eVariX™ Quick Start Guide

SILKAN

January 2018

Welcome to this quick start guide of eVariX™. It will show you how to accelerate your MATLAB® models by generating standalone executables, and will make you familiar with the most useful optimization and parallelization options to help you make the most of it.

Apart from this guide, eVariX™ comes with a full documentation:

— an extensive *User Guide*; in particular, it describes how to install eVariX™, and how to write eVariX™ compliant code; it also includes a complete description of the supported operators and functions;

— a *Quick Reference Guide*, which lists all the available options and summarizes eVariX™ capabilities.

In the following sections, we will guide you through the acceleration of four examples covering several fields: Mandelbrot, conventional beam forming, Lattice-Boltzmann, and N-body simulations. For each example, we will show you how the eVariX™ options can impact the performances, and give you some tips.

All our performance results have been obtained on a Linux platform described in the Appendix on page 5 and are summarized in the conclusion.

1 Mandelbrot

We begin with a standard Mandelbrot computation example. Two files are provided:

— `mandelbrot.m` provides the mandelbrot function; this version intentionally uses single array element accesses instead of whole matrix operations to exemplify eVariX™ loop auto-parallelization features;

— `run_mandelbrot.m` calls 10 times the mandelbrot function, displays the intermediary results and computes the average time of a call. Displaying the results is here compulsory, otherwise eVariX™ would eliminate the calls to the mandelbrot function as dead code.

To generate a sequential standalone executable named `seq`, run the command:

```
evarix run_mandelbrot.m -o seq
```

If in addition you want some information about the compilation process, and to have a look at the generated C++ code, run:

```
evarix run_mandelbrot.m -o seq -v --keep
```

The C++ file is named `seq.cold.cpp` after the executable output file name. The generated executable can be run with the following command line:

```
seq
```

On our platform (see the appendix for the experimental conditions), the average running time for the mandelbrot function in MATLAB® is around 19s, while it is around 2.7s for the executable generated by eVariX™, so more than 7 times faster.

1.1 Optimizing array usage: the `-asi` option

eVariX™ tries to precisely determine if the size of an array may change or not during the program execution. If it finds out that it doesn't change, then the allocation is optimized and array bound checks are avoided as well as array resizing operations. Depending on the application, this may have a noticeable impact on performances.

When array elements are written using indexed references (such as `z(l,j)`), eVariX™ cannot determine whether these references may trigger a growth of the array, and it thus conservatively assumes that the size of the array may change. However, if you are the author of the Mandelbrot program or have a sufficient knowledge of it, you know that the indexed accesses never go beyond the current array size, and you can specify it with the `--array-static-indexing` option, or its short form `-asi`:

```
evarix run_mandelbrot.m -o seq -asi
```

On our platform, the execution time is now around 1.74s, which is 11 times faster than the Matlab execution.

1.2 Automatic `for` loop parallelization

The Mandelbrot computation is well known for begin intrinsically parallel. So let us now examine the possibility of automatically parallelizing the loops in our example. The eVariX™ option to trigger this capability is `--auto-parallelization-level=2`. In addition, the option `--parallelization-report-level=1` lets you know about the loops which can or cannot be parallelized by eVariX™.

```
evarix run_mandelbrot.m -o par -asi --auto-parallelization-level=2 \
       --parallelization-report-level=1
```

The `-parallelization-report-level=1` options give you the following report:

```
---------- Coarse Grain Parallelization Report --------------------------------

mandelbrot.m:18:
  inter-iterations array dependences:
    ** WW: z c
    ** WR: z c
    ** RW: z c
  => loop on index k is NOT PARALLELIZABLE
mandelbrot.m:19:
  loop on index l is PARALLELIZABLE
mandelbrot.m:20:
  loop on index j is PARALLELIZABLE
run_mandelbrot.m:3:
  write effects on global or break variables/objects:  IOEffect TimeEffect
  => loop on index i is NOT PARALLELIZABLE

-------------------------------------------------------------------------------

---------- Parallelization Policy Report --------------------------------------

mandelbrot.m:19:
  loop on index l set as PARALLEL
mandelbrot.m:20:
  loop on index j finally not set as PARALLEL

-------------------------------------------------------------------------------
```

Loops l and j are identified as being parallelizable, and the dependences that still preclude parallelization are given for the loop k. And finally, only the loop j is set as parallel by the internal heuristic [1].
The execution time then becomes 0.22s, a speed-up of 85 compared to Matlab.

1.3 The `--fes-level=1` option

This option triggers a series of optimizations. In most cases it gives some additional performance gains, and we highly recommend that you use it systematically. The command is then:

```
evarix run_mandelbrot.m -o par -asi --auto-parallelization-level=2 \
       --parallelization-report-level=1 --fes-level=1
```

Running the generating executable on our platform gives an average time of 0.177 s, which means a speed-up near 110 compared to Matlab.

1. Beware that not using the `-asi` option in this case will preclude the parallelization, as eVariX™ conservatively avoids to parallelize loops writing arrays which may have variable sizes.

1.4 Automatic loop parallelization and gpu

Finally, if we now embed the main loop nest inside a gpu section (see files `run_mandelbrot_gpu.m` and `mandelbrot_gpu.m`), the executable generated with the same command as in the previous section runs in 0.028s in single precision, a speed-up of 686 compared to Matlab.

1.5 Summary

The table below summarizes the different options and their impact on performances:

Mandelbrot	time (s)	Speed-up / MATLAB®
MATLAB®	19.22	
eVariX™ (cpu)		
(no option)	2.7	7.2
-asi	1.74	11
-asi –auto-parallelization-level=2	0.22	85.9
-asi –auto-parallelization-level=2 –fes-level=1	0.177	**108**
eVariX™ (gpu)		
-asi –auto-parallelization-level=2 –fes-level=1	0.028	**686**

2 Conventional Beam Forming

Our second program is a simple conventional beam forming example, provided in file `v_cbf.m`. The best sequential performances can be reached with the following command:

```
evarix v_cbf.m -o seq -asi --fes-level=1
```

The running time for the main computation loop is then 6.2s, 2.3 faster than Matlab 14.6s.

The main loop in `v_cbf.m` is intrinsically parallel, and the program also contains several element wise array operations which can be automatically parallelized by eVariX™. The `--auto-parallelization-level=3` option triggers both kind of parallelization:

```
evarix v_cbf.m -o par -asi --auto-parallelization-level=3 --fes-level=1
```

The running time for the main computation loop then becomes 3.43s, 4.25 times faster than Matlab.

CBF	time (s)	Speed-up / MATLAB®
MATLAB®	14.6	
eVariX™		
-asi –fes-level=1	6.2	2.3
-asi –fes-level=1 –auto-parallelization-level=3	3.43	**4.25**

3 Lattice-Boltzmann

The third example, provided in file `cylinder_evarix.m`, has been taken from the Internet and slightly modified. You may in particular notice the use of the `ignore` annotation, which comments out some parts of the code[2] which are not relevant for or supported by eVariX™. This allows you to keep a single source code for both Matlab and eVariX™. We have also added some initializations of arrays, to ensure that their sizes are constant throughout the program.

You must now be familiar with the command to get a sequential version, knowing that your indexed array accesses don't go beyond the current arrays sizes:

```
evarix cylinder_evarix.m -o seq -asi
```

and how to turn on additional optimizations with the `--fes-level=1` option:

2. Here calls to the clear () function.

```
evarix cylinder_evarix.m -o seq_fes -asi --fes-level=1
```

On our platform, the execution time of the main loop within MATLAB® is 209s, 137s for `seq` and 86s for `seq_fes`, which means a speed-up of 1.52 and 2.43 respectively.

This example is mainly made of array element wise operations performed inside a timing loop, which is not parallel. Hence, it is sufficient to trigger the automatic parallelization of these array element wise operations, which is achieved with the `--auto-parallelization-level=1` option:

```
evarix cylinder_evarix.m -o par -asi --fes-level=1  --auto-parallelization-level=1
```

On our platform this leads to a further reduction of the main loop execution time, which falls to 52s, meaning a speed-up of 4 compared to Matlab.

The execution time can even be lowered by enforcing the number of threads and fixing their processor affinity. Our machine has 12 cores, which means up to 24 threads with hyperthreading. In our case, the command line to avoid hyperthreading, enforce a number of 12 threads, and fix the affinity so that that the threads run on different cores is:

```
OMP_NUM_THREADS=12 taskset -c 0-11 par
```

The execution time falls to 46.7s, to reach a speed-up of 4.47 compared to Matlab.

Lattice-Boltzmann	**time (s)**	**Speed-up / MATLAB®**	threads limitation
MATLAB®	209		none
eVariX™			
-asi	137	1.52	none
-asi –fes-level=1	86	2.43	none
-asi –fes-level=1 –auto-parallelization-level=1			
	52	4	none
	46.7	**4.47**	yes (12 threads)

4 N-Body Simulation

The last use case simulates the gravitational movement of set of objects and comes from McGill University. The source code is split into three files:

- `nbody1d.m` is the main computational function;
- `vrand1.m` implements a deterministic random vector initializer;
- `run_nbody1d.m` is the main script; it initializes the program data and calls the `nbody1d` function; the last three lines display the mean of some of the computed data to prevent eVariX™ considerating the call to `nbody1d` as dead code.

This gives us the opportunity to introduce the `--inline-level=2` option which triggers the inline expansion of functions (see the user guide for more details). Although in this case it brings little performance gains, it may be useful to know how to use it for larger cases for which it may make a huge difference.

Also, as we know that there is some intrinsic loop parallelism in the `nbody1d` function, we directly use the `--auto-parallelization-level=2` option:

```
evarix run_nbody1d.m -o par -asi --auto-parallelization-level=2 --fes-level=1
```

and for the version with the inlining option:

```
evarix run_nbody1d.m -o par_inl -asi --auto-parallelization-level=2 --fes-level=1 \
       --inline-level=2
```

When the value of the `scale` variable is set to 10, the execution time within MATLAB® is 2.88s. It falls to 0.083s with `par` and to 0.076s with `par_inl`, to reach speed-ups of 34.5 and 37.8 respectively compared to MATLAB®. By fixing the number of threads and the affinity, we can further decrease the execution time, which lowers to 0.062s for `par` (speed-up of 46) and to 0.056s for `par_inl` (speed-up of 51).

However, when size of the problem is increased by setting the value of the `scale` variable to 1000, there is no more need to restrict the use of the number of threads, and the maximum speed up is directly reached by simply running `par` or `par_inl`. The results are given in the following table:

N-Body	**time (s)**	**Speed-up / MATLAB®**	threads limitation
Scale 10			
MATLAB®	2.88		none
eVariX™			
-asi –fes-level=1	0.36	8	none
-asi –fes-level=1 –auto-parallelization-level=2	0.083	34.5	none
	0.062	46	yes (12 threads)
-asi –fes-level=1 –auto-parallelization-level=2 –inline-level=2	0.076	37.8	none
	0.056	**51**	yes (12 threads)
Scale 1000			
MATLAB®	278		
eVariX™			
-asi –fes-level=1	143.8	1.93	none
-asi –fes-level=1 –auto-parallelization-level=2	12.16	22.9	none
-asi –fes-level=1 –auto-parallelization-level=2 –inline-level=2	12.07	**23**	none

5 Conclusion

eVariX™ provides several options to help you adapt the generated code to your environments and your needs. When you don't use any option, the generated C++ code is very close to your original MATLAB® code. If you favor performance over output code readability, you may trigger one or more of the following options:

— **`-asi`** or **`--array-static-indexing`** specifies that no indexed access to array elements goes beyond the current array size, which is a recommended coding practice to get good performances even in MATLAB®; beware that eVariX™ cannot check the validity of using this option: if you use it non-adequately, you may get erroneous results;

— **`--fes-level=1`** usually brings performance gains, so we advise you to always turn it on;

— **`--inline-level=2`** should be tried if your code features several functions; notice however that in the case of very large codes, this may lead to performance loses;

— **`--auto-parallelization-level`** sets the level of parallelization; if your code uses element wise operations, set the level to 1; if it potentially has parallel for loops, set it to 2; and of course, if it has both, try level 3.

If you are unsure of which options you should use, just try them: generating an executable with eVariX™ only takes a few seconds!

The next figures summarize the speed-ups wrt MATLAB® 2017 obtained on our platform for our 4 examples. To simplify the graphics, eVariX™ options are designated by the following symbols:

— **N** for no option;

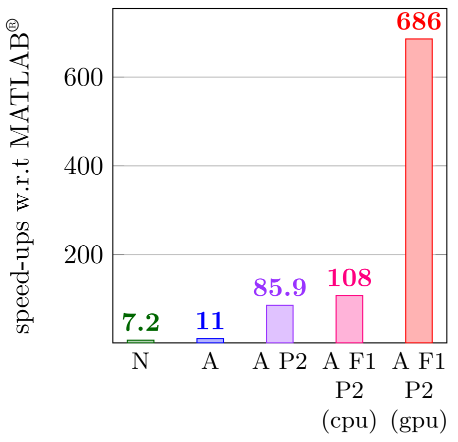
— **A** for `-asi`

— **F1** for `--fes-level=1`;

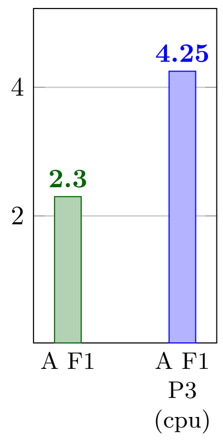
— **P1**, **P2** and **P3** for `--auto-parallelization-level=1,2,3`;

— **I2** for `--inline-level=2`.

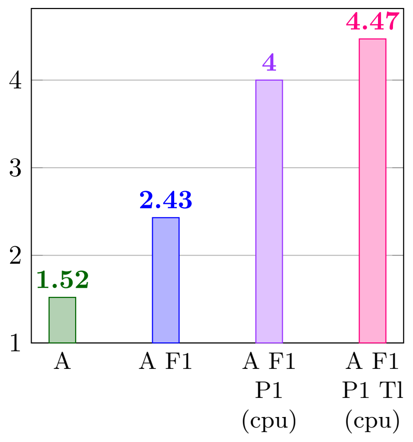
In addition, **Tl** stands for *thread limitation*, meaning that the number of threads is fixed as well as their affinity for the execution of the parallel executable generated by eVariX™ (see Section 3).



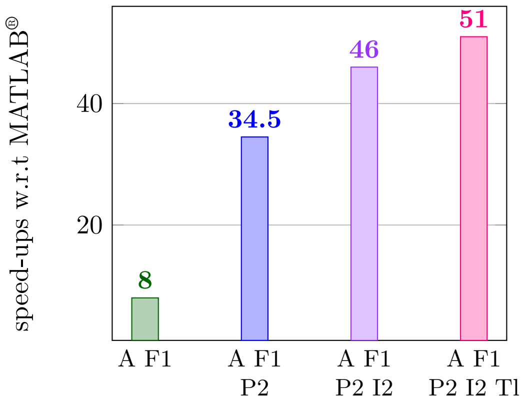
(1) Mandelbrot



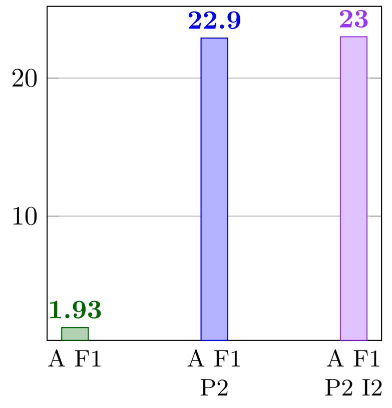
(2) CBF



(3) Lattice Boltzmann



(4) N-Body: Scale 10



(5) N-Body: Scale 1000

eVariX™ speed-ups for different sets of options

Appendix: Experimental conditions

All the performance results reported here have been obtained on a Linux platform with the following characteristics:

- Intel Core i7 3770@3.40GHz, 16Go ram
- Nvidia GeForce GTX 1080
- Ubuntu 14.04
- Matlab 2017
- eVariX 2.6.alpha3
- gcc 4.8.4
- OpenCL 1.2

The options used to compile the generated C++ codes are the default options specified in the configuration file provided in the eVariX™ package.