# Universidade do Minho

## 4º ano do Mestrado Integrado em Engenharia Informática

### Métodos Formais em Engenharia de Software

### Análise e Teste de Software

### Ano Letivo 2016/2017

---

# *CaparicaPost*

---


**Membros do Grupo:**

Diogo José Linhares Couto – 71604

Gil Gonçalves-a67738

Pedro Silva-67751


Braga, 27 de janeiro de 2017

# Introdução

Neste presente relatório é detalhadamente explicado as estratégias utilizadas na segunda fase de otimização da performance energética de uma aplicação que gere uma agência noticiosa, desenvolvida com a utilização da linguagem de programação JAVA.

Esta aplicação foi desenvolvida por alunos no seu primeiro curso de programação em JAVA.

Como tal é natural que devido a sua falta de experiência em programação JAVA, as decisões tomadas durante o processo de desenvolvimento nem sempre a tivessem eficiência do programa como objetivo principal.

Isso dá origem a pedaços de código ineficientes e com a possibilidade de serem otimizados.

Foi então utilizada a ferramenta *jRAPL*, para fazer a análise de performance do software fornecido tendo em consideração três fatores, tempo de execução, consumo de memória e o consumo de energia

Usando essa análise como base prossegue-se à localização e otimização de secções de código ineficientes, e a sua respetiva comparação dos resultados de performance entre os vários estados de otimização.

Ao contrário da primeira fase em que otimização foi meramente em termos de desempenho, que por si só gera ganhos energéticos, esta segunda fase teve apenas em conta a melhoria do consumo energético.

Para esse efeito foram aplicadas as técnicas de Programação Verde lecionadas e feita a análise do seu impacto.

Tal como na primeira fase as alterações feitas não devem mudar a funcionalidade do projeto.

# Metodologia de Teste

Foi então novamente utilizada a *framework black-box testing* disponibilizada para este projeto para garantir que a funcionalidade do mesmo se mantinha. Esta *framework* consiste em fornecer para cada um dos dois ficheiros de inputs diferentes, *t1.txt e t2.txt*, o output esperado correspondente.

Desta forma, desde que se invoque a classe *Main* para cada um dos inputs, e se obter o seu output esperado o projeto faz o que é pedido.

Assim para facilitar estes testes foi usada a *bash script* com o nome de *"run.sh"*, criada durante a primeira fase, que devolve *"OK"*, se o output do programa for igual ao output esperado ou "KO", no caso contrário. Este script também fornece a performance do programa para cada um dos outputs.

De modo a apresentar resultados mais fiéis e descartar ligeiras oscilações de valores, cada um dos testes foi executado 5 vezes para cada um dos inputs, e foi calculada a média desses resultados.

Foram utilizados dois computadores para verificar se os resultados eram semelhantes em máquinas diferentes, deste modo eliminando a influência do estado geral da máquina.

O PC-1 tem um processador Intel® Core™ i7-4750HQ, uma gráfica GeForce950m e 8.00GB de memória RAM.

O PC-2 tem um processador Intel® Core™ i5 Dual Core -3210M 2,5 GHZ, uma gráfica GeForce GT 640m 2GB e 6.00GB de memoria RAM.

Nas tabelas apresentadas as colunas de ganho são os ganhos das otimizações comparativamente à versão original.

De seguida estão apresentados os resultados finais obtidos durante a otimização da primeira fase, estes resultados foram o ponto de partida para este segundo passo na otimização. Os ficheiros de input não foram modificados e são denominados de *t1.txt* e *t2.txt*.

**Tabela 1 - t1.txt**

| | PC - 1 | PC - 2 |
|---|---|---|
| **GPU (J)** | 0.07924779999 | 0.6300908 |
| **CPU (J)** | 17.8698362 | 33.9977356 |
| **Package (J)** | 31.091931 | 45.0657078 |
| **Time (Seconds)** | 1.4439319634 | 2.7017811838 |

**Tabela 2 - t2.txt**

| | PC - 1 | PC - 2 |
|---|---|---|
| **GPU (J)** | 0.057837 | 1.1025938 |
| **CPU (J)** | 10.225354 | 22.875485 |
| **Package (J)** | 16.2525632 | 30.7739626 |
| **Time (Seconds)** | 0.6094794914 | 1.6971776764 |

# Otimizações

Depois de retirar os resultados preliminares, foram aplicadas três técnicas distintas de programação verde de modo obter não só melhorias na performance do programa, mas também para torna-lo mais “Verde”, ou seja, para reduzir o seu consumo de energia.

## Manipulação de *Strings*

Em *JAVA* quando estamos a realizar operações com *Strings* é natural utilizar as primitivas oferecidas como o operador “+”. No entanto essas primitivas são custosas especialmente no caso da concatenação de *Strings* pois acaba por criar objetos *String* extras e desnecessários durante a execução do programa. Depois de alguma pesquisa e nos apontamentos disponibilizados nas aulas decidimos então utilizar a classe *StringBuilder* como forma de concatenar *Strings* substituindo assim o operador “+”.

Para além das melhorias a nível de performance a classe *StringBuilder* oferece várias funções para manipular *Strings* estando estas otimizadas para ser mais eficientes em termos energéticos.

Então a primeira operação realizada neste processo de otimização foi substituir todas as manipulações de *String* feitas com as primitivas do *JAVA*, de modo a utilizarem a classe *StringBuilder.*

A comparação é feita com a versão inicial.

Após essa alteração foram realizados os testes e obteve-se os seguintes resultados:

**Tabela 3 – Otimização 1 para o ficheiro t1.txt**

| | PC - 1 | PC - 2 | Ganho% PC - 1 | Ganho% PC - 2 |
|---|---|---|---|---|
| **GPU (J)** | 0.0511470 | 0.423004 | 1.549412 | 1.489562 |
| **CPU (J)** | 13.321106 | 30.48973 | 1.341468 | 1.115053 |
| **Package (J)** | 27.87518 | 40.43132 | 1.115398 | 1.114624 |
| **Time (Seconds)** | 1.799860 | 2.440326 | 0.802247 | 1.107152 |

**Tabela 4– Otimização 1 para o ficheiro t2.txt**

| | PC - 1 | PC - 2 | Ganho% PC - 1 | Ganho% PC - 2 |
|---|---|---|---|---|
| **GPU (J)** | 0.0129400 | 0.47546399 | 4.469629 | 2.318985 |
| **CPU (J)** | 8.8202519 | 14.90980500 | 1.159304 | 1.534258 |
| **Package (J)** | 16.6256100 | 19.75991799 | 0.615036 | 1.557914 |
| **Time (Seconds)** | 0.9185966 | 1.11577197 | 0.066349 | 1.521079 |

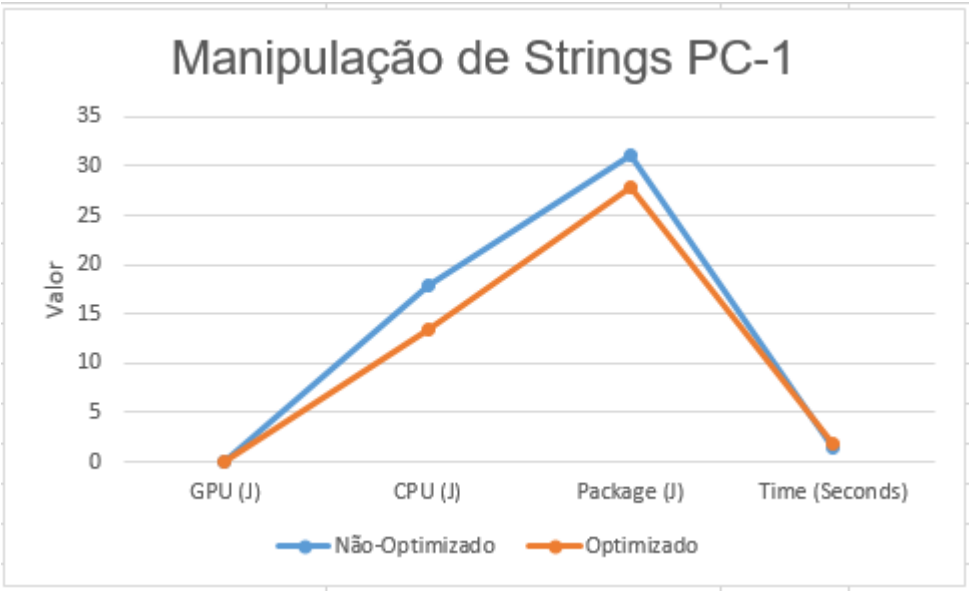


**Figura 1-Gráfico para o PC-1 e ficheiro t1.txt**

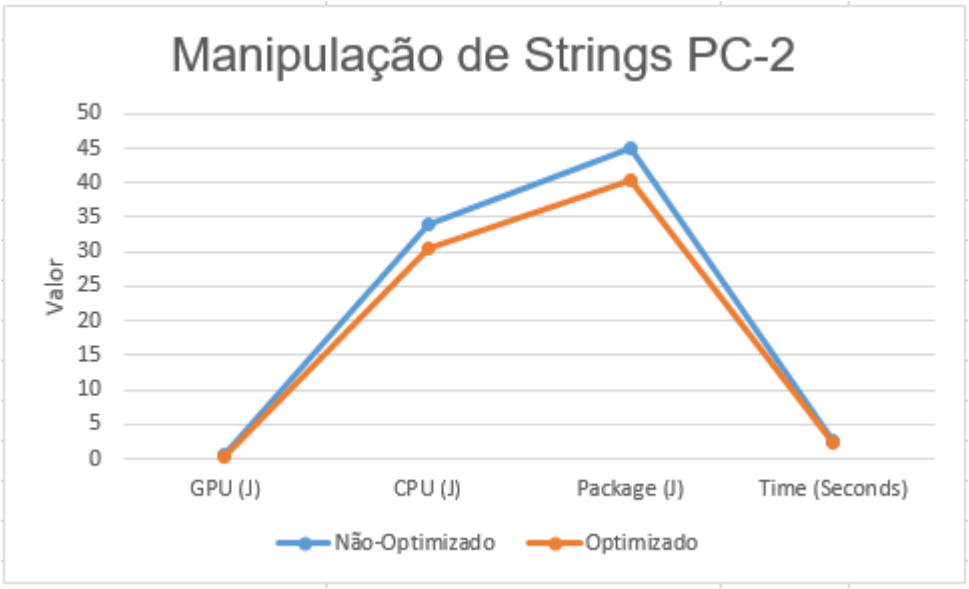


**Figura 2-Gráfico para o PC-2 e ficheiro t1.txt**

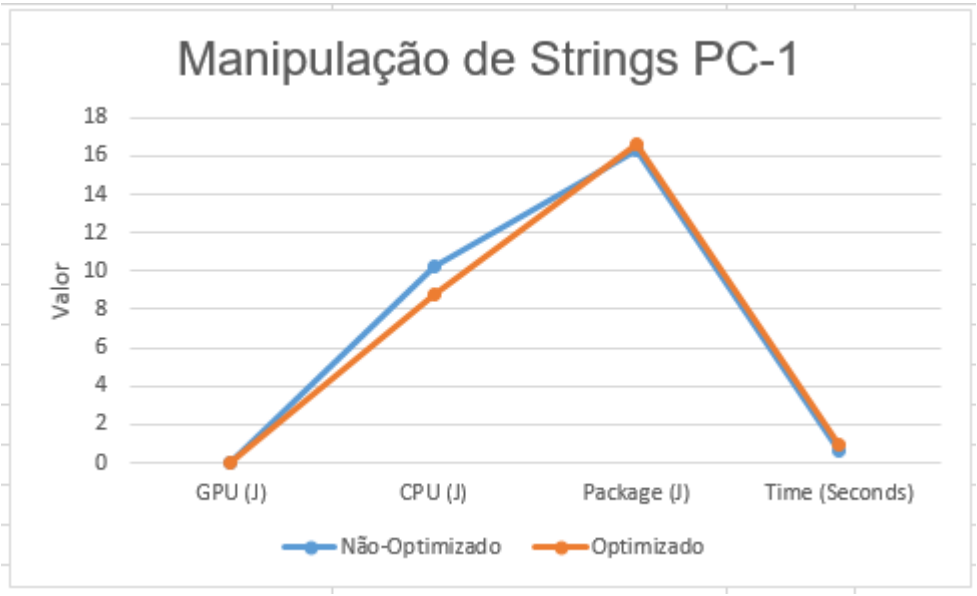


**Figura 3-Gráfico para o PC-1 e ficheiro t2.txt**

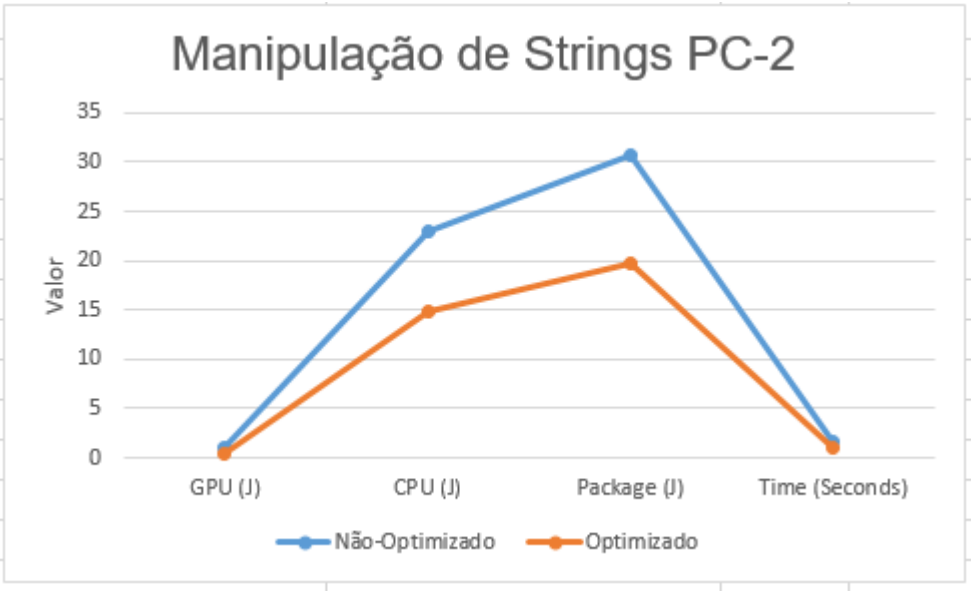


**Figura 4-Gráfico para o PC-2 e ficheiro t2.txt**

Como é possível observar através dos gráficos com a utilização do *StringBuilder* em vez do operador "+" para concatenar *Strings* gerou-se uma diminuição da utilização do CPU, GPU e Time, mas no caso do PC-1 demorou mais tempo a produzir resultados.

## Switch Statements e Comando

A utilização dos *Switch Statments* é uma prática muito usada em programação, contudo é uma má prática e deve ser evitado o seu uso. O seu uso está muito relacionado com o facto de em vez de criar muitas condições com *if/elses* usa-se o *switch* e consegue-se obter melhorias em termos de performance, contudo em termos de volatilidade tanto num caso como no outro ambos não reagem bem a mudanças. No primeiro caso a inserção de mais um *if* poderá fazer com que o código fique mais confuso e difícil de perceber, no caso do *switch* esta alteração poderá não ser tão simples como criar um novo "*case*", mas poderá levar à restruturação do código.

Como é possível constatar ambos os casos não lidam muito bem com mudanças o que torna este tipo de implementações difíceis de reutilizar originando perda enormes de tempo só para mudar a sua estrutura base.

Então para se ter uma estrutura mais versátil decidiu-se implementar uma nova estrutura. Esta estrutura consiste na criação de uma classe que irá conter a coleção de todos os comandos a serem utilizados no programa e sempre que o utilizador efetuar um comando o programa o que faz é, ir a lista de comandos e se o comando existir na lista retorna a classe que está associada ao comando. Deste modo se houver alterações nas funcionalidades basta só alterar a classe que está associada ao comando.

Neste tipo de implementações sempre que se criar uma nova funcionalidade será necessário adicionar essa funcionalidade à coleção e criar a classe associada a essa coleção.

A comparação é feita com a versão inicial.

**Tabela 4 – Otimização 2 para o ficheiro t1.txt**

| | PC - 1 | PC - 2 | Ganho% PC - 1 | Ganho% PC - 2 |
|---|---|---|---|---|
| **GPU (J)** | 0.249695 | 0.3744199 | 0.317378 | 1.682845 |
| **CPU (J)** | 14.405762 | 25.15982 | 1.240464 | 1.351271 |
| **Package (J)** | 30.826966 | 33.295531 | 1.008595 | 1.353506 |
| **Time (Seconds)** | 1.981281 | 2.059980 | 0.728787 | 1.311557 |

**Tabela 5 - Otimização 2 para o ficheiro t2.txt**

| | PC - 1 | PC - 2 | Ganho% PC - 1 | Ganho% PC – 2 |
|---|---|---|---|---|
| **GPU (J)** | 0.034240 | 0.503754 | 1.689165 | 2.188754 |
| **CPU (J)** | 9.308349 | 13.23261 | 1.098514 | 1.728721 |
| **Package (J)** | 17.949524 | 17.78877 | 0.905459 | 1.729966 |
| **Time (Seconds)** | 1.014592 | 1.056397 | 0.600714 | 1.606572 |

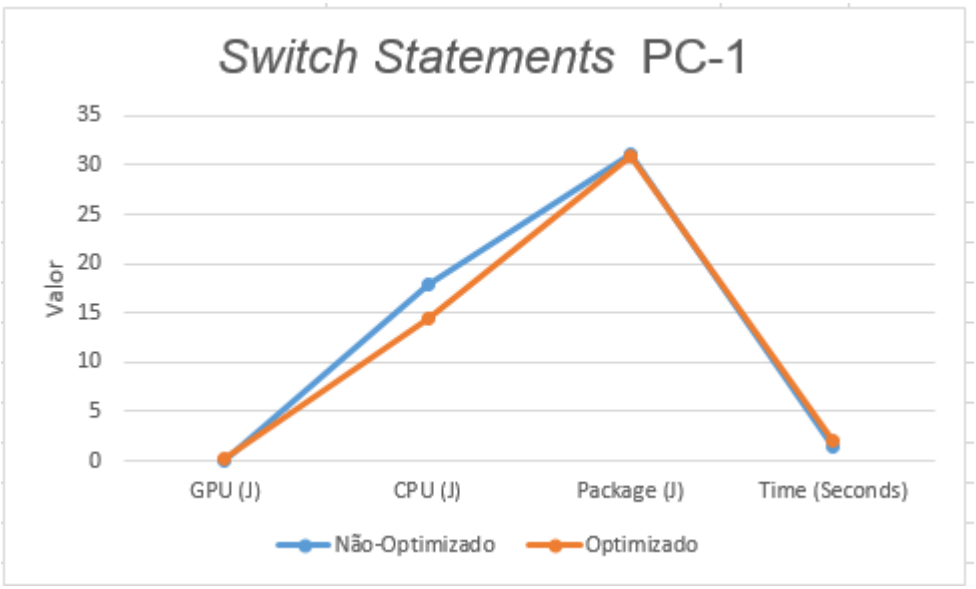


**Figura 5-*Switch Statments* para o PC-1 e ficheiro t1.txt**

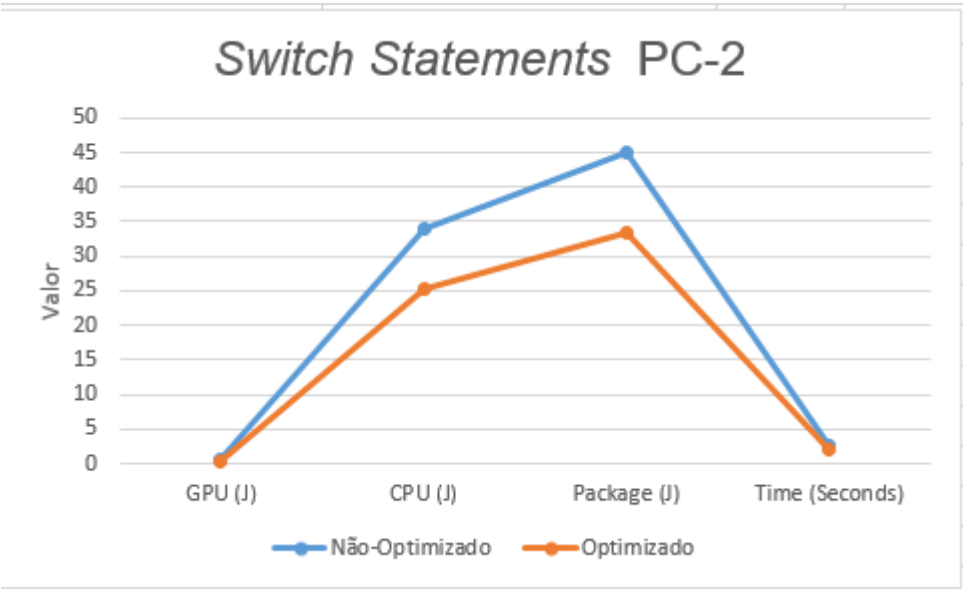


**Figura 6-*Switch Statments* para o PC-2 e ficheiro t1.txt**

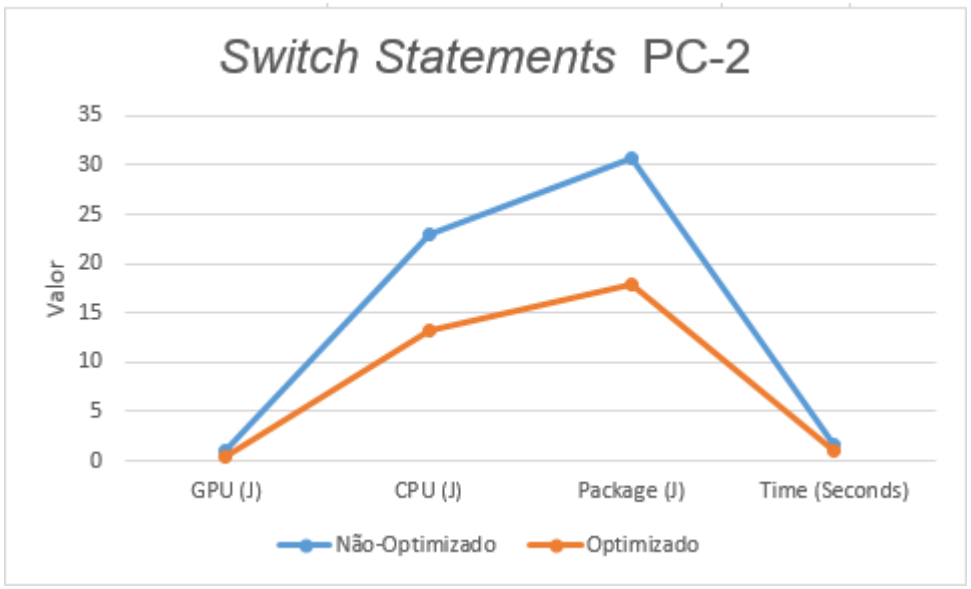


**Figura 7- *Switch Statments* para o PC-2 e ficheiro t2.txt**

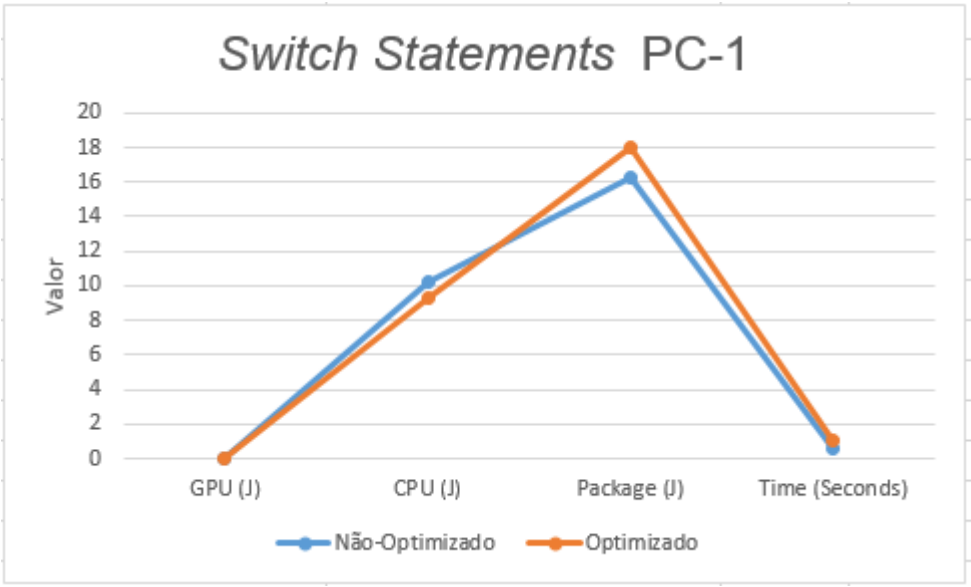


**Figura 8-*Switch Statments* para o PC-1 e ficheiro t2.txt**

Como era de esperar com a introdução de uma nova classe os *packages* aumentam porque agora é feito o importe da classe Comando que depois será utilizada na classe *main* para que sempre que haja introdução de um comando a classe Comando procura na sua lista de comandos disponíveis e

verifica se pode ou não executar aquele comando, caso possa executar o comando verifica se obtém os parâmetros corretos para o executar. Caso não possa executar o comando, ou tenha os parâmetros errados informa o utilizador que o comando é inválido ou que os parâmetros passados são inválidos.

## Alteração da estrutura de dados

A terceira técnica utilizada foi tirar vantagem da partilha de funções entre várias estruturas de dados, de modo a escolher a mais eficiente para as funcionalidades necessárias. Estas alterações são de fácil implementação graças à partilha de funções entre diferentes classes.

Para proceder a alteração das estruturas de dados fez-se um *profiling* aos inputs recebidos. Foi analisado para cada ficheiro quais eram as operações mais utilizadas obtendo os seguintes resultados:

| | T1.TXT | T2.TXT |
|---|---|---|
| **REGISTER** | 551 | 757 |
| **LOGOUT** | 1009 | 1053 |
| **ADD** | 4436 | 1587 |
| **LIKE** | 5321 | 1456 |
| **LOGIN** | 1009 | 1053 |
| **APPROVE** | 626 | 305 |
| **ASSIGN** | 73 | 38 |
| **LIST** | 2693 | 997 |

Desta forma conseguimos perceber que os métodos mais aplicados às nossas estruturas de dados são métodos *"add"* na operação *"ADD"* e métodos *"get"* para as operações de *"LOGIN"* e *"LIST"*.

Com este *profiling* aos dados é necessário escolher as estruturas de dados mais eficientes para o problema. A análise a este tipo de operações é demorada e pode levar a conclusões erradas, por isso e de maneira a utilizar-se os dados corretos a escolha das estruturas foi baseada num artigo disponibilizado nas aulas[1].

Devido à constante ordenação das coleções algumas das estruturas de dados não podem ser alteradas por não serem subclasses da coleção *List*.

Assim, optou-se por comparar a execução do programa para a atual coleção *TreeMap* e para a coleção *Hashtable* que segundo o artigo[1] possuí melhores resultados a nível energético para métodos de *get* e *set*.

[1] http://greenlab.di.uminho.pt/wp-content/uploads/2016/06/greens.pdf

Os resultados obtidos encontram-se demonstrados a seguir:

**Tabela 5 – Para o ficheiro t1.txt e PC-1**

| | TreeMap | Hashtable | Ganho% TreeMap | Ganho% Hashtable |
|---|---|---|---|---|
| **GPU (J)** | 0.249695 | 0.101257 | 0.317378 | 0.78264 |
| **CPU (J)** | 14.405762 | 17.438232 | 1.240464 | 1.02475 |
| **Package (J)** | 30.826966 | 35.181091 | 1.008595 | 0.883768 |
| **Time (Seconds)** | 1.981281 | 2.066638 | 0.728787 | 0.698686 |

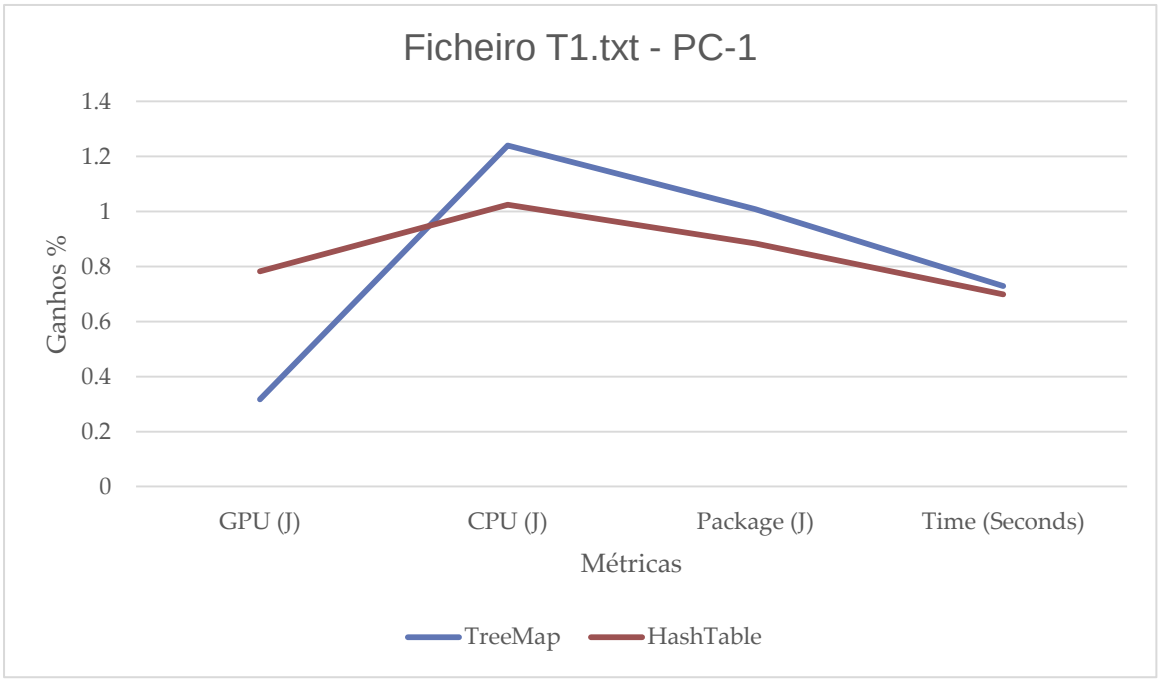


**Figura 9-Ficheiro T1.txt - PC-1**

Como se pode observar no gráfico, a versão *TreeMap* supera sempre a versão Hashtable para todas as métricas expecto em termos de GPU onde a versão *Hashtable* apresenta melhores resultados. Visto que as diferenças nos outros parâmetros não são muito significativas, é possível afirmar que a versão *Hashtable* é apresenta melhores resultados a nível energético.

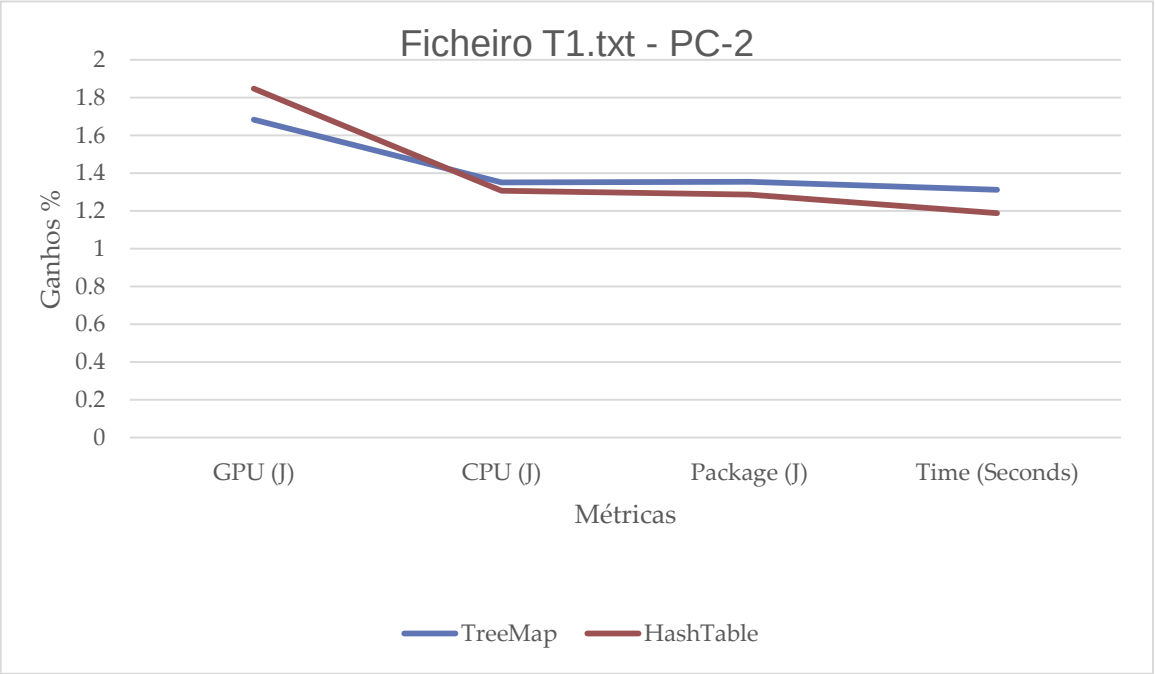


Figura 10-Ficheiro T1.txt - PC-2

**Tabela 6 – Para o ficheiro t1.txt e PC-2**

| | TreeMap | Hashtable | Ganho% TreeMap | Ganho% Hashtable |
|---|---|---|---|---|
| **GPU (J)** | 0.3744199 | 0.340928 | 1.682845 | 1.848164 |
| **CPU (J)** | 25.15982 | 26.010162 | 1.351271 | 1.307094 |
| **Package (J)** | 33.295531 | 35.050232 | 1.353506 | 1.285746 |
| **Time (Seconds)** | 2.059980 | 2.274111 | 1.311557 | 1.18806 |

Para o PC-2, os resultados foram ligeiramente melhores para a versão com *TreeMap*, existindo apesar de menos evidente uma melhoria a nível de CPU para a versão *Hashtable*.

**Tabela 5 – Para o ficheiro t2.txt e PC-1**

| | TreeMap | Hashtable | Ganho% TreeMap | Ganho% Hashtable |
|---|---|---|---|---|
| **GPU (J)** | 0.034240 | 0.034240 | 1.689165 | 1.689165 |
| **CPU (J)** | 9.308349 | 11.235656 | 1.098514 | 0.910081 |
| **Package (J)** | 17.949524 | 21.087525 | 0.905459 | 0.770719 |
| **Time (Seconds)** | 1.014592 | 1.090047 | 0.600714 | 0.559131 |

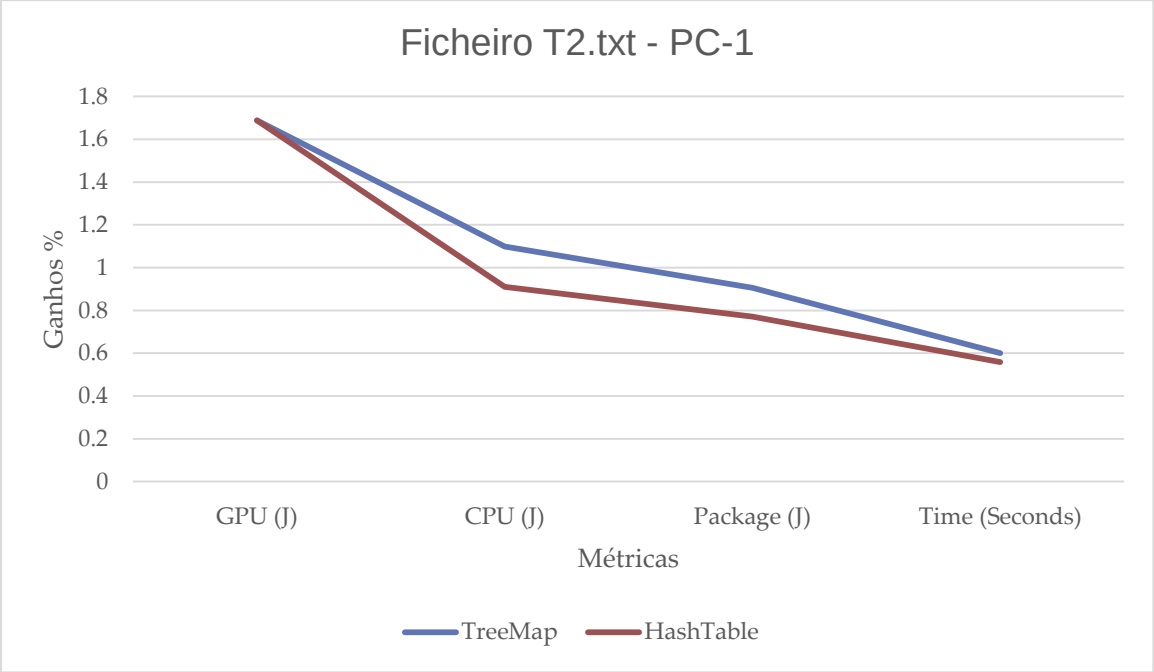


**Figura 11-Ficheiro T2.txt - PC-1**

Para o ficheiro *t2.txt* no *PC-1* obteve-se, em todos os parâmetros, melhores resultados usando *TreeMap*. Isto deve-se ao facto de o ficheiro *t2.txt* apresentar muito menos operações no geral que o ficheiro *t1.txt*, e sobretudo operações *"ADD"* onde os ganhos com a coleção *Hashtable* seriam muito mais evidentes.

**Tabela 6 – Para o ficheiro t2.txt e PC-2**

| | TreeMap | Hashtable | Ganho% TreeMap | Ganho% Hashtable |
|---|---|---|---|---|
| **GPU (J)** | 0.503754 | 0.965591 | 2.188754 | 1.141885 |
| **CPU (J)** | 13.23261 | 13.303221 | 1.728721 | 1.719545 |
| **Package (J)** | 17.78877 | 18.376845 | 1.729966 | 1.674605 |
| **Time (Seconds)** | 1.056397 | 1.08535 | 1.606572 | 1.563715 |

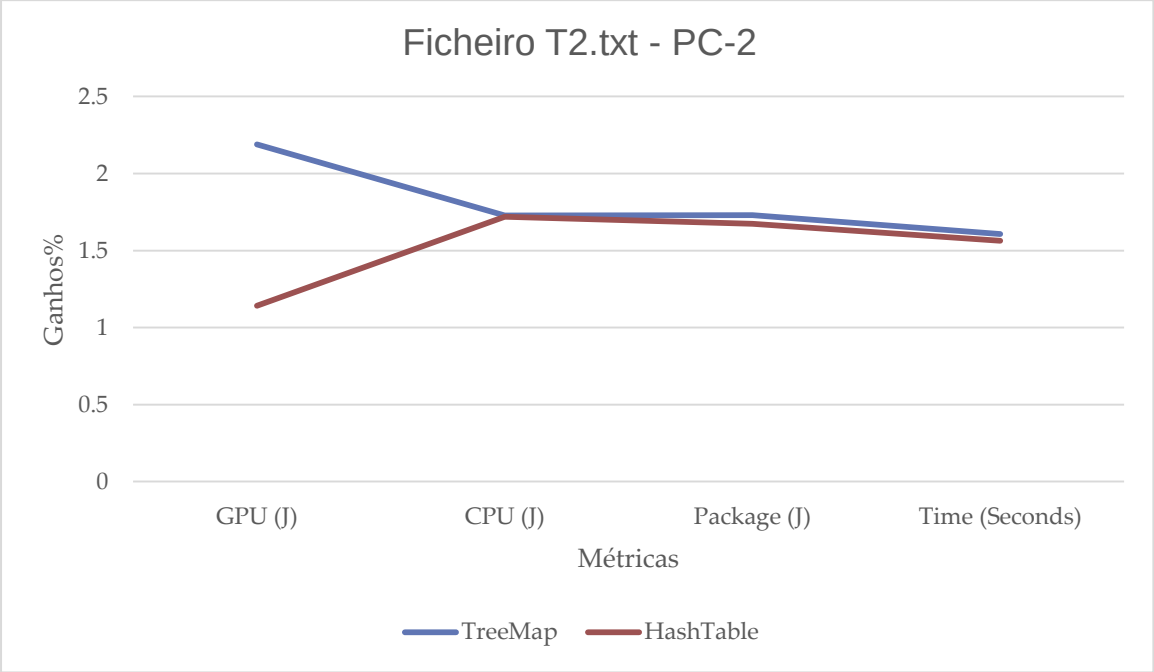


**Figura 12-Ficheiro T2.txt - PC-2**

No *PC-2* verificou-se o mesmo que no *PC-1*, ou seja, a versão com *Hashtable* embora menos evidente, não teve ganhos significativos.

# Conclusões

Nesta fase conseguiu-se melhores resultados a nível de performance que na segunda fase. Algumas das técnicas que acabam por tornar o programa mais eficiente energeticamente também acabam por trazer, em alguns casos, ganhos significativos a nível de performance.

Apesar destes ganhos de performance o grupo considerou que algumas destas técnicas como por exemplo a de manipulação de *Strings* acabou por tornar o código menos percetível.

Uma das otimizações de código realizadas foi a retirada de chamadas de métodos redundantes, visto que o custo de chamar esse método é elevado.

Nesta fase apercebeu-se que o método *HasUser* e *HasArticle* se baseiam em consultar o conteúdo de coleções e por isso optou-se pela sua remoção e substituição, pelas respetivas operações, em todas as invocações. Obtendo assim ganhos a nível de performance.

Assim, de um modo geral, obteve-se os seguintes resultados com todas as otimizações feitas até agora:

**Tabela 6 – Para o PC-1**

| | T1.txt - Inicio | T1.txt - Otimizado | T2.txt - Inicio | T2.txt - Otimizado |
|---|---|---|---|---|
| **GPU (J)** | 0.07924779999 | 0.249695 | 0.057837 | 0.034240 |
| **CPU (J)** | 17.8698362 | 14.405762 | 10.225354 | 9.308349 |
| **Package (J)** | 31.091931 | 30.826966 | 16.2525632 | 17.949524 |
| **Time (Seconds)** | 1.4439319634 | 1.981281 | 0.6094794914 | 1.014592 |

**Tabela 6 – Para o PC-2**

| | T1.txt - Inicio | T1.txt - Otimizado | T2.txt - Inicio | T2.txt - Otimizado |
|---|---|---|---|---|
| **GPU (J)** | 0.6300908 | 0.3744199 | 1.1025938 | 0.503754 |
| **CPU (J)** | 33.9977356 | 25.15982 | 22.875485 | 13.23261 |
| **Package (J)** | 45.0657078 | 33.295531 | 30.7739626 | 17.78877 |
| **Time (Seconds)** | 2.7017811838 | 2.059980 | 1.6971776764 | 1.056397 |