Monitorização do Consumo de Energia com Intel RAPL

PG57549 - Hugo Ricardo Macedo Gomes

Universidade do Minho

2024/2025

Introdução

A crescente preocupação com a eficiência energética dos sistemas computacionais tem impulsionado o desenvolvimento de ferramentas para a monitorização do consumo de energia.
O Intel Running Average Power Limiting (RAPL) é uma interface que permite obter estimativas precisas do consumo energético de processadores modernos, sem necessidade de hardware externo.
Esta apresentação aborda a utilização de um programa em C que integra a biblioteca RAPL para medir o consumo energético de aplicações, preservando a integridade do código original e minimizando a interferência nos resultados.

Sensores de Temperatura

A medição consistente do consumo de energia exige que as execuções sejam realizadas sob condições térmicas semelhantes. O programa desenvolvido utiliza a biblioteca *lm-sensors* para monitorizar a temperatura média dos núcleos da CPU, assegurando que cada execução começa apenas quando a temperatura estabiliza em um estado de inatividade.

Objetivos

- Implementar um sistema não intrusivo de monitorização energética.
- Garantir a precisão das medições de energia com base na temperatura.
- Automatizar múltiplas execuções para obter resultados estatisticamente relevantes.

Máquina de testes

A máquina utilizada, foi o meu computador pessoal, que conta com as seguintes configurações:

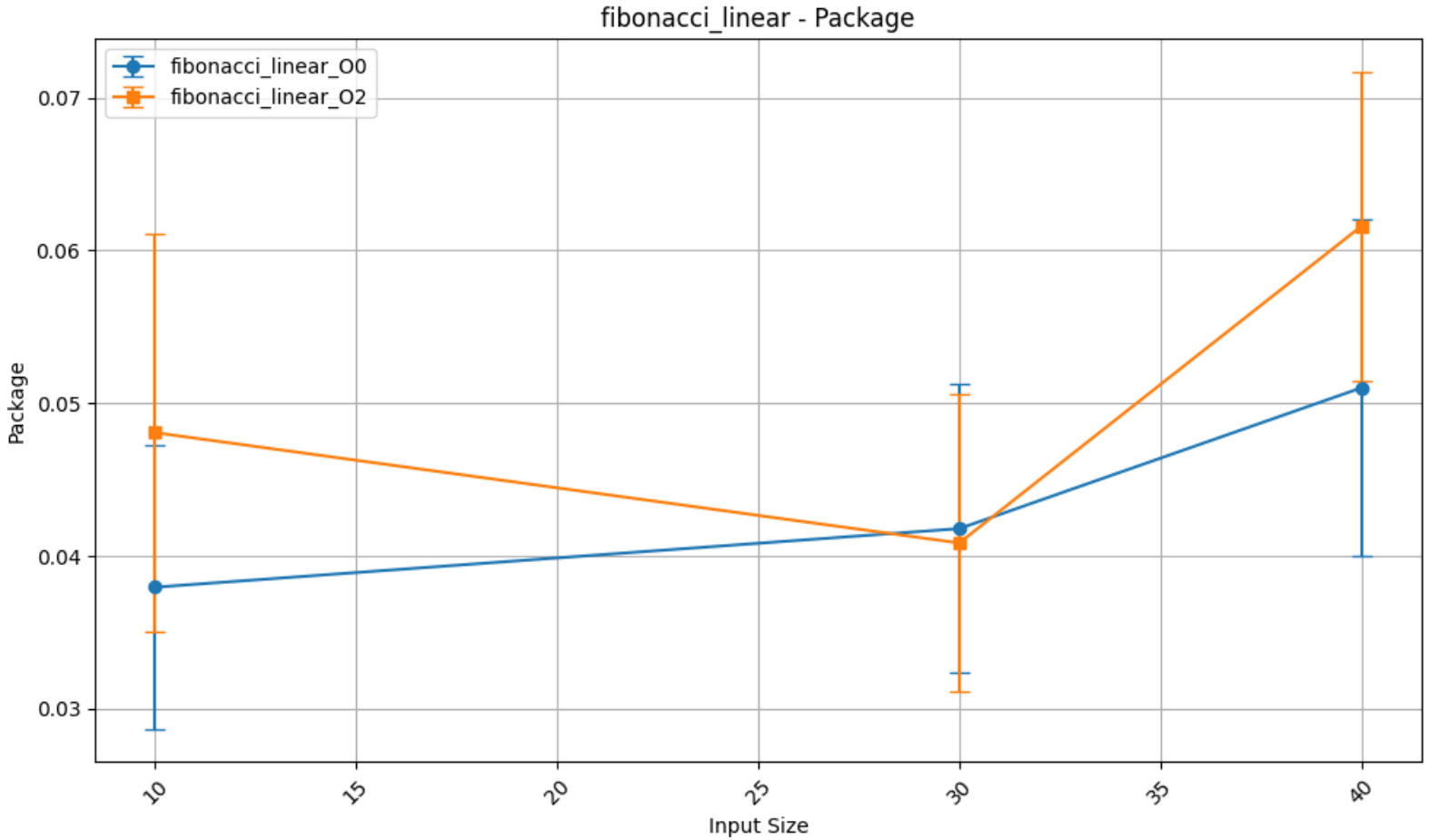
- HP ProBook 450 G7
- Memória: 8 GB
- Processador: Intel® Core™ i7-10510U CPU @ 1.80GHz × 8
- Placa Gráfica: Mesa Intel® UHD Graphics (CML GT2)
- SO: Ubuntu 22.04.5 LTS 64-bit

Fibonacci - C

Comecei por implementar uma versão linear e uma versão recursiva (não linear) de complexidade $O(2^N)$. Comparei vários aspetos de *performance*, nomeadamente: *Speedup, Greenup e Powerup*. Além disso, utilizei otimizações de compilador, especificamente a flag -O2.
Os testes foram realizados para o cálculo do 10º, 30º e 40º termo da sequência de Fibonacci.

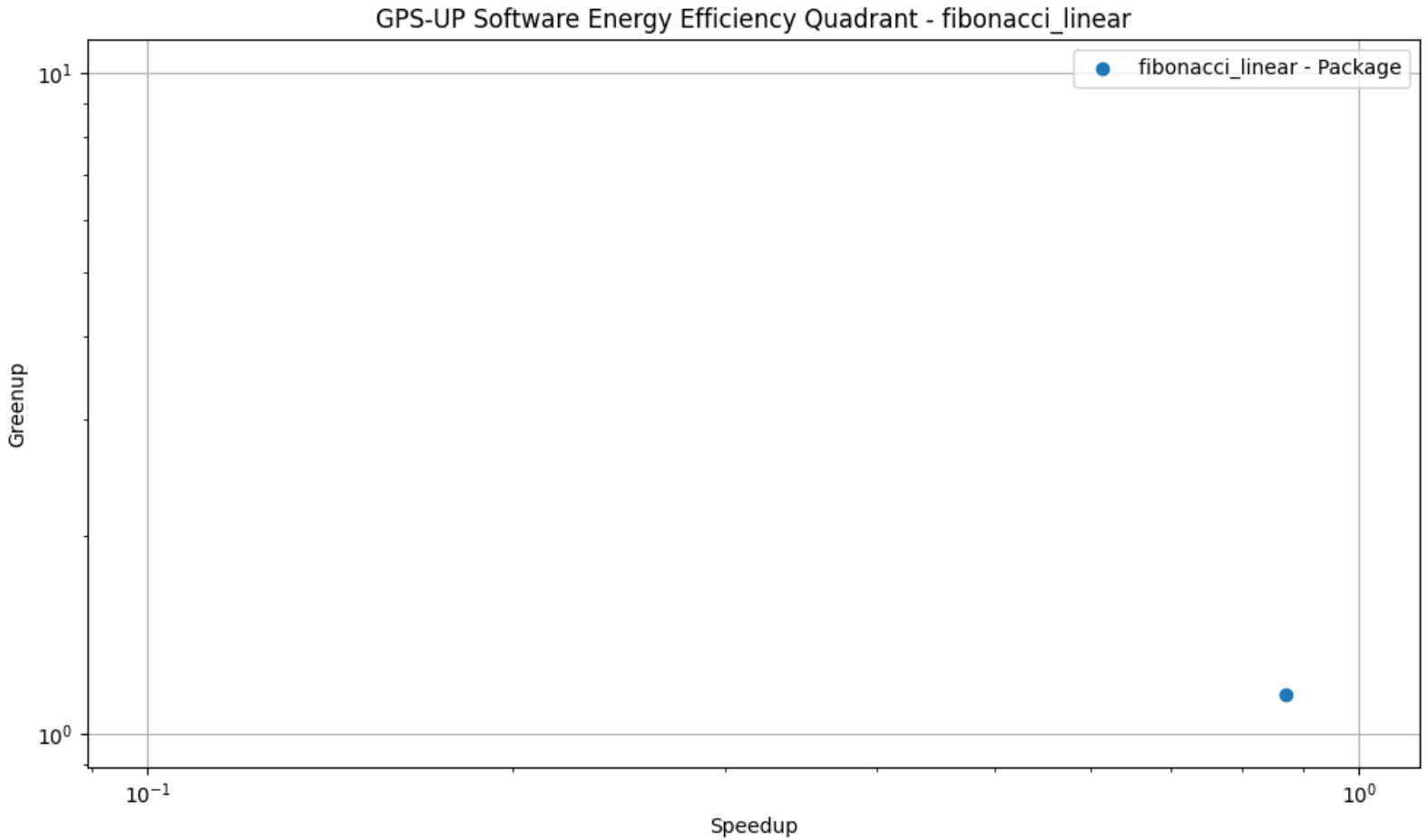
Fibonaccci (linear) - Package

Este gráfico demonstra o consumo de energia de todo o *socket*. A versão otimizada (O2) apresenta um comportamento misto: para alguns tamanhos de entrada, o consumo energético é menor, mas para valores maiores, pode superar a versão sem otimização (O0).



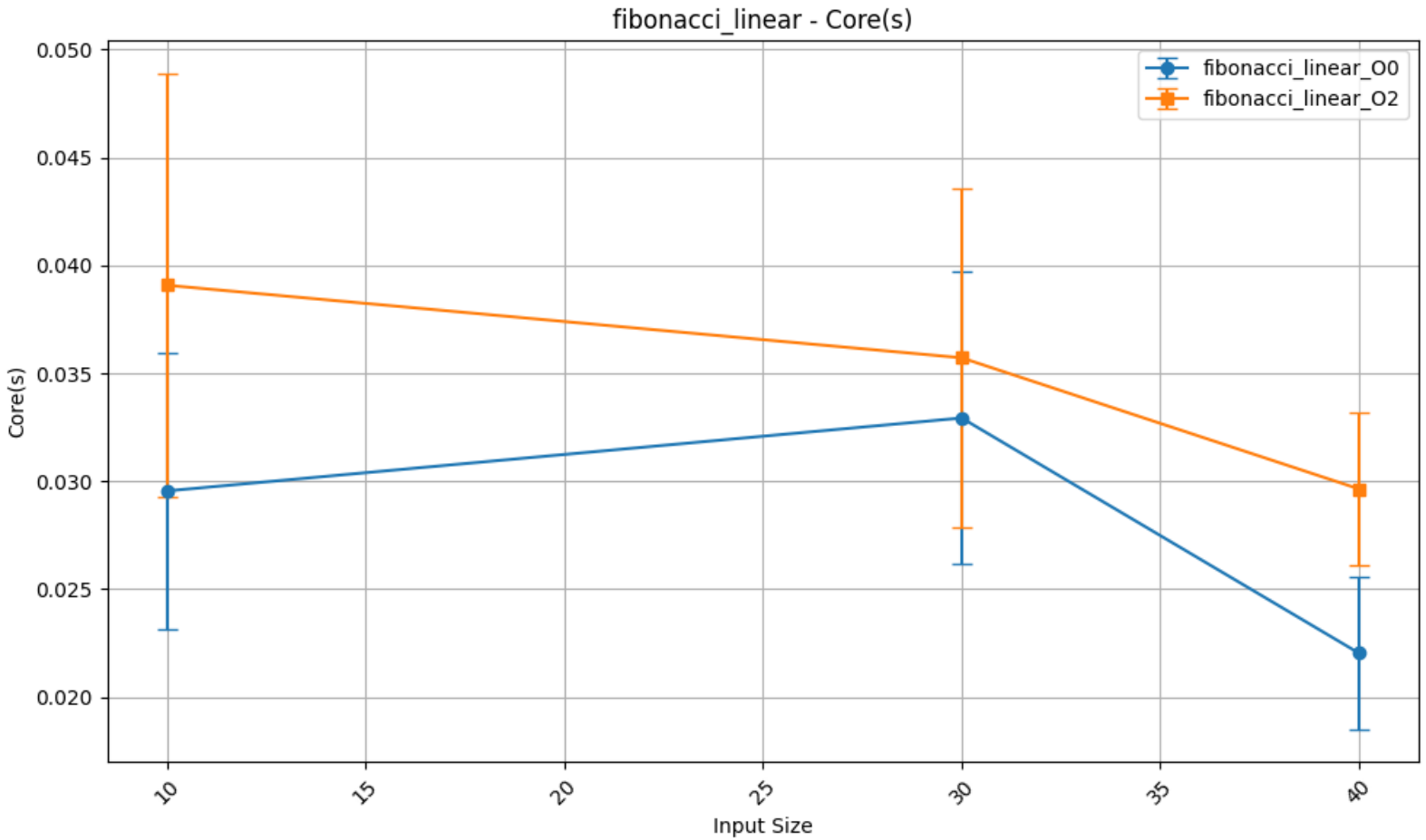
Fibonacci (linear) - Package (gps_up)

O ponto encontra-se no quadrante inferior direito, indicando que a implementação linear do Fibonacci apresenta melhorias significativas em speedup, no entanto é menos eficiente do que a versão não otimizada.



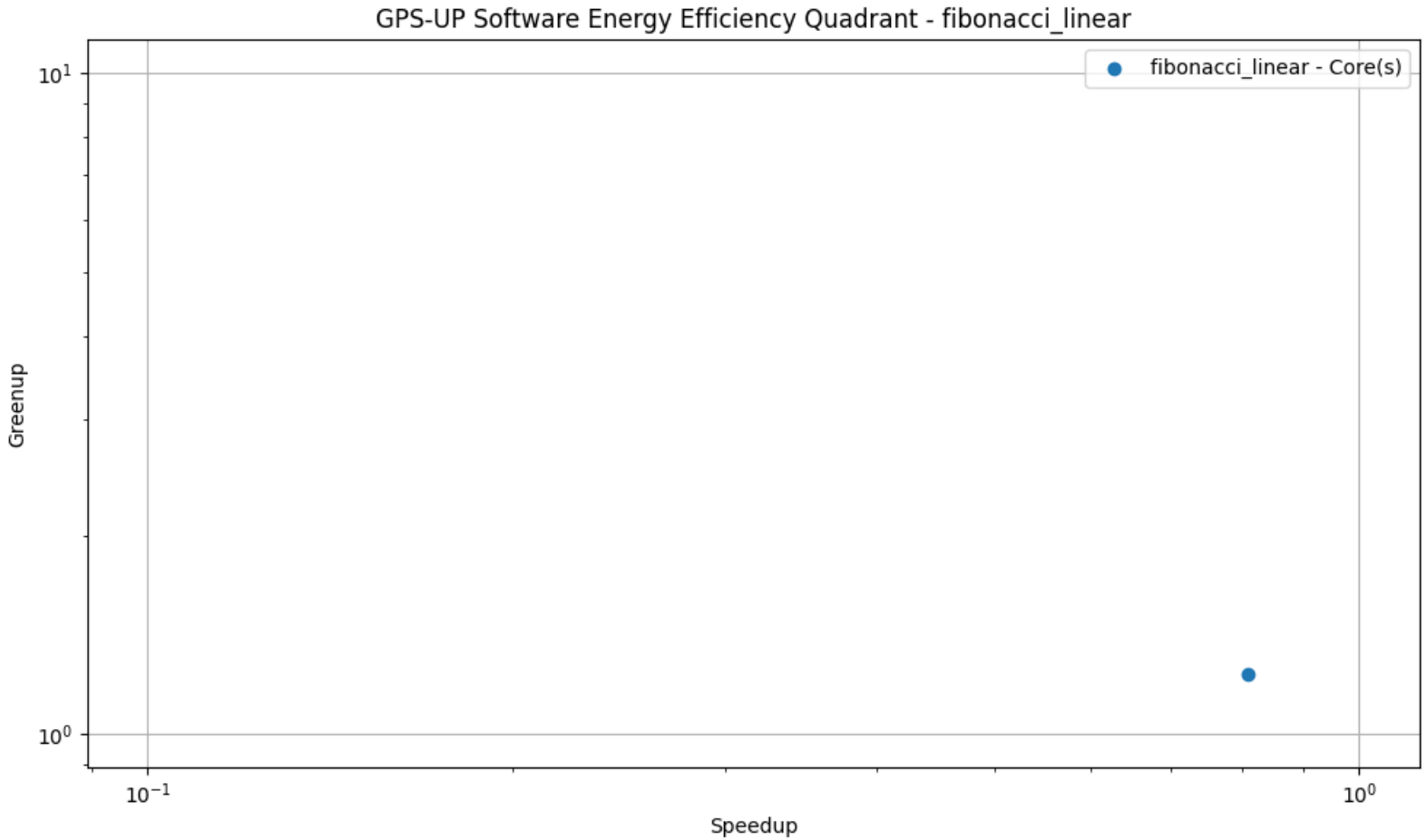
Fibonaccci (linear) - Core

Esta variável representa a energia consumida por todos os processadores e caches. A implementação com otimizações tem um desempenho pior, mas mais consistente em comparação com a versão sem otimizações, especialmente para tamanhos de entrada maiores.



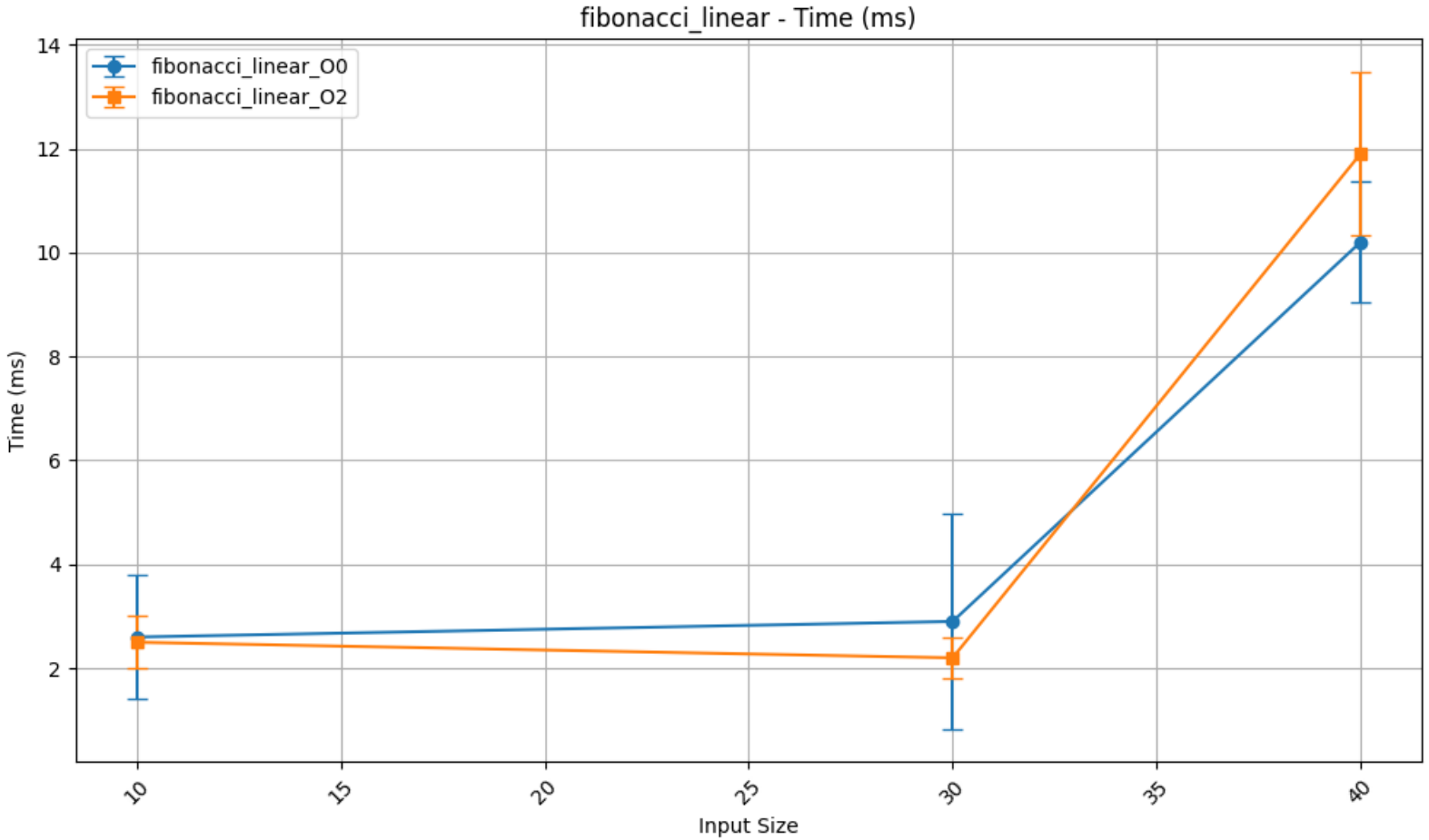
Fibonacci (linear) - Core (gps_up)

O ponto encontra-se no quadrante inferior direito, indicando que a implementação linear do Fibonacci apresenta melhorias significativas em speedup, no entanto é menos eficiente do que a versão não otimizada.



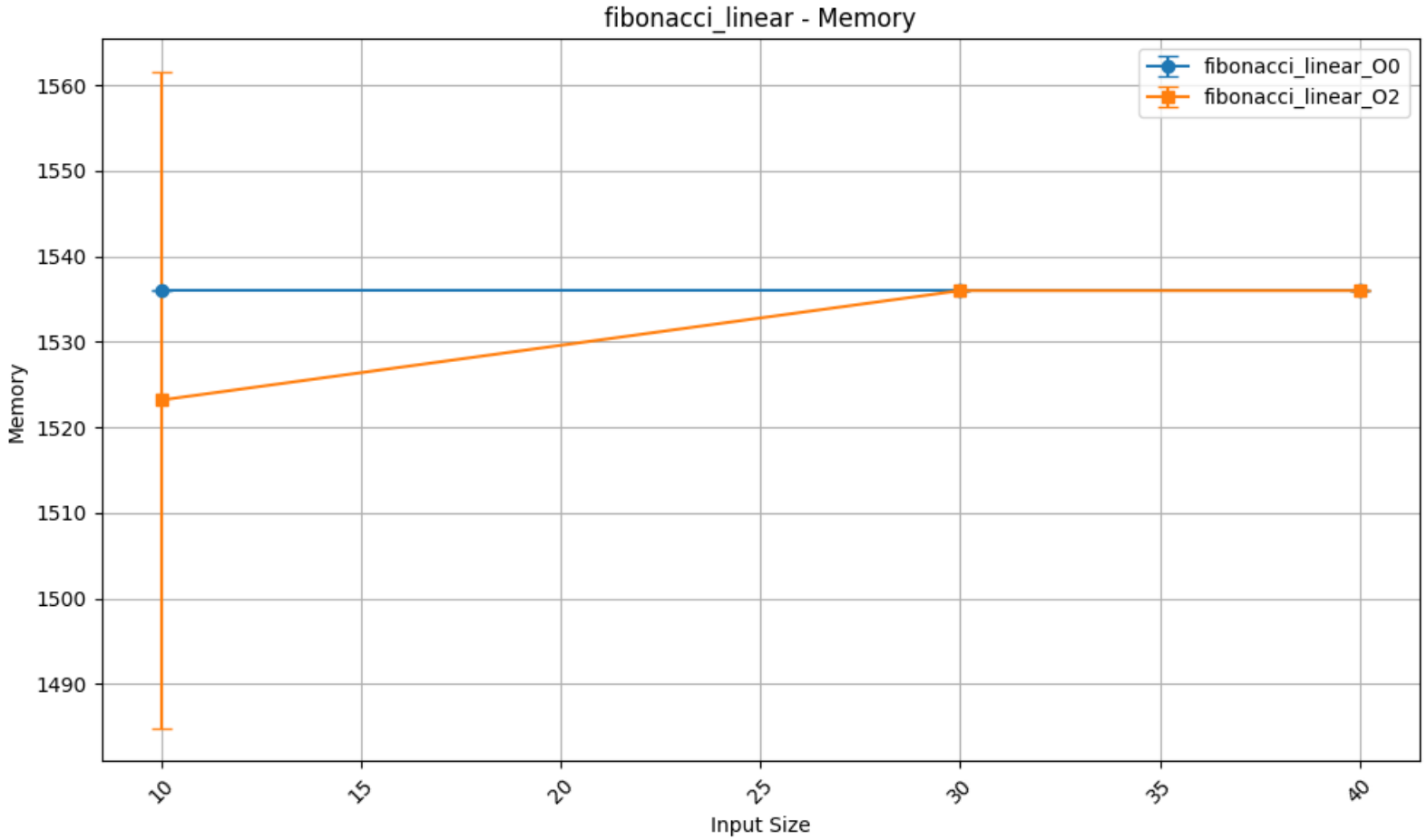
Fibonacci (linear) - Time

A implementação com otimizações tem um desempenho ligeiramente melhor para tamanhos de entrada menores, mas para entradas maiores (acima de 30), o tempo de execução aumenta ligeiramente em comparação com a versão sem otimizações.



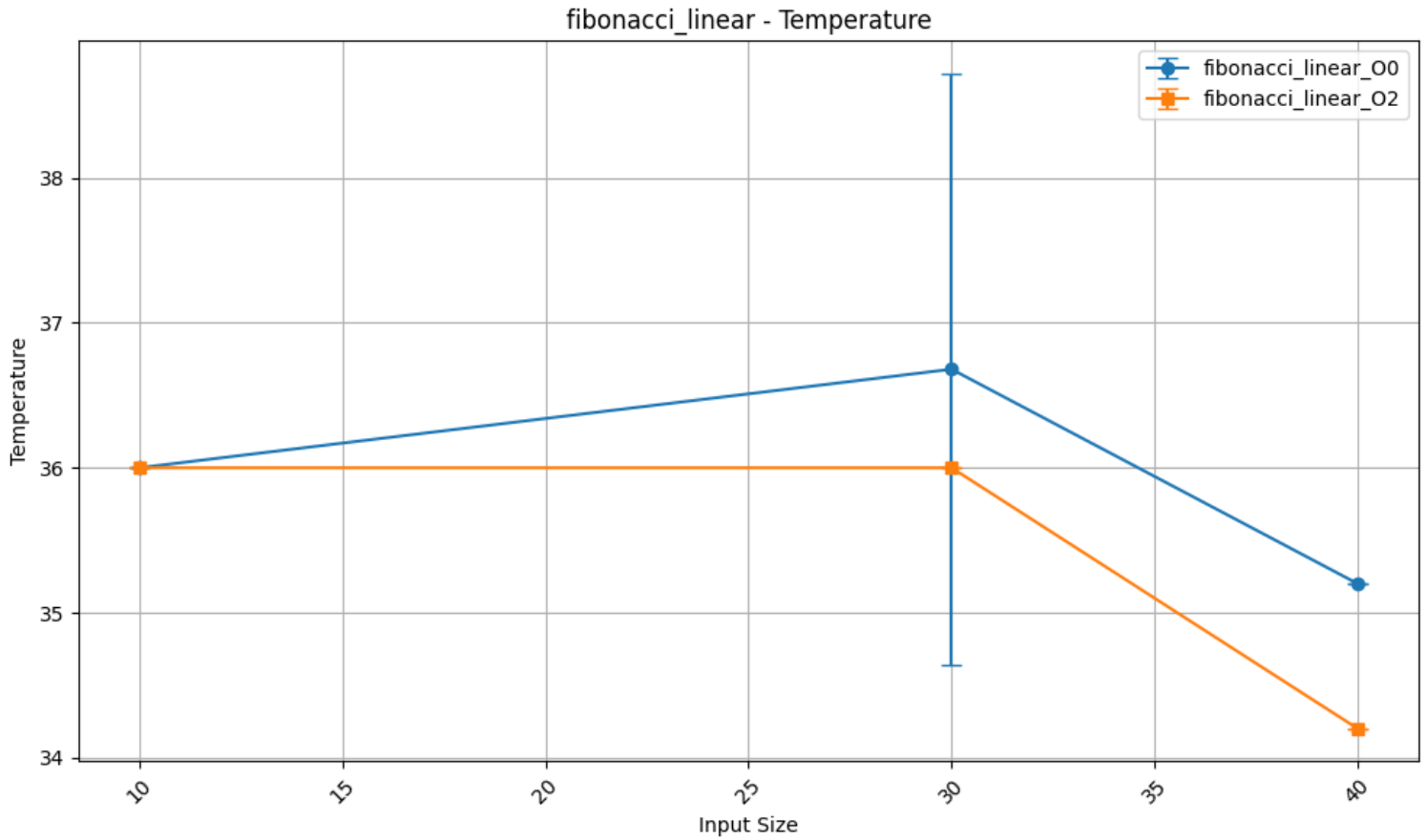
Fibonacci (linear) - Memory

A implementação sem otimizações mantém um consumo de memória constante em torno das 1540 KBytes para diferentes tamanhos de entrada. A implementação com otimizações apresenta um consumo de memória grosseiramente semelhante.



Fibonacci (linear) - Temperature

A implementação sem otimizações apresenta uma maior variação de temperatura, especialmente para tamanhos de entrada superiores, onde a temperatura é significativamente mais alta. A implementação com otimizações mantém uma temperatura mais estável e geralmente mais baixa.

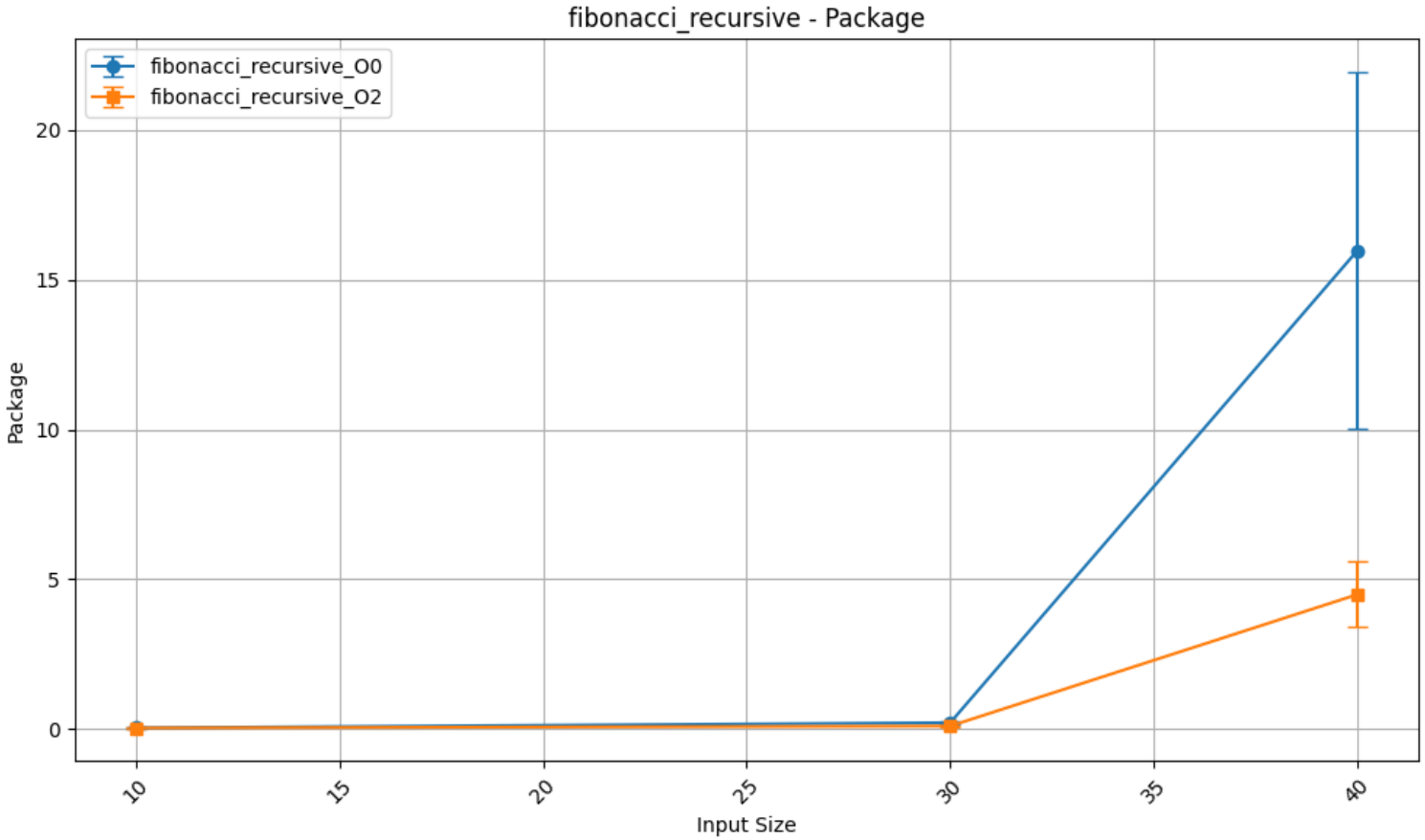


Fibonacci (linear) - Conclusão

Devido à complexidade desta função ser linear, a utilização de otimizações não vale a pena, tanto que influencia, negativamente, todas as variáveis estudadas. O overhead da utilização da otimização '-O2' será a provável causa desta influência negativa.

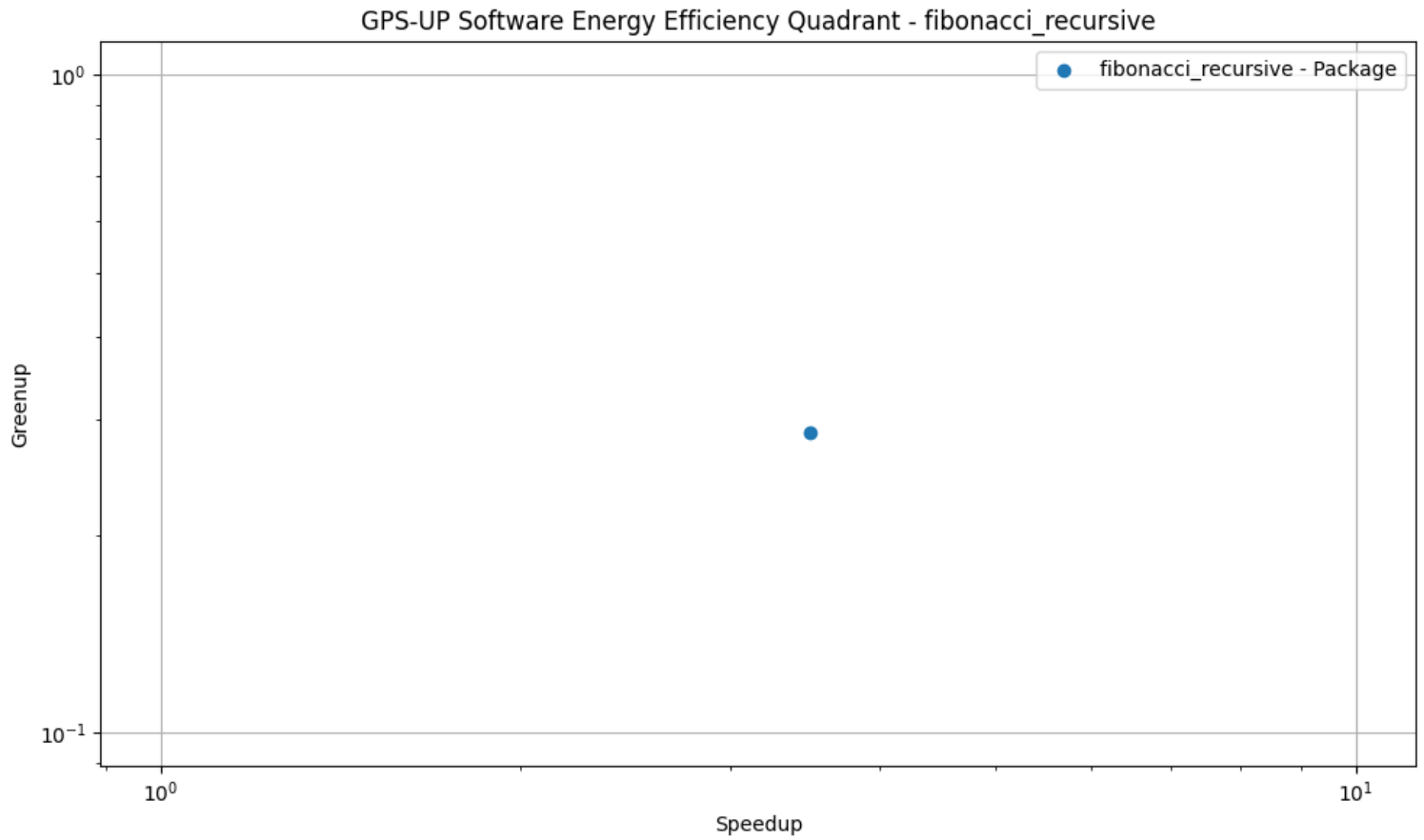
Fibonacci (recursive) - Package

A implementação sem otimizações tem gastos de energia significativamente maior para tamanhos de entrada maiores, especialmente a partir de 30, com grande variabilidade. A implementação com otimizações é mais eficiente, apresentando gastos de energia menores e mais consistentes.



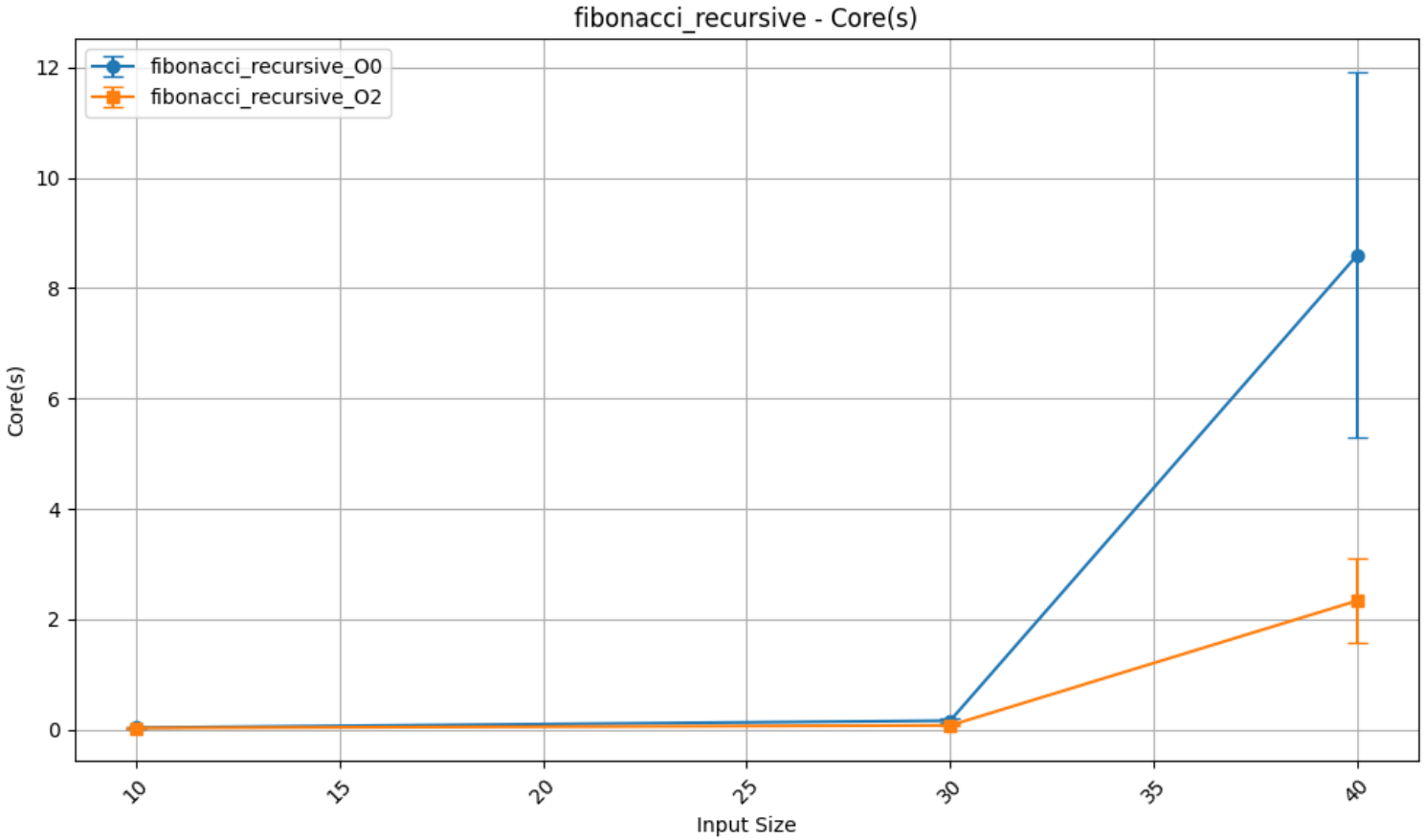
Fibonacci (recursive) - Package (gps_up)

O ponto encontra-se muito próximo do centro, o que indica que tem uma melhoria moderada, tanto em termos de velocidade como em termos de eficiência energética, isto em comparação com a versão não otimizada.



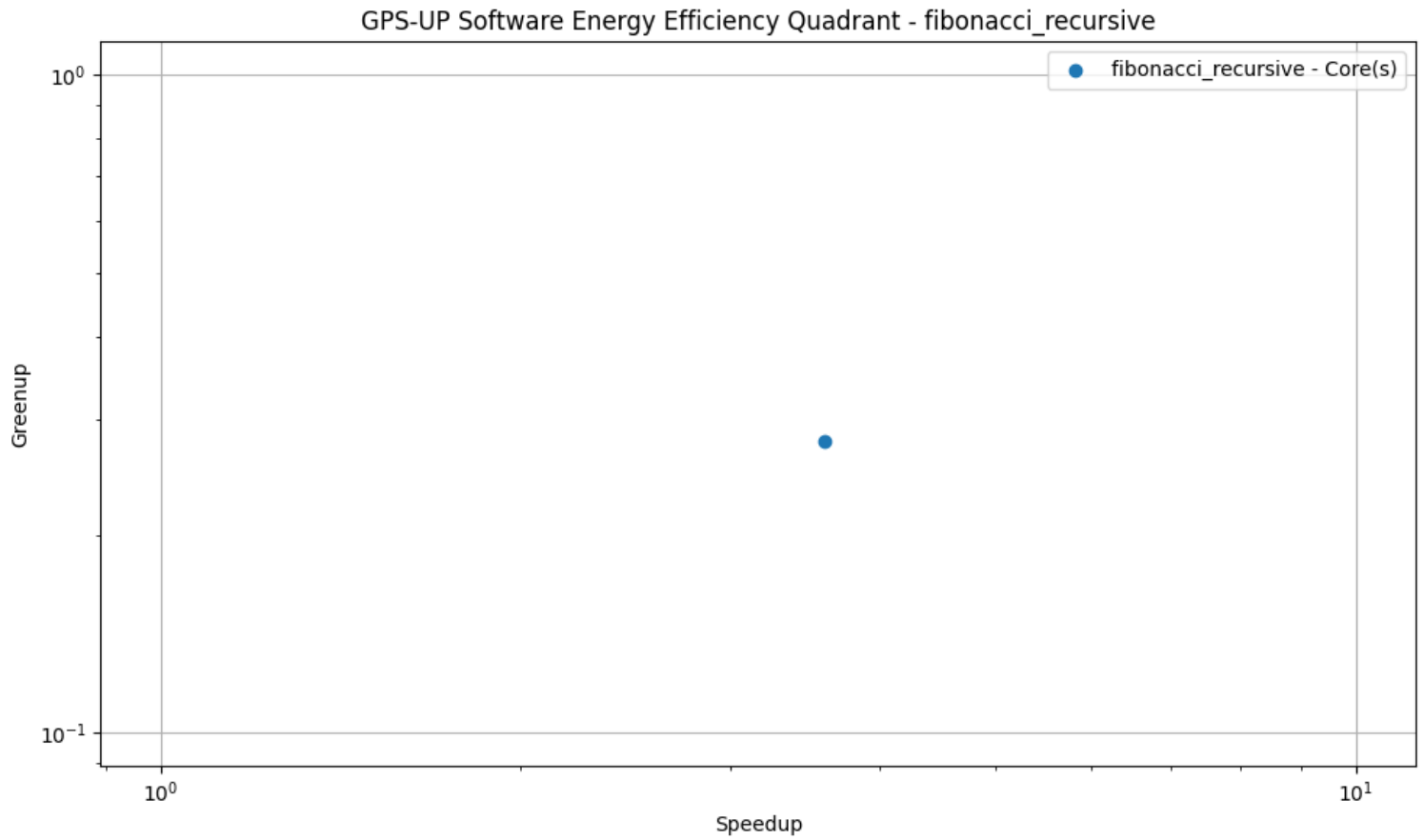
Fibonacci (recursive) - Core

A implementação sem otimizações tem gastos de energia significativamente maior para tamanhos de entrada maiores, especialmente a partir de 30, com grande variabilidade. A implementação com otimizações é mais eficiente, apresentando gastos de energia menores e mais consistentes.



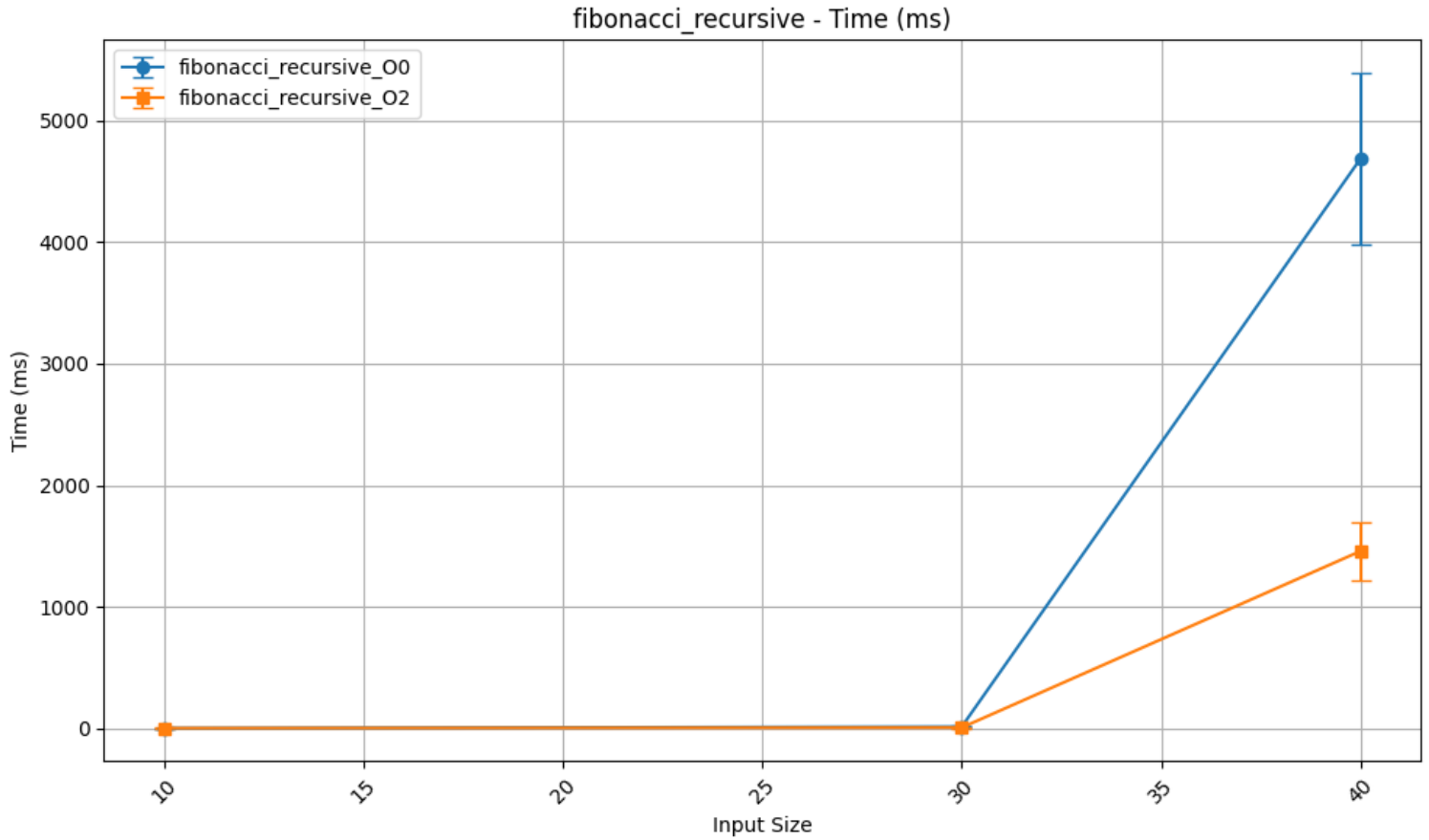
Fibonacci (recursive) - Core (gps_up)

O ponto encontra-se muito próximo do centro, o que indica que tem uma melhoria moderada, tanto em termos de velocidade como em termos de eficiência energética, isto em comparação com a versão não otimizada.



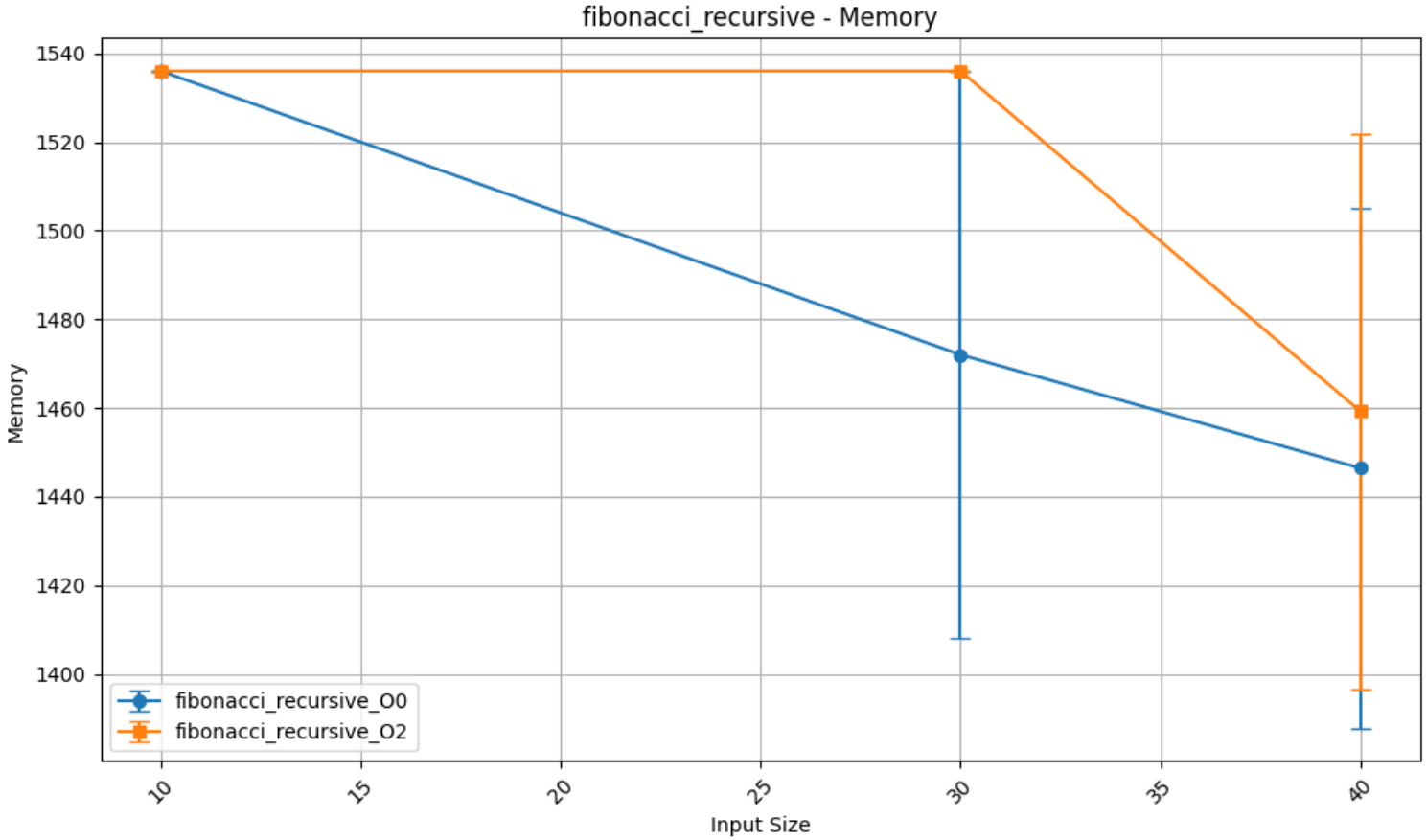
Fibonacci (recursive) - Time

Para termos de Fibonacci baixos, os tempos de execução são muito semelhantes, porém sendo a versão otimizada ligeiramente mais rápida. A discrepância é notória no cálculo do termo 40, onde existe uma diferença significativa entre ambas as versões e uma variação de memória muito mais instável, na versão menos otimizada.



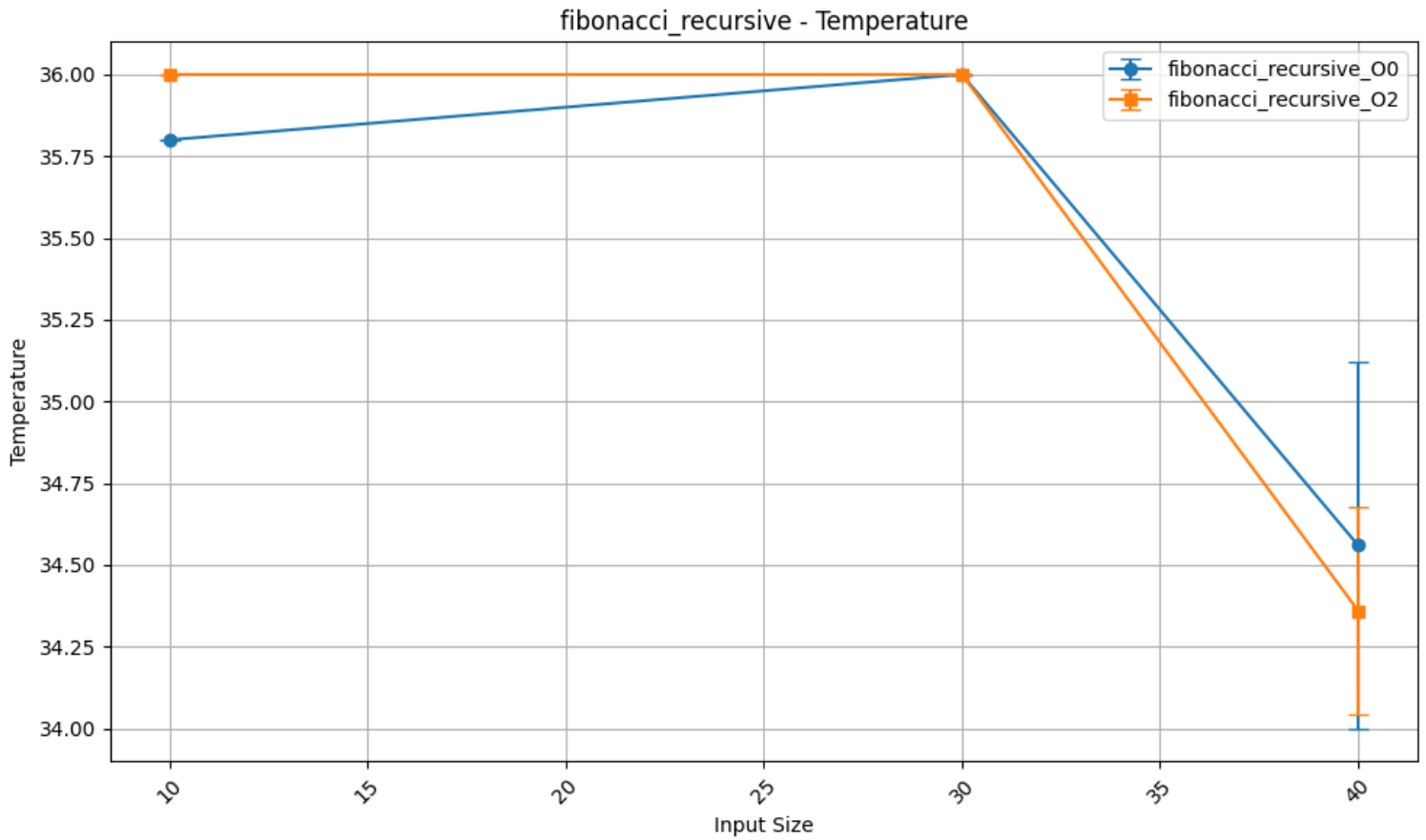
Fibonacci (recursive) - Memory

As diferenças de memória é notável no cálculo do termo 30 de Fibonacci, onde os valores oscilaram entre 1408 e os 1536. A versão otimizada tornou apenas esse valor estável. No cálculo do termo 40 de Fibonacci, as oscilações entre os dois valores mencionados é notável.



Fibonacci (recursive) - Temperature

Houve uma queda significativa de temperatura para o cálculo do termo 40 de Fibonacci, ao qual não compreendi. De resto, a temperatura mantém-se praticamente a mesma, sem muitas oscilações.



Fibonacci (recursive) - Conclusão

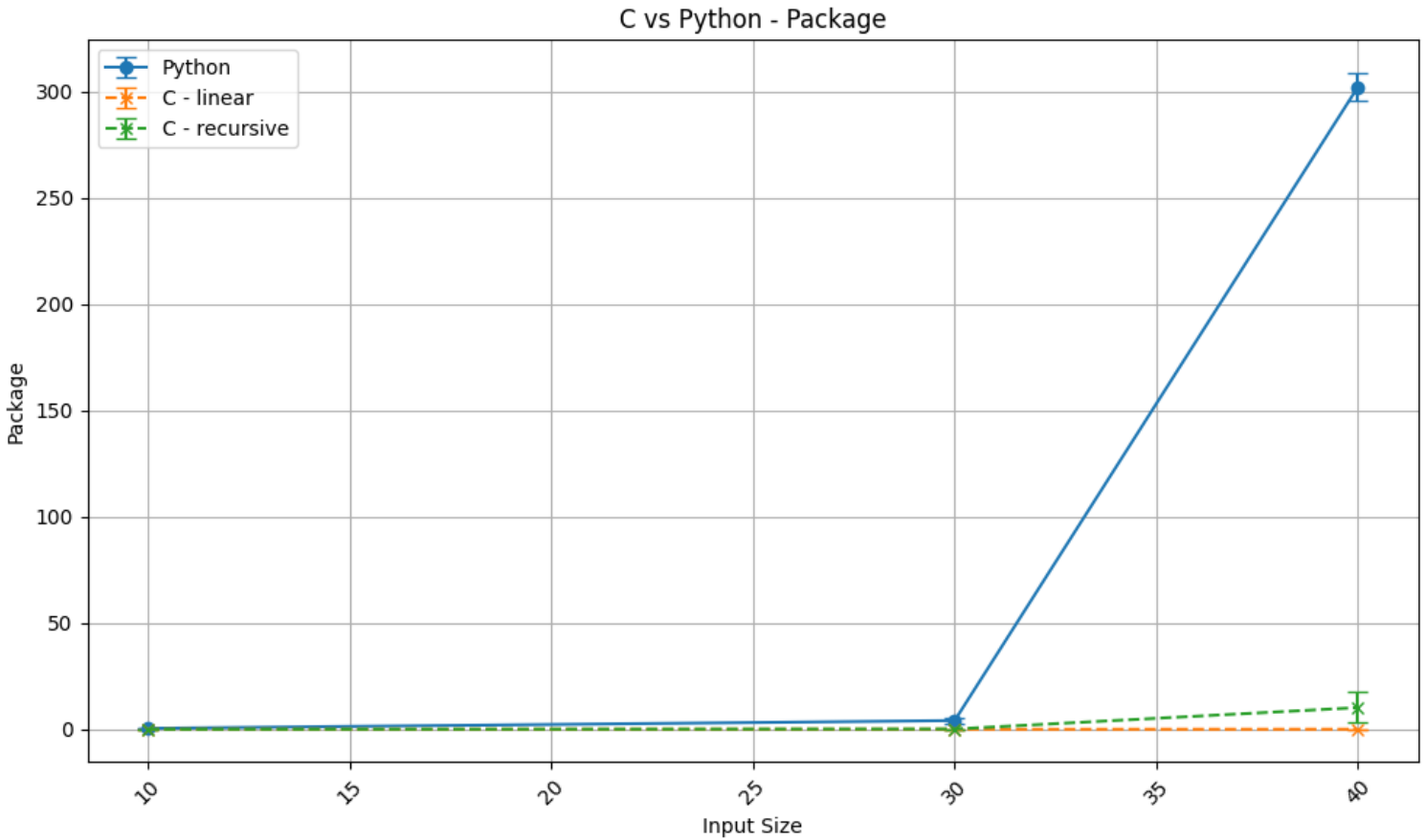
De forma geral, a utilização da otimização '-O2' no compilador influencia, positivamente, tanto a energia consumida, bem como o tempo utilizado para fazer os cálculos dos termos. Neste caso, a utilização da flag compensou, mesmo os testes terem sido pequenos. Acredito que para resultados de termos superiores, seria mais visível as melhorias da otimização.

Fibonacci - Python

A título de curiosidade das aulas práticas de Tópicos de Desenvolvimento de Software, resolvi criar uma função *Python* do Algoritmo de Fibonacci, com um termo de complexidade $O(2^N)$. Não utilizei qualquer tipo de otimizações, mantive a mesma máquina e os testes realizados, anteriormente.

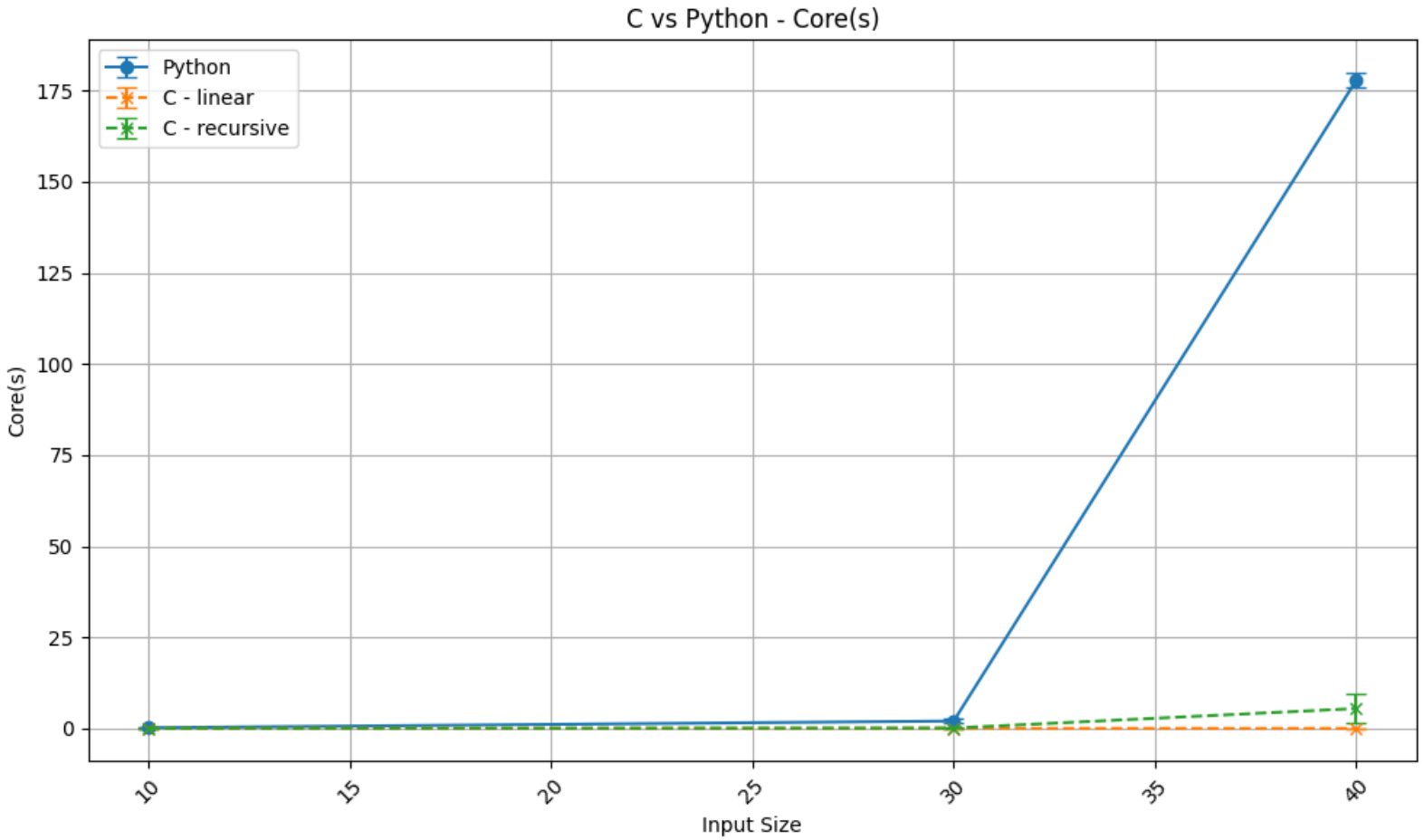
Fibonacci (Python) - Package

As implementações em C são significativamente mais eficientes energeticamente do que a implementação em Python, especialmente para tamanhos de entrada maiores. A implementação em Python não escala bem em termos de consumo de energia, enquanto as implementações em C mantêm um consumo de energia estável e baixo.



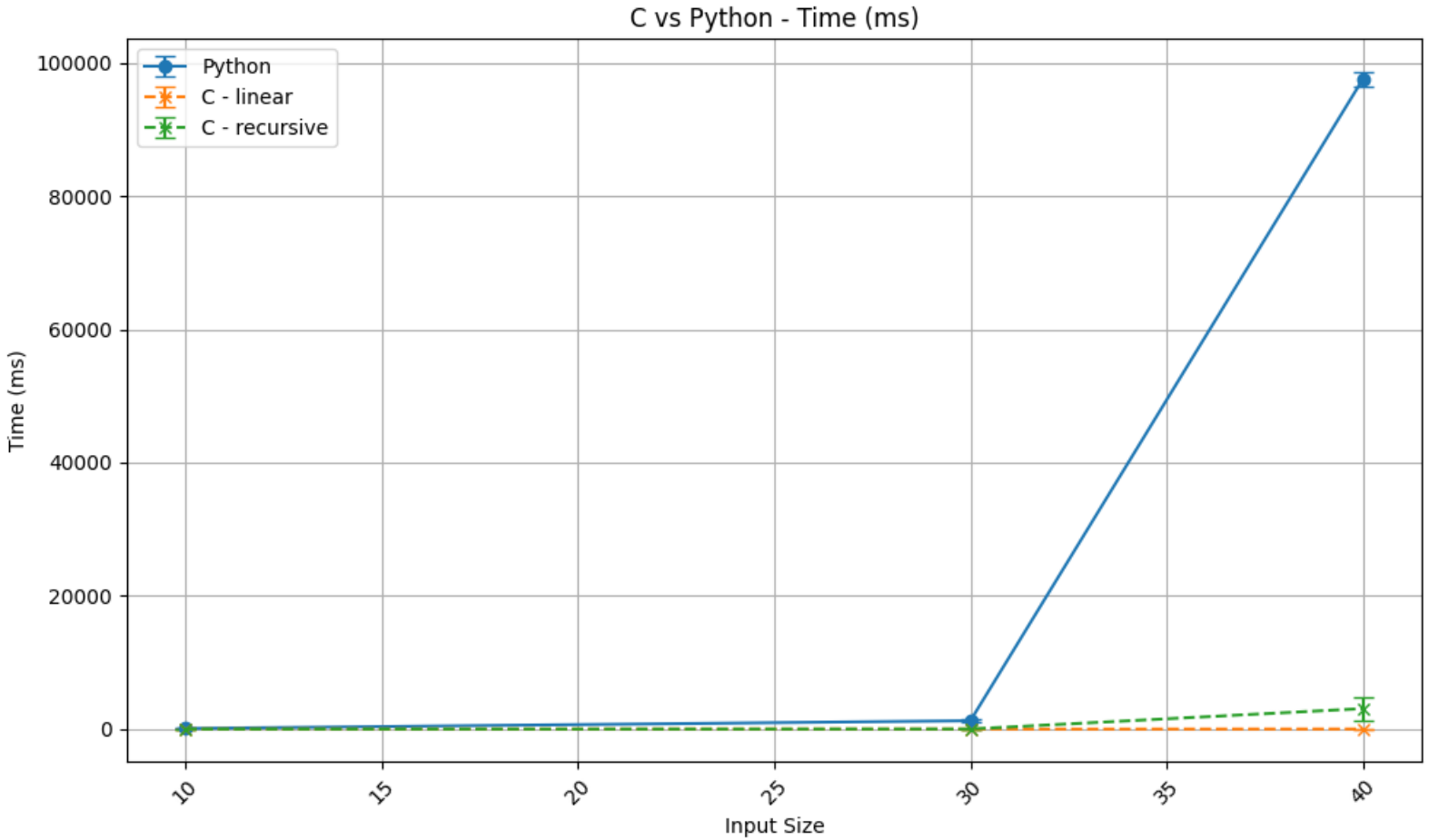
Fibonacci (Python) - Core

As mesmas evidências são encontradas neste gráfico.



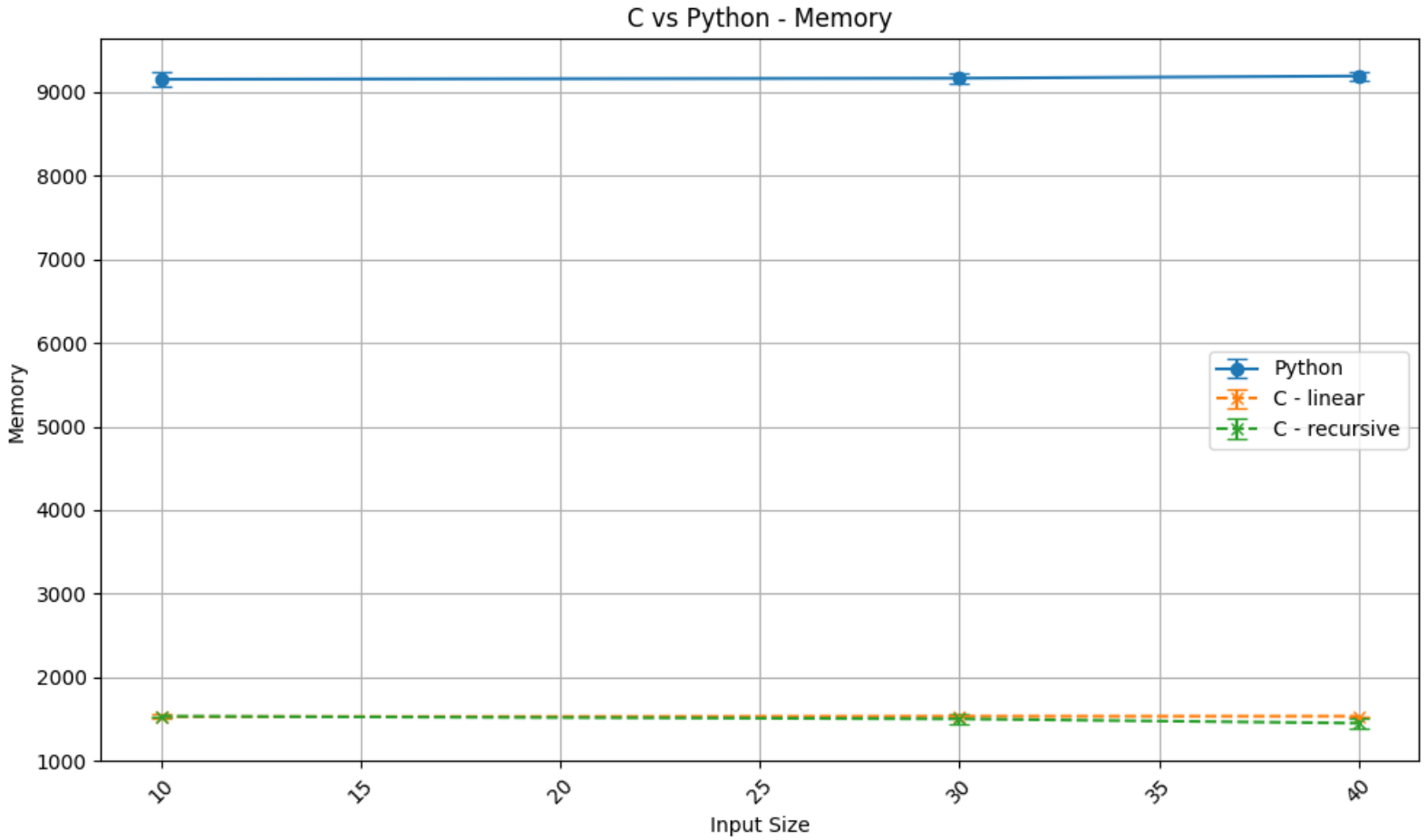
Fibonacci (Python) - Time

As implementações em C são significativamente mais rápidas do que a implementação em Python, especialmente para tamanhos de entrada maiores, onde a diferença é absurda. Até a escala deixa de ser ótima devido ao imenso tempo utilizado no cálculo do termo 40 de Fibonacci.



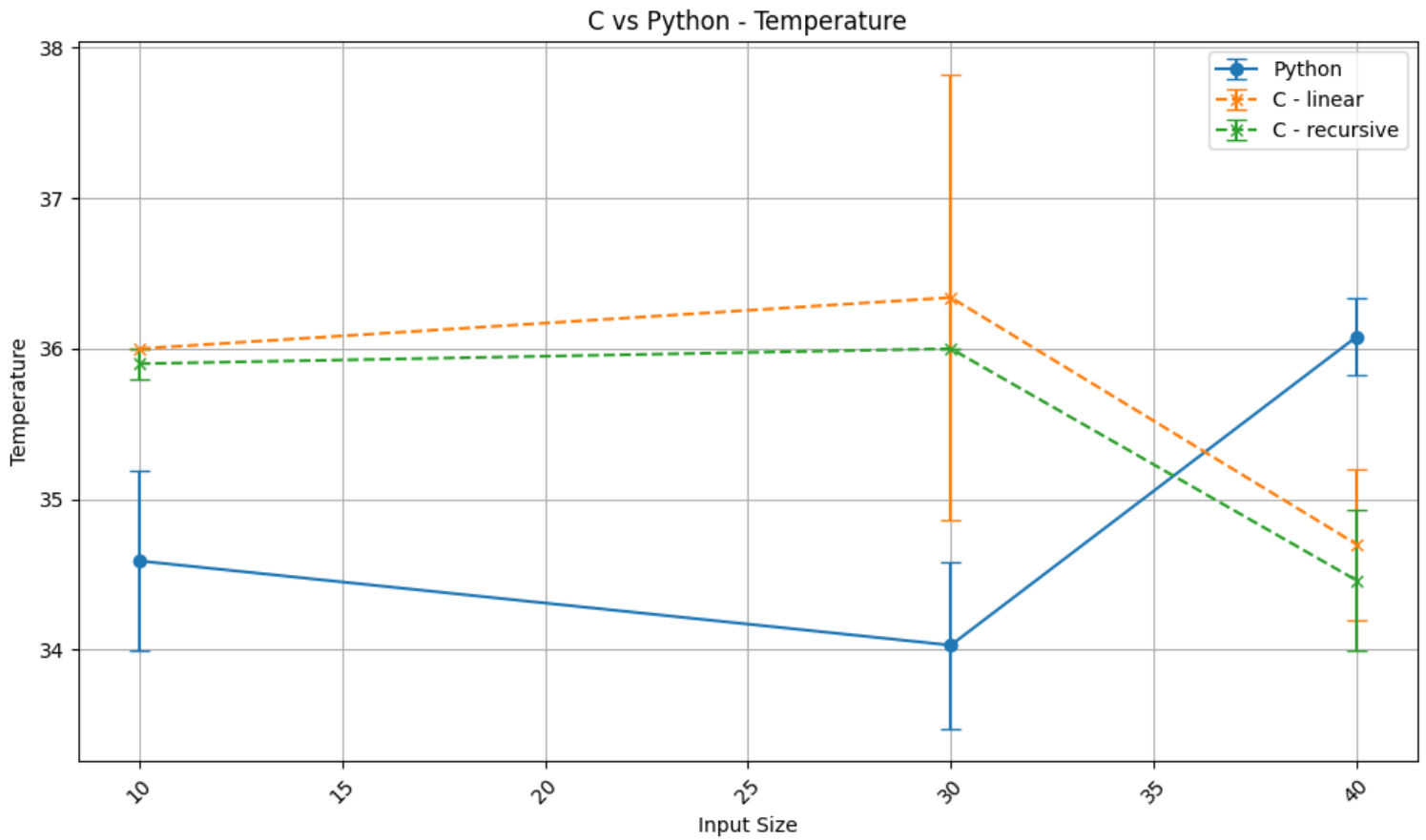
Fibonacci (Python) - Memory

Mais uma vez, comprova-se que o mesmo algoritmo em Python consome mais de 9000 Kbytes de espaço em memória, seja qualquer os valores de termos de Fibonacci a serem calculados.



Fibonacci (Python) - Temperature

A implementação em C recursiva é a mais estável em termos de temperatura, enquanto a implementação em C linear mostra uma variação significativa para o tamanho de entrada 30. A implementação em Python, apesar de apresentar variações, mantém uma temperatura relativamente baixa para tamanhos de entrada maiores.



Fibonacci (Python vs C) - Conclusão

Como seria esperado, para valores pequenos a diferença no tempo de execução entre as implementações em Python e C é mínima. No entanto, à medida que a complexidade dos cálculos aumenta, a diferença no tempo de execução torna-se bastante notável.
Em resumo, as implementações em C, tanto linear quanto recursiva, são muito mais eficientes em termos de tempo de execução em comparação com a implementação em Python, especialmente para tamanhos de entrada maiores.

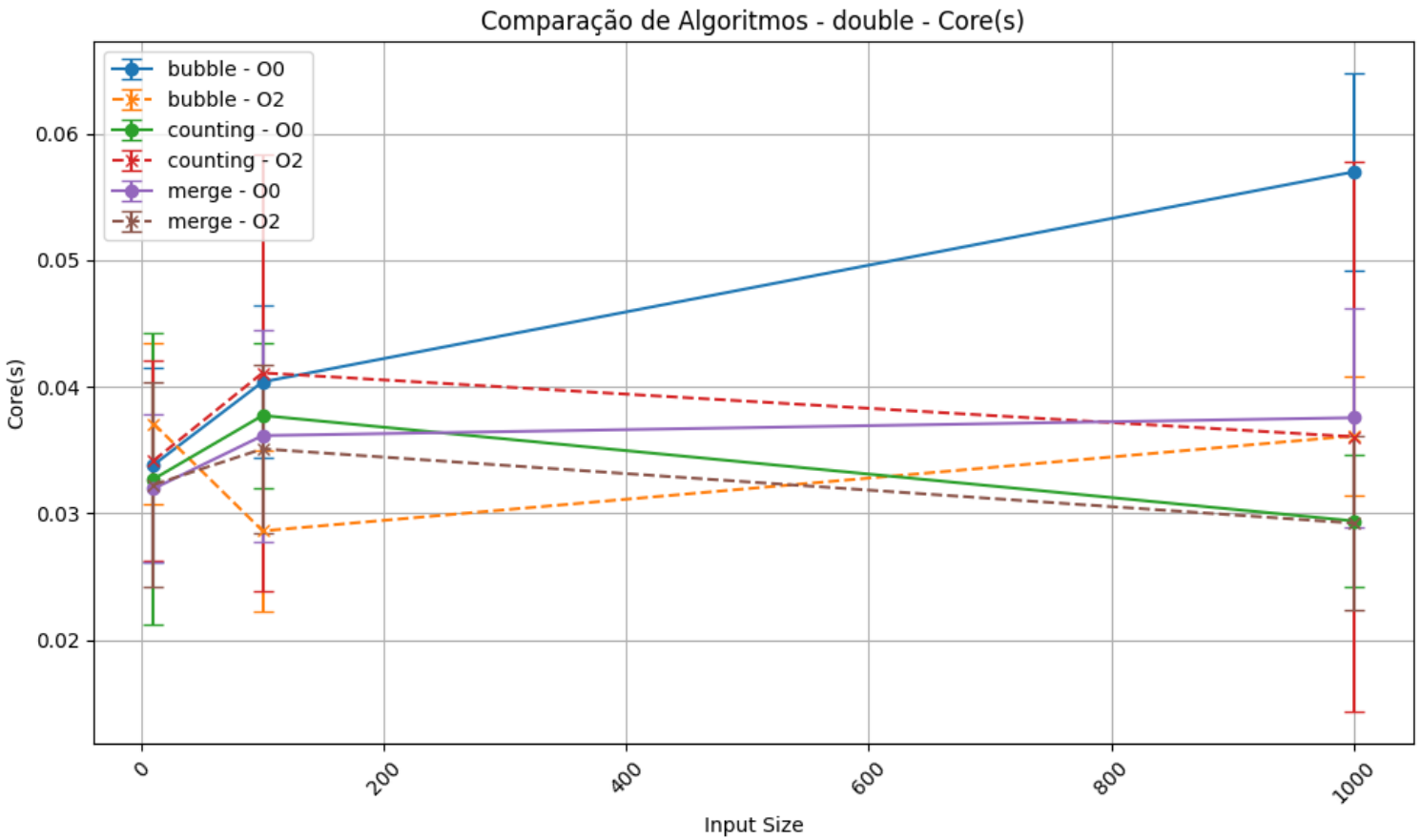
Algoritmos de Ordenação

Conforme sugerido pela equipa docente decidi escolher 3 algoritmos de ordenação, onde utilizei arrays de vários tamanhos (10 , 100 e 1000) e também usei vários tipos de dados (*double, float, int e string*):

- Bubble Sort: $O(n^2)$
- Merge Sort: $O(n/logn)$
- Counting Sort: $O(n + k)$

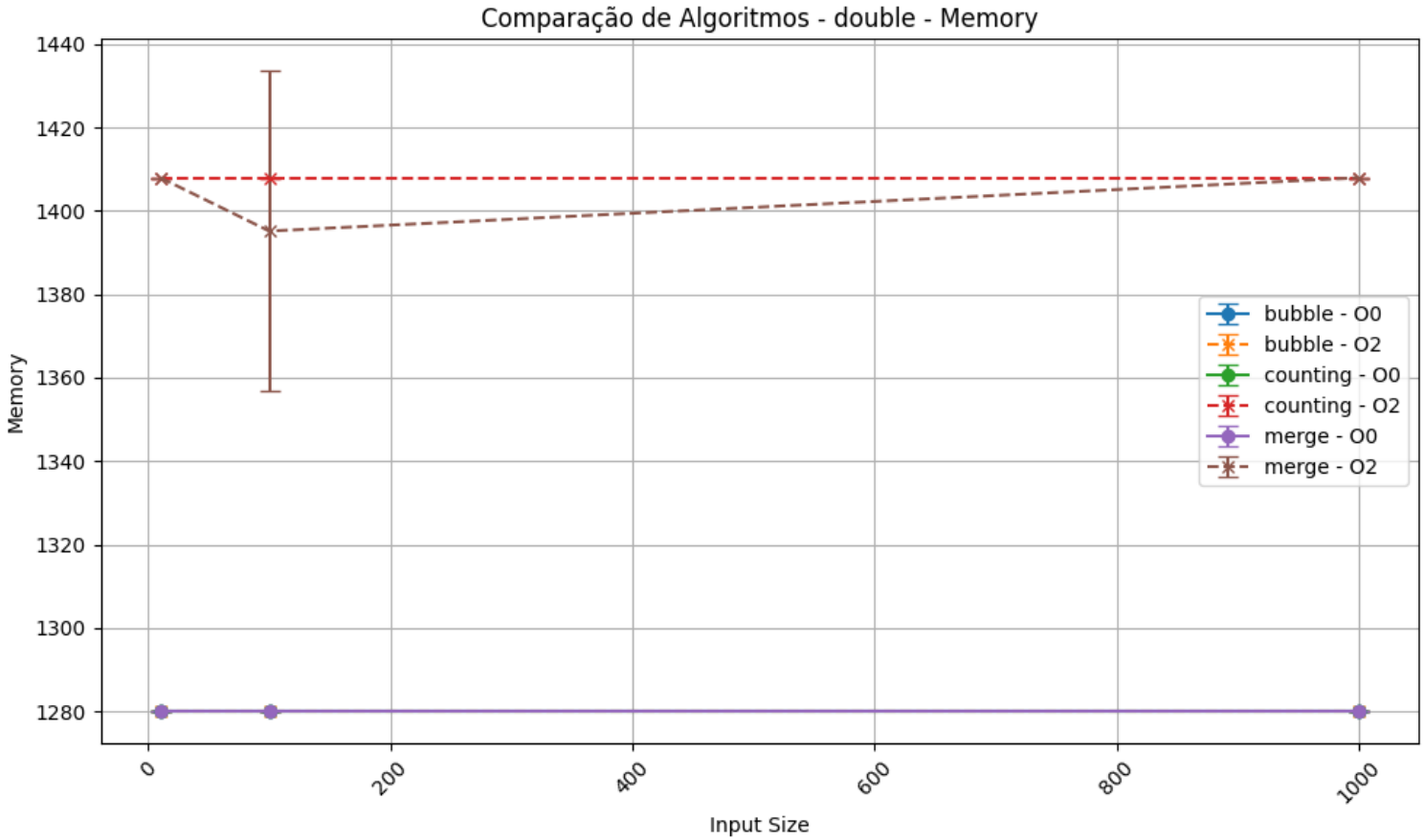
Data Type: Double - Core

O *counting sort* é o algoritmo mais eficiente e estável para os tamanhos de entrada testados, seguido pelo *merge sort*. O *bubble sort*, como esperado, tem o pior desempenho, especialmente sem otimização. A otimização do compilador '-O2' melhora o desempenho, mas o impacto varia entre os algoritmos.



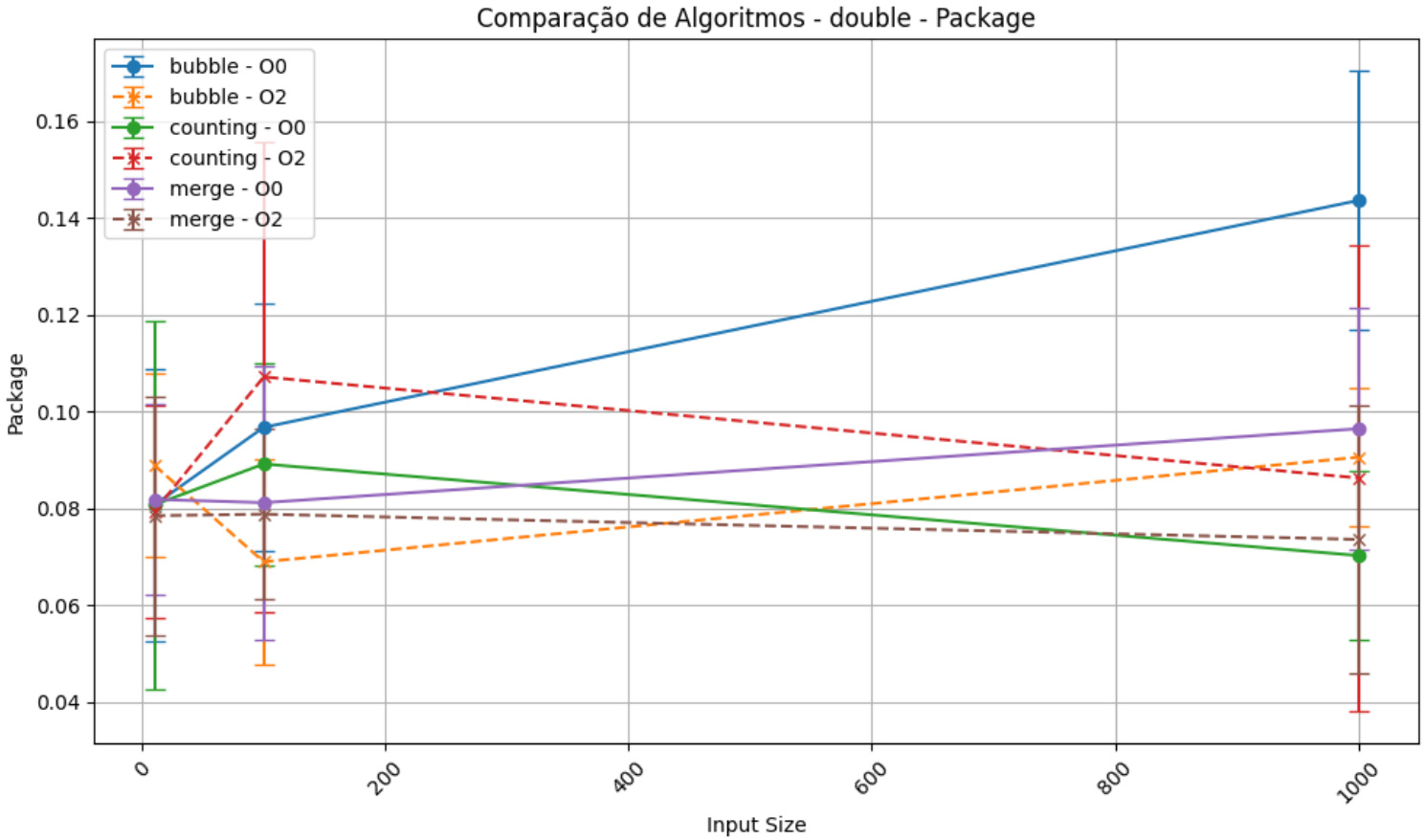
Data Type: Double - Memory

Apenas as versões de *counting* e *merge sort* que utilizem a flag '-O2' é que tem valores superiores de utilização de memória, as restantes implementações apenas utilizam 1280 Kbytes.



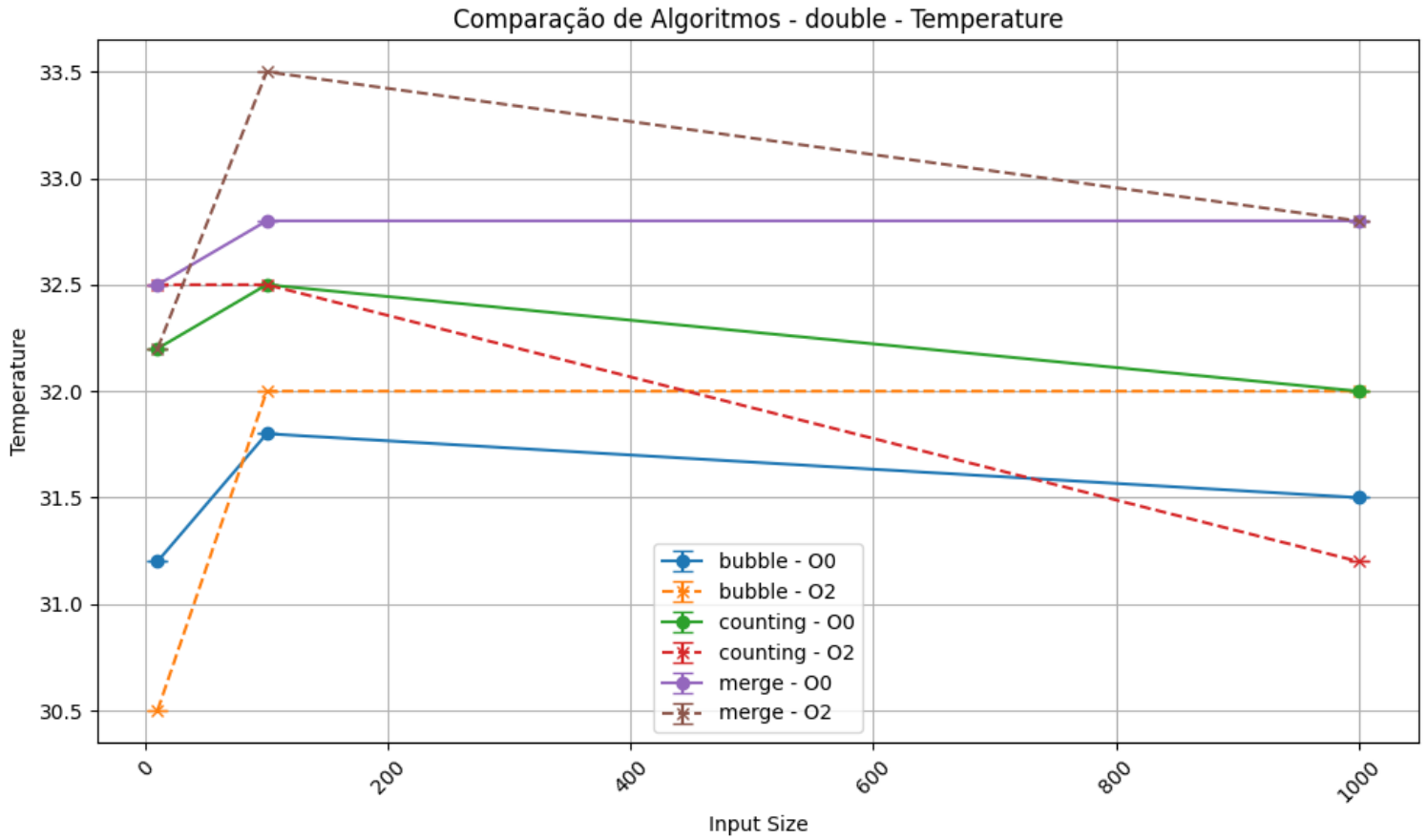
Data Type: Double - Package

As mesmas observações feitas acerca dos valores do Package, são feitas nesta variável de energia.



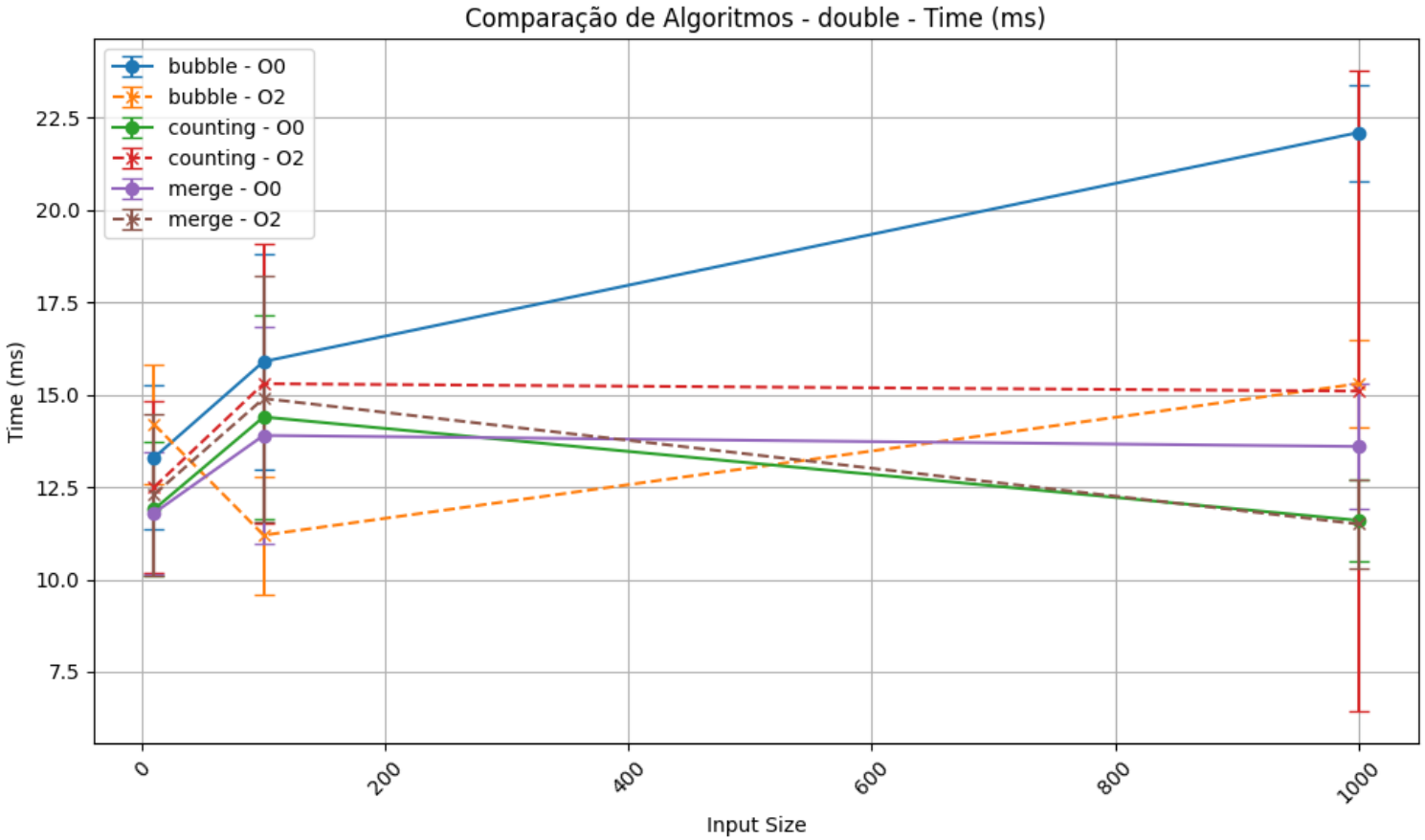
Data Type: Double - Temperature

A temperatura durante os cálculos do *bubble sort* foram inferiores aos restantes algoritmos. É de notar no pico de temperatura atingida durante o algoritmo *merge sort* com a utilização da flag '-O2'.



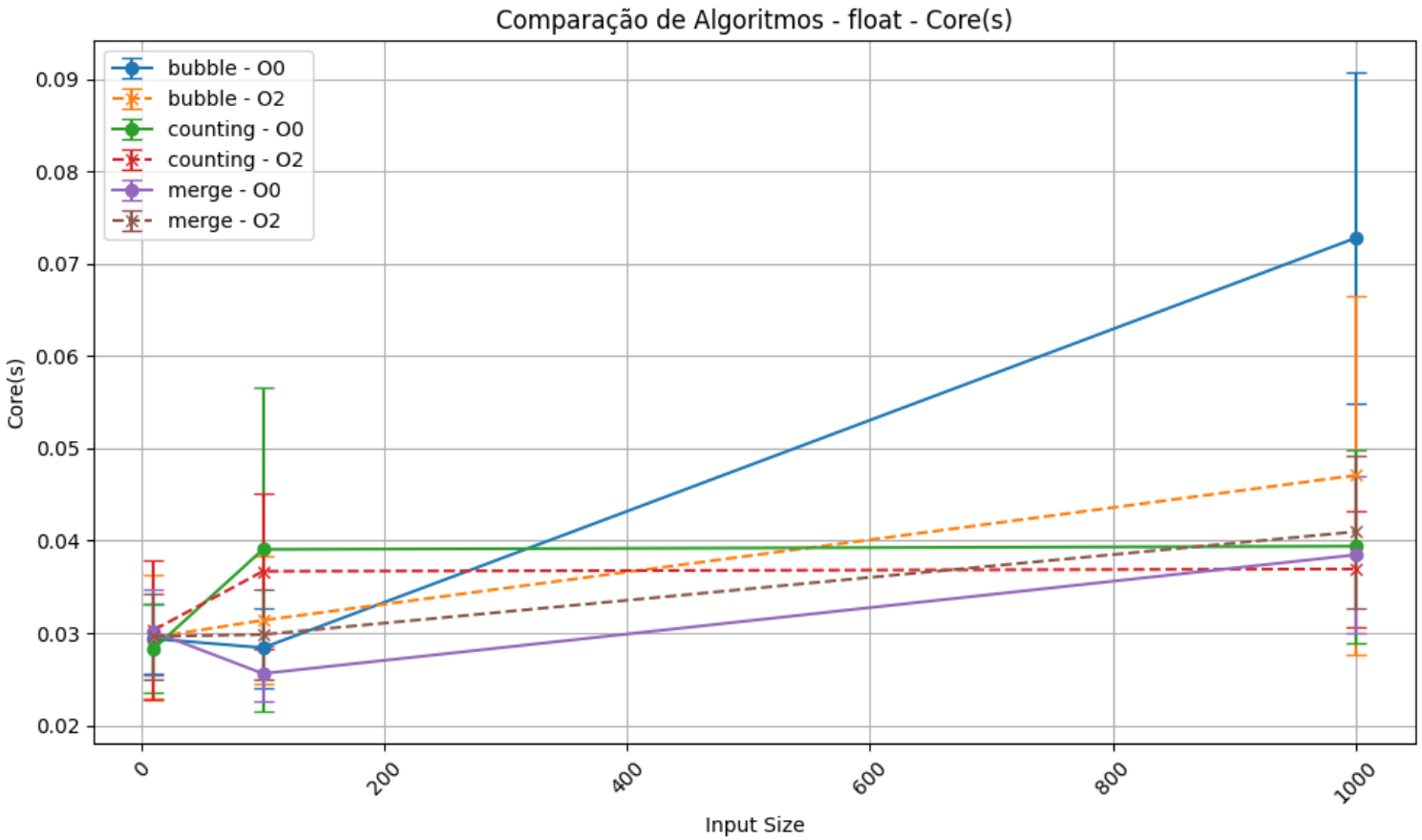
Data Type: Double - Time

Os valores tiveram tempos de execução bastante semelhantes, exceto o algoritmo *bubble sort* sem qualquer tipo de otimização. É de notar que o algoritmo *counting sort* é o mais rápido, mesmo para valores superiores.



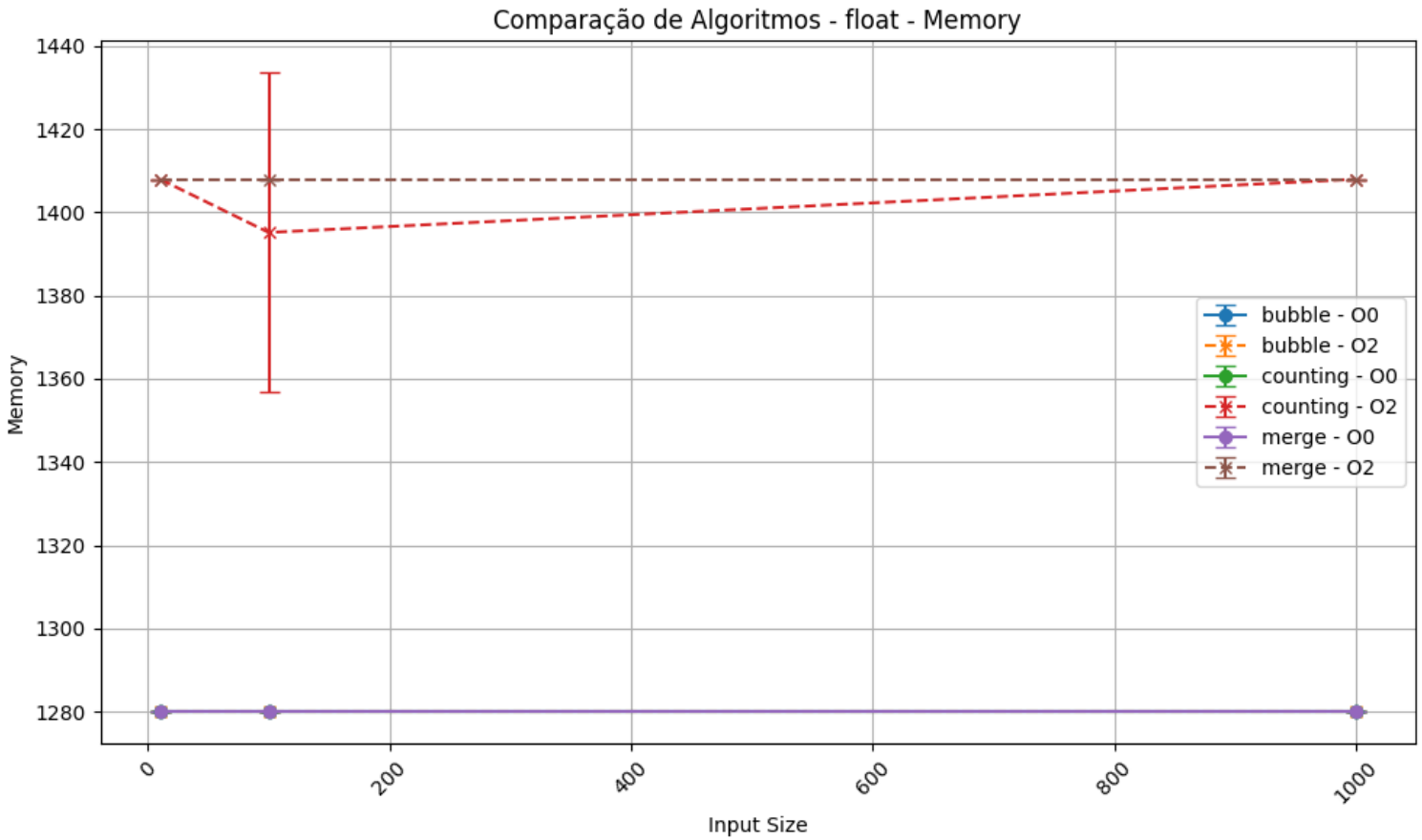
Data Type: Float - Core

Os resultados são muito semelhantes, à exceção do algoritmo *bubble sort* que tem um gasto de energia muito superior para valores mais altos.



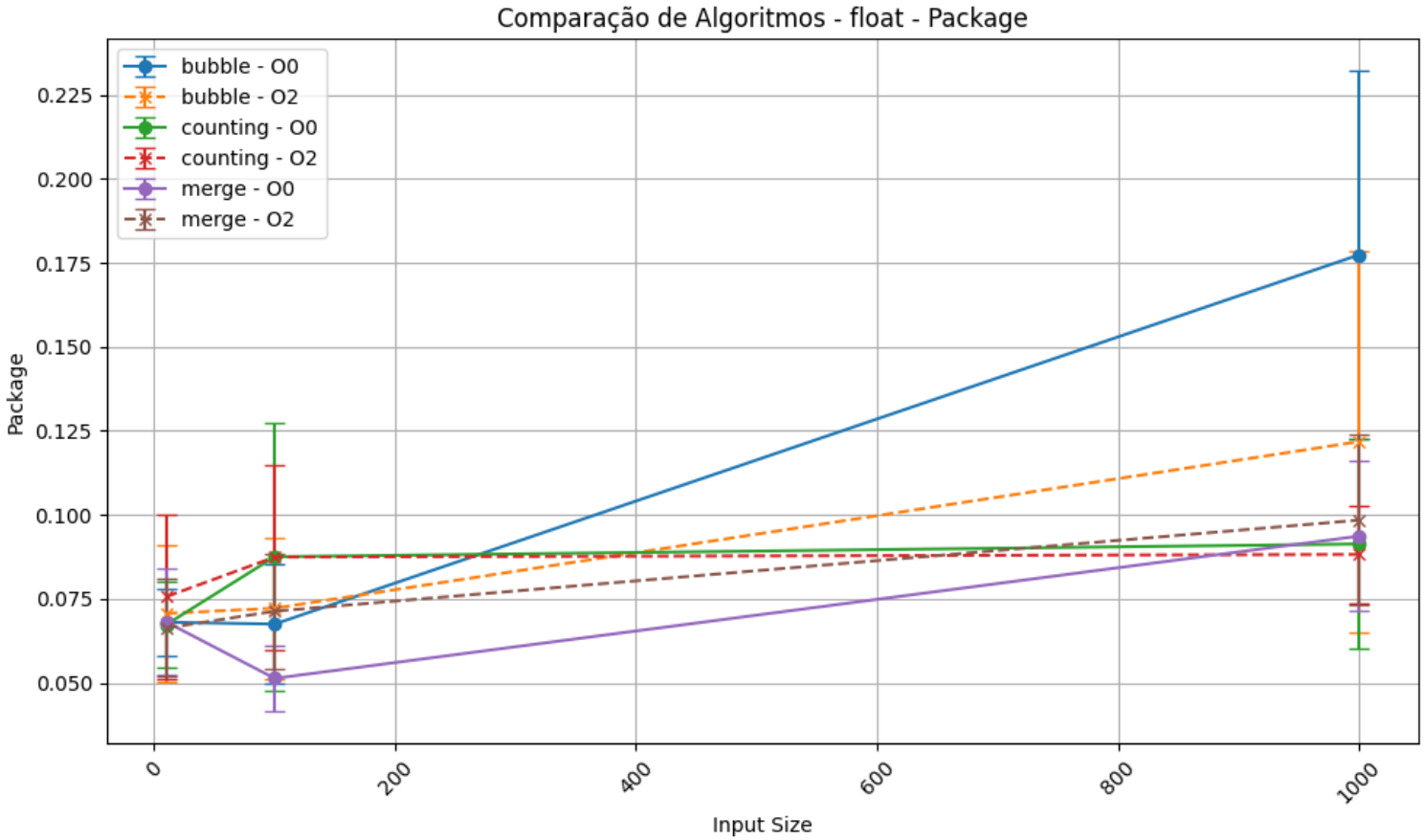
Data Type: Float - Memory

Os resultados observados com o *data type double* são rigorosamente iguais.



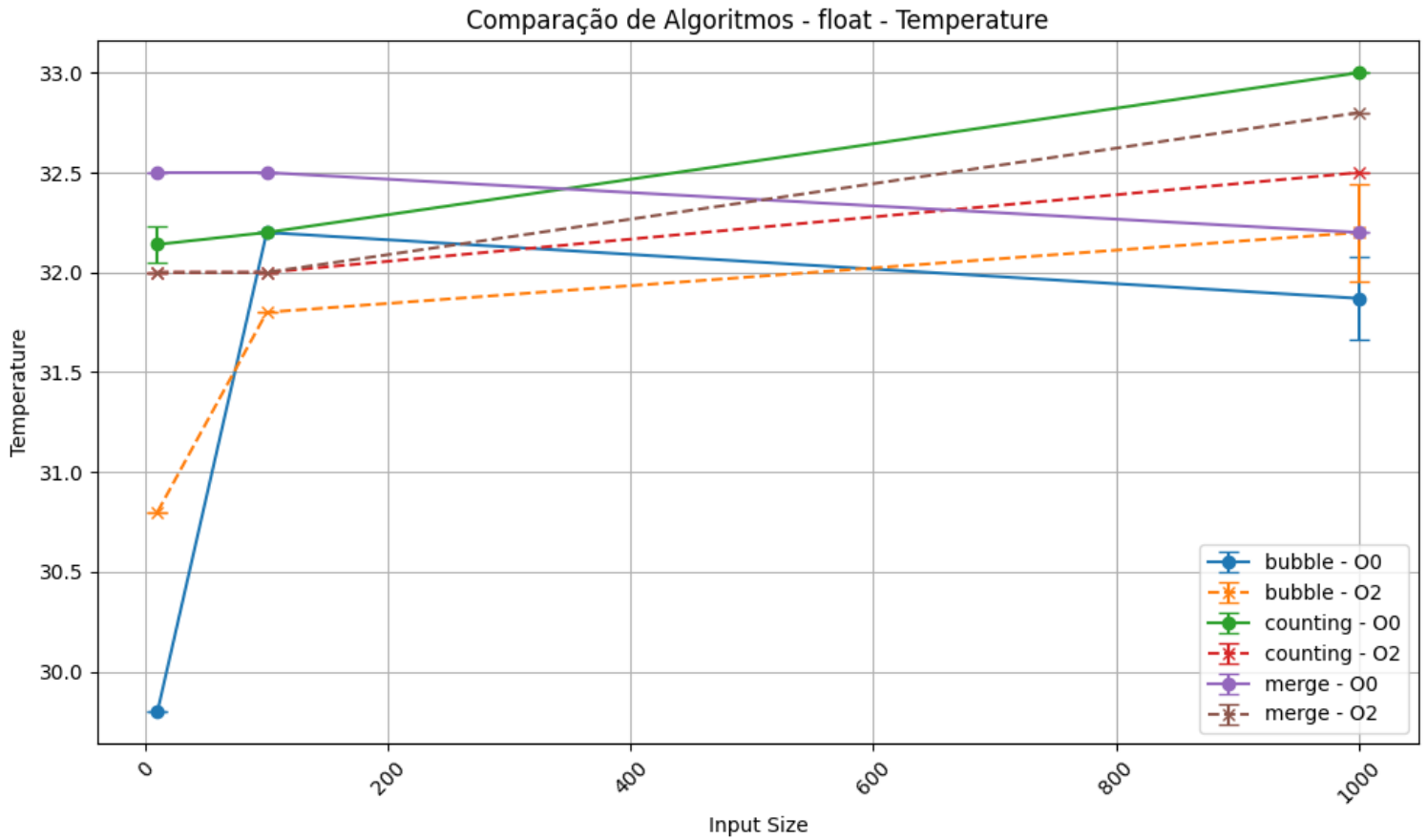
Data Type: Float - Package

Os resultados são muito semelhantes, à exceção do algoritmo *bubble sort* que tem um gasto de energia muito superior para valores mais altos.



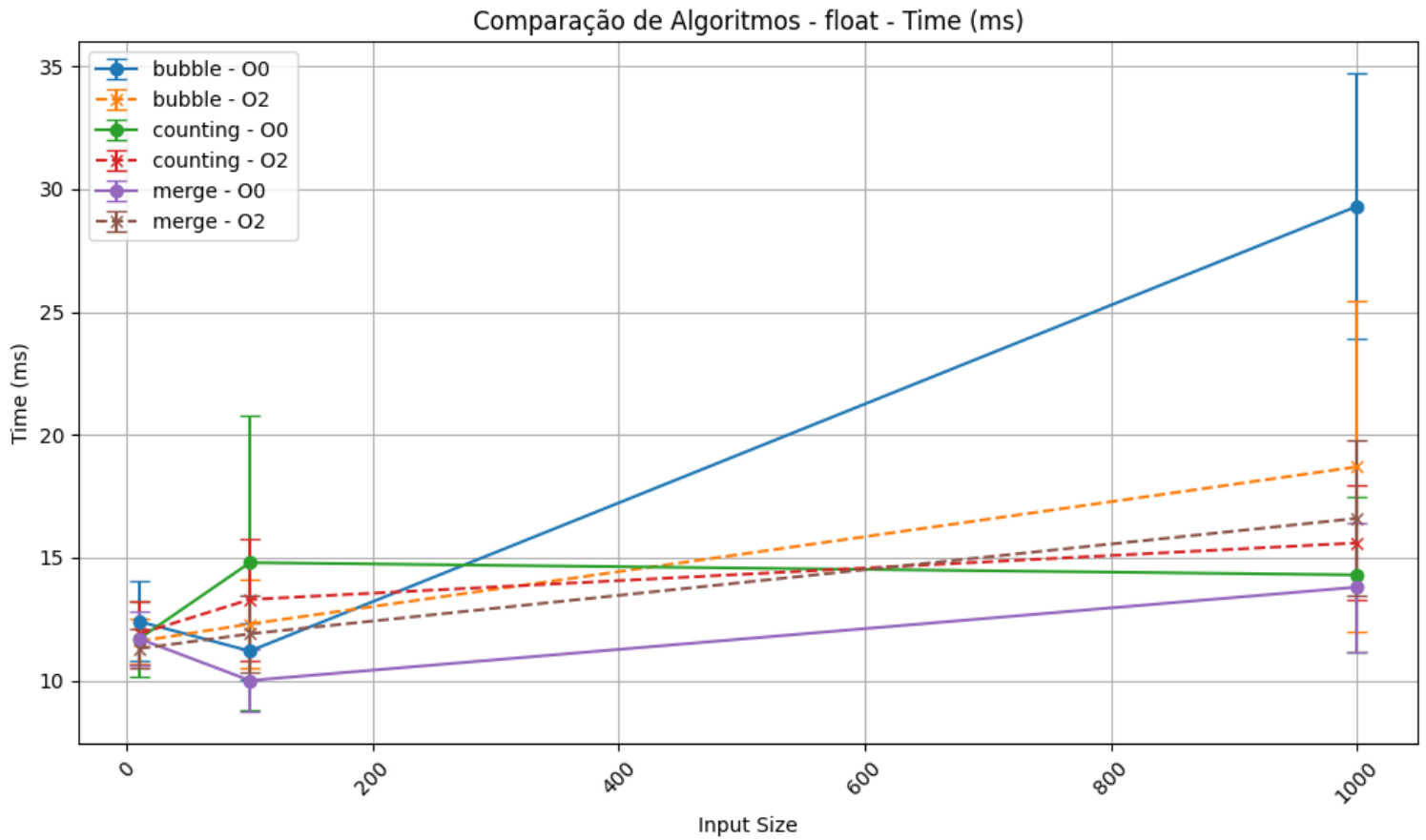
Data Type: Float - Temperature

As variâncias de temperatura são quase nulas, apenas no arranque dos testes, a temperatura aumentou conforme a ordem dos testes realizados.



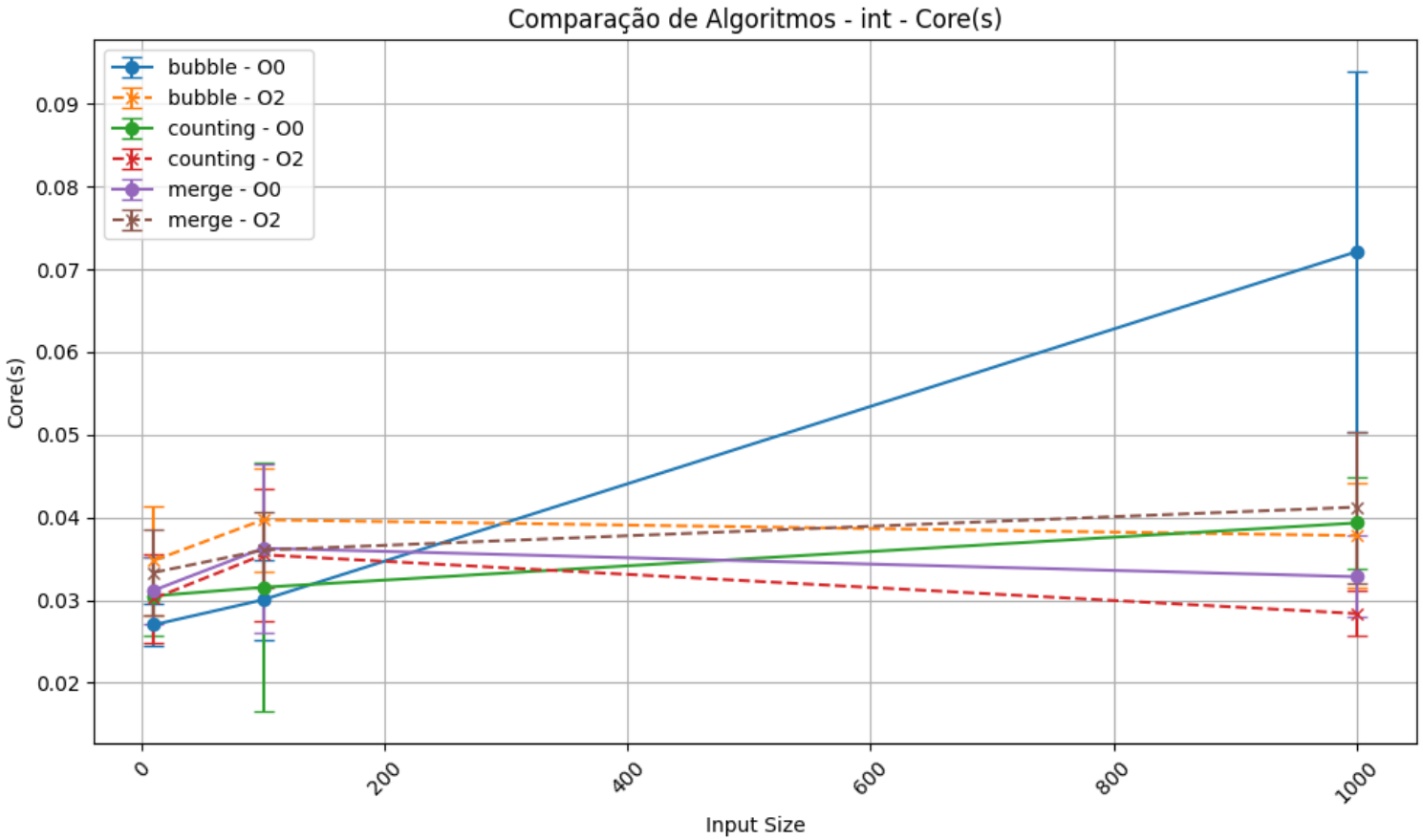
Data Type: Float - Time

Os valores tiveram tempos de execução bastante semelhantes, exceto o algoritmo *bubble sort* sem qualquer tipo de otimização. É de notar que o algoritmo *counting sort* e o *merge sort* são os mais rápidos, mesmo para valores superiores.



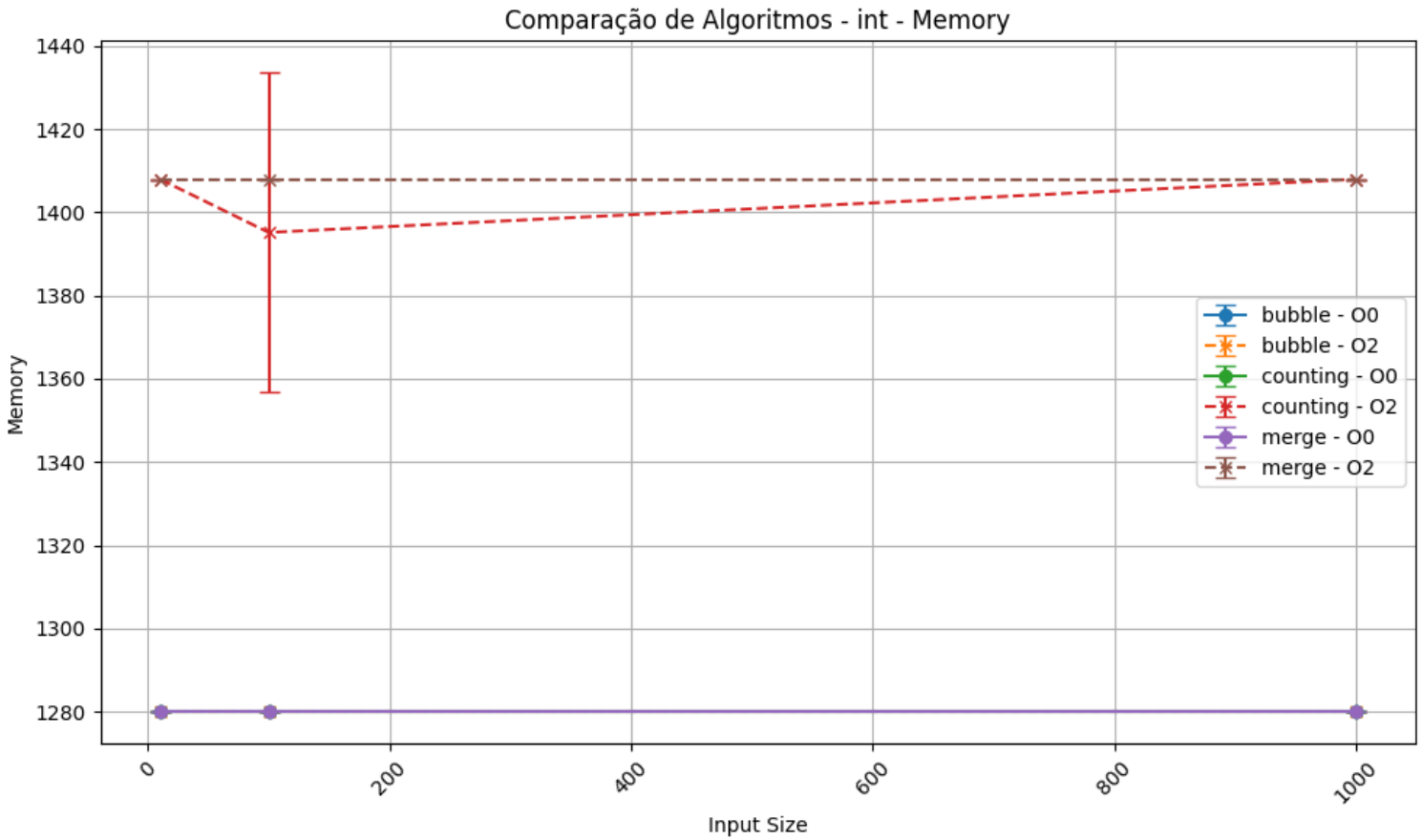
Data Type: Int - Core

O *counting sort* é o algoritmo mais eficiente e estável para os tamanhos de entrada testados, seguido pelo *merge sort*. O *bubble sort*, como esperado, tem o pior desempenho, especialmente sem otimização. A otimização do compilador '-O2' melhora o desempenho, mas o impacto varia entre os algoritmos.



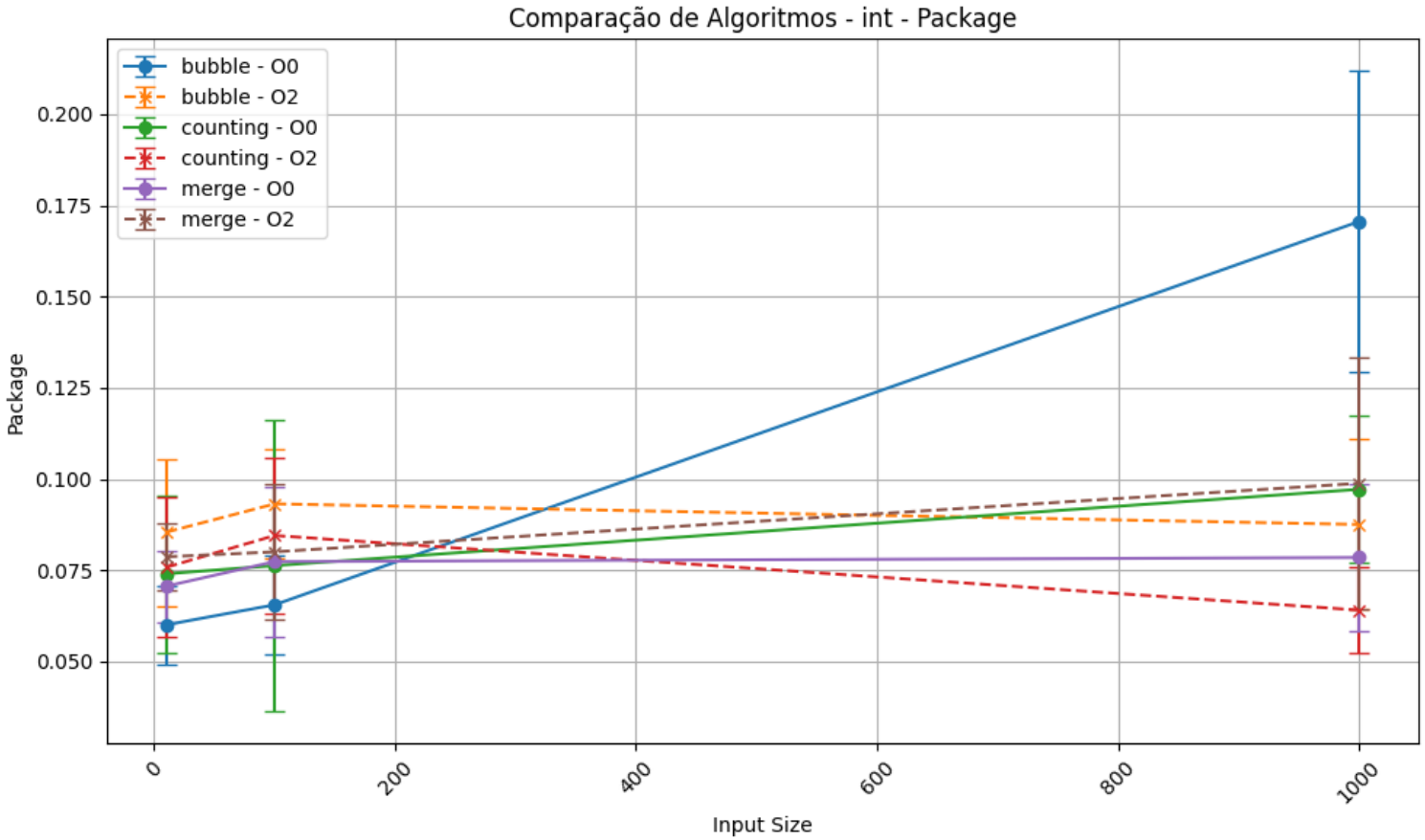
Data Type: Int - Memory

Os resultados observados com o *data type double* são rigorosamente iguais.



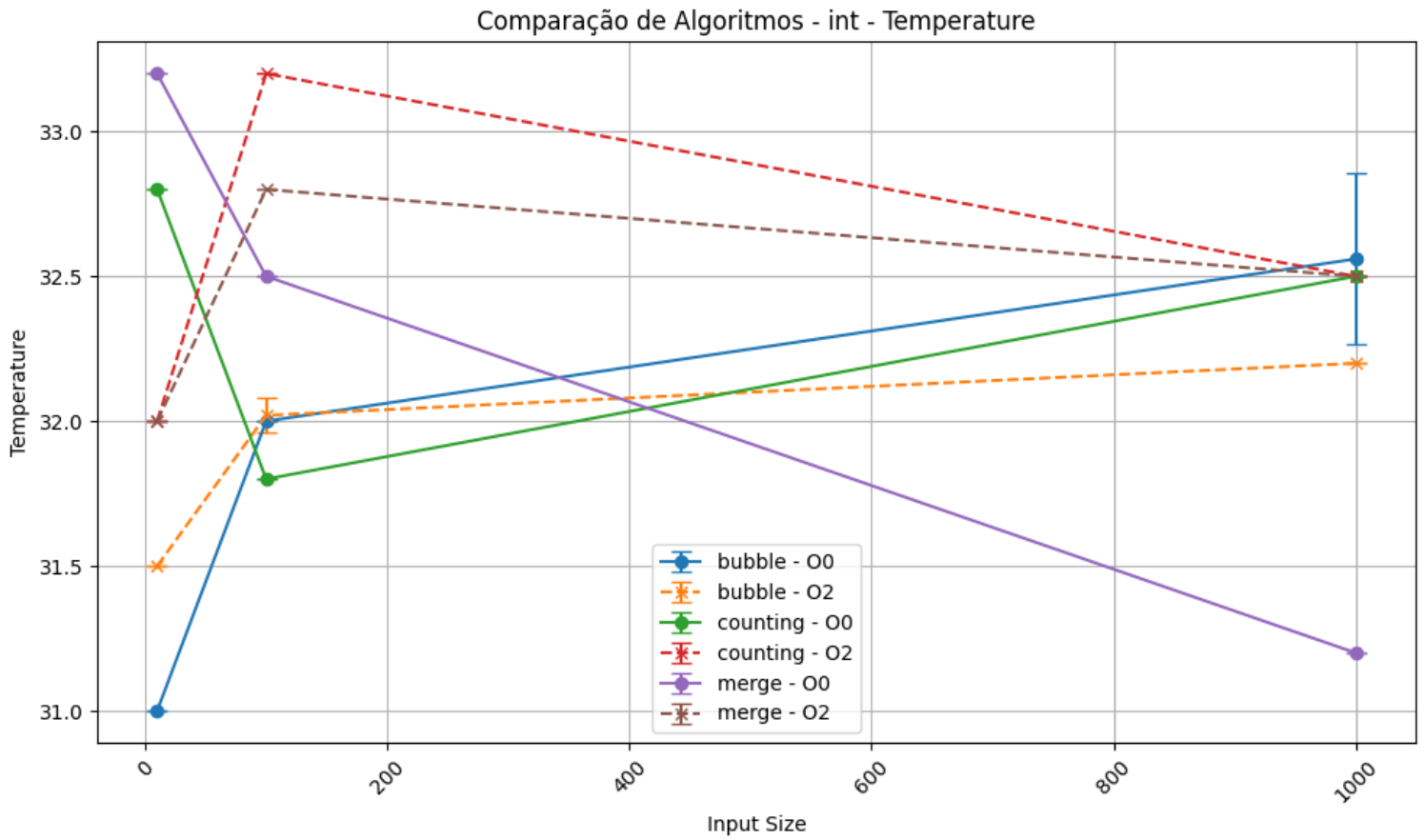
Data Type: Int - Package

Os resultados são extremamente semelhantes aos resultados apresentados pela variável float.



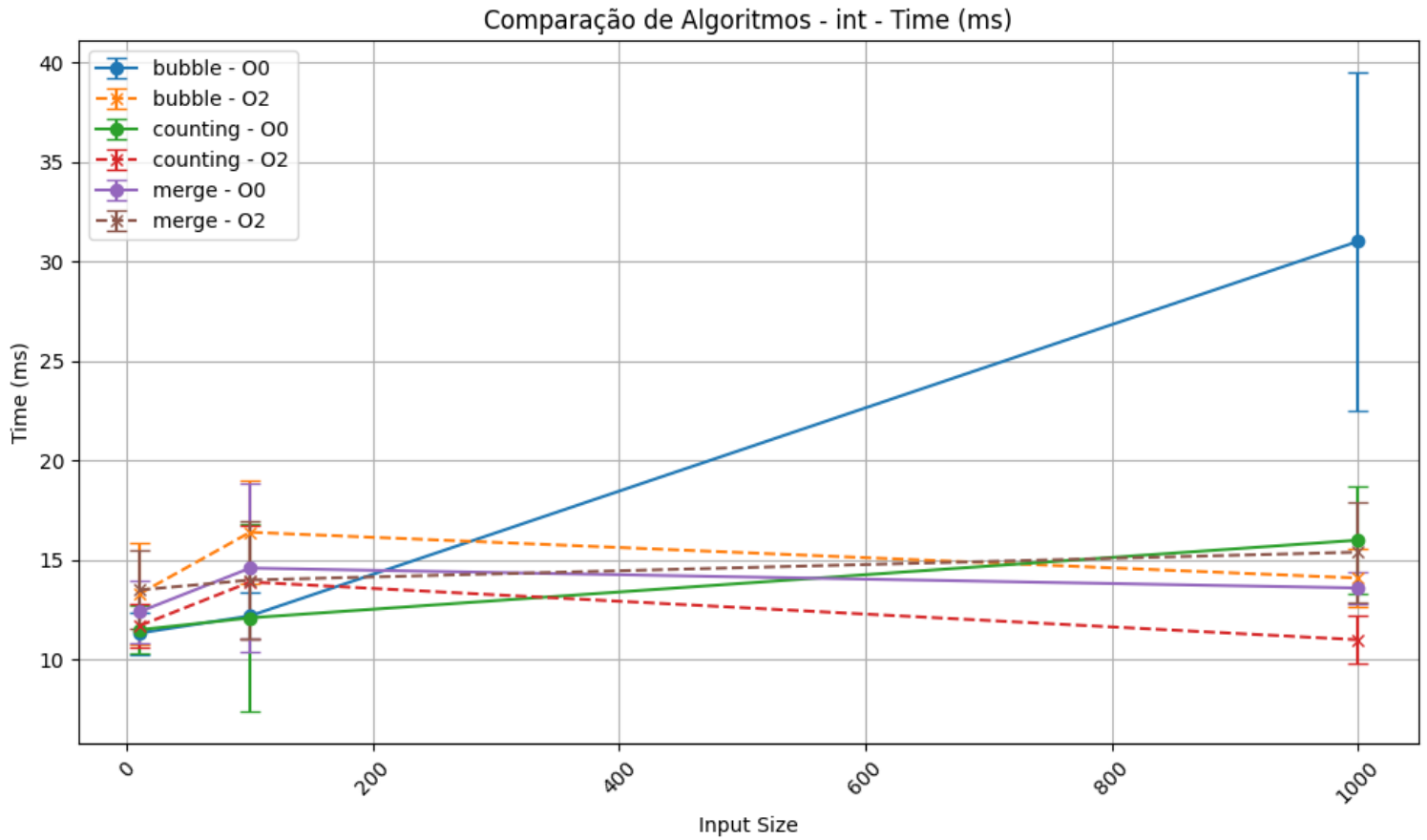
Data Type: Int - Temperature

A temperatura durante os cálculos do *bubble sort* foram inferiores aos restantes algoritmos. É de notar no pico de temperatura atingida durante o algoritmo *merge sort* com a utilização da flag '-O2'.



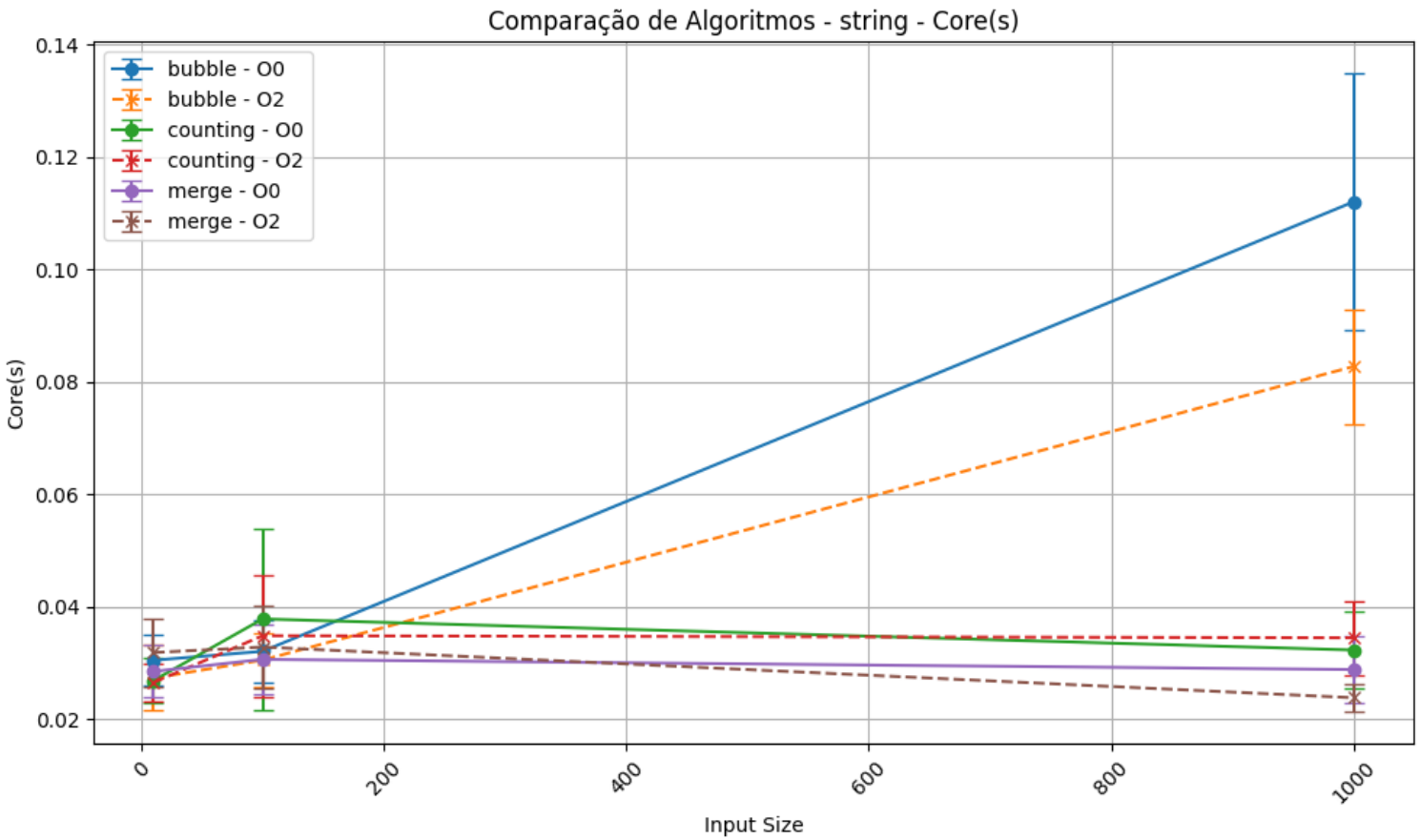
Data Type: Int - Time

Mais uma vez, os resultados são muito semelhantes aos do *data type* float.



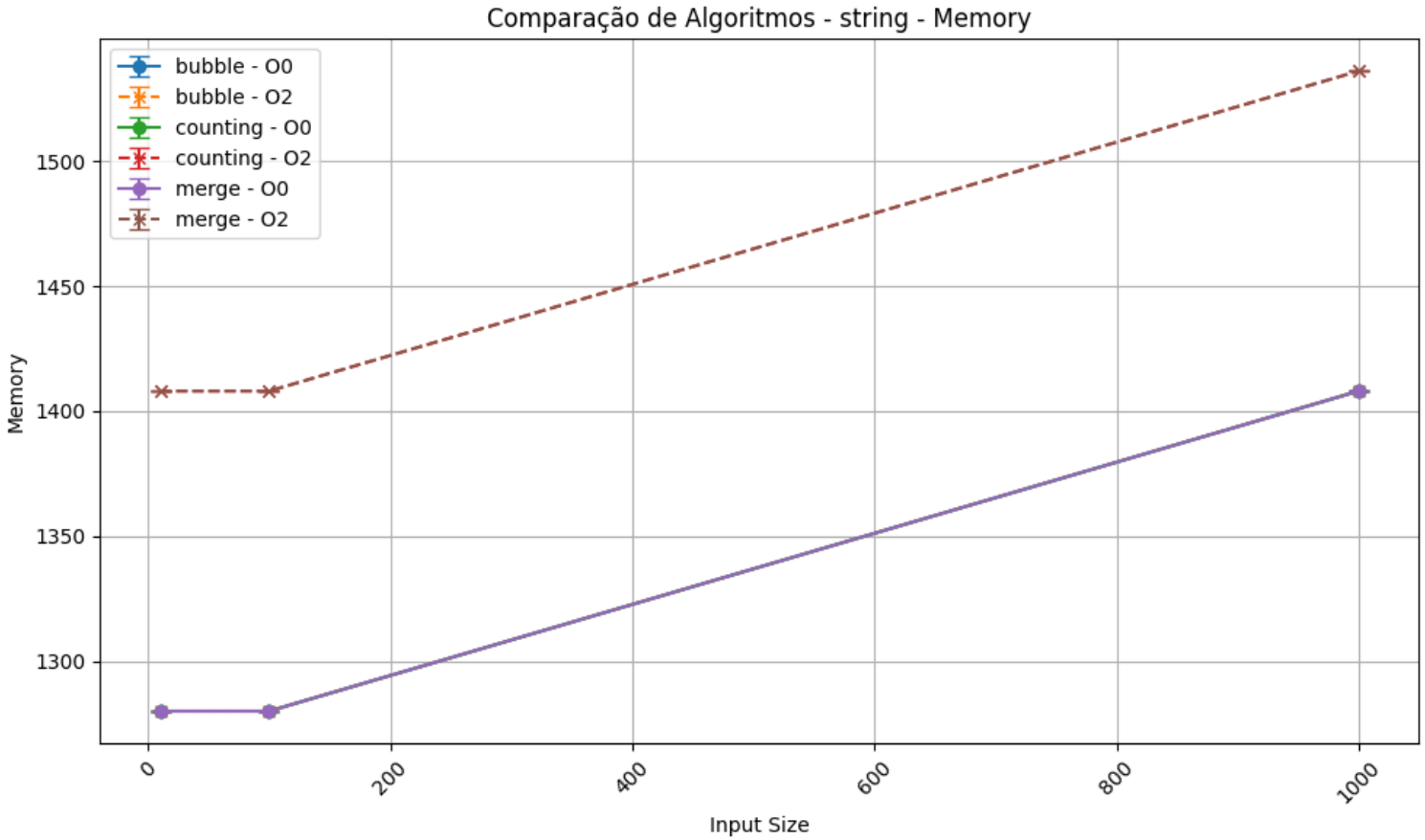
Data Type: String - Core

O *counting sort* é o algoritmo mais eficiente e estável para os tamanhos de entrada testados, seguido pelo *merge sort*. O *bubble sort*, como esperado, tem o pior desempenho, especialmente sem otimização. A otimização do compilador '-O2' melhora o desempenho, mas o impacto varia entre os algoritmos.



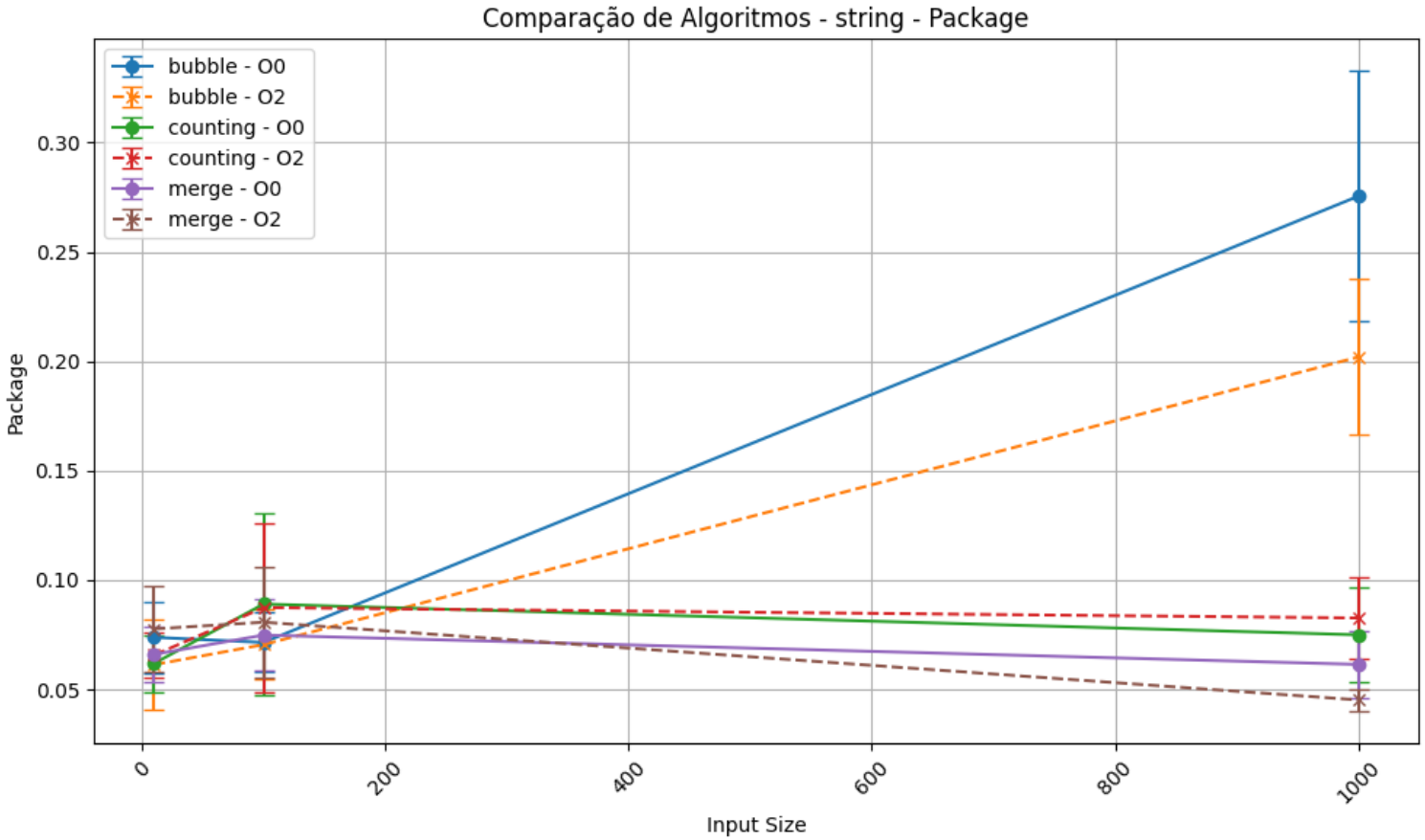
Data Type: String - Memory

O *bubble sort* tem o uso de memória mais estável e previsível, enquanto o *counting sort* e o *merge sort* têm um uso de memória maior e crescente com o aumento do tamanho da entrada. A otimização do compilador '-O2' não tem um impacto significativo no uso de memória. Alguns resultados estão sobrepostos, devido aos mesmos valores.



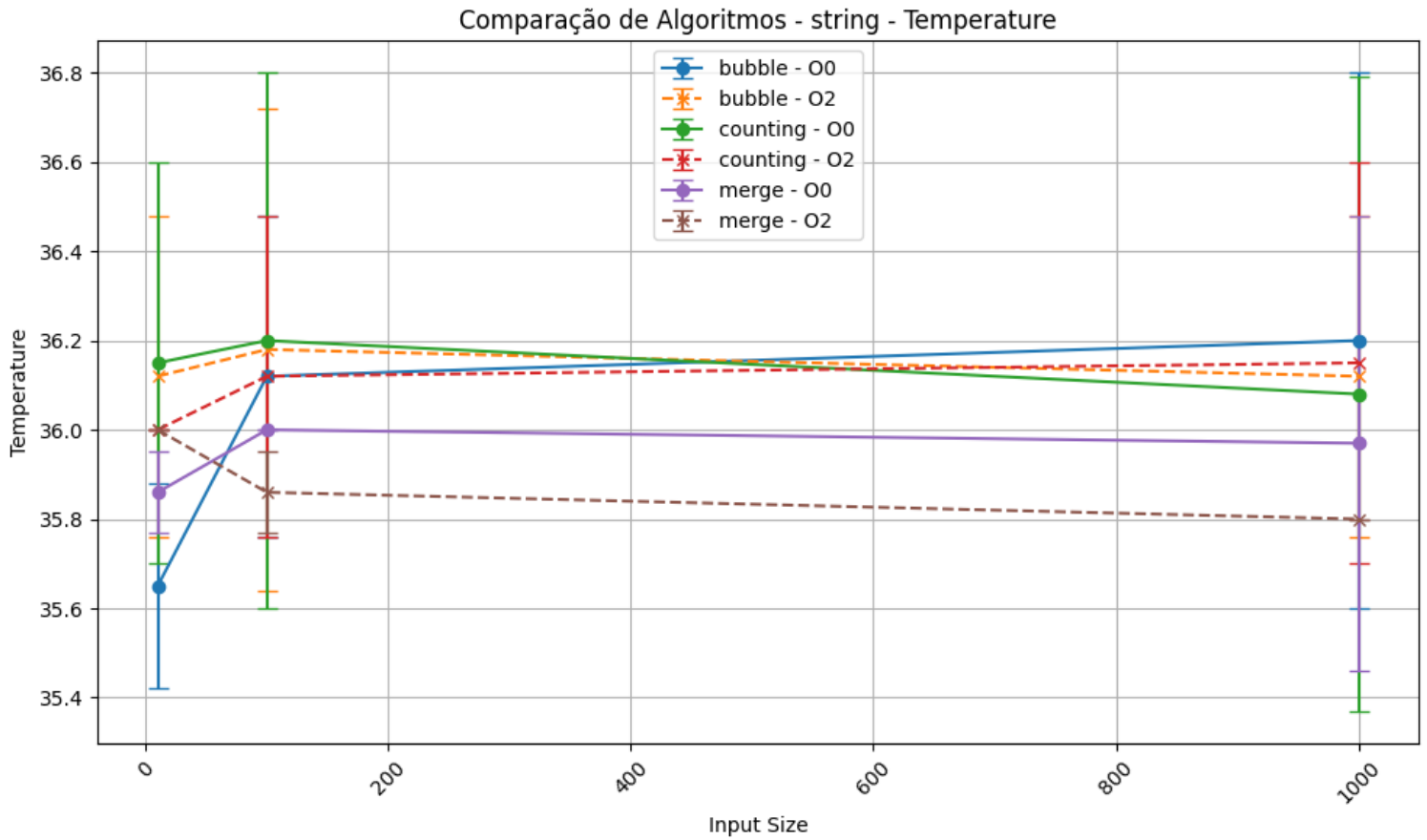
Data Type: String - Package

Os resultados são muito semelhantes, à exceção do algoritmo *bubble sort* que tem um gasto de energia muito superior para valores mais altos. Porém existe um gasto geral maior para este tipo de data type, comparando com os restantes.



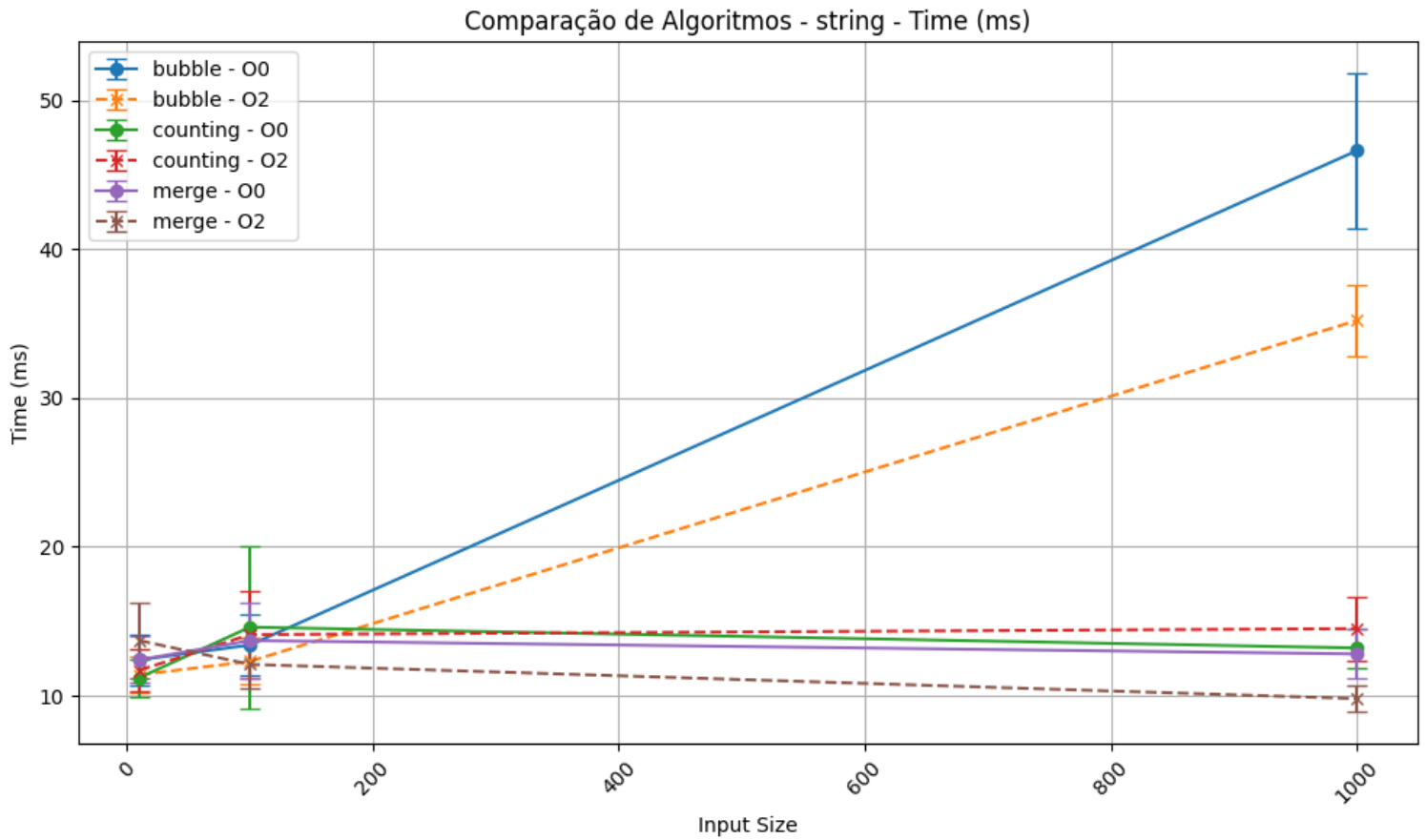
Data Type: String - Temperature

Nestes testes, a temperatura foi bastante constante e sem grande informação a retirar.



Data Type: String - Time

Os valores tiveram tempos de execução bastante semelhantes, exceto o algoritmo *bubble sort* sem qualquer tipo de otimização. É de notar que o algoritmo *counting sort* e o *merge sort* são os mais rápidos, mesmo para valores superiores. Mais uma vez, é de notar que o tempo de execução é superior aos restantes data types.



Data Type - Conclusão

O *data type string* demora muito mais tempo a ordenar que os restantes *data types*, pois as comparações envolvem múltiplas operações de leitura e comparação caractere por caractere.
Os data types int, float e double são otimizados para comparação em hardware e possuem tamanho fixo, tornando a ordenação mais eficiente.
O overhead de memória e cache impacta negativamente a ordenação de strings.

Trabalho na Íntegra

O código-fonte deste trabalho está disponível no meu repositório no GitHub (link). Os gráficos não estão incluídos no repositório, mas podem ser gerados seguindo as instruções fornecidas no README.