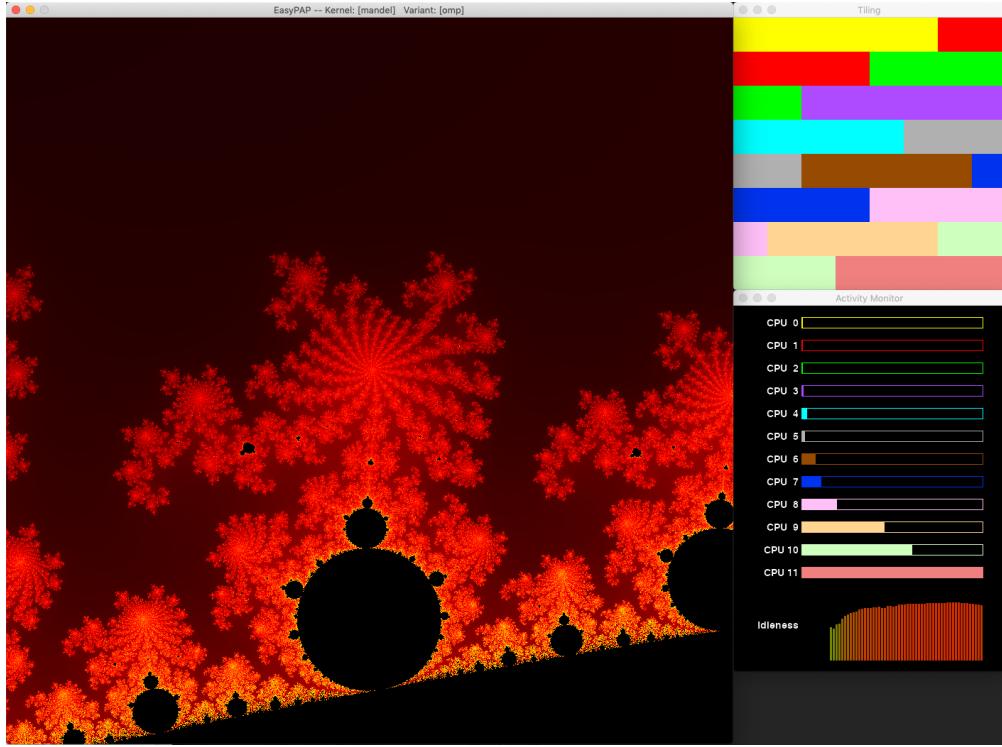


Figure 7: Snapshot of monitored execution of the `mandel` kernel. One can clearly observe that the compute load is unbalanced between CPUs.

Figure 7 (page 14) shows a snapshot taken during the parallel execution of the `mandel` kernel. The monitoring mode introduces two additional windows on the right side of the main one:

Tiling window This window reflects the way tiles have been assigned to threads at each iteration. Each thread is assigned a different color⁶ which is consistent with the color assigned to CPUs in the *Activity Monitor* window. By observing Figure 7, we can see that the image has been divided in eight rows of eight tiles (64 tiles in total), and that these tiles have been assigned to threads in contiguous blocks, in accordance to the *static* loop scheduling policy.

Note that the display mode used by the Tiling Window can be toggled between normal and *heatmap* by pressing `h`. In heatmap mode, the brightness of tiles displayed in the Tiling Window reflects the duration of the corresponding tasks: the brighter an area is, the more time-consuming it is. Figure 8 shows a capture of the Tiling Window in heatmap mode during the execution of a tiled sequential version of `mandel`. We can neatly distinguish the shape of the Mandelbrot set.

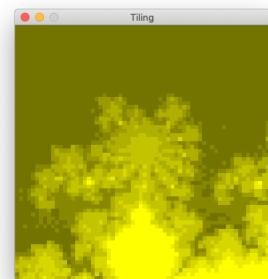


Figure 8: Heatmap mode

⁶This is true to some extent. When the number of threads exceeds the maximum number of predefined colors, the color is computed using `thread_number` modulo `MAX_COLORS`.

Activity Monitor window This window reports the real-time load of each CPU. This load is a percentage representing the amount of time spent in computations over the duration of the iteration. In contrast with system wide perfometers, the activity monitor only reflects the kernel behavior. For instance, the overhead of updating the main graphical window is excluded from the stats. At the bottom of the window, a history diagram shows the evolution of *cumulated idleness* over time. In figure 7, one can clearly observe a load imbalance between CPUs, where CPU9, CPU10 and CPU11 are much busier than the others. The reason comes from the fact that the image is statically divided into contiguous blocks of squared tiles: this is unfortunate because the bottom black area of the image (i.e. pixels belonging to the Mandelbrot set) involves much more computations than the other areas.

As for the main window, the contents of both monitoring windows is refreshed at the end of each iteration.

3.2 Post-mortem trace analysis

Although the monitoring facilities can greatly help to detect and understand some flaws in the execution of kernels, a real-time tool can not always capture some subtle properties such as the heterogeneity of tasks duration, the correct implementation of task dependencies, etc.

When a fine grain analysis is required, EASYPAP can record the events related to the execution of tiles (i.e. start time, end time, tile coordinates, cpu) into a trace file. Section 3.2.1 explains how to trigger trace generation, and Section 3.2.2 shows how to run the trace visualizer.

3.2.1 Trace generation

Use the `--trace` option to enable events recording during the execution of a kernel. To avoid any extra overhead which would spoil the trace, it is strongly recommended to use the tracing option in conjunction with the `--no-display` mode. As a consequence, the number of iterations must be bound using the `--iterations` option.

```
./run --kernel mandel --variant omp --trace --no-display --iterations 10
```

This kernel execution eventually creates a trace file named “`ezv_trace_current.evt`” under the `data/traces` subdirectory. If the file already exists, it is backed up as `ezv_trace_previous.evt`. As a consequence, the default behavior leads to only keep the last two traces in `data/traces`.

```
[my-machine] ls data/traces
ezv_trace_current.evt          ezv_trace_previous.evt
```

To keep more than two trace files simultaneously, rename the `ezv_trace_current.evt` file between runs.

3.2.2 Visualizing traces

To visualize the more recently generated trace, simply run:

```
./view
```

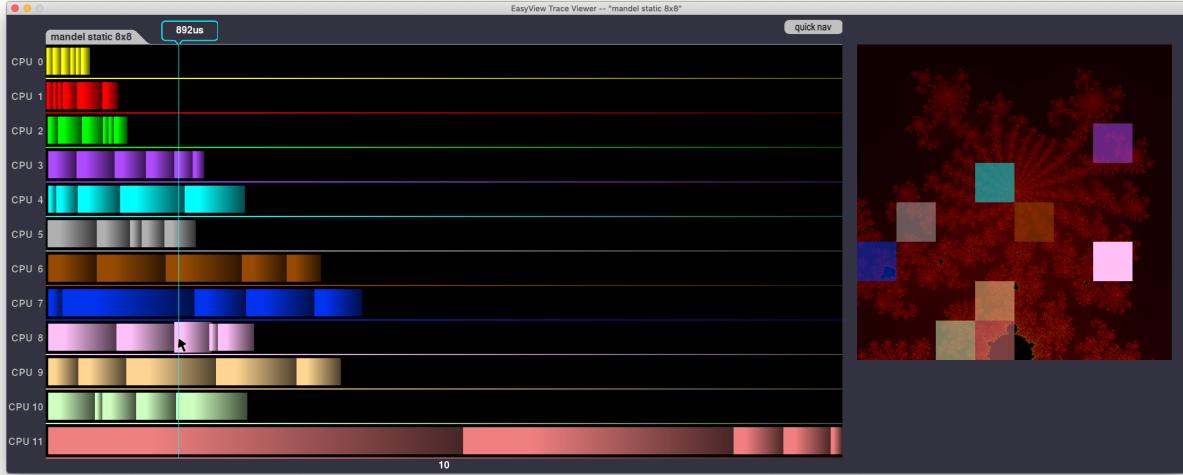


Figure 9: Post-mortem visualization of the 10th iteration of the mandel OpenMP variant using a static scheduling policy. Moving the mouse over a task in the Gantt diagram displays its duration (bubble pop-up at top of window). Tasks intersecting the x-axis of the mouse cursor have their corresponding tile highlighted on the right image thumbnail.

The trace visualizer window is subdivided in two parts.

On the left side, a Gantt chart displays a range of iterations. By default, the first iteration is displayed, but several command line options allow to specify a specific range⁷. Tiles computed by the same CPU have the same color, and are displayed on the same timeline. When moving the mouse over a task, a pop-up bubble displays the task duration (e.g. On Figure 9, the duration of the selected pink task executed by CPU 8 is 892 μ s).

On the right side, the square represents a reduced view of the image. Whenever the x-axis of the mouse intersects tasks in the Gantt chart, the corresponding tiles are highlighted on this reduced image, helping to localize computations. As a consequence, starting on the left side of the Gantt chart and moving smoothly the mouse towards the right side reveals the order in which tiles have been computed.

Interacting through the keyboard/mouse can change the range of tasks displayed in the Gantt chart, as detailed in Table 2 (Page 18).

You can toggle between this vertical mouse mode and an horizontal mode by pressing **x**. In this latter mode, the y-axis of the mouse selects a particular CPU and displays the tiles computed during the current period. This is illustrated in Figure 10 (Page 17).

Finally, pressing **f** enters (or leaves) the *footprint mode*, which is an extension of the horizontal mode where all tiles corresponding to on-screen tasks are displayed. This typically allows to observe the whole tile distribution during a given iteration.

⁷Use `--iteration <i>` to start with a specific iteration, `--range <i> <j>` to start with a range of iterations, or `--whole-trace` to display all recorded iterations.

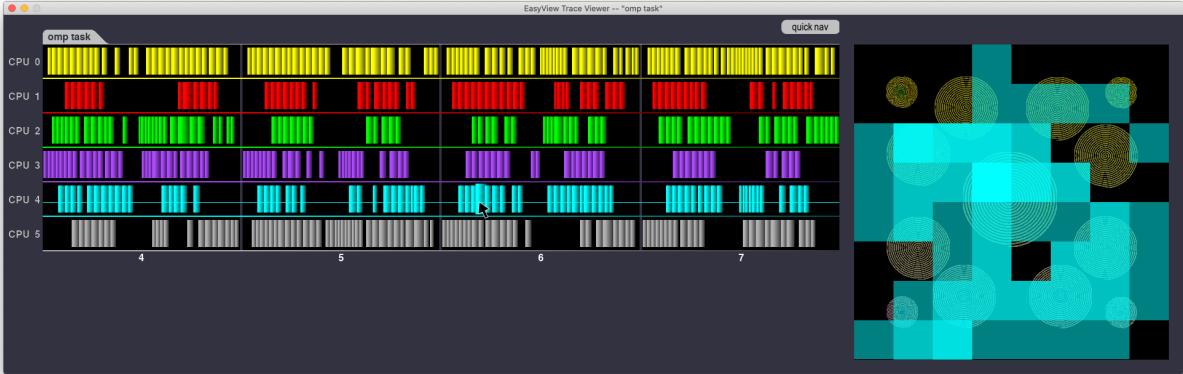


Figure 10: In horizontal mouse mode, it is possible to observe the set of tiles computed by a selected CPU. In this example, one can observe the spreading of tiles computed by CPU #4 during four iterations.

3.2.3 Thumbnails generation

As one can observe in Figure 9, a reduced view of the Mandelbrot set is displayed in the background of the left figure. By default, the background is black. To use a *thumbnail* reflecting your kernel output at each iteration, these thumbnails must be generated during a preliminary execution, using the `--thumbnails` (or `-tn` for short).

```
./run --kernel mandel --variant omp --thumbnails --iterations 10
```

The thumbnails are small PNG files generated in the `data/traces` subdirectory. Once generated, they can be used by multiple subsequent trace visualizations (as long as the kernel stays the same, obviously).

Trick: Since the thumbnails should be identical with respect to the variant used, the fastest one should be preferred.

```
# Dry run to first generate thumbnails
./run --kernel mandel --variant omp --thumbnails --iterations 10
# Then generate a first trace
OMP_SCHEDULE=static ./run --kernel mandel --variant omp --trace --iterations 10
# Observe the trace
./view
# Try a different schedule
OMP_SCHEDULE=dynamic ./run --kernel mandel --variant omp --trace --iterations 10
# Observe
./view
```

Figure 11: Typical workflow when using execution traces.

As a recap, Figure 11 (page 17) shows a typical sequence of commands to analyze the impact of the OpenMP loop scheduling policy on the execution of the `mandel` kernel. Actually, this example

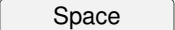
Key/Mouse control	Action
	toggle the “quick-nav” mode On/Off. In quick-nav mode, iterations are always completely displayed. As a consequence, the  and  keys directly shift the iteration range. When not in quick-nav mode, zooming on specific parts of iterations can be performed, and the  and  keys smoothly shift the bounds of the displayed time interval.
 	shift the displayed time range.
 	zoom/unzoom the displayed time range.
	reveal the whole trace.
	toggle between vertical (default) and horizontal mouse selection mode.
	enter/leave <i>footprint mode</i> , where all tiles corresponding to on-screen tasks are displayed.
	toggle <i>tracking mode</i> .
Mouse scroll	shift the displayed range.
Mouse click-and-drag	select a specific time range.
	zoom to current selection.
	take a screenshot.

Table 2: Navigating inside traces.

represents a case for a more convenient feature: the ability to display two traces simultaneously! This is precisely the topic of the next Section.

3.2.4 Trace comparison

To display two traces at the same time, run `view` with two filenames as arguments, e.g.:

```
./view data/traces/ezv_trace_1.evt data/traces/ezv_trace_2.evt
```

That said, in many cases we just want to compare the last two traces. In such cases, the `--compare` option (or `-c` for short) is a convenient shortcut:

```
./view -c
```

It is equivalent to

```
./view data/traces/ezv_trace_current.evt \
    data/traces/ezv_trace_previous.evt
```

Figure 12 (page 19) shows an example where two traces are displayed simultaneously. In this specific example, one can observe the different task scheduling strategies used by respectively the `gcc9` and `clang8` OpenMP runtime systems.

You can navigate over the traces using the same keyboard/mouse control as seen previously (Table 2). In addition, a new feature is available when comparing traces: you can press `a` to toggle “auto-align” mode. In *auto-align* mode, the beginning of each iteration is synchronized in both traces, by adding extra padding between iterations if necessary. This typically allows to conveniently compare the same iteration range in both traces. In Figure 12, iterations 12 and 13

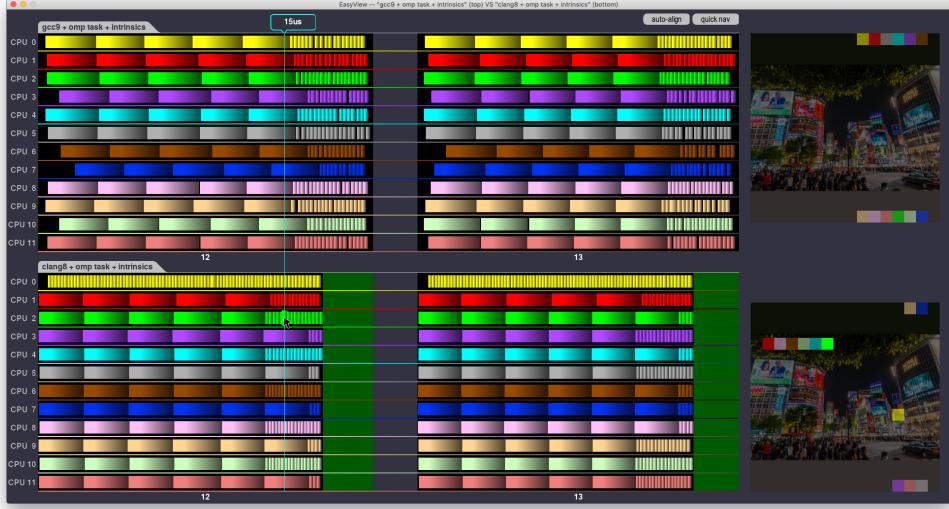


Figure 12: Comparing two traces with *auto-align* enabled.

are displayed, and it is easy to observe that the bottom trace results from a more efficient execution of the kernel. Green areas materialize the amount of time saved per iteration by the faster run.

3.2.5 Comparison between different granularities

When comparing two traces corresponding to using different tile sizes (that is, different granularities), it is often useful to compare the time spent to compute a given region on both sides. On one hand, a large granularity may decrease the overhead incurred by the underlying management of inter-tile parallelism (e.g. task creations). On the other hand, with kernels where not all tiles are systematically calculated (e.g. lazy evaluation), a small tile granularity will probably help to save computations at the frontiers between inactive and active areas.

To this end, the trace visualizer offers a *tracking mode* in which the user can select a task and see which tasks, in the other trace, have worked on the same region (see Figure 13, page 20). Most importantly, the cumulated duration of these tasks is displayed and can be directly compared with the duration of the selected task. To toggle this mode, simply press `t`.

4 Distributed Computing with MPI

EASYPAP allows the implementation of distributed computations using the MPI high performance communication library. The integration of MPI into EASYPAP was done in such a way that programmers can focus on the design of their kernels without worrying (too much) about low-level details related to handling SDL events or coping with the distribution of the initial content of data.



Figure 13: Comparing two traces with “*tracking mode*” enabled. When selecting a coarse grain task in one trace, the corresponding fine grain tasks are highlighted (in white color) in the other trace, and the accumulated execution time is indicated.

4.1 A simple example

Let us go back to our `spin` kernel. Its parallelization is trivial due to the fact that the computation of each pixel is totally independent from the others.

The `mpi` variant of `spin` is displayed in Figure 14 (page 21). In this implementation, the work is evenly distributed across MPI processes by dividing the image in horizontal stripes of equal thickness. For each process, the computation of the bounds of its tile are computed once in the `spin_init_mpi` function which is called automatically during the initialization phase of EASYPAP (see Section 6.1.1 for more details).

The `spin_compute_mpi` kernel function is remarkably simple: upon each invocation of the kernel, each process performs a burst of iterations consisting in computing its horizontal stripe (calling `do_tile`, Figure 14, line 27). Then, the process participates to a collective MPI communication to regroup all the contributions on the master node (`MPI_Gather`, lines 32–33).

4.2 Running MPI variants

MPI implementations usually come with a `mpirun` command which takes care of properly launching the requested number of processes, possibly using resource allocation managers in some environments.

To launch MPI variants of kernels, the EASYPAP `run` script calls `mpirun` with the parameters supplied using the `--mpirun` (or `-mpi`) option. As an illustration, here is how to run the `mpi` variant of `spin` using 2 processes:

```
./run --kernel spin --variant mpi --mpirun "-np 2"
```

```

1 static int mpi_y      = -1;
2 static int mpi_h      = -1;
3 static int mpi_rank   = -1;
4 static int mpi_size   = -1;
5
6 void spin_init_mpi (void)
7 {
8     easypap_check_mpi (); // check if MPI was correctly configured
9
10    MPI_Comm_rank (MPI_COMM_WORLD, &mpi_rank);
11    MPI_Comm_size (MPI_COMM_WORLD, &mpi_size);
12
13    mpi_y = mpi_rank * (DIM / mpi_size); // first line to start with
14    mpi_h = (DIM / mpi_size);           // number of lines to compute
15 }
16
17 unsigned spin_compute_mpi (unsigned nb_iter)
18 {
19     for (unsigned it = 1; it <= nb_iter; it++) {
20
21         do_tile (0, mpi_y, DIM, mpi_h, 0);
22         rotate ();
23     }
24
25    MPI_Gather ((mpi_rank == 0 ? MPI_IN_PLACE : image + mpi_y * DIM), mpi_h * DIM,
26                MPI_INT, image, mpi_h * DIM, MPI_INT, 0, MPI_COMM_WORLD);
27
28    return 0;
29 }

```

Figure 14: Simple MPI variant of the spin kernel. Calls to the MPI library are highlighted.

To prevent users from forgetting the `--mpirun` option when using MPI variants, a call to `easypap_check_mpi ()` can be done in the variant-specific initialization function. This is illustrated in Figure 14 at line 8.

4.3 Debug mode

By default, only the windows (i.e. main window plus, optionnally, monitoring windows) associated to the master process are showing up. The other processes are running “in the background” but are still capable of handling the fact that `esc` was pressed in the master window, for instance.

The debug mode of MPI can be activated by including the `M` character in the `--debug` flags:

```
./run --kernel spin --variant mpi --mpirun "-np 2" --debug M
```

This forces the display of all processes’ windows, helping to track down work distribution bugs by inspecting “*which process computes what pixel.*”, as illustrated in Figure 15.

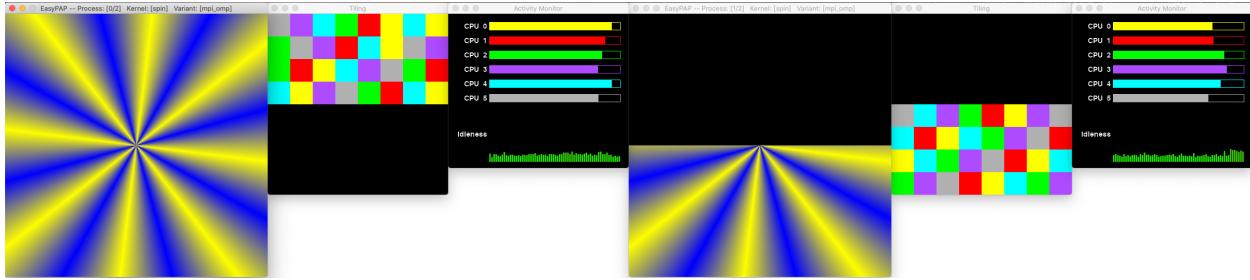


Figure 15: MPI+OpenMP variant of spin running in debug mode. Two MPI processes were launched, each containing 6 threads.

4.4 Traces

Trace generation of MPI variants will lead to the creation of one separate trace file per process. The command line options are no different from the non-MPI variants:

```
./run --kernel spin --variant mpi --mpirun "-np 2" --no-display --trace
```

In this example, two processes were launched, leading to the generation of two trace files: `data/traces/ezv_trace_current.0.evt` and `data/traces/ezv_trace_current.1.evt`. They can be visualized side by side using the *compare* capabilities of the trace viewer:

```
./view data/traces/ezv_trace_current.0.evt \
        data/traces/ezv_trace_current.1.evt
```

5 Exploiting GPU accelerators

5.1 OpenCL

EASYPAP allows to implement and run OpenCL kernels as easily as regular C kernels. When using the main⁸ GPU as a target, the image buffers used by EASYPAP are shared between OpenGL and OpenCL so that the results of OpenCL computations are rendered *in place* and require no data transfer between the host memory and the GPU.

5.2 Checking the OpenCL configuration

Many OpenCL implementations rely on an OpenCL Installable Client Driver (ICD). This mechanism allows OpenCL implementations from multiple vendors to coexist on a system.

To display the list of devices detected by OpenCL, just run:

```
[my-machine] ./run --show-devices
1 OpenCL platforms detected
Platform 0: Apple (Apple)
--- Device 0 : CPU [Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz]
+++ Device 1 : GPU [Intel(R) UHD Graphics 630]
--- Device 2 : GPU [AMD Radeon Pro 560X Compute Engine]
Note: OpenGL renderer uses [AMD Radeon Pro 560X OpenGL Engine]
```

⁸That is, the GPU in charge of the screen display

Note which device is selected by default (line beginning with `+++`). To be able to run EASYPAP interactively, please make sure that the selected device corresponds to renderer driver of OpenGL (which is not the case in our example). In performance mode (i.e. no display), you may choose any OpenCL device as a target.

To specify a particular platform or device, please set the `PLATFORM` and/or `DEVICE` shell variables. For instance:

```
[my-machine] DEVICE=2 ./run --show-devices
1 OpenCL platforms detected
Platform 0: Apple (Apple)
--- Device 0 : CPU [Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz]
--- Device 1 : GPU [Intel(R) UHD Graphics 630]
+++ Device 2 : GPU [AMD Radeon Pro 560X Compute Engine]
Note: OpenGL renderer uses [AMD Radeon Pro 560X OpenGL Engine]
```

5.2.1 Writting and executing 2D kernels

OpenCL kernels are located in the `kernel/ocl/` subdirectory.

To Appear Soon

5.3 NVIDIA Compute Unified Device architecture

To Appear Soon

6 Advanced topics

6.1 Initialization hooks

Before the execution of a kernel starts, a number of user prelude functions can be called by EASYPAP in order to initialize/allocate data structures, perform initial computations the matrices, or even trigger page-based *first-touch* allocations.

```
1 [my-machine] ./run -d i -k hook_funcs
2 Using kernel [hook_funcs], variant [seq]
3 Init phase 1 : SDL initialized (DIM = 1024)
4 Init phase 2 : [OpenCl init not required]
5 Hello from the initialization function! Image size is 1024x1024
6 Init phase 3 : init() hook called
7 Init phase 4 : images allocated
8 Init phase 5 : [first-touch policy not activated]
9 Init phase 6 : kernel-specific draw() hook called
10 Init phase 7 : [no OpenCL data transfer involved]
```

Figure 16: Detailed typical EASYPAP initialization sequence.

File `kernel/c/hook_funcs.c` demonstrates how the user can define several functions that will be called at specific steps during the initilizaion process. Figure 16 shows the output obtained when running the `hook_funcs` kernel with the `--debug i` flag set. Seven distinct phases are involved:

1. The SDL library is initialized and the image size (i.e. `DIM`) is calculated ;

2. If required, the OpenCL library is initialized ;
3. If defined, the `init` kernel-specific function is called (see Section 6.1.1) ;
4. Images are allocated using the `mmap` system call ;
5. If defined, the first-touch kernel-specific function is called (see Section 6.1.2) ;
6. If defined, the `draw` kernels-specific function is called (see Section 6.1.3) ;
7. Data are transferred to the GPU if needed.

Note: For all kernel-specific functions (e.g. `init`), EASYPAP first looks for the variant-specific one (e.g. `<kernel>_init_<variant>`) and, if not found, looks for the general one (e.g. `<kernel>_init`).

6.1.1 Initialization

The user-defined `init` function is called after the size (i.e. `DIM`) is known. It is typically the place where *worker threads* can be spawned, in case a specific task-based scheduler⁹ is desired for instance. It is also the place where various variables can be initialized. As an illustration, the `init` function defined in `kernels/c/mandel.c` is displayed in Figure 17.

```

1 static float leftX    = -0.2395;
2 static float rightX   = -0.2275;
3 static float topY     = .660;
4 static float bottomY = .648;
5
6 static float xstep;
7 static float ystep;
8
9 void mandel_init (void)
10 {
11     xstep = (rightX - leftX) / DIM;
12     ystep = (topY - bottomY) / DIM;
13 }
```

Figure 17: Initialization of global variables using the `init` hook function.

Note that, at the time the `init` function is called (see Figure 16, line 6), images are not yet allocated. If accessing `cur_img` or `next_img` is needed, the code should be placed in the drawing hook (Section 6.1.3).

6.1.2 First-touch data allocation

Because most parallel nodes have a *non-uniform memory access* (i.e. NUMA) architecture, it is often desirable to control the allocation of data structures at the page level. On Linux, for instance, the `numactl` utility can help to run EASYPAP under a specific memory allocation policy.

⁹Using worker threads is illustrated by the `sched` variant of the `max` kernel. Check `max_init_sched` and `max_finalize_sched` functions for more details.

In addition, EASYPAP provides an optional *first touch* feature: when the `--first-touch` option appears on the command line, a “*first touch hook*” is called (if defined). Since the function is called before any access has been performed on the images (see Figure 16, line 6), it is possible to force physical allocation near specific cores from within a parallel region.

Figure 18 illustrates how the `omp` variant of the `transpose` kernel distributes data allocations evenly among cores.

```

1 void transpose_ft_omp (void)
2 {
3     #pragma omp parallel for
4     for (int i = 0; i < DIM; i++)
5         for (int j = 0; j < DIM; j += 512)
6             next_img (i, j) = cur_img (i, j) = 0;
7 }
```

Figure 18: First-touch data allocation in an OpenMP kernel using static scheduling.

6.1.3 Drawing hook

The ability to define a `draw` function has already been presented in Section 2.8.

Let us just observe that, since this hook function receives a command line string as a parameter, it can be used for other purposes too. For instance, in the `pixelize` kernel, it is used to parametrize the size of squares on the pixelized image, as shown in Figure 19.

```

1 void pixelize_draw (char *param)
2 {
3     unsigned n;
4
5     if (param != NULL) {
6         n = atoi (param);
7         if (n > 0)
8             PIX_BLOC = n;
9     }
10 }
```

Figure 19: Using the command line argument to parametrize the `pixelize` kernel.

6.2 Using your own data structures

For convenience, EASYPAP provides two predefined images to work with – `cur_img` and `next_img` – which are `DIM×DIM` arrays of `unsigned int` elements.

However, there are many 2D kernels which require to allocate more complex data structures (e.g. Adaptive mesh refinement codes, sparse matrices, etc.), or 2D matrices of a different type.

Implementing the *Game of Life*, for instance, is probably more cache-effective if the cells are stored using a few bits rather than using an `unsigned int` per cell.

The `lifec` kernel is such an implementation of *Game of Life*. It uses a memory footprint of one byte per cell. Two matrices, `cur_table` and `next_table`, are allocated (resp. destroyed) in the `init` (resp. `finalize`) hook function, as shown in Figure 20.

```

1 static char *restrict _table = NULL,
2      *restrict _alternate_table = NULL;
3
4 static inline char *table_cell (char *restrict i, int y, int x)
5 {
6     return i + y * DIM + x;
7 }
8
9 #define cur_table(y, x) (*table_cell (_table, (y), (x)))
10 #define next_table(y, x) (*table_cell (_alternate_table, (y), (x)))
11
12 void lifec_init (void)
13 {
14     _table = mmap (NULL, DIM * DIM * sizeof (char), PROT_READ | PROT_WRITE,
15                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
16     _alternate_table =
17         mmap (NULL, DIM * DIM * sizeof (char), PROT_READ | PROT_WRITE,
18                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
19 }
20
21 void lifec_finalize (void)
22 {
23     munmap (_table, DIM * DIM * sizeof (char));
24     munmap (_alternate_table, DIM * DIM * sizeof (char));
25 }
```

Figure 20: Allocating/freeing ad hoc data structures in the `lifec` kernel.

The side effect of using arbitrary data structures is that the `cur_img` must be regularly updated to refresh the display window. To avoid performing unnecessary updates, and even avoiding all updates in case no display was requested (i.e. *performance mode*), EASYPAP calls a `refresh_img` function whenever the display must be refreshed. Figure 21 (page 26) shows how the `cur_img` image is updated in the `lifec` kernel.

```

1 void lifec_refresh_img (void)
2 {
3     for (int i = 0; i < DIM; i++)
4         for (int j = 0; j < DIM; j++)
5             cur_img (i, j) = cur_table (i, j) * color;
6 }
```

Figure 21: Updating `cur_img` when using ad hoc data structures is performed via a `refresh_img` function.

7 Plotting performance graphs with EASYPLOT

7.1 Introduction

Each time EASYPAP is invoked in performance mode (i.e., with the `--no-display` flag), it reports the completion time as well as all execution and configuration parameters in a *Comma Separated Value* (CSV) file. The collected data can then be exploited by EASYPLOT, a plotting facility based on the Seaborn¹⁰ data visualization library and the pandas¹¹ data analysis toolkit. One can easily filter and select appropriate data from the performance file. Other options are available to specify figure-level or axes-level parameters. A key feature of EASYPLOT is that speedup ratios and legends are automatically generated from the data. Once data have been filtered, constant parameters are put aside, and the names of plot-lines are set using the remaining parameters (see Fig. 23b). This guarantees that experiments conducted in different conditions will not silently be incorporated in the same graph.

7.2 Production of experimental data

As soon as we want to observe the influence of few parameters, it is preferable to explore them using scripts to systematically generate data. Several experimentation scripts are available (see `plots/run-xp-*.py` files). As an illustration, Figure 22 presents a simple script which automates the execution of the OpenMP tiled variant of the Mandelbrot kernel for several image sizes, tile sizes, and scheduling policies. Note that the data produced by this script is placed in the `mandel-data.csv` file.

7.3 Production of graphs

The Python script EASYPLOT generates line plots, scatter plots, and heat maps from a CSV file produced by EASYPAP. While execution time can naturally be used for the y-axis, the default performance measure is speedup. Additionally, it is possible to utilize parallel efficiency, throughput in pixels per second, or attributes obtained via performance counter measurements.

Moreover, users have the flexibility to define new attributes by adapting the EASYPLOT script. As an example, the script `plot-freq-stall.py` employs hardware performance counter values to compute the mean frequency (cpu-frequency) and the time wasted in stalls for each run.

In the figure 23 we show how to build a heatmap and a speedup graph with EASYPLOT.

7.3.1 Data selection

The script's options are derived from attribute names in EASYPAP.

Example:

```
./plots/easyplot.py --kernel mandel --size 1024 --tileh 4 8 16 \
--input mandel-data.csv --output mandel.pdf
```

Data selection's options:

`--attribute values` Selects rows that match with any provided value for the specified attribute.

¹⁰<https://seaborn.pydata.org>

¹¹<https://pandas.pydata.org>

```

#!/usr/bin/env python3
from expTools import *

easypapOptions = {
    "--output-file": ["mandel-data.csv"]
    "--kernel": ["mandel"],
    "--variant": ["omp_tiled"],
    "--iterations": [20],
    "--size": [1024, 2048],
    "--tile-height": [2**i for i in range(0, 5)],
    "--tile-width": [2**i for i in range(3, 11)]
}

ompICV = {    # OMP Internal Control Variable
    "OMP_SCHEDULE": ["dynamic", "static,1"],
    "OMP_NUM_THREADS": [46],
    "OMP_PLACES": ["cores"]
}

execute('./easypap', ompICV, easypapOptions, nbruns=5)

```

Figure 22: Experiments automation script to investigate the role of tile size.

--**delete** *attributes* Eliminates complete columns identified by their attribute from the dataset.

--**input**, -**if** *filename* Specifies the path of the input file, with the default being easypap/data/perf.csv.

--**output**, -**of** *filename* Specifies the path of the ouput file, with the default being plot.pdf. The available file formats include eps, jpeg, jpg, pdf, pgf, png, ps, raw, rgba, svg, svgz, tif and tiff.



Tip

Utilize the --verbose option to receive feedback regarding data selection.

7.3.2 Data presentation

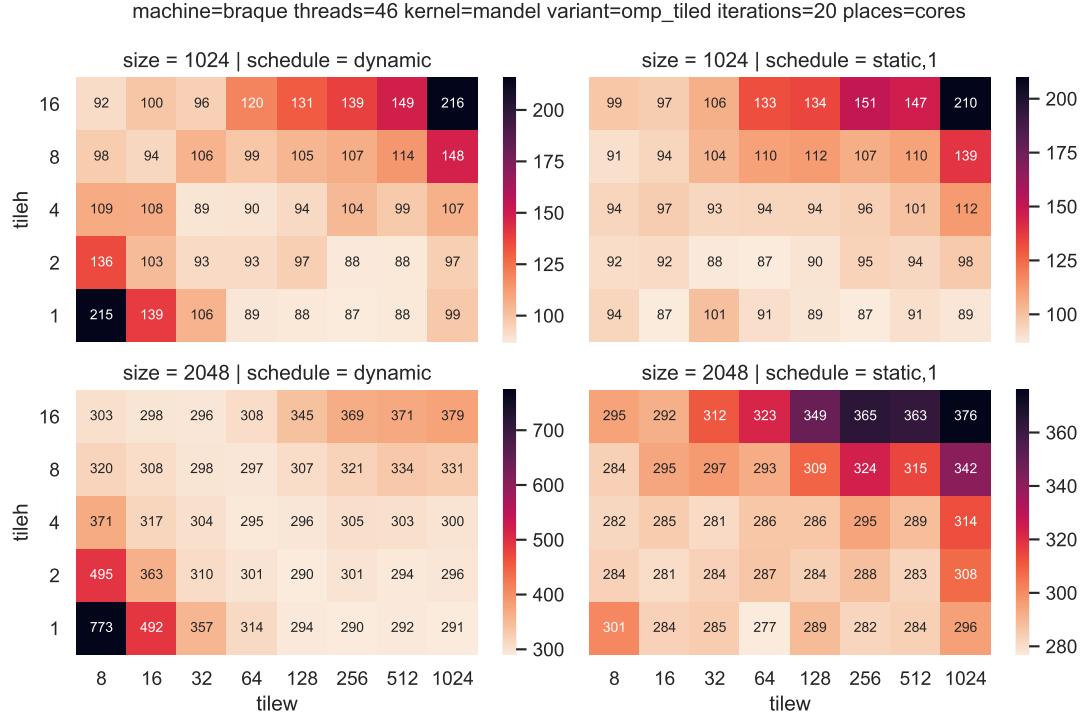
Options for data presentation directly sourced from Seaborn / Matplotlib.

Example :

```
./plots/easyplot.py --plottype catplot -y time -yscale log -- row=size col=schedule
```

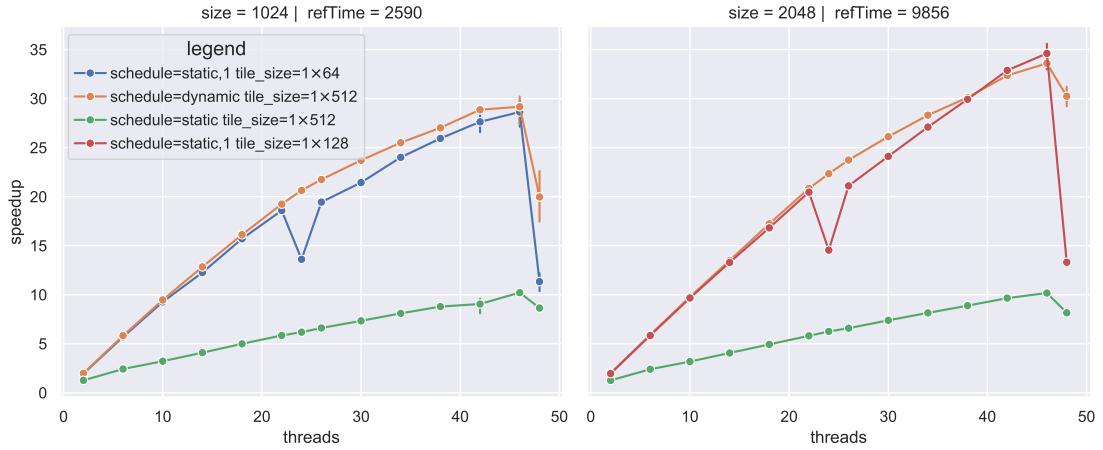
--**plottype** *kind* Seaborn's lineplot, catplot and heatmap are available.

-**y** *measure-attributes* Specifies the measure-attributes to be displayed on the y-axis.



(a) Heat map revealing the influence of tile geometry on performance. Numbers inside tiles denote the average duration time (in ms). This heat map was obtained using the following command:

```
easyplot.py --input mandel.csv --plottype heatmap \
-y time -heatx tilew -heaty tileh -- col=schedule row=size
machine=braque kernel=mandel variant=omp_tiled iterations=20 places=cores label=speedUp
```



(b) Speedup plots of mandel kernel. The speedup ratios were computed against the best sequential execution time (2590 ms for 1024×1024 and 9856 ms for 2048×2048). This graph was obtained using the following command:

```
easyplot.py --input mandel-speedup.csv -- col=size legend_out=false sharey=row
```

Figure 23: Some plots and their associated command line.

-y2 *measure-attributes* Attributes related to the second y-axis, displayed on the right side of the graph.

-x *non-measure-attribute* For the x-axis, you can specify a non-measure, ie. categorical, attributes (eg. tiles size, images dimension, variants name,...). The default is threads.

--heatx, **--heaty** *non-measure-attribute* Indeed, heat maps need two axis.

--xscale, **--yscale**, **--yscale2** *scale* For setting axis scales. By default linear scales are used, however log and log2 scales may be useful.

Customization options for Catplot and Lineplot graphs align with the corresponding arguments of Seaborn's catplot and, in the case of Lineplot, FacetGrid methods. These arguments are specified in the form of an `arg=value` assignment. For instance, to create a multi-plot grid where graphics share the same size on each row and the same variant on each column, simply define the `row=size` `col=variant` variables. It's important to note that only boolean, float, integer, and string parameters can be set via the command line. Please note that it might be necessary to use `--` to separate a list of arguments placed before another option. For example: `--size 512 1024 -- row=none`.

FacetGrid's and catplot's arguments are detailed in Seaborn's documentation at the following links: <https://seaborn.pydata.org/generated/seaborn.facetgrid.html> and <https://seaborn.pydata.org/generated/seaborn.catplot.html>. Among these arguments, the following are common main arguments:

row, col Categorical variables determining the faceting of the grid.

sharex, sharey Controls sharing of properties among x (sharex) or y (sharey) axes:

- True or all: x- or y-axis will be shared among all subplots.
- False or none: each subplot x- or y-axis will be independent.
- row: each subplot row will share an x- or y-axis.
- col: each subplot column will share an x- or y-axis.

height Height (in inches) of each facet.

aspect Aspect ratio of each facet, where aspect * height gives the width of each facet in inches.

legend_out If True, the figure size extends, and the legend is drawn outside the plot on the center right.

Additionally, Catplot introduces the following arguments:

kindstr The type of plot to draw, corresponding to the name of a categorical axes-level plotting function. Options include: "strip," "swarm," "box," "violin," "boxen," "point," "bar," or "count."

native_scale When True, numeric or datetime values on the categorical axis maintain their original scaling rather than being converted to fixed indices.

7.3.3 About speedup computations

As previously mentioned, EASYPLOT automatically calculates speedup ratios, which are derived from reference durations representing minimal times observed during sequential executions with a thread count set to 1 (using `OMP_NUM_THREADS=1`). However, in some cases, optimal sequential performance is achieved through a vectorized version, prompting the need to opt for a purely sequential version:

--RefTimeVariants, **--RefTimeTiling** functions Choose specific rows for speedup ratio calculations.

--noRefTime Suppress the display of the duration used in the legend for cosmetic reasons.

7.3.4 Cosmetic options

--noSort Follows the data order to process the graphics (i.e., does not sort data based on y-axis values).

--fontScale *float* Scales the font of the title and the legend.

--adjustTop *float* Adjusts the space between the title and the graphs.

7.3.5 Plotting multiple attributes

The `-c` option in EASYPAP facilitates the collection of measurements from hardware counters. The script `plot-freq-stall.py` generates graphs using these measurements, defining attributes `frequency` (average core frequency) and `stall_ratio`. This ratio indicates the sum of stall cycles across all cores divided by the total number of execution cycles. Two examples of these graphs are illustrated in 24.

8 Installing EASYPAP

EASYPAP runs on both Linux and Mac OS X systems.

8.1 Prerequisites

EASYPAP is written in C. and uses OpenMP/Pthreads to exploit multicore machines.

On Mac OS X systems, make sure that the C compiler you are using is understanding OpenMP directives (the default Apple clang compiler does not support OpenMP).

8.2 Required packages

8.2.1 SDL2

EASYPAP relies on the [SDL](#) 2.0 library for fast 2D graphics display and requires the following packages:

- `libsdl2-dev` (Linux) / `libsdl2` (Mac OS X)

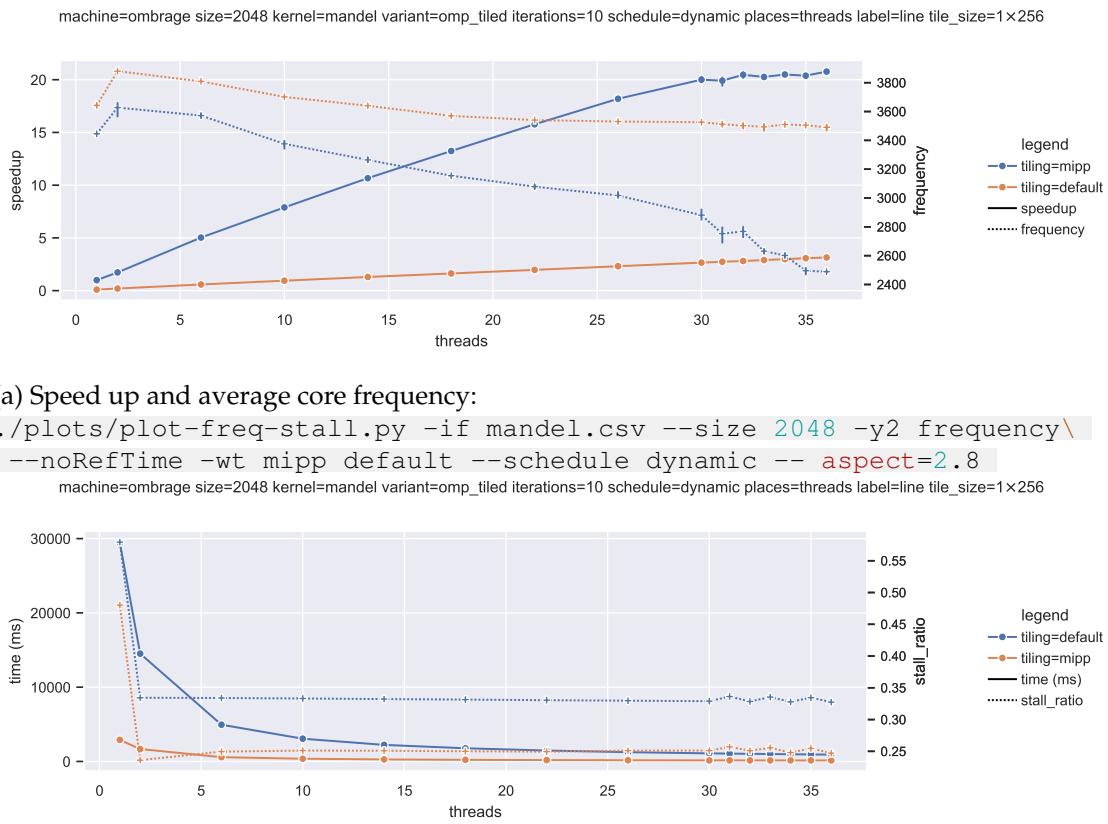


Figure 24: Some plots and their associated command line.

- `libsdl2-image-dev` (Linux) / `libsdl2_image` (Mac OS X)
- `libsdl2-ttf-dev` (Linux) / `libsdl2_ttf` (Mac OS X)

8.2.2 Hwloc

EASYPAP relies on the [Portable Hardware Locality \(hwloc\)](#) library to discover the underlying hardware topology and place threads accordingly. Please make sure the following package is installed:

- `libhwloc-dev` (Linux) / `hwloc` (Mac OS X)

8.2.3 Lex

EASYPAP relies on the LEX lexical analyzer generator to parse Run-Length Encoded (RLE) files, typically files describing *Game of Life* configurations. Please make sure the following package is installed:

- `flex` (Linux) / Xcode Command Line Tools (Mac OS X)

8.2.4 Cglm

EASYPAP relies on the `cglm` optimized 3D math library to perform its geometrical calculations. Please make sure the following package is installed:

- `libcglm-dev` (Linux) / `cglm` (Mac OS X)

8.3 Optional packages

8.3.1 FxT (recommended)

EASYPAP relies on the [Fast User Tracing utility](#) to generate and display execution traces. This package is required if `ENABLE_TRACE` set in the Makefile (see Section 8.5, page 35). If needed, please install the following package:

- `libfxt-dev` (Linux when available) / — (manual installation required on Mac OS X and some Linux distributions)

On Mac OS X systems, or on Linux distributions that do not provide a `libfxt` package, please download the latest version of FxT sources from [Savannah](#), and follow the instructions of the `README` file:

```
./configure
make
sudo make install
```

8.3.2 Scotch (recommended)

EASYPAP relies on the [Scotch partitioning library](#) to split meshes into subdomains. To use this feature, please make sure that `USE_SCOTCH` is defined both in `libezv`'s Makefile, and that the following package is installed:

- `libscotch-dev` (Linux) / `scotch` (Mac OS X)

8.3.3 OpenCL

[OpenCL](#) is a portable programming environment to exploit hardware accelerators. To use OpenCL to generate kernels, please set `ENABLE_OPENCL` in the Makefile (see Section 8.5, page 35), and install at least the following package:

- `ocl-icd-opencl-dev` (Linux) / Xcode Command Line Tools (Mac OS X)

Depending on your hardware, you may need to install additional packages (e.g. AMD OpenCL drivers).

8.3.4 CUDA (Linux only)

If your machine is equipped with an NVIDIA graphical processing unit (GPU), you might want to manage your hardware with CUDA. In this case, set `ENABLE_CUDA` in the Makefile (and make sure that `ENABLE_OPENCL` is unset), and install the following package:

- `nvidia-cuda-toolkit` (Linux) / — (NVIDIA no longer supports CUDA on Mac OS X)

8.3.5 MPI

To build an MPI-ready version of EASYPAP (i.e. with `ENABLE_MPI` set in the Makefile), you need to install an MPI development environment. Although EASYPAP could probably work with any available MPI implementation, it has only been tested with the [Open MPI](#) implementation. So please install the following package:

- `libopenmpi-dev` (Linux) / `openmpi` (Mac OS X)

8.3.6 OpenSSL

To use sha256 encryption methods to signing computation outputs, please make sure `ENABLE_SHA` is defined in the Makefile and install the following package:

- `libssl-dev` (Linux) / `openssl13` (Mac OS X)

8.3.7 MIPP

The [MIPP](#) software is provided as a git submodule of the EASYPAP project. To install MIPP, simply type:

```
git submodule update --init
```

To develop and run MIPP kernels, you'll also have to set the `ENABLE_MIPP` flag in the Makefile (see Section 8.5, page 35).

8.3.8 PAPI (Linux only)

EASYPAP can collect hardware performance counters (using the `--pref-counters` command line option) using the [PAPI Library](#) (Performance Application Programming Interface). To use this feature, please make sure `ENABLE_PAPI` is defined in the Makefile and the following package is installed:

- `libpapi-dev` (Linux) / — (PAPI is not available on Mac OS X)

8.4 Troubleshooting



Warning

The EASYPAP Makefile uses the `pkg-config` utility to collect compile and link flags for each of the aforementioned packages. Please make sure that your `PKG_CONFIG_PATH` variable is properly set to find the config files associated to these packages. You may typically run the following command and check if the result contains the `-I<includedir>` paths corresponding to your local setup:

```
pkg-config --cflags sdl2 fxt hwloc
```

8.5 Customizing EASYPAP

Before compiling EASYPAP, you may check the Configuration Section at the top of the Makefile (see Figure 25) to enable/disable some functionalities.

```
1 ##### Config Section #####
2
3 ENABLE_MONITORING = 1
4 ENABLE_VECTO     = 1
5 ENABLE_TRACE     = 1
6 ENABLE_MPI       = 1
7 ENABLE_SHA       = 1
8 ENABLE_OPENCL    = 1
9 #ENABLE_CUDA      = 1
10 #ENABLE_MIPP     = 1
11 #ENABLE_PAPI     = 1
12
13 #####
```

Figure 25: The config Section of the Makefile allows to easily disable some functionalities.

ENABLE_MONITORING To collect precise performance numbers and compute CPU activity periods, the code of EASYPAP is instrumented with conditional “`gettimeofday`” calls. Since

the instrumentation was carefully done to minimize execution overhead, we advise to keep monitoring always enabled.

ENABLE_VECTO Some kernel variants are using *Intrinsics* macros to generate vectorized code. In principle, EASYPAP only uses vector instructions supported by the underlying processor. Should you encounter problems coming from improper vectorization settings, disable this flag.

ENABLE_TRACE EASYPAP currently uses the `libfxt` fast tracing library to generate execution trace files. If this library is not installed on your machine, you should disable this flag. The installation of the `libfxt` package is obviously the preferred option.

ENABLE_MPI This flag should be set if you want to use the [Open MPI](#) communication library.

ENABLE_SHA EASYPAP can generate SHA256 signatures of the computation output at any iteration. This allows to check the correctness of a given implementation by comparing the obtained signature with the one obtained using a reference implementation. Enable this flag to use such a feature.

ENABLE_OPENCL Enable this flag to use the [OpenCL](#) programming environment to exploit hardware accelerators, including GPUs.

ENABLE_CUDA Enable this flag to use the [NVIDIA CUDA](#) programming toolkit to exploit NVIDIA GPUs. CUDA kernels must be placed into the `kernel/cuda` subdirectory.

ENABLE_MIPP [MIPP](#) (MyIntrinsics++) is a portable and Open-source C++ wrapper for vector intrinsic functions (SIMD). It can be easily fetched as a git submodule of your EASYPAP repository. Enable this flag to allow kernels in `kernel/mipp/` to be included at compile time.

ENABLE_PAPI EASYPAP is able to collect hardware performance counters when the [PAPI Library](#) (Performance Application Programming Interface) is installed on your system. If not, you should disable this flag.

Index of options

--arg, 10
--perf-counters, 35
--debug, 21
--first-touch, 24
--iterations, 9
--kernel, 6
--load-image, 9
--monitoring, 13
--mpirun, 20
--no-display, 9
--pause, 9
--refresh-rate, 8
--show-devices, 22
--size, 6
--thumbnails, 17
--tile-height, 11
--tile-size, 12
--tile-width, 11
--trace, 15
--variant, 6
--with-tile, 12