High Performance Python (from Training at EuroPython 2011)

Release 0.2

Ian Ozsvald (@ianozsvald)

July 24, 2011

CONTENTS

Author:

- Ian Ozsvald (ian@ianozsvald.com)

Version:

- 0.2_improved_in_evenings_over_the_last_few_weeks_20110724

Websites:

- http://IanOzsvald.com (personal)
- http://twitter.com/ianozsvald
- http://MorConsulting.com (my Artificial Intelligence and High Performance Computing consultancy)

Source:

- http://tinyurl.com/europyhpc # zip file of build-as-you-go training src (but may get out of date)
- https://github.com/ianozsvald/EuroPython2011_HighPerformanceComputing # full src for all examples (up to date)
- Use "git clone git@github.com:ianozsvald/EuroPython2011_HighPerformanceComputing.git" to get the source or download a zipped snapshot from the page above
- http://ep2011.europython.eu/conference/talks/experiences-making-cpu-bound-tasks-run-much-faster # slides
- http://www.slideshare.net/IanOzsvald/euro-python2011-high-performance-python # same slides in SlideShare

Questions?

- If you have Python questions then the Python Tutor list is an excellent resource
- If you have questions about a specific library (e.g. pyCUDA) then go to the right user group for the best help
- You can contact me if you have improvements or if you've spotted errors (but I can't help you learn Python, sorry!)

CHAPTER

ONE

TESTIMONIALS FROM EUROPYTHON 2011

- *@ianozsvald does an excellent workshop on what one needs to know about performance and python #europython* @**LBdN**
- *Ozsvald's training about speeding up tasks yesterday was awesome! #europython* @**Mirko_Rossini**
- *PDF from @ianozsvald's High Performance Python workshop http://t.co/TS94l3V It allowed us to make parts of @setjam code 2x faster. Read it!* @**mstepniowski**
- *Yup. I call it "Advanced Toilet Literature" http://lockerz.com/s/115235120* @**emilbronikowski**
- *#EuroPython high performance #Python workshop by @ianozsvald is most excellent! Learned about RunSnakeRun, line profiler, dis module, Cython* @**mstepniowski**
- *@mstepniowski @ianozsvald line profiler is amazing, and such a hidden gem* @**zeeg**
- *Inspired to try out pp after @ianozsvald #EuroPython training* @**ajw007**
- *@ianozsvald's talk on speeding up #python code is high speed itself! #europython* @**snakecharmerb**
- *Don't miss this, Ian's training was terrific! RT @ianozsvald: 43 pages of High Performance Python tutorial PDF written up #europython* @**europython**
- *"@ianozsvald The #Europython2011 HighPerf #python material is absolutely amazing o/ Thanks for that !"* @**BaltoRouberol**
- *@ianozsvald looks great and possibly more content than in the talk! [...]* @**ajw007**
- *First half of the optimization training with @ianozsvald (http://t.co/zU16MXQ) has been fun and really interesting #europython* @**pagles**

Figure 1.1: My happy class at EuroPython 2011

CHAPTER

TWO

MOTIVATION

I ran a 4 hour tutorial on High Performance Python at EuroPython 2011. I'd like to see the training go to more people so I've written this guide. This is based on the official tutorial with some additions, I'm happy to accept updates.

The slides for tutorial are linked on the front page of this document.

If you'd like some background on programming for parallelised CPUs then the Economist has a nice overview article entitled "Parallel Bars" (June 2nd 2011): http://www.economist.com/node/18750706. It doesn't mention CUDA and OpenCL but the comment thread has some useful discussion. GvR gets a name-check in the article.

I'll also give myself a quick plug - I run an Artificial Intelligence consultancy (http://MorConsulting.com) and rather enjoy training with Python.

I'd like to note that this report is a summary of work over many weeks preparing for EuroPython. I didn't perform statistically valid tests, I did however run the timings many times and can vouch for their stability. The goal isn't to suggest that there is "one best way" to do things - I'm showing you several journeys that takes different routes to faster execution times for this problem.

If you're curious to see how the stock CPython interpreter compares to other languages like C and JavaScript then see this benchmark: http://shootout.alioth.debian.org/u32/which-programming-languages-are-fastest.php - you'll note that it does rather poorly (up to 100* slower than C!). It also compares poorly against JavaScript V8 which is dynamically typed and interpreted - much like CPython. Playing with comparisons against the JavaScript V8 examples got me started on this tutorial.

To see how CPython, PyPy, ShedSkin, IronPython and Jython compare to other languages (including C and V8) see this benchmark: http://attractivechaos.github.com/plb/ - as shown we can make Python run to 2-6* slower than C with little effort, and the gap with C is shrinking all the time. The flipside of course is that developing with Python is far faster than developing with C!

2.1 Changelog

- v0.2 July 2011 with longer write-ups, some code improvements
- v0.1 earliest release (rather draft-y) end of June 2011 straight after EuroPython 2011

2.2 Credits

- Thanks to my class of 40 at EuroPython for making the event so much fun :-)
- The EuroPython team for letting me teach, the conference was a *lot* of fun
- Mark Dufour and ShedSkin forum members

- Cython team and forum members
- Andreas Klöckner for pyCUDA
- Everyone else who made the libraries that make my day job easier

2.3 Other talks

The following talks were all given at EuroPython, many have links to slides and videos:

- "Debugging and profiling techniques" by Giovanni Bajo: http://ep2011.europython.eu/conference/talks/debugging-and-profiling-techniques
- "Python for High Performance and Scientific Computing" by Andreas Schreiber: http://ep2011.europython.eu/conference/talks/python-for-high-performance-and-scientific-computing
- "PyPy hands-on" by Antonio Cuni - Armin Rigo: http://ep2011.europython.eu/conference/talks/pypy-hands-on
- "Derivatives Analytics with Python & Numpy" by Yves Hilpisch: http://ep2011.europython.eu/conference/talks/derivatives-analytics-with-python-numpy
- "Exploit your GPU power with PyCUDA (and friends)" by Stefano Brilli: http://ep2011.europython.eu/conference/talks/exploit-your-gpu-power-with-cuda-and-friends
- "High-performance computing on gamer PCs" by Yann Le Du: http://ep2011.europython.eu/conference/talks/high-performance-computing-gamer-pcs
- "Python MapReduce Programming with Pydoop" by Simone Leo: http://ep2011.europython.eu/conference/talks/python-mapreduce-programming-with-pydoop
- "Making CPython Fast Using Trace-based Optimisations" by Mark Shannon: http://ep2011.europython.eu/conference/talks/making-cpython-fast-using-trace-based-optimisations

CHAPTER

THREE

THE MANDELBROT PROBLEM

In this tutorial we'll be generating a Mandelbrot plot, we're coding mostly in pure Python. If you want a background on the Mandelbrot set then take a look at WikiPedia.

We're using the Mandelbrot problem as we can vary the complexity of the task by drawing more (or less) pixels and we can calculate more (or less) iterations per pixel. We'll look at improvements in Python to make the code run a bit faster and then we'll look at fast C libraries and ways to convert the code directly to C for the best speed-ups.

This task is embarrassingly parallel which means that we can easily parallelise each operation. This allows us to experiment with multi-CPU and multi-machine approaches along with trying NVIDIA's CUDA on a Graphics Processing Unit.

This is the output we're after:

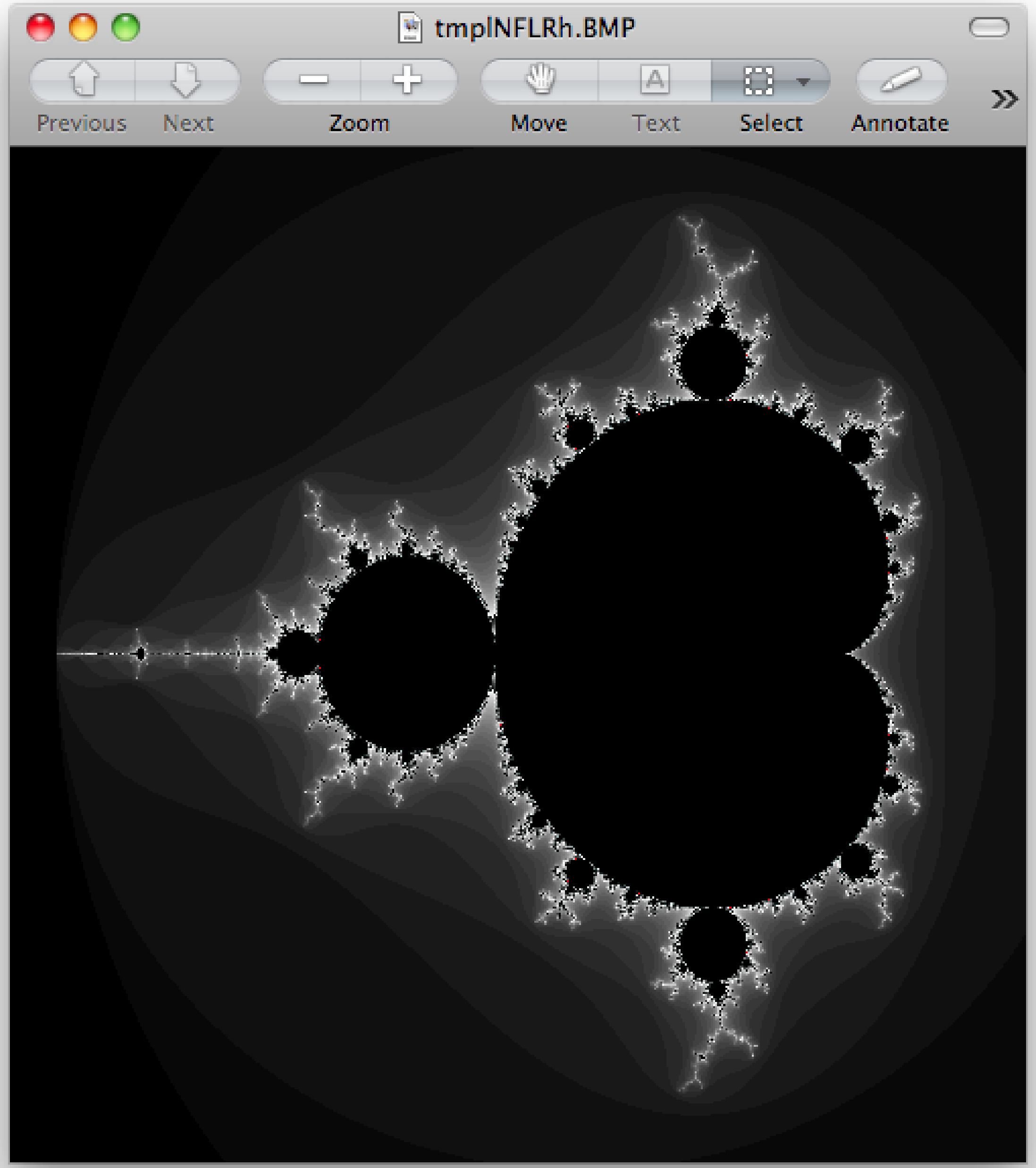

Figure 3.1: A 500 by 500 pixel Mandelbrot with maximum 1000 iterations

CHAPTER

FOUR

GOAL

In this tutorial we're looking at a number of techniques to make CPU-bound tasks in Python run much faster. Speed-ups of 10-500* are to be expected if you have a problem that fits into these solutions.

In the results further below I show that the Mandelbrot problem can be made to run 75* faster with relatively little work on the CPU and up to 500* faster using a GPU (admittedly with some C integration!).

Techniques covered:

- Python profiling (cProfile, RunSnake, line_profiler) - find bottlenecks
- PyPy - Python's new Just In Time compiler
- Cython - annotate your code and compile to C
- numpy integration with Cython - fast numerical Python library wrapped by Cython
- ShedSkin - automatic code annotation and conversion to C
- numpy vectors - fast vector operations using numpy arrays
- NumExpr on numpy vectors - automatic numpy compilation to multiple CPUs and vector units
- multiprocessing - built-in module to use multiple CPUs
- ParallelPython - run tasks on multiple computers
- pyCUDA - run tasks on your Graphics Processing Unit

4.1 MacBook Core2Duo 2.0GHz

Below I show the speed-ups obtained on my older laptop and later a comparitive study using a newer desktop with a faster GPU.

These timings are taken from my 2008 MacBook 2.0GHz with 4GB RAM. The GPU is a 9400M (very underpowered for this kind of work!).

We start with the original `pure_python.py` code which has too many dereference operations. Running it with PyPy and no modifications results in an easily won speed-up.

Tool	Source	Time
Python 2.7	pure_python.py	49s
PyPy 1.5	pure_python.py	8.9s

Next we modify the code to make `pure_python_2.py` with less dereferences, it runs faster for both CPython and PyPy. Compiling with Cython doesn't give us much compared to using PyPy but once we've added static types and expanded the `complex` arithmetic we're down to 0.6s.

Cython with `numpy` vectors in place of `list` containers runs even faster (I've not drilled into this code to confirm if code differences can be attributed to this speed-up - perhaps this is an exercise for the reader?). Using ShedSkin with no code modificatoins we drop to 12s, after expanding the `complex` arithmetic it drops to 0.4s beating all the other variants.

Be aware that on my MacBook Cython uses `gcc 4.0` and ShedSkin uses `gcc 4.2` - it is possible that the minor speed variations can be attributed to the differences in compiler versions. I'd welcome someone with more time performing a strict comparison between the two versions (the 0.6s, 0.49s and 0.4s results) to see if Cython and ShedSkin are producing equivalently fast code.

Do remember that more manual work goes into creating the Cython version than the ShedSkin version.

Tool	Source	Time	Notes
Python 2.7	pure_python_2.py	30s	
PyPy 1.5	pure_python_2.py	5.7s	
Cython	calculate_z.pyx	20s	no static types
Cython	calculate_z.pyx	9.8s	static types
Cython	calculate_z.pyx	0.6s	+expanded math
Cython+numpy	calculate_z.pyx	0.49s	uses numpy in place of lists
ShedSkin	shedskin1.py	12s	as pure_python_2.py
ShedSkin	shedskin2.py	0.4s	expanded math

Compare CPython with PyPy and the improvements using Cython and ShedSkin here:

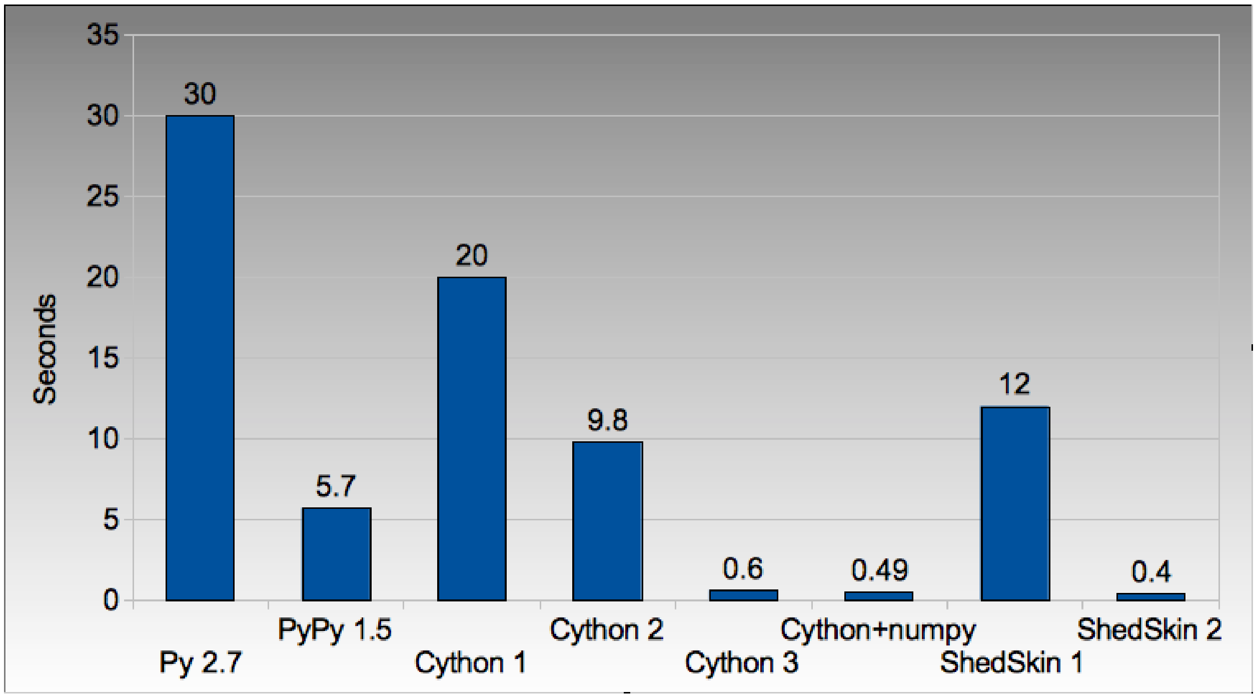

Figure 4.1: Run times on laptop for Python/C implementations

Next we switch to vector techniques for solving this problem. This is a less efficient way of tackling the problem as we can't exit the inner-most loops early, so we do *lots* of extra work. For this reason it isn't fair to compare this approach to the previous table. Results within the table however can be compared.

`numpy_vector.py` uses a straight-forward vector implementation. `numpy_vector_2.py` uses smaller vectors that fit into the MacBook's cache, so less memory thrashing occurs. The `numexpr` version auto-tunes and auto-vectorises the `numpy_vector.py` code to beat my hand-tuned version.

The pyCUDA variants show a `numpy`-like syntax and then switch to a lower level C implementation. Note that the 9400M is restricted to single precision (`float32`) floating point operations (it can't do `float64` arithmetic like the rest of the examples), see the GTX 480 result further below for a `float64` true comparison.

Even with a slow GPU you can achieve a nice speed improvement using pyCUDA with `numpy`-like syntax compared to executing on the CPU (admittedly you're restricted to `float32` math on older GPUs). If you're prepared to recode the core bottleneck with some C then the improvements are even greater.

Tool	Source	Time	Notes
numpy	numpy_vector.py	54s	uses vectors rather than lists
numpy	numpy_vector_2.py	42s	tuned vector operations
numpy	numpy_vector_numexpr.py	19.1s	'compiled' with numexpr
pyCUDA	pycuda_asnumpy_float32.py	10s	using old/slow 9400M GPU
pyCUDA	pycuda_elementwise_float32.py	1.4s	as above but core routine in C

The reduction in run time as we move from CPU to GPU is rather obvious:

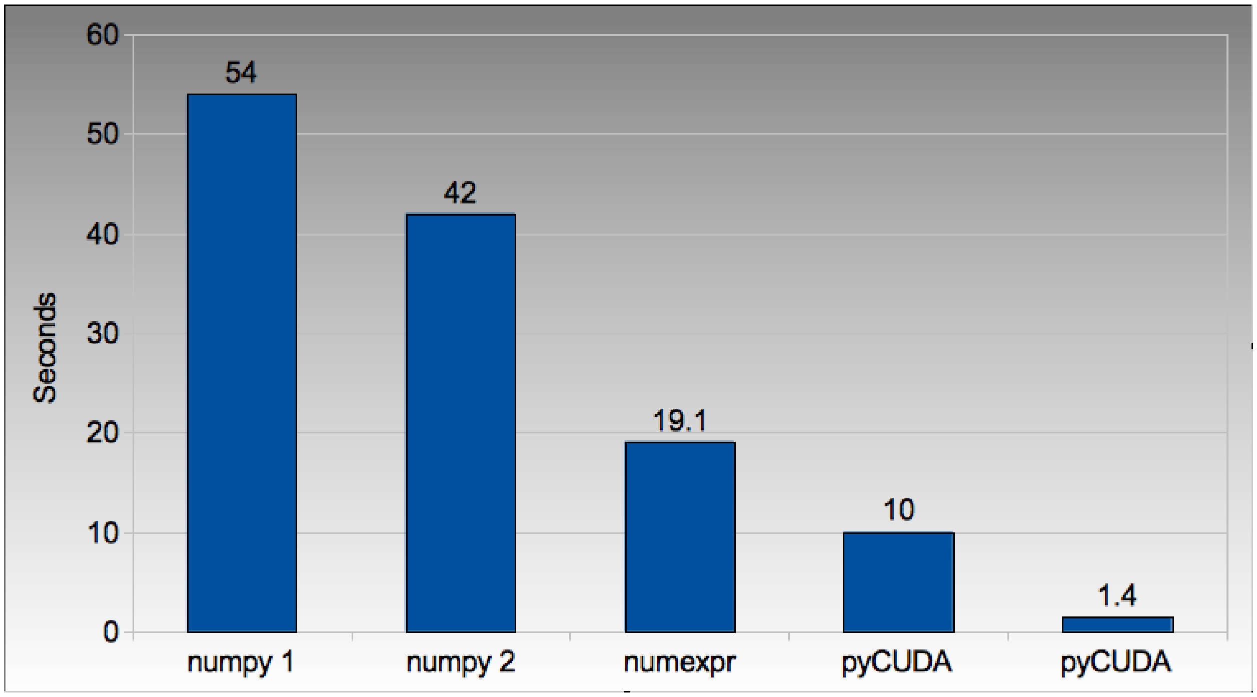

Figure 4.2: Run times on laptop using the vector approach

Finally we look at using multi-CPU and multi-computer scaling approaches. The goal here is to look at easy ways of parallelising to all the resources available around one desk (we're avoiding large clusters and cloud solutions in this report).

The first result is the `pure_python_2.py` result from the second table (shown only for reference). `multi.py` uses the `multiprocessing` module to parallelise across two cores in my MacBook. The first ParallelPython example works exaclty the same as `multi.py` but has lower overhead (I believe it does less serialising of the environment). The second version is parallelised across three machines and their CPUs.

The final result uses the 0.6s Cython version (running on one core) and shows the overheads of splitting work and serialising it to new environments (though on a larger problem the overheads would shrink in comparison to the savings made).

Tool	Source	Time	Notes
Python 2.7	pure_python_2.py	30s	original serial code
multiprocessing	multi.py	19s	same routine on two cores
ParallelPython	parallelpython_pure_python.py	18s	same routine on two cores
ParallelPython	parallelpython_pure_python.py	6s	same routine on three machines
ParallelPython	parallelpython_cython_pure_python.py	1.4s	0.6s cython version on two cores

The approximate halving in run-time is more visible in the figure below, in particular compare the last column with Cython 3 to the results two figures back.

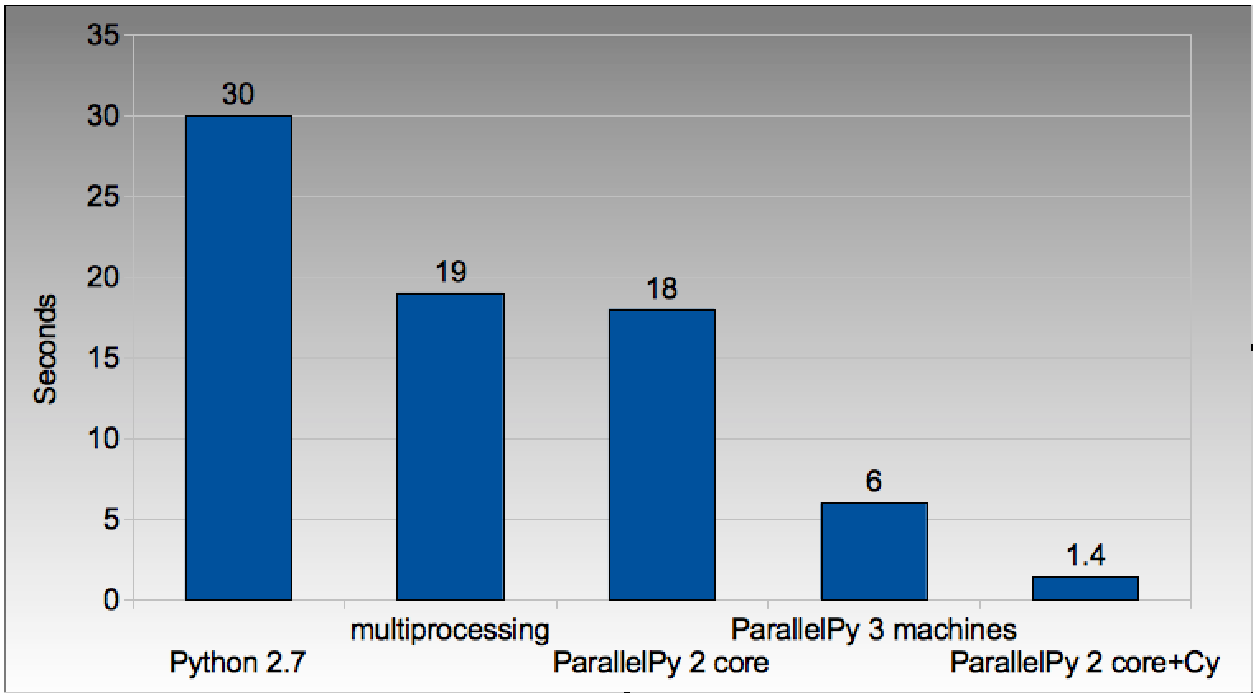

Figure 4.3: Run times on laptop using multi-core approaches

4.2 2.9GHz i3 desktop with GTX 480 GPU

Here I've run the same examples on a desktop with a GTX 480 GPU which is far more powerful than my laptop's 9400M, it can also support double-precision arithmetic. The GTX 480 was the fastest consumer-grade NVIDIA GPU during 2010, double precision arithmetic is slower than single precision arithmetic (the double-precision in the scientific C series was even faster, with a big price hike).

The take-home message for the table below is that re-coding a vector operation to run on a fast GPU may bring you a 10* speed-up with very little work, it may bring you a 500* speed-up if you're prepared to recode the heart of the routine in C.

Tool	Source	Time	Notes
Python 2.7	pure_python_2.py	35s	(slower than laptop - odd!)
pyCUDA	pycuda_asnumpy_float64.py	3.5s	GTX480 with float64 precision
pyCUDA	pycuda_elementwise_float64.py	0.07s	as above but core routine in C

The 500* speed-up is somewhat more visible here:

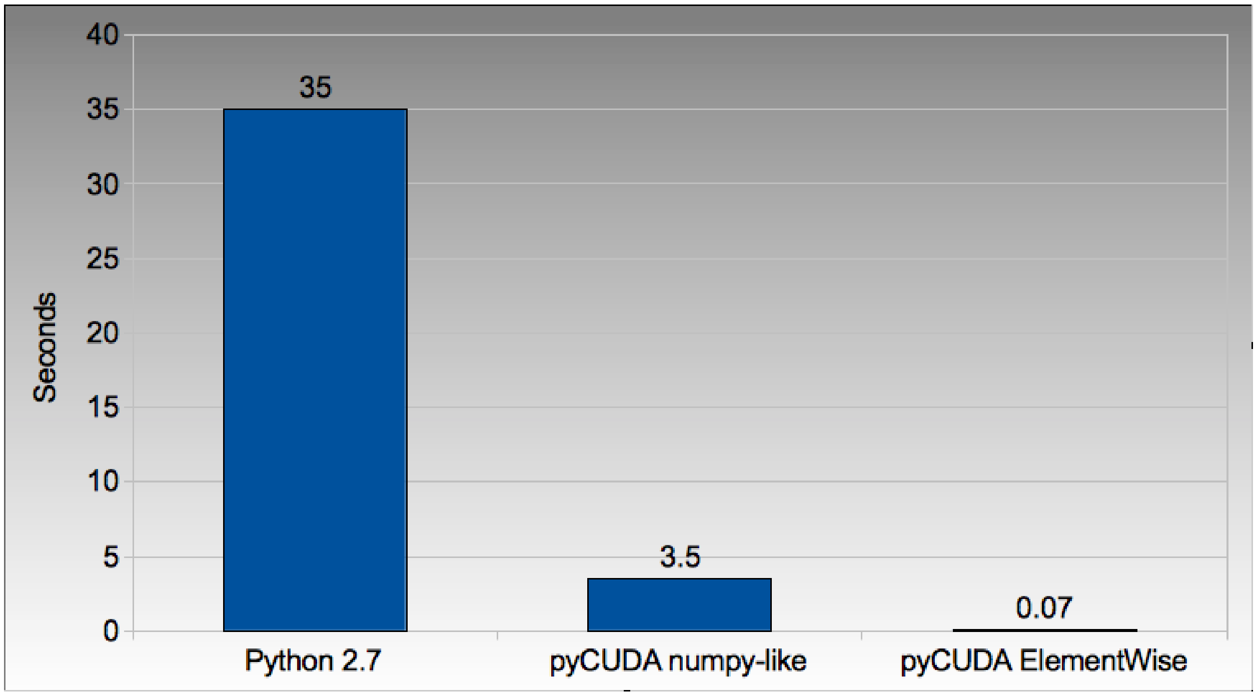

Figure 4.4: Run times on i3 desktop with GTX 480 GPU

CHAPTER

FIVE

USING THIS AS A TUTORIAL

If you grab the source from https://github.com/ianozsvald/EuroPython2011_HighPerformanceComputing (or Google for “ianozsvald github”) you can follow along. The github repository has the full source for all these examples (and a few others), you can start with the `pure_python.py` example and make code changes yourself.

You probably want to use `numpy_loop.py` and `numpy_vector.py` for the basis of some of the `numpy` transformations.

CHAPTER

SIX

VERSIONS AND DEPENDENCIES

The tools depend on a few other libraries, you'll want to install them first:

- CPython 2.7.2
- line_profiler 1.0b2
- RunSnake 2.0.1 (and it depends on wxPython)
- PIL (for drawing the plot)
- PyPy pypy-c-jit-45137-65b1ed60d7da-osx64 (from the nightly builds around July 2011)
- Cython 0.14.1
- Numpy 1.5.1
- ShedSkin 0.8 (and this depends on a few C libraries)
- NumExpr 1.4.2
- pyCUDA 0.94 (HEAD as of June 2011 and it depends on the CUDA development libraries, I'm using CUDA 4.0)