Hochschule
für Technik
Stuttgart

High-performance computing / Scientific Computing

Table of contents

Hochschule
für Technik
Stuttgart

Document information

Document Type	Dokumentation Anwendung
Author	phelstab
Target groups	-
Status	work in progress
Classification of Information	open source
Location	GitHub

Change History

Version	Date	Author	Change
1.1	10.01.2023	phelstab	code formatting + docs
1.0	11.12.2022	phelstab	init

References and helpful links

Title	DOI	Source
Cython Docs	-	stackoverflow.com/questions/62249186/how-to-use-prange-in-cython
Intel Docs to Cython	-	www.intel.com/content/www/us/en/developer/articles/technical/thread-parallelism-in-cython.html
Cython Blog post	-	nealhughes.net/parallelcomp2/

Chapter 1. Motivation

Die Motivation für das Modul HPC kam unter anderem bezüglich meines Bachelor Thesis Themas, das sich mit Agenten-basierten diskreten Event Simulationen zur Untersuchung von auftretenden deterministischen Anomalien auf Referenzmärkten beschäftigt. Marcos Lopez de Prado, ein quantitative finance Professor an der University of California, San Diego, spricht in seinem Buch "Advances in Financial Machine Learning" davon, dass quantitative finance als eines der am schnellsten an Bedeutung gewinnenden Disziplinen in der Finanzwelt angesehen wird, da es immer weniger menschliche Entscheidungen in der Finanzwelt gibt. Firmen wie Citadel, Optiver, Two Sigma, Jane Street, etc. sind nur einige der Firmen, die sich mit quantitative finance beschäftigen um damit Milliardengewinne zu erzielen (Einblick). Dabei ist High-Frequency Trading (HFT) ein sehr wichtiger Teil der quantitative finance und wird von Market Maker und Hedgefunds genutzt, um unter anderem Arbitragegewinne durch Zeitvorteile zu erzielen (gutes Buch bezüglich dieser Revolution Flash Boys). Das Wettrennen schneller zu sein als andere durch performanteren Code (see HPC bei Optiver) oder embedded programming von Strategien auf ASIC's, finde ich hoch spannend, da es meiner Meinung bei vielen Unternehmen als nicht mehr als allzu wichtig gesehen wird, da Leistung günstig geworden ist.

Monte Carlo Simulationen sind ein wichtiges Tooling für Quants und werden vor allem im Risikomanagement verwendet, da sie es ermöglichen, Risiken simulativ zu quantifizieren und schnell Entscheidungen getroffen werden können, besonders in hochvolatilen Märkten wie es der Fall Navinder Singh Sarao (see Flash Crash) 2010 gezeigt hat. Dabei ist es besonders wichtig, dass die Monte Carlo Simulation eine hochperformante Runtime erzielt, um auch auf kleinen Zeitreihen Einblicke zu ermöglichen.

Die weitere Motivation für mich war DeepMind's AlphaTensor mit dem sie den Strassen Algorithmus nach 50 Jahren als performantesten Algorithmus zur Matrixmultiplikation ablösten und dass statt eines mathematischen Beweises über eine Reinforcement Learning Simulation (see DeepMind, GitHub, Nature).

Chapter 2. Definition und Problemstellung

Python bietet die Möglichkeit, schnell Softwarelösungen zu entwickeln und Daten mit Bibliotheken wie matplotlib und plotly einfach visualisieren zu können. Allerdings ist Python nicht für hochperformante Anwendungsbereiche (GIL, lässt nur ein Thread parallel ausführen) und somit auch nicht für z. B. Einsatzbereiche wie HFT Analysen und Strategien geeignet. Durch die Nutzung von Cython bekommt man die Vorteile aus beidem. Interpretiertes Python und Hardwarenahes C zusammengebracht und somit die Laufzeiteffizienz gesteigert. Über ein paar Umwege ist zusätzlich auch die Nutzung von HPC-Bibliotheken und Paradigmen wie z. B. OpenMP, OpenCL, SIMD-Instruktionen, etc. möglich.

Als Beispiel möchte ich mit diesem Projekt diese Vorteile nutzen, um die Performance von Monte Carlo Simulationen durch hardwarenahes C und HPC-Paradigmen zu steigern. Dies möchte ich in der Theorie ermöglichen, indem ich den compute intensiven Teil auslagere in hochperformanten C code und anschließend die simulierten Daten zur Visualisierung in Python zurückgebe.

2.1. Meine hardware specs

MacBook Pro 13" 2018

```
# CPU info
sysctl -n machdep.cpu.brand_string
#Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz
```

Meine CPU specs (Source)

Anzahl der Kerne 4
Anzahl der Threads 8
Max. Turbo-Taktfrequenz 3.80 GHz
Intel® Turbo-Boost-Technik 2.0 Taktfrequenz‡ 3.80 GHz
Grundtaktfrequenz des Prozessors 2.30 GHz
Cache 6 MB Intel® Smart Cache
Bus-Taktfrequenz 4 GT/s
Verlustleistung (TDP) 28 W
Frequenz der konfigurierbaren TDP-down 1.10 GHz
Konfigurierbare TDP-down 20 W
Befehlssatzerweiterungen Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2
Innovative technische Funktionen
Intel® Optane™ Speicher unterstützt ‡ Ja
Intel® Speed Shift Technology Ja
Intel® Turbo-Boost-Technik‡ 2.0
Intel® Hyper-Threading-Technik ‡ Ja
Intel® TSX-NI Nein
Intel® 64 ‡ Ja
Befehlssatz 64-bit
Befehlssatzerweiterungen Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2

Intel® My WiFi Technologie Ja
Ruhezustände Ja
Erweiterte Intel SpeedStep® Technologie Ja
Thermal-Monitoring-Technologien Ja
Intel® Flex-Memory-Access Ja
Intel® Identity-Protection-Technik ‡ Ja

2.2. Python setup

```
# Python 3.9.13 recommended
# Update pip and create venv with example name venv_lambda
python -m pip install --upgrade pip
python -m venv venv_lambda

# Activate venv on Linux
source lambda/bin/activate

# Activate venv on Windows
.\lambda\Scripts\activate

# Activate venv on mac and without and with fishshell
source lambda/bin/activate
. lambda/bin/activate.fish

# Install libs
pip install -r requirements.txt
```

2.3. C setup

```
# Test if openmp is installed (unix only)
gcc -fopenmp multi_test.c -o multi_test
./multi_test
```

2.4. Commands

```
# Decompiler use flag --cplus, when compiling with cpp headers
cython -a x.pyx
# Compile .pyx cython
python setup.py build_ext --inplace
# Run our test
python main.py
```

2.5. Asciidoc

```
# Asciidoc Vorlage erstellt von @phelstab
# Erstellung eines PDFs aus der README.adoc
brew install asciidoctor
asciidoctor-pdf README.adoc
```

Chapter 3. Lösungsansatz

Mein Lösungsansatz ist es, die Monte Carlo Simulation zunächst sehr quick and dirty in Python zu implementieren, um es anschließend im Step-by-Step Ansatz optimieren zu können. Zunächst habe ich eine Architektur entwickelt, in dem sich Monte-Carlo Simulationen basierend auf der Volatilität der letzten N Tagen eines Wertpapieres (daten von yahoo finance) mit dem Startwert den letzten gehandelten Preis (closing price) berechnen und plotten lassen.

Figure 1. Architektur der Anwendung und Simulation

Die Grundlegende Formel einer Monte Carlo Simulation lautet (compute intensiver teil):

```
FOR x in RANGE(0, num_simulations, 1)::
    FOR y in RANGE(0, (num_days - 1), 1)::
        IF y == 0::
            price[0] = last_traded_price * (1 + (random.normal(0, volatility)))
        ELSE::
            price[y] = price[y-1] * (1 + (random.normal(0, volatility)))
        ENDIF::
    ENDFOR::
ENDFOR::
```

Dieser Teil ist der compute intensive Teil der Simulation und muss vollständig in C geschrieben sein, sodass verschiedene HPC-Paradigmen angewendet werden können, dadurch ist man zunächst einmal verpflicht, alle bekannten Pythonbibliotheken wie Numpy, Math, Time, etc. herunterzubrechen, um sie in C erneut zu implementieren. Die innere Schleife ist deterministisches Chaos und kann daher nicht parallelisiert werden.

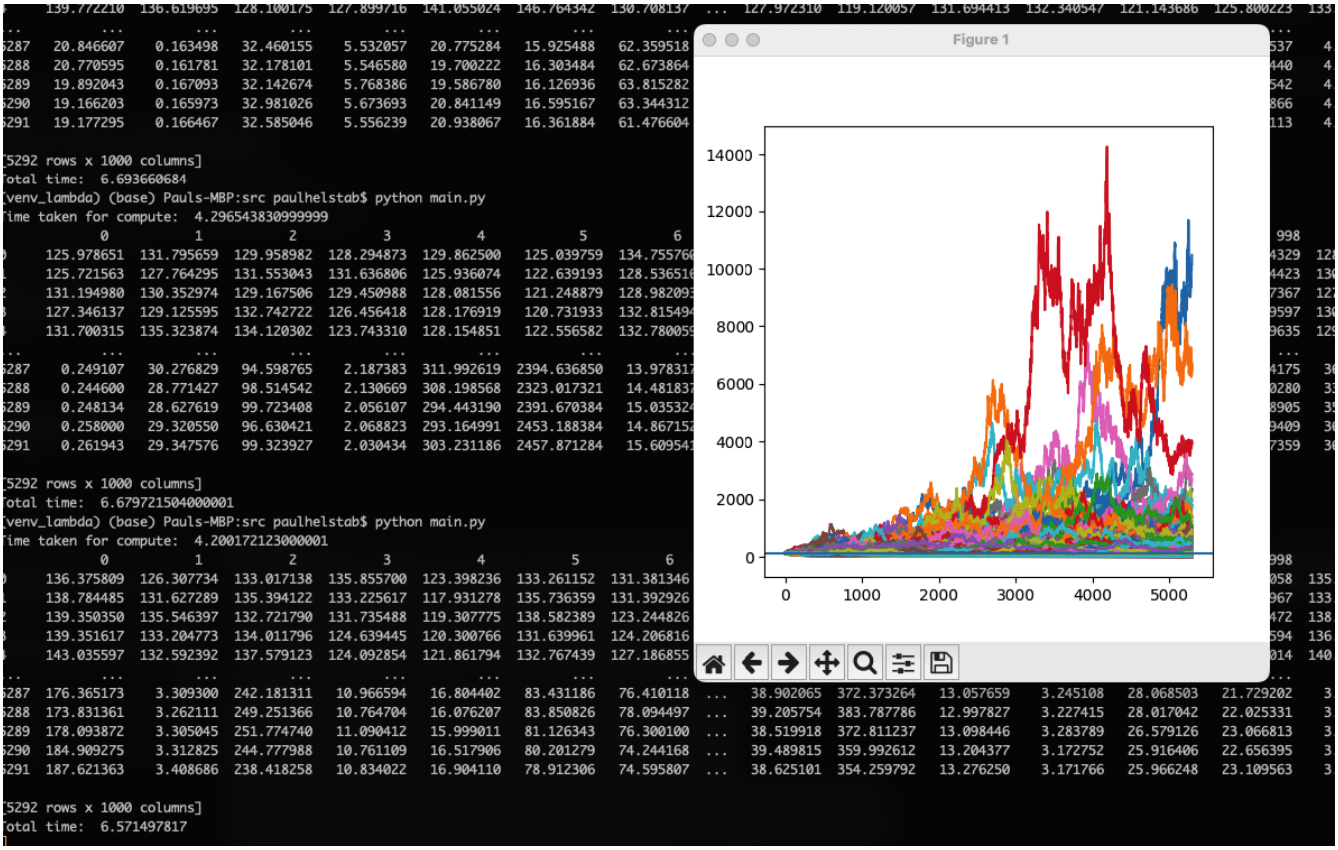

Figure 2. Durchlauf, Datenpunkte und plot der Simulation

3.1. Benchmarking

Als Benchmarking definiere ich die reine computation Zeit der Monte Carlo Simulation. Das bedeutet, ich übergebe an den Cython Wrapper die Anzahl der Tage, die simuliert werden sollen, die Anzahl der Simulationen, den letzten gehandelten Preis und eine Liste mit allen Preisen der letzten N Tage.

Sobald diese Parameter übergeben wurden, wird die Monte Carlo Simulation ausgeführt und die computation time gemessen, bis ein 2-dimensionaler Array mit den Simulationen von der Runtime Logik zurückgegeben wird und in Python mit Matplotlib geplotted werden kann.

Chapter 4. Implementation

Die ersten Schritte die getan wurden:

1. Entwicklung des Wrappers für die Monte Carlo Simulation in Python
2. Entwicklung der Monte Carlo Simulation in Python
3. Python optimierung
4. Cypthon wrapper für die Monte Carlo Simulation
5. Precompiling des Python codes in Cython über die .pyx (anschließend Benchmarking)
6. Python funktionen in natives C überführen

Als ich dachte, den Code nahezu komplett in C überführt zu haben, tauchte das erste Problem mit der Berechnung des Zufalls auf. Da die Berechnung des Zufalls in C sehr komplex ist, habe ich mich dazu entschieden, die Berechnung des Zufalls so weit es geht, herunter zu brechen. Leider musste ich dabei auf die Systemnanosekunden Funktion von Python zurückgreifen, da ich keine andere Möglichkeit gefunden habe, um saubere Seeding für die Zufallszahlengenerierung hochperformant zu erhalten (meine C-Implementierung war zu langsam).

Ein Versuch, den Zufall über CPP Header zu generieren, war leider nicht teilweise erfolgreich obwohl alle Funktionen sauber implementiert waren, geriet ich an irgendeiner Stelle in einen Infinite Loop und der Rechner freezed (ist als Codeleiche noch zu finden und kann gerne ausprobiert werden:D). Ich denke, dass ich dort allerdings sehr nah am Ziel war, allerdings meine CPP Kenntnisse noch nicht ausgereicht hatten, um das Problem in nicht wochenlanger Arbeit gelöst zu bekommen. Dennoch war es mir möglich, über einen Workaround auf CPP Code in Cython zurückzugreifen, was für zukünftige Projekte sehr hilfreich sein wird und sich mehr Zeit ergibt, sich damit ausgiebig zu beschäftigen.

Da der Code allerdings vollständig in C überführt werden muss, um den GIL deaktivieren zu können, musste ich den ersten Kompromiss eingehen, die Zufälle local vorzugenerieren um diese dann im HPC Part für die Berechnung verwenden zu können. Dies kostet mich allerdings num_days * num_simulations an Iterationen.

Anschließend wollte ich die Performance der Monte Carlo Simulation durch die Nutzung von HPC-Paradigmen und Frameworks steigern. Dazu zähle ich in der Theorie z. B.:

- Weitere C/CPP code optimierung durch
  - Obsoleszieren von unnötigen Kopiervorgängen
  - Obsoleszieren von unnötigen Schleifen
  - Obsoleszieren von unnötigen Funktionen
  - Obsoleszieren von unnötigen Variablen
- OpenMP (Cython Cython.parallel Lib)
  - Parallelierung von Schleifen
    Konnte erfolgreich umgesetzt werden, allerdings kaum Performance Vorteile aufgrund von keiner verbesserten memory performance.

- OpenCL (PyOpenCL)
    - Compute intensive Berechnungen auf der GPU (Memory vorteile ausnutzen)
      Erfolgreich umgesetzt, allerdings hatte ich mir single precision Probleme bei der Ergebnisqualität starke Probleme, daher läuft das ganze auf double precision und ist daher unnötige Arbeit gewesen, da wie wir Wissen, die meisten GPUs keine double precision unterstützen.
- SIMD-Intrinsics
    - Berechnungen in Vektoren (z. B. 2x float oder 4x float) um die Performance zu steigern

Die verwendung von SIMD-Intrinsics ist an vielen stellen als Codeleiche zu finden. Allerdings gab es für mich bei diesem Ansatz keine sinvolle Möglichkeit SIMD-Intrinsics einzusetzen, da vektorisierung an vielen punkten keine performance Vorteil ergeben hätte da man die Values wieder aufwendig in Schleifen vorbereiten hätten müssen.

Chapter 5. Bewertung

Bewertung des Ansatzes und der performance-limitierenden Faktoren (1-2 Seiten)

Besonders interessant war zu sehen das bestimmte Python Bibliotheken zu denen auch weit verbreitete Bibliotheken wie Numpy, Math etc. gehören ineffizient sind und sich durch Hardwarenahen C-code ersetzen lassen.

Ergebnisse der definierten HPC paradigmen:

Table 1. Ergebnisstufen der optimierung

Optimierungsstufe	Compute Zeit	Relational zum Ursprung
Interpretierter Python code	4.2 - 4.5 sec	100%
Precompiled Cython code unoptimiert	3.6 - 3.8 sec	80%
Sequenzielles optimiertes C (80-90% C)	1.0 - 1.2 sec	25%
Optimiertes C (80-90%) + OMP	0.9 - 1 sec	22%
Optimiertes C (80-90%) + OpenCL auf CPU (double precision)	2.5 - 2.6 sec	60%
Optimiertes C (80-90%) + OpenCL auf CPU (single precision)	Ergebnis nicht aussagekräftig	-
Optimiertes C (80-90%) + OpenCL auf GPU (single precision)	Ergebnis nicht aussagekräftig	-

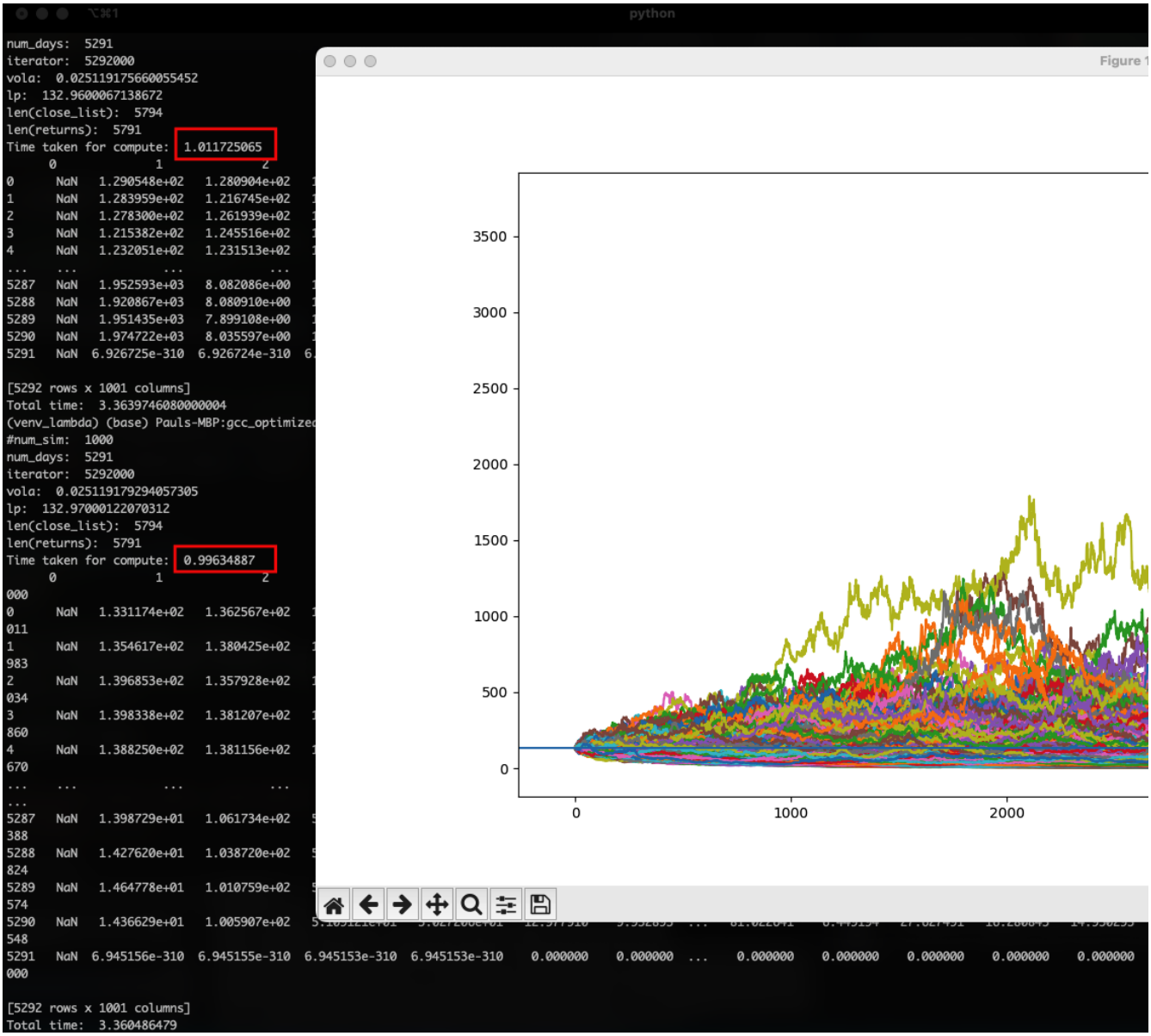

Figure 3. C optimierung seriell

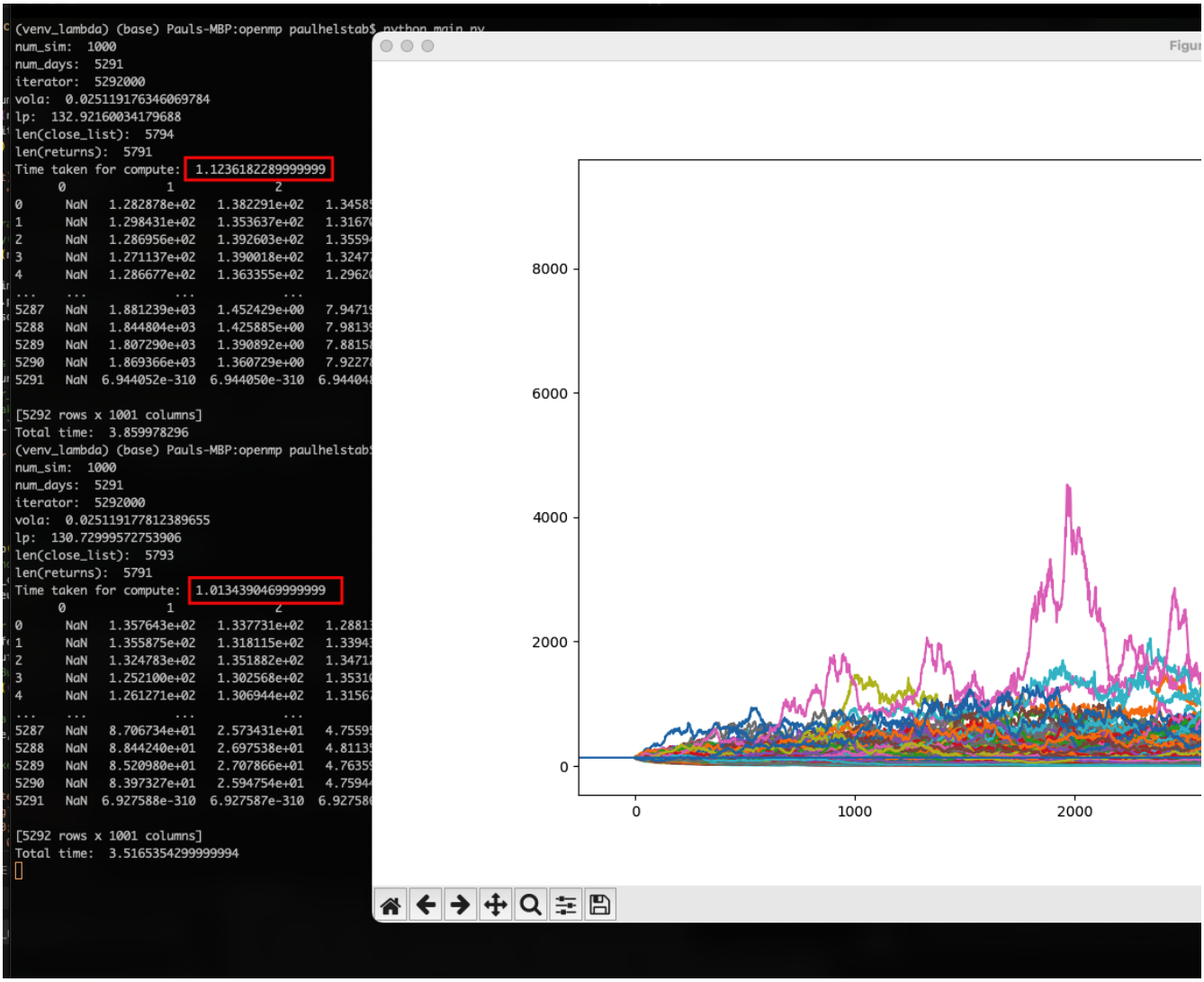

Figure 4. C optimierung parallel

Hier sieht man leider das Ergebnis der nach single precision (float32) formatierten Simulation. Ich kann es aus meiner Sicht nicht erklären und müsste stand jetzt tiefere Recherche betreiben, um die Ursache zu finden.

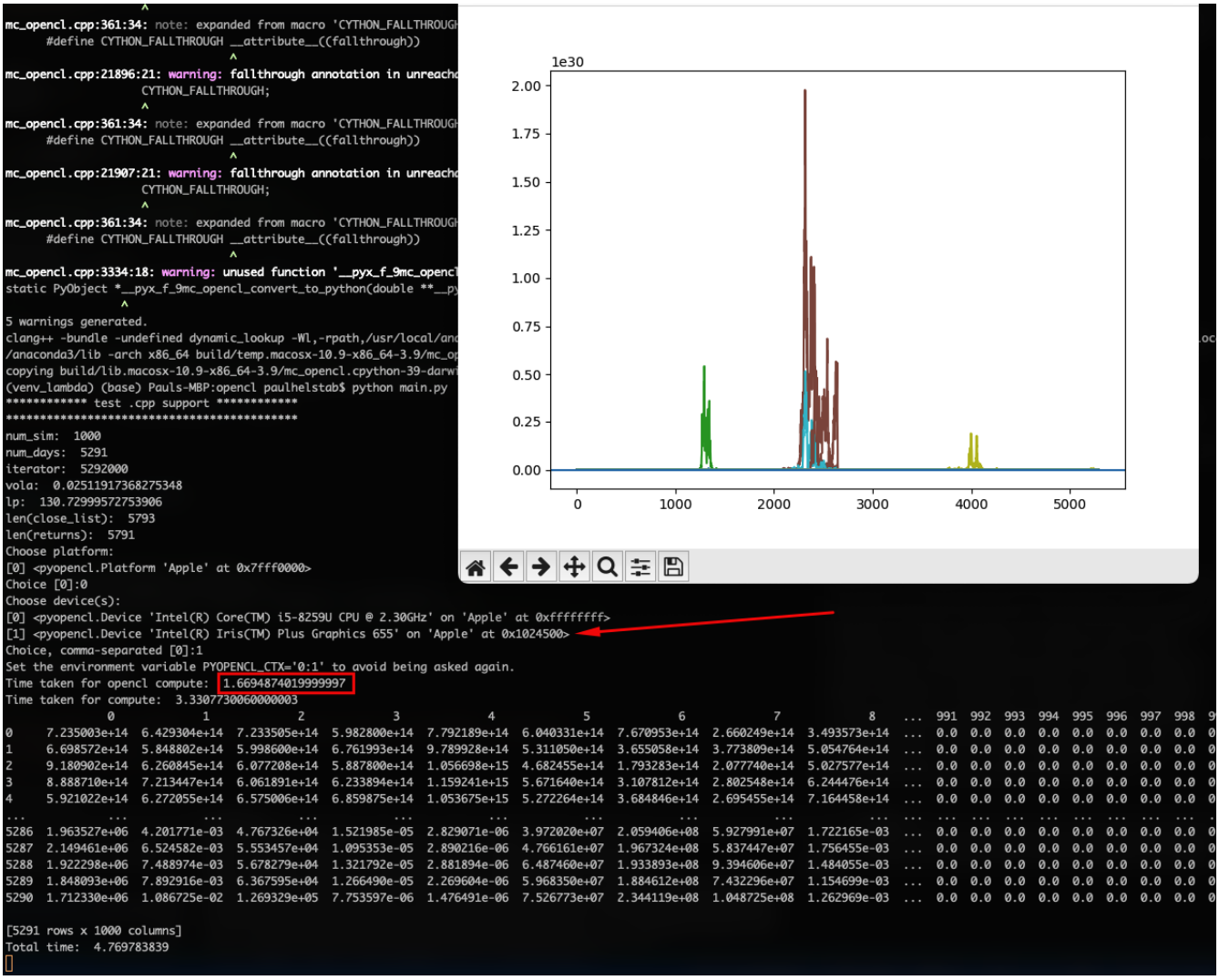

Figure 5. OpenCL single precision problem

Hier sieht man wenig überraschend, dass es teuer ist, wenn OpenCL zunächst Buffern muss und die Daten auf die CPU kopiert. Es ist ein wenig ärgerlich bezüglich des Problems single precision problems, ich hätte gerne die Performance mit der GPU beobachtet, da ich denke, dass hier die Performance für diese Simulation vor allem im Hinblick auf Skalierung den größten Vorteil hätte.

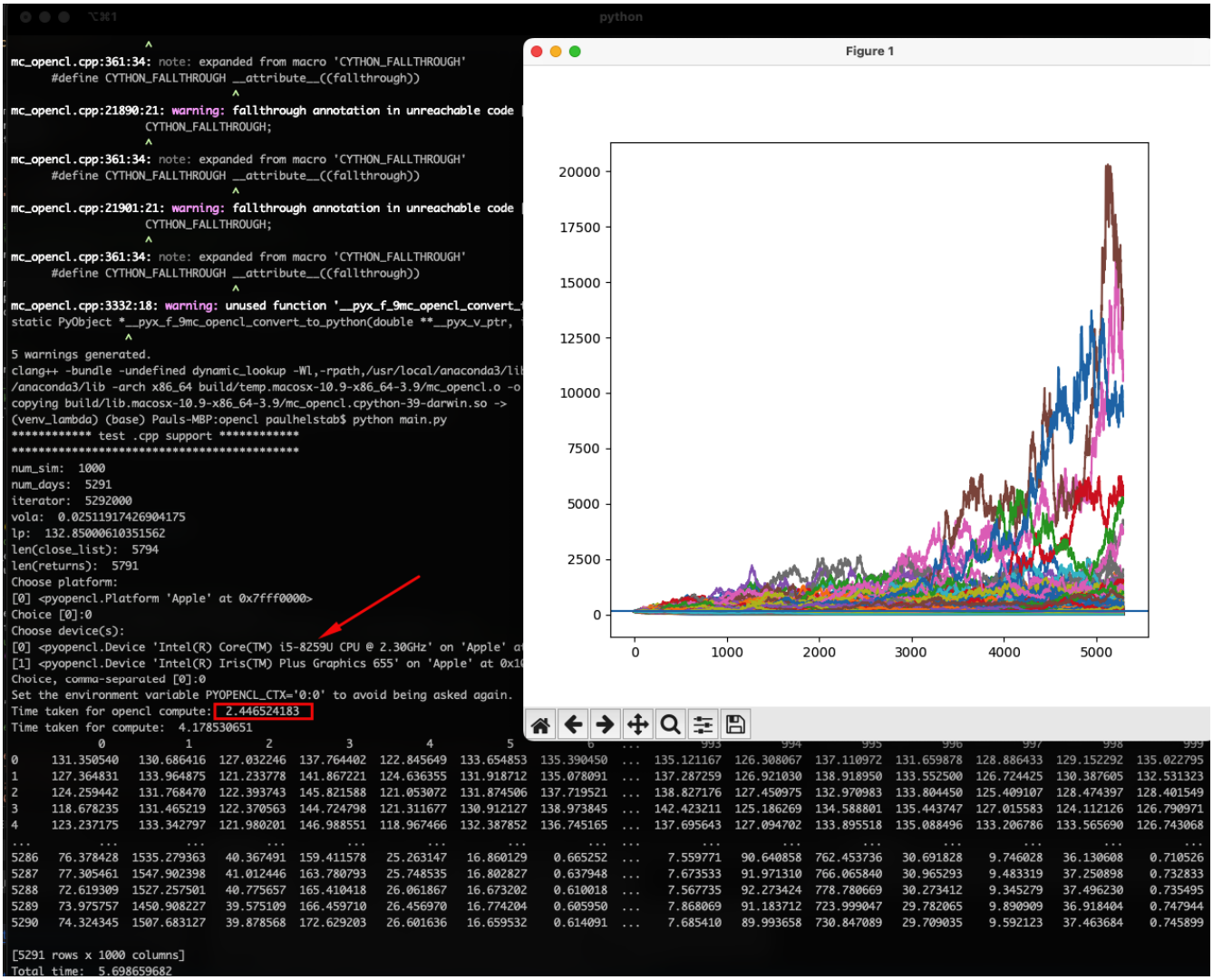

Figure 6. OpenCL double precision auf CPU

Hier sieht man die Cython Debugger Ansicht als HTML. Gelb markiert Python Code, der interpretiert werden muss. Falls gelb schwach markiert ist, so wird ein geringer Teil in C interpretiert. Man sieht hier, dass der compute Part vollständig in C kompiliert wurde und aktuell sequenziell durchiteriert.

```
+103: cpdef run(close_list, int num_sim, int num_days, double last_price):
+104:     cdef long int* timestamps = <long int*> malloc((num_sim*num_days+100) * sizeof(long int))
+105:     cdef array.array n_close_list = array.array('d', close_list)
+106:     returns = get_pct_change(n_close_list)
+107:     cdef double vola = get_standard_deviation(returns)
+108:     cdef double lp = last_price
 109:     cdef int x, y
+110:     cdef signed int iterator = num_sim*num_days
+111:     cdef double* randoms = <double*>malloc((iterator * sizeof(double)) + 15)
+112:     cdef double** sim_array = <double**>malloc((num_sim * sizeof(double*)) + 15)
 113:
+114:     for x in range(num_sim):
+115:         sim_array[x] = <double*>malloc((num_days * sizeof(double)) + 15)
 116:
+117:     for i in range(num_sim*num_days+20):
+118:         timestamps[i] = int(time.perf_counter() * 1e9)
 119:
 120:     cdef double rnd
+121:     cdef double pi = 3.14159265358979323846
+122:     for x in range(num_sim*num_days):
+123:         srand48(timestamps[x])
+124:         randoms[x] = 0 + vola * sqrt(-2.0 * log(drand48())) * cos(2 * pi * drand48())
 125:
 126:
 127:
+128:     print("num_sim: ", num_sim)
+129:     print("num_days: ", (num_days - 1))
+130:     print("iterator: ", iterator)
+131:     print("vola: ", vola)
+132:     print("lp: ", lp)
+133:     print("len(close_list): ", len(close_list))
+134:     print("len(returns): ", len(returns))
 135:
 136:     # multiprocessing not very effective due lack of memory on CPU
+137:     for y in range(num_sim):
 138:         # 1 simulation
+139:         for x in range(num_days - 1):
+140:             if x == 0:
+141:                 sim_array[y][0] = lp * (1 + randoms[y * num_days + x])
 142:             else:
+143:                 sim_array[y][x] = sim_array[y][x-1] * (1 + randoms[y * num_days + x])
 144:
+145:     return convert_to_python(sim_array, num_sim, num_days)
```

python

Figure 7. Python vs C-Code

Chapter 6. Fazit

Ich habe als Fazit gelernt, dass es wichtig ist, bevor man sich auf Details konzentriert, die Hauptlogik des Problems zu verstehen und sich genug Zeit zu nehmen, dieses zu Bewerten.
Nur dadurch spart man sich viel Zeit in Form von unnötigen Trial and Error Zyklen, da man oft sich in Sackgassen verirrt.

Als Beispiel: Man parsed einen Teil des Codes in performanten C-Code, sodass sich ein größerer compute intensiver Teil parallelisieren lässt. Später stellt man fest, dass sich der Code nicht wie erwartet umgesetzt werden kann oder die Performance sich sogar durch zu viele Eingriffe verschlechtert. In Form von:

- Zu viele Schleifen, um die Daten vorzubereiten
- Viele unnötige Kopiervorgänge
- CPU Laufzeit-Stack kommt an seine Grenzen bei der eigentlichen Parallelisierung

Leider konnte ich das Single Precision Problem bei OpenCL nicht lösen, da mir leider nach unzähligen Stunden Recherche keine Lösung eingefallen ist und meine Skills leider an ihre Grenzen gestoßen sind. Ich bin mir aber sehr sicher, dass die Lösung nur noch minimal Aufwand ist. Und man somit korrekte Ergebnisse in single precision für die GPU erzielen kann.

Eine gewonnene Erfahrung mit diesem Projekt ist es, in Zukunft gezielt bei neuen Projekten vor der Entwicklung auf Bottlenecks zu achten und diese nach HPC-Prinzipien bewerten und ggfls. vorab theoretisch zu lösen.

Weitere Schritte die für dieses Projekt aufgeführt werden können:

- Optimierung des C-Codes durch die saubere Einführen des Zufalls über CPP Header somit keine verwendung von interpretiertem Python ensteht
- Entfernung von unnötigen Schleifen
- Code weniger statisch für Daten machen

6.1. Aufwand

Ich würde den Aufwand für dieses Projekt als sehr hoch einschätzen, sich erst mal ein geeignetes Thema zu überlegen. Das vollständige Verständnis eines Anwendungsproblemes hat mich am meisten Zeit und immer wieder Lehrgeld gekostet, da man immer wieder in Sackgassen gelaufen ist.
Es war viel Recherche notwendig und nochmals doppelt so viel Trial and Error. Die meisten Probleme, die ich hatte, kamen von der schlechten Dokumentation der verwendeten Bibliotheken. Zusätzlich findet man kaum gute Beispiele über Cython besonders wenn es zu komplexeren Einsatzbereichen kommt, obwohl es in nahezu allen Frameworks eingesetzt wird wie z. B. scikit-learn. Durch intensives Verstehen der Cython Dokumentation ist man dann nur wenig schlau geworden und musste daher immer wieder auf ein Google Forum Board zurückgreifen (see Forum).

Es war, denke ich, ein sehr gewagter Ansatz, da man sehr schnell den Überblick verlor und in nahezu jedem Schritt an die Grenzen der jeweiligen Bibliothek oder Programmiersprache gestoßen

wurde. Dadurch wurde der Rattenschwanz immer länger und man wusste nicht mehr, was eigentlich wichtig war. Man wusste allerdings im Unterbewusstsein ab einem gewissen Punkt war es nicht mehr performant und verwarf die ursprüngliche Intention wieder.

Zum Abschluss würde ich sagen, dass ich sehr viel lernen konnte, vor allem was hardwarenahe Entwicklung angeht und es mir eine große Wissenslücke aufgezeigt hat.
Dennoch konnte ich durch dieses Projekt meine Skills in C, C++, Cython, OpenCL und Python deutlich verbessern.

Das Projektthema war ein eher triviales Thema und ich hätte mit einem komplexeren Thema wahrscheinlich größere Erfolge beim Ergebnis der Optimierung erzielen können. Ich denke allerdings, dass ich dadurch einen geringeren Lernerfolg gehabt hätte, da man es dadurch eher mit Problemen nicht HPC-Ursprungs zu tun bekommen hätte.

Chapter 7. Appendix

List of Figures

List of tables

7.1. Conventions

The following conventions are used in the document and are specially marked:

Note

Warning

Important

@todo - ...

- Todos are marked accordingly and usually highlighted in yellow. There should be no more todos in the final version.

7.2. License

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.