

Relatório de performance utilizando likwid

Gabriel Lüders - GRR20190172

Richard Fernando Heise Ferreira - GRR20191053



Introdução

Neste trabalho, feito em C, resolvemos funções de Rosenbrock utilizando o método de Newton em duas formas diferentes: o padrão, com eliminação de Gauss e retrosubstituição; e o método de Newton inexato, que resolve um sistema linear pelo método de Gauss-Seidel.

Ainda, utilizando um software de engenharia de performance (likwid) avaliamos a velocidade de resolução das funções de Rosenbrock, para sistemas de 10 a 4096 variáveis – com alguns saltos –, como um todo, bem como a velocidade de cada método, do cálculo da matriz hessiana e do gradiente. Também avaliamos o tráfego de dados da cache L3, a quantidade de *cache misses* em L2 e a quantidade de FLOPS por segundo (AVX e DP) em cada item supracitado.

Na segunda parte deste trabalho buscamos otimizar, utilizando métodos de otimização em arquitetura x86 – especificamente *unroll & jam*, *loop blocking* e *padding* –, cada item que nos é de interesse.

Ao fim, comparamos as métricas selecionadas através de tabelas que usamos para gerar gráficos (as tabelas foram geradas por um script em bash e os gráficos por um script em python) e tiramos conclusões e interpretações de ganho de performance – ou não – depois das otimizações feitas no código original.

Descrição do computador utilizado para rodar os experimentos

CPU name: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz

CPU type: Intel Coffeelake processor

CPU stepping: 9

Hardware Thread Topology

Sockets: 1

Cores per socket: 4

Threads per core: 1

HWThread	Thread	Core	Socket	Available
0	0	0	0	*
1	0	1	0	*
2	0	2	0	*
3	0	3	0	*

Socket 0: (0 1 2 3)

```
*****
```

Cache Topology

```
*****
```

Level: 1

Size: 32 kB

Type: Data cache

Associativity: 8

Number of sets: 64

Cache line size: 64

Cache type: Non Inclusive

Shared by threads: 1

Cache groups: (0)(1)(2)(3)

Level: 2

Size: 256 kB

Type: Unified cache

Associativity: 4

Number of sets: 1024

Cache line size: 64

Cache type: Non Inclusive

Shared by threads: 1

Cache groups: (0)(1)(2)(3)

Level: 3

Size: 6 MB

Type: Unified cache

Associativity: 12

Number of sets: 8192

Cache line size: 64

Cache type: Inclusive

Shared by threads: 4

Cache groups: (0 1 2 3)

NUMA Topology

NUMA domains: 1

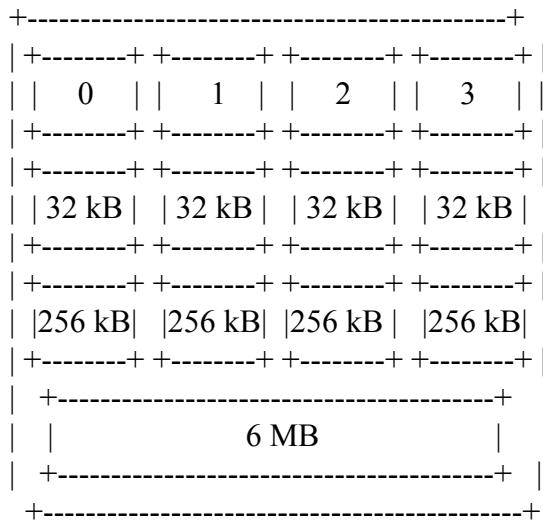
Domain: 0

Processors: (0 1 2 3)

Distances: 10
Free memory: 4277.68 MB
Total memory: 7863.17 MB

Graphical Topology

Socket 0:



Descrição de como rodar

Para rodar o código basta que o usuário descompacte os arquivos e rode o script *run.sh*, que pode ser rodado utilizando *>/run.sh* ou *>bash run.sh*. Esse script executará o script *gera_rosenbrock.sh* do diretório *rosenbrock*, que criará as funções de entradas no arquivo *entrada.dat*. Esse arquivo será usado nos macros *make runL2*, *make runL3* e *make runFLOPS_DP* que terão suas saídas respectivas em *saidaL2.dat*, *saidaL3.dat* e *saidaFDP.dat*. Esses arquivos são movidos para a pasta de scripts a fim de estarem propriamente preparados para serem acessados futuramente.

Em seguida, as entradas são copiadas para a pasta *rosenbrock-opt*, que possui o código otimizado, e os mesmos macros do *make* são rodados para gerar saídas que serão concatenadas nos arquivos de saída que foram criados anteriormente.

Por fim, o script chamará o script em python 3 que acessa outros scripts geradores de tabelas e, depois, o python usará bibliotecas próprias para gerar os gráficos requeridos a partir das tabelas. Vale ressaltar que caso o usuário queira apenas gerar os gráficos basta rodar o script em python na pasta de scripts com o comando *>python3 makeGraphs.py*.

Descrição das otimizações efetuadas

No arquivo *methods.c* a função que calcula a matriz Hessiana, chamada de calcHessian(), fora feita utilizando dois laços para percorrer cada elemento a ser calculado da matriz na sua versão padrão. Já na versão otimizada o laço interno foi desenrolado (*unroll & jam*) e foi feito blocagem nesses laços (*loop blocking*) a fim de melhorar a distribuição dos dados na cache e melhorar o acesso de cada elemento da matriz. Como trabalhamos com matrizes que não necessariamente são múltiplas do nosso tamanho de bloco (4) decidimos implementar mais dois laços de cálculo de resíduo dos blocos nesses casos em específico.

Na função que calcula o gradiente, chamada de calcGradient(), fizemos um simples *unroll & jam* do vetor a fim de melhorar a distribuição e acesso às linhas de cache.

Na função do método de Gauss-Seidel, chamada gaussSeidel(), fizemos o *unroll & jam* dos laços internos de forma a melhorar a distribuição dos dados nas linhas de cache e acesso aos dados no geral, além da ativação do SIMD observada pelo aumento em FLOPS_AVX.

No arquivo *newton.c* modificamos os dois métodos, o padrão em newtonDefault() e o inexato em newtonGS(), da mesma forma: fizemos um *unroll & jam* no cálculo de cada elemento do vetor X a fim de melhorar a distribuição dos dados na cache.

No arquivo *utils.c*, na função de alocação de matrizes mallocMatrix(), fizemos a adição de um *padding* de 1 caso o valor original da dimensão da matriz seja múltiplo de 2, isso garante que não haverá *cache trashing*.

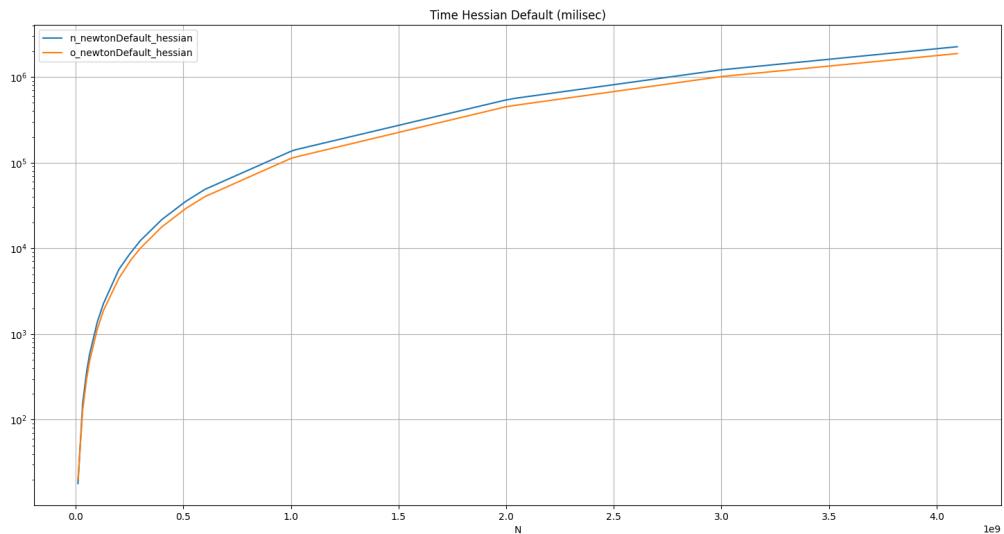
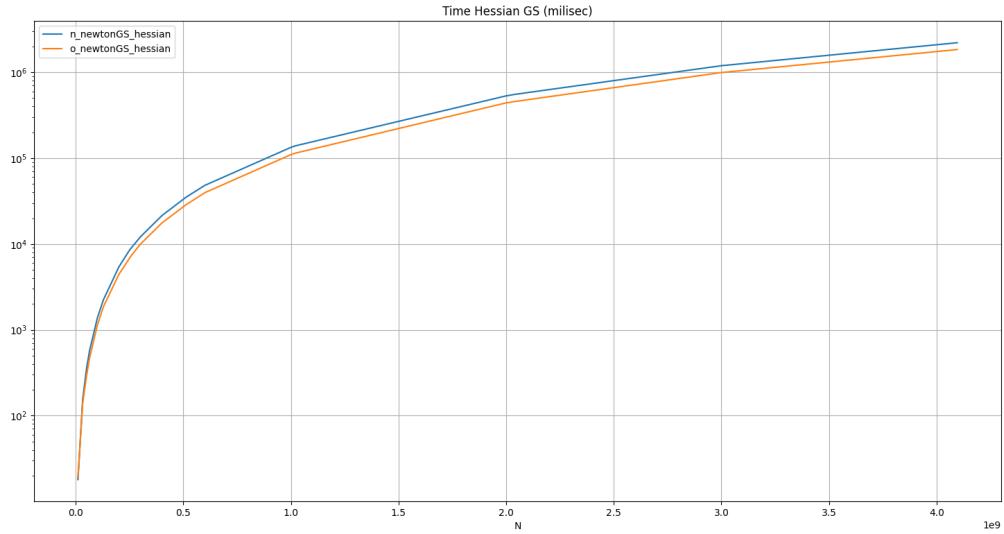
Essas modificações foram o resultado de diversos testes e interpretação dos gráficos que serão expostos abaixo; ainda, contudo, há pontos que gostaríamos de ressaltar desde já:

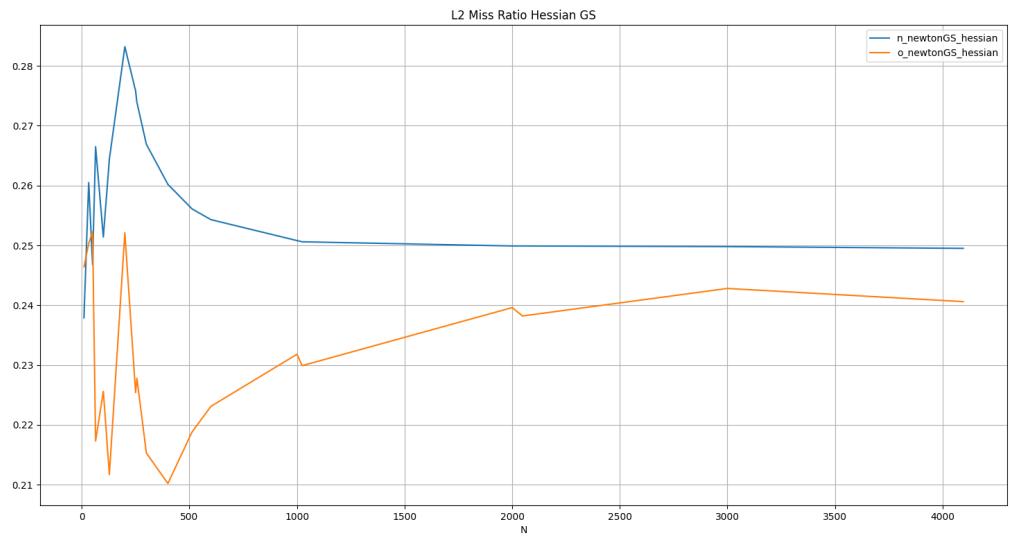
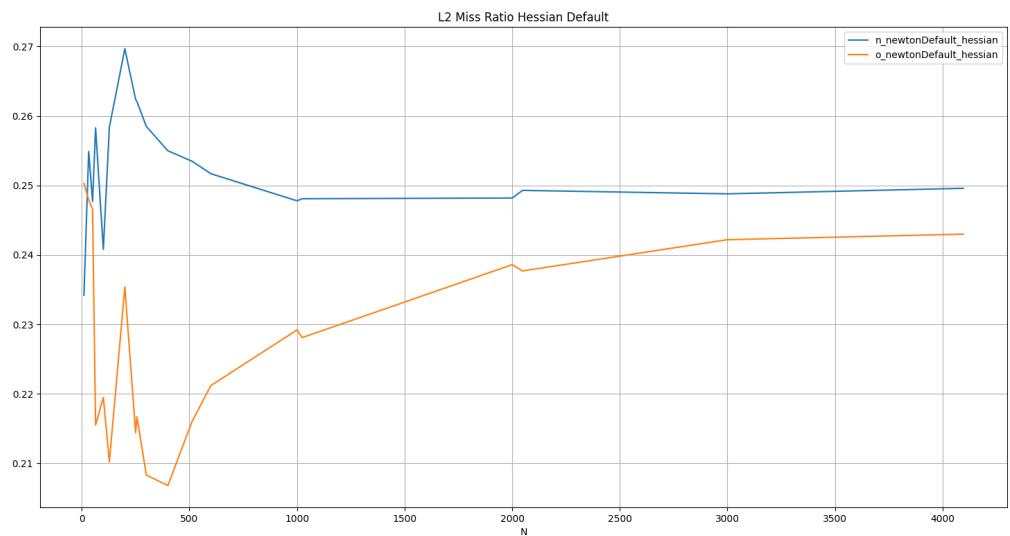
- Não podemos garantir que as otimizações funcionam corretamente para Ns menores que 10 devido ao *loop blocking* feito no cálculo da matriz Hessiana.
- Tivemos poucas otimizações no geral devido às estruturas de dados já bem implementadas na parte não otimizada do trabalho, observando os gráficos averiguamos que a versão não otimizada já estava, de certa forma, bem otimizada devido ao compilador ativar automaticamente algumas otimizações como SIMD.
- Algumas alterações que realizamos para otimizar o código tiveram o efeito contrário inicialmente, mesmo sendo feitas corretamente, notoriamente na função de cálculo do gradiente fizemos *unroll* e *blocking* na primeira versão da otimização, porém essa modificação aumentou significativamente o tempo de execução da função e resolvemos deixá-la apenas com *unroll* que, sem o *blocking*, pode acarretar em um maior número de *cache misses* em Ns muito grandes devido, possivelmente, aos vetores utilizados nessa função não caberem completamente na cache. Vale ressaltar, porém, que o cálculo do gradiente se mostrou mais rápido sem o *blocking* mesmo tendo um número mais alto de *cache misses*.

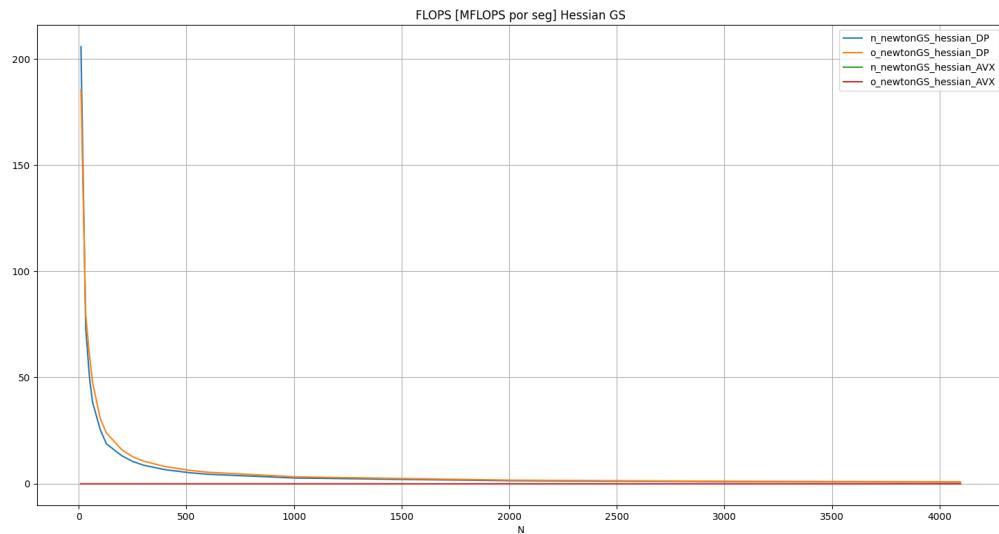
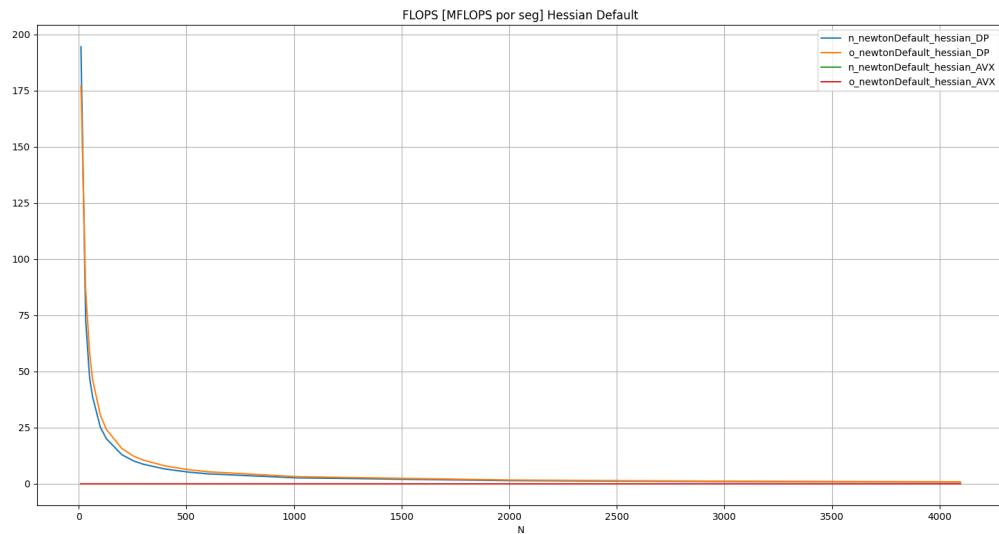
- No cálculo da Hessiana começamos somente com *unroll & jam* e averiguamos que o tempo de execução ficou terrivelmente pior; aplicamos, então, a blocagem e observamos uma clara melhora no desempenho da função em geral.
- Na função que resolve o sistema linear em `newtonDefault()`, a função `solveSL()` – utilizada para chamar a eliminação de Gauss e a retrosubstituição – começamos com *unroll* nos laços internos da eliminação de Gauss e da retrosubstituição, porém observamos que o número de *cache misses* aumentou significativamente, o que nos levou a tentativa de realizar uma blocagem que aumentou ainda mais o número de *cache misses* e piorou significativamente o tempo de execução do código. Portanto, resolvemos não fazer otimizações aqui.

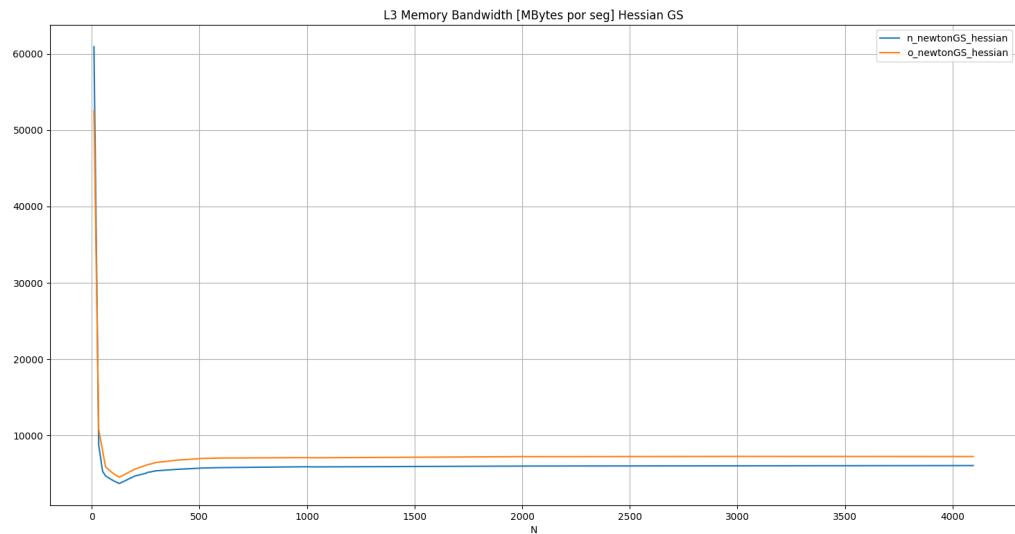
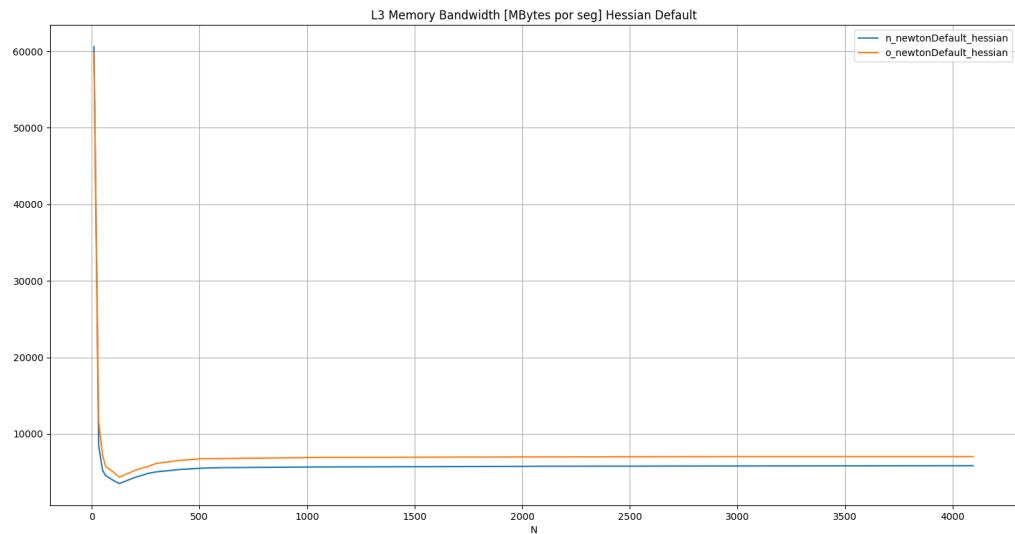
Gráficos e resultados

Começaremos falando sobre os gráficos da Hessiana: no código da hessiana, fizemos *unroll & jam* no laço interno e *loop blocking* em ambos. A nossa ideia com essas modificações era deixar o cálculo da matriz hessiana mais rápido e otimizar os acessos às linhas de cache a fim de diminuir a quantidade de cache miss das L2. Obtivemos sucesso em ambas as modificações como corroboram os gráficos de tempo e os gráficos de L2 cache miss abaixo que demonstram claramente o ganho de performance do otimizado para o normal. Fora isso, gostaríamos de comentar um pouco sobre o gráfico de FLOPS da hessiana: acreditamos que a curva tende a zero porque o tempo de execução cresce desproporcionalmente com relação a quantidade de flops executadas nesse trecho de código. Mesmo com esse comportamento peculiar, é possível notar neles também que a quantidade de flops por segundo do algoritmo otimizado é maior, o que também era esperado devido ao *unroll & jam* feito. Sobre os gráficos de tráfego de memória pela L3, notamos que a quantidade do tráfego de dados do otimizado subiu, o que também é efeito do *unroll & jam*, demonstrando mais uma vez o sucesso da modificação.

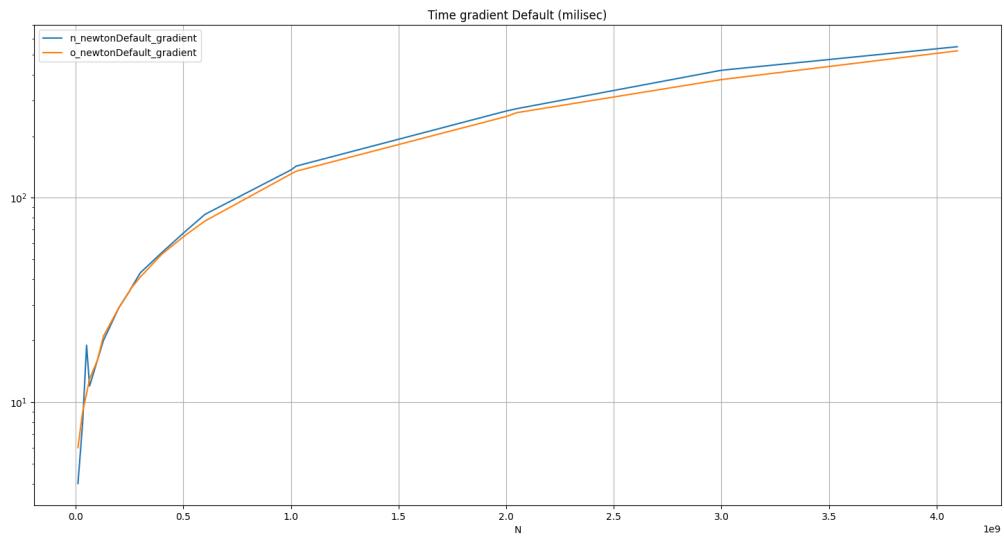
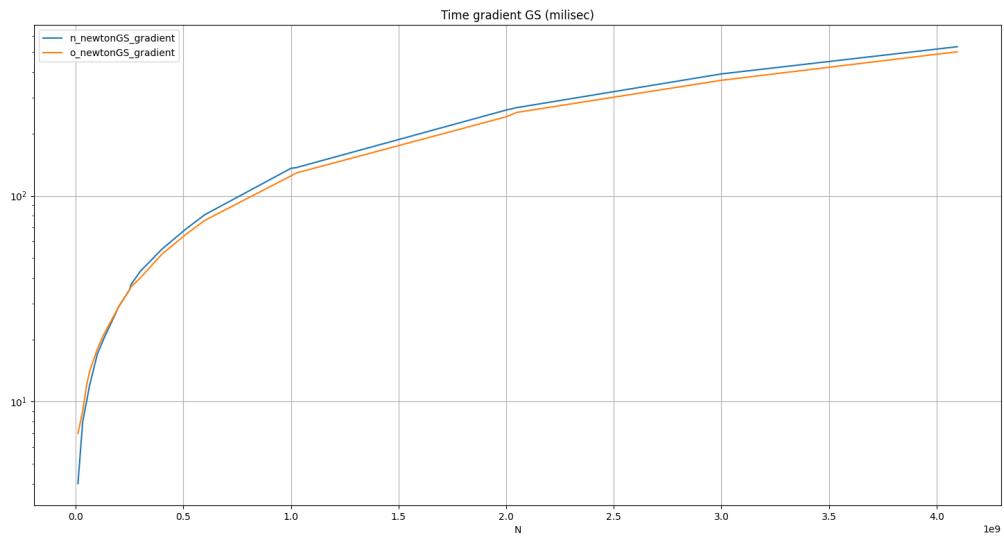


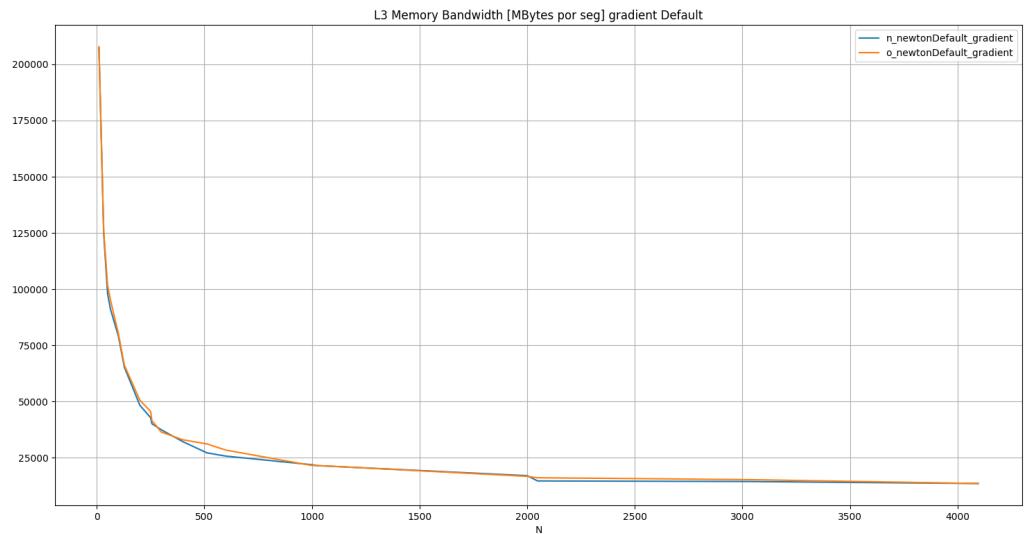
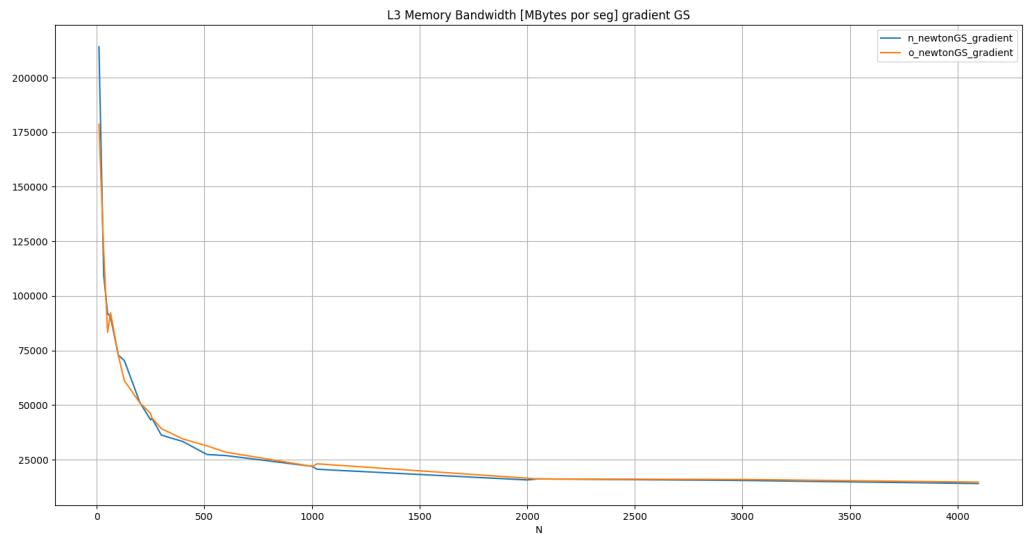


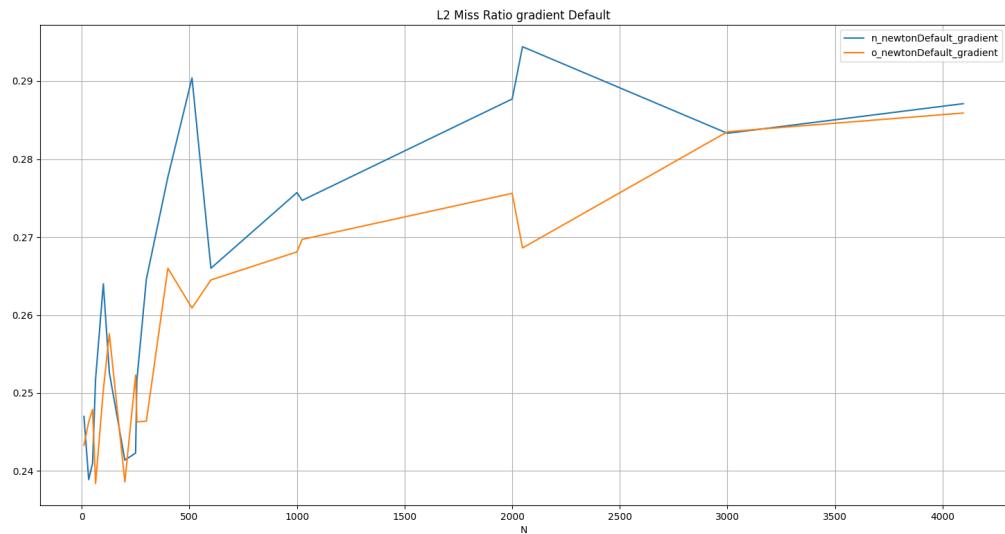
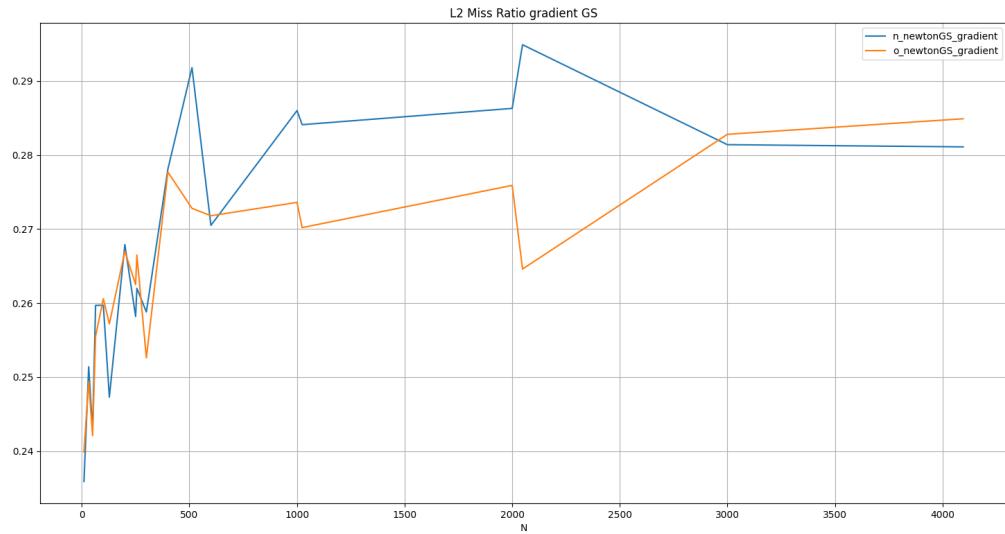


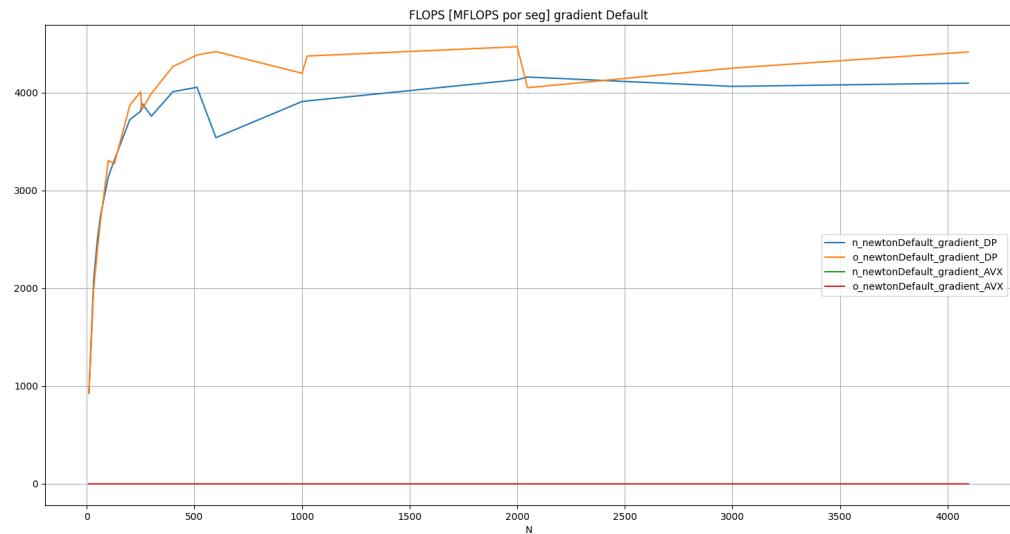
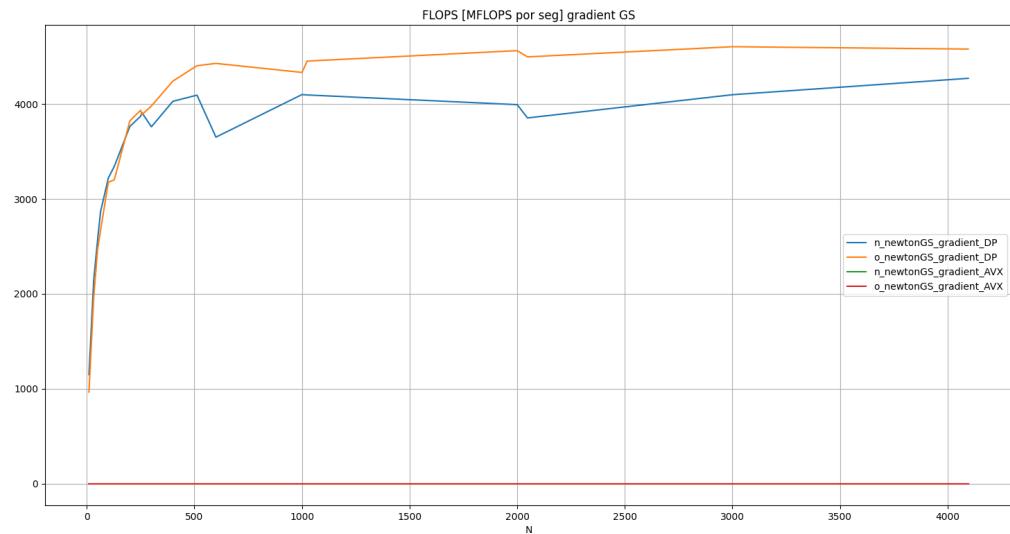


Os gráficos abaixo tratam do gradiente: fizemos um *unroll* no newton otimizado a fim de melhorar o acesso às linhas de cache ao acessar os vetores utilizados no cálculo do gradiente, esperávamos também que a quantidade de *cache misses* seria menor devido à melhor distribuição dos dados pela cache; que o tráfego de dados por L3 deveria ser maior, pois estamos puxando mais dados de uma só vez da memória; o número de operações de ponto flutuante por segundo deve ser maior e que o tempo de execução da função deve ser menor pelos motivos acima. Observando os gráficos podemos averiguar que tivemos os resultados esperados: o tempo diminuiu, no geral a quantidade de tráfego de dados por L3 é maior no otimizado, o *cache miss ratio* do otimizado é menor em geral, com alguns *spikes* pontuais e um comportamento peculiar perto e após $N = 3000$ que especulamos ser daquela forma devido ao tamanho dos vetores utilizados no cálculo do gradiente serem muito grandes - no nosso código, calculamos o gradiente negado e o gradiente simultaneamente. Mantivemos assim pois separar em dois loops, apesar de diminuir o cache miss, aumentava significativamente o tempo de execução do cálculo do gradiente - e, portanto, não caberem simultaneamente na cache, o que faz com que precisem ser realocados constantemente cada vez que são acessados. Já os FLOPS, vemos que não há ativação de SIMD (AVX zerados) e que no otimizado o número relativo de operações realmente aumentou, como esperado, devido ao maior número de dados acessados simultaneamente.

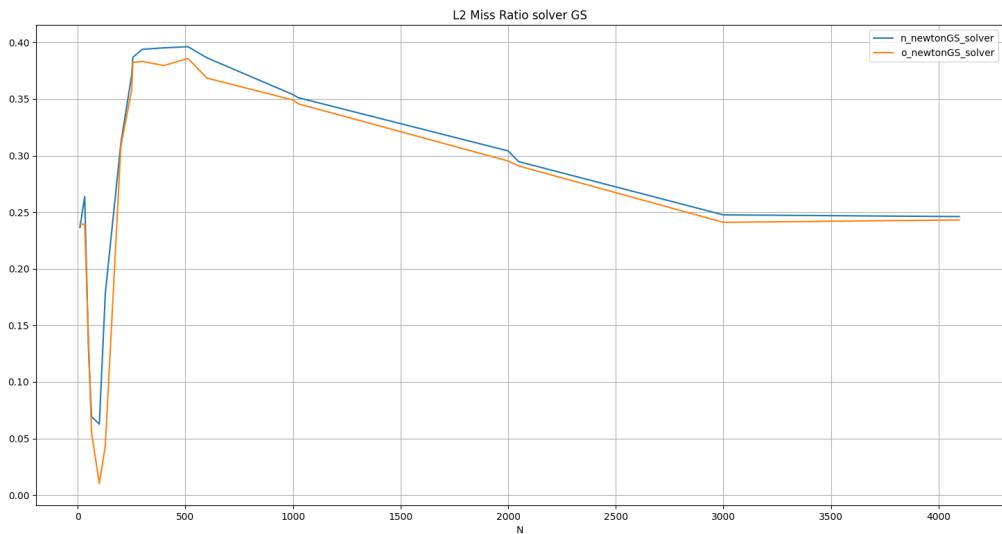
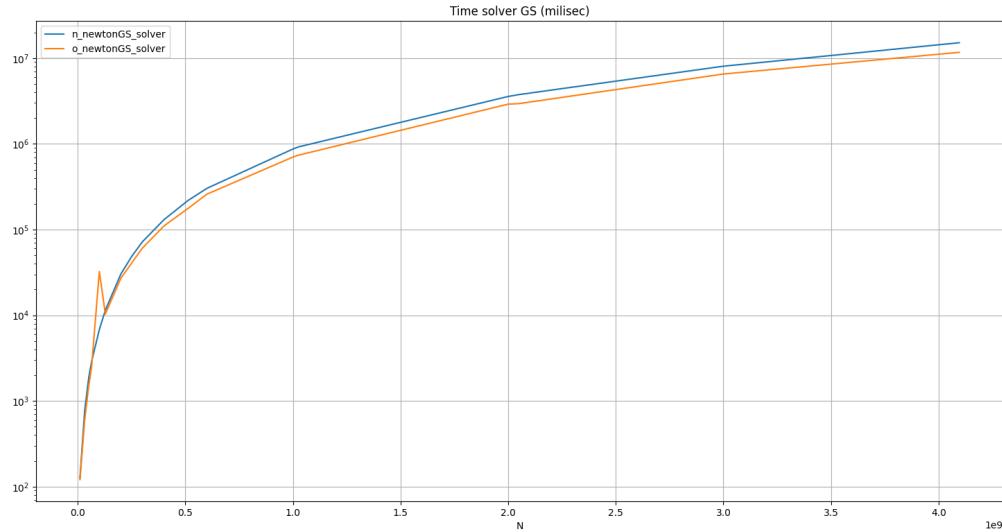


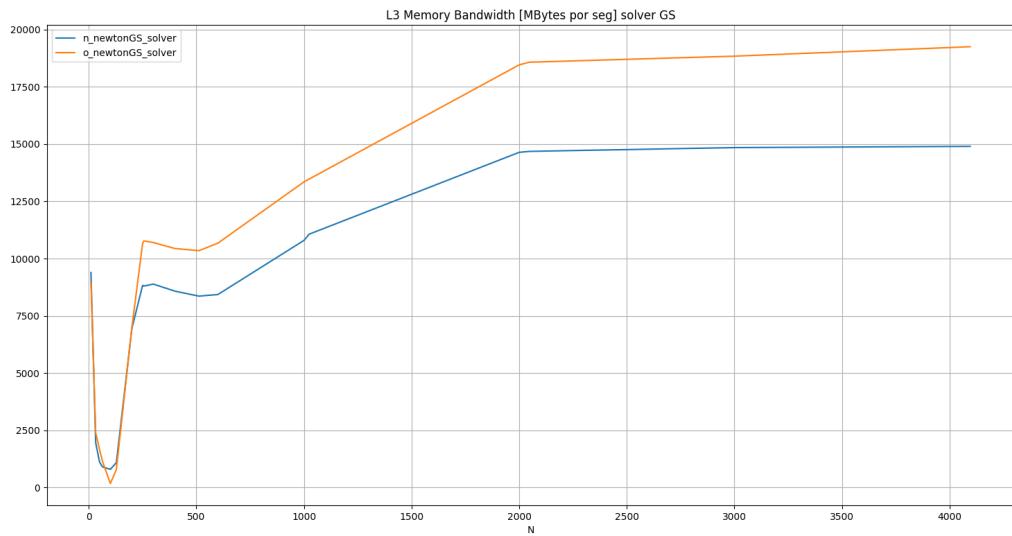
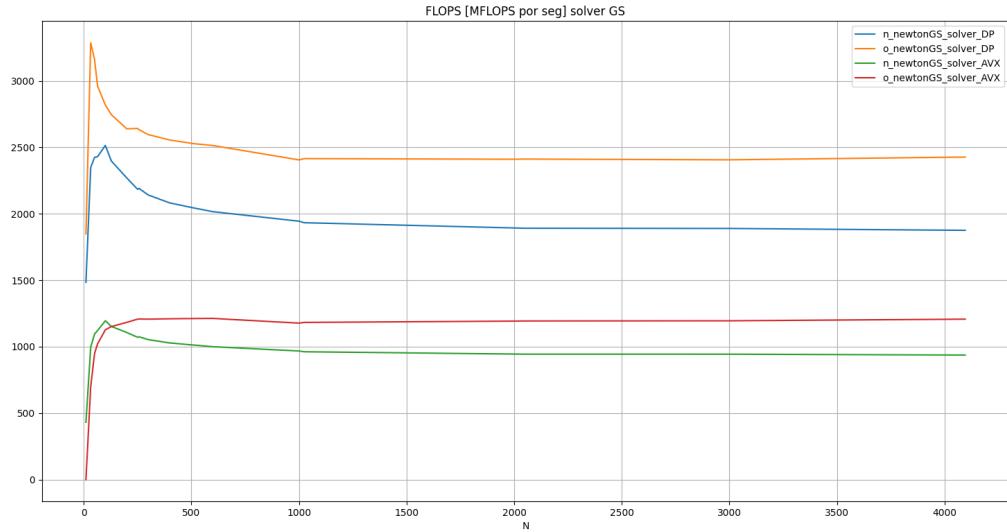




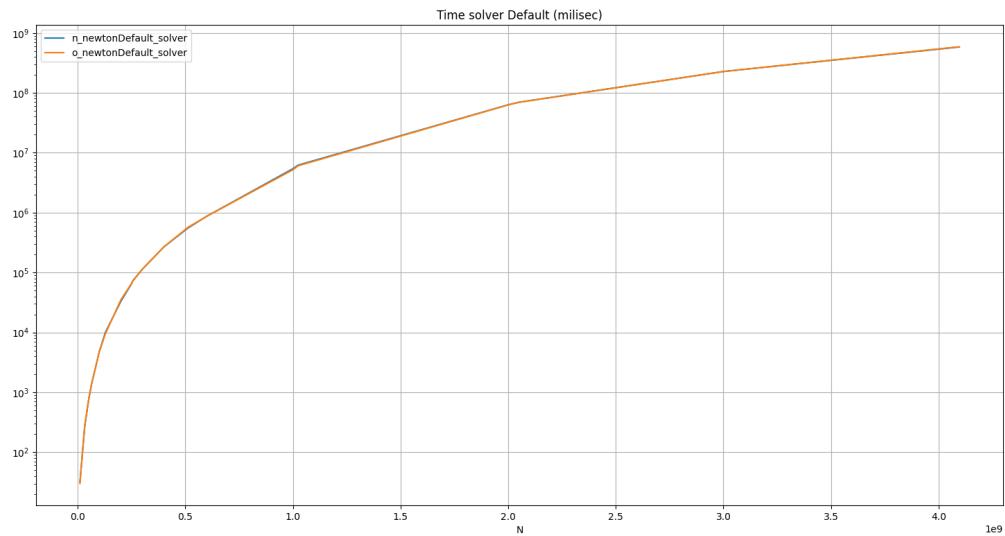


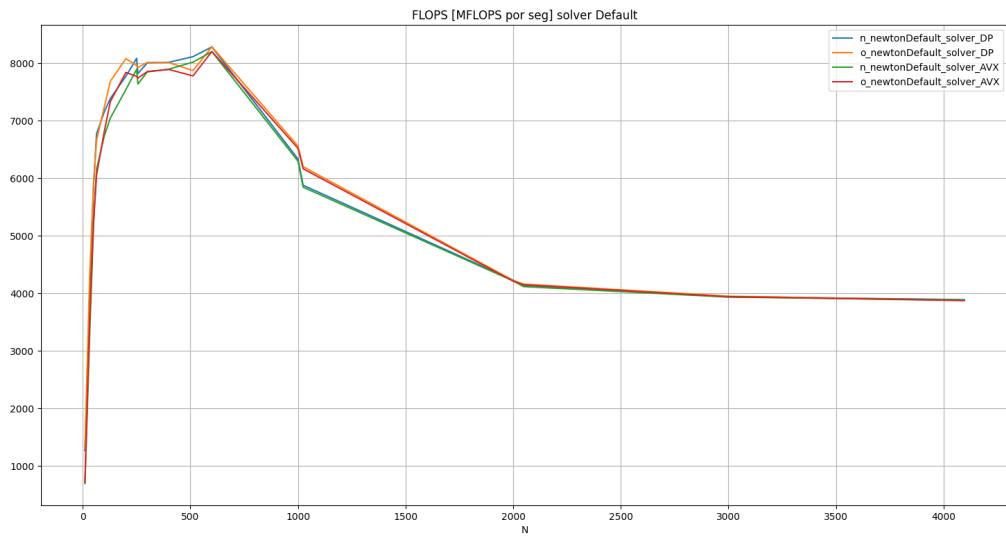
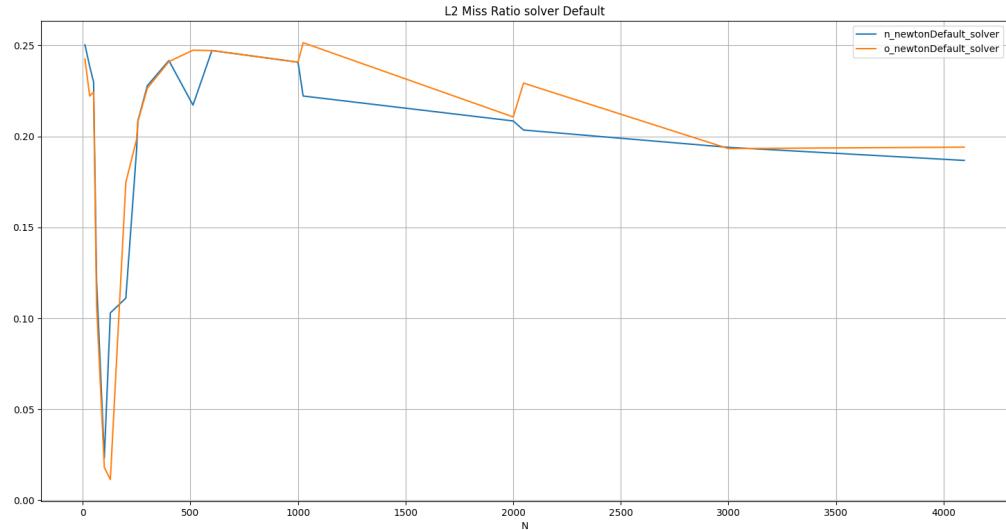
Agora, iremos discutir um pouco sobre o solver do método inexato: a principal modificação realizada foi o *unroll & jam* dos laços internos do gauss-seidel a fim de ativar operações AVX (SIMD) e melhorar o acesso às linhas de cache, diminuindo cache miss da L2. Ambas as modificações cumpriram seu papel como podemos ver nos gráficos abaixo: o gráfico de tempo mostra uma redução clara no tempo de resolução do solver, enquanto o gráfico de L2 miss ratio demonstra uma diminuição nos cache misses. Além disso, é possível ver claramente que aumentamos a quantidade de operações AVX, o que demonstra que o *unroll & jam* teve efeito desejado e ativou SIMD, ao passo que o gráfico do tráfego de memória pela L3 aponta um aumento significativo no código otimizado, o que também é consequência do *unroll & jam*. Vale ressaltar que a equipe fez experimentos aplicando blocking ao solver e obteve resultados piores. Dessa forma, escolhemos deixar apenas *unroll & jam*.

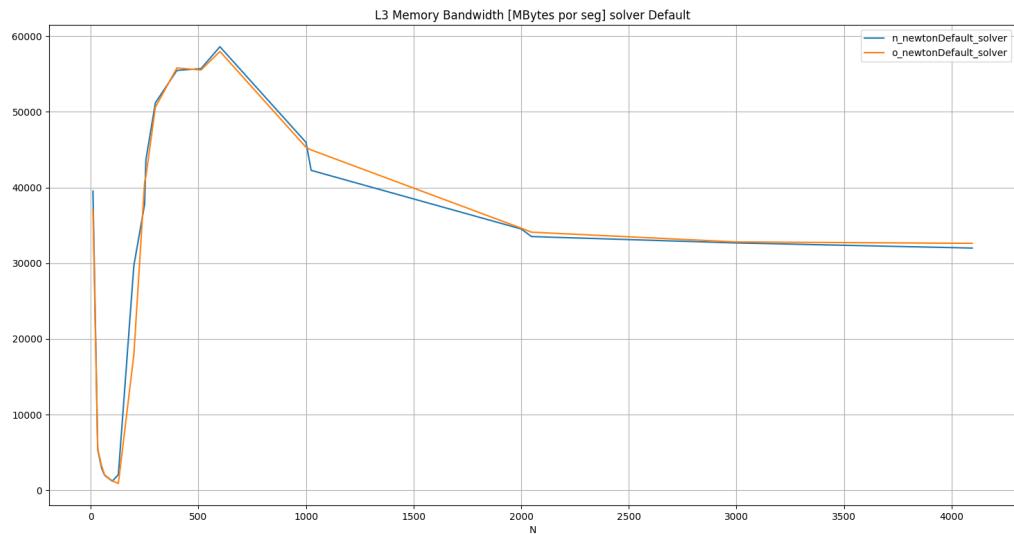




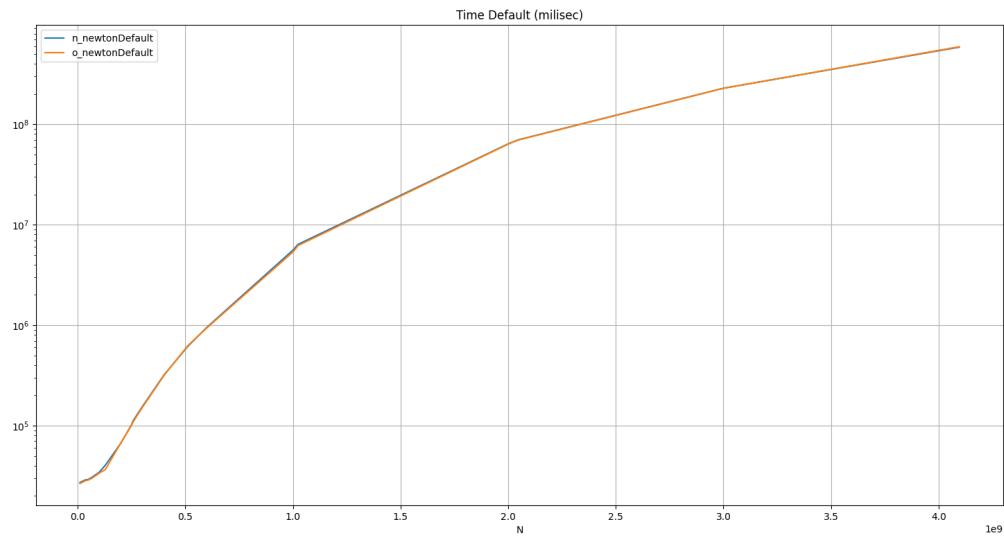
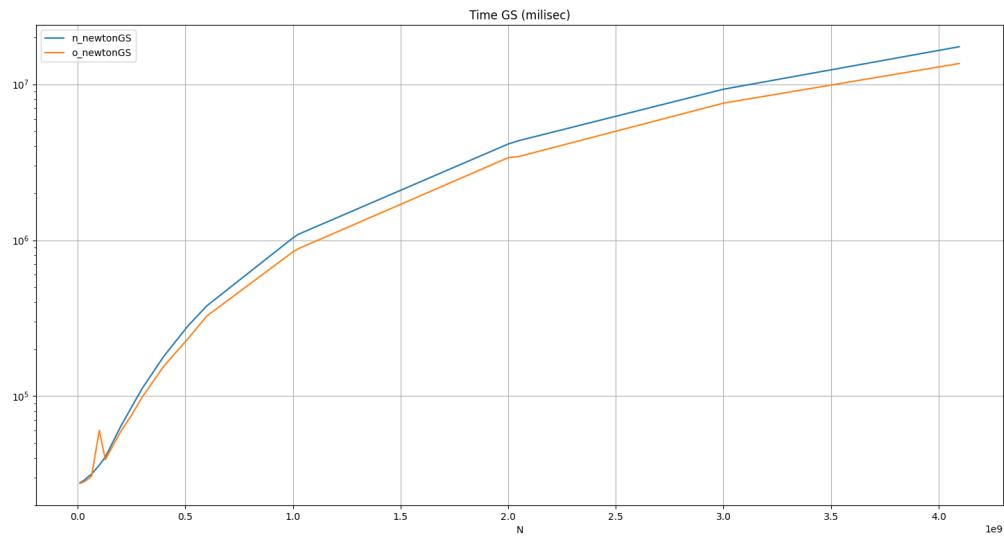
Mudando um pouco o foco para o solver do método exato: aqui, tivemos os piores resultados. A ideia era adicionar *unroll & jam* nos laços mais internos da retrosubstituição e da eliminação de Gauss, mas isso se mostrou completamente ineficiente ao aumentar demais o número de cache miss da L2 e o tempo de execução do método. Especulamos que a falha poderia ser pela falta de *loop blocking*, então fizemos outra tentativa combinando *unroll & jam* e *loop blocking*. Estávamos errados, o tempo de execução ficou ainda maior. Então, como discutido na descrição das otimizações no tópico anterior, já que o compilador estava conseguindo otimizar de maneira razoável a versão do código não otimizado, permitindo uso de SIMD naturalmente, decidimos não mudar nada nessa parte do código. Os gráficos finais estão logo abaixo: como nenhuma modificação foi feita, os gráficos seguem as mesmas tendências e não possuem nada de interessante, com exceção de alguns spikes no L2 Miss Ratio que acreditamos serem apenas ruídos ocasionados pelo maior número de dados da hessiana presentes na cache logo antes do solver ser chamado.

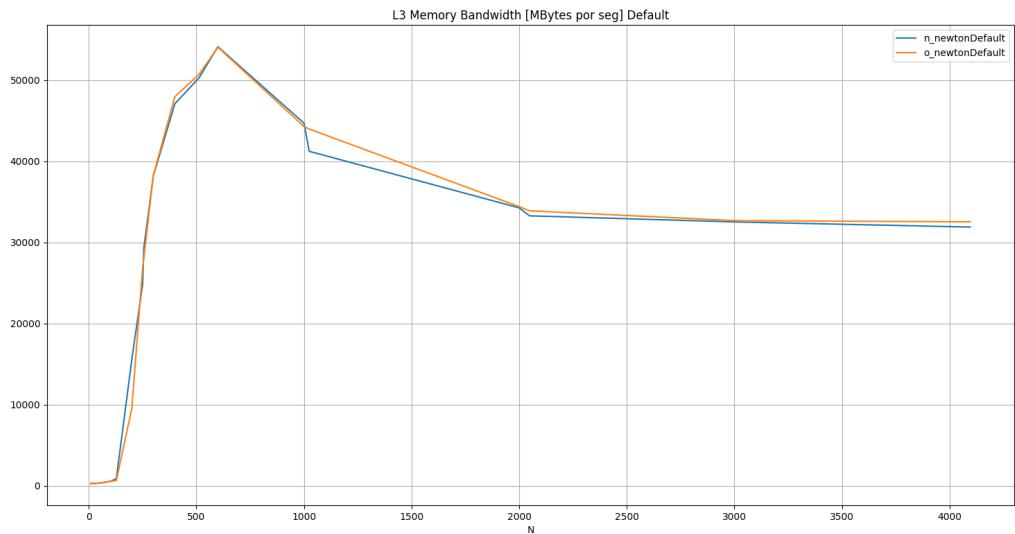
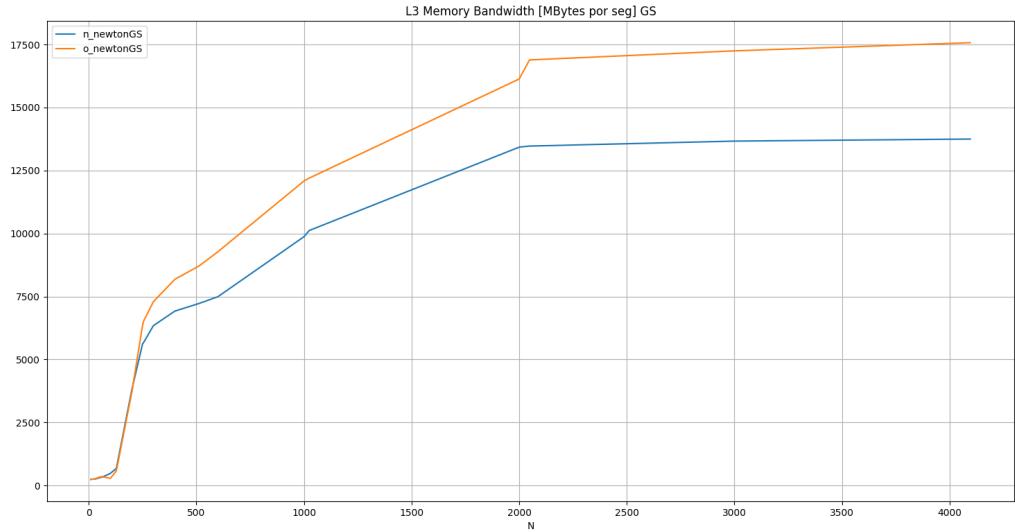


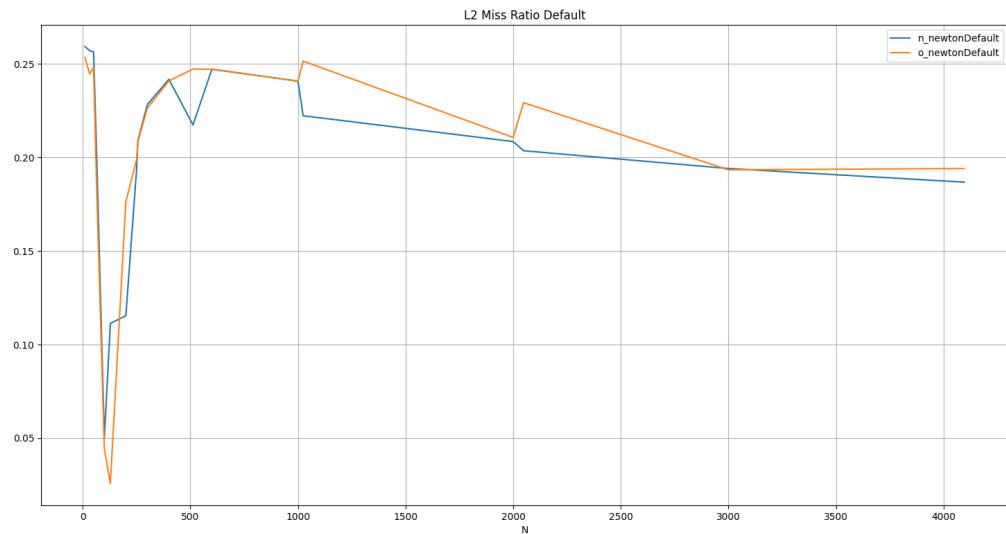
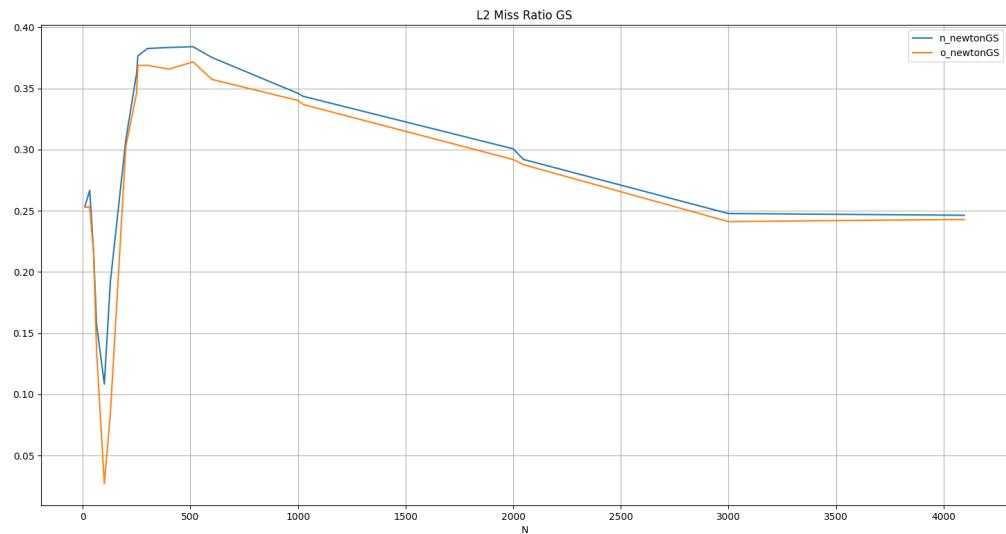


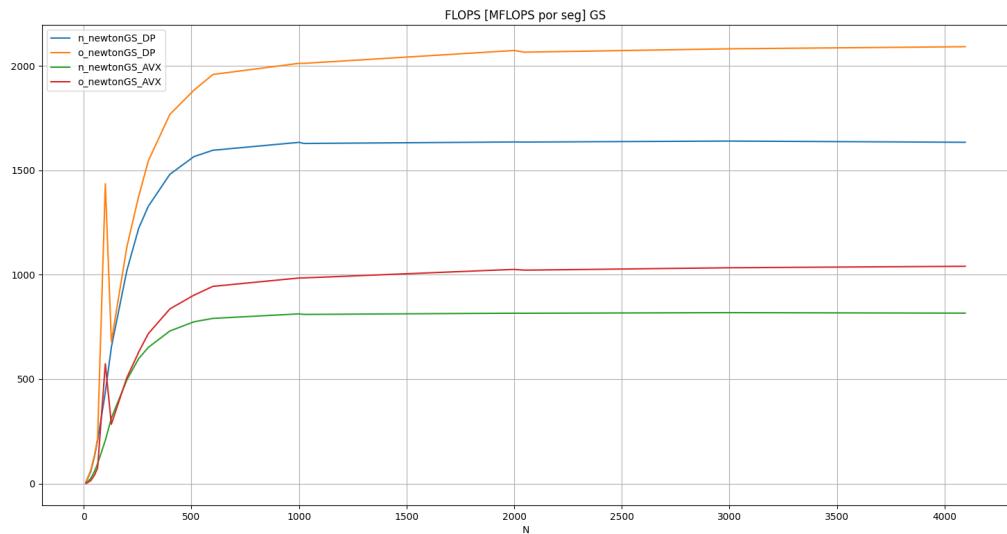
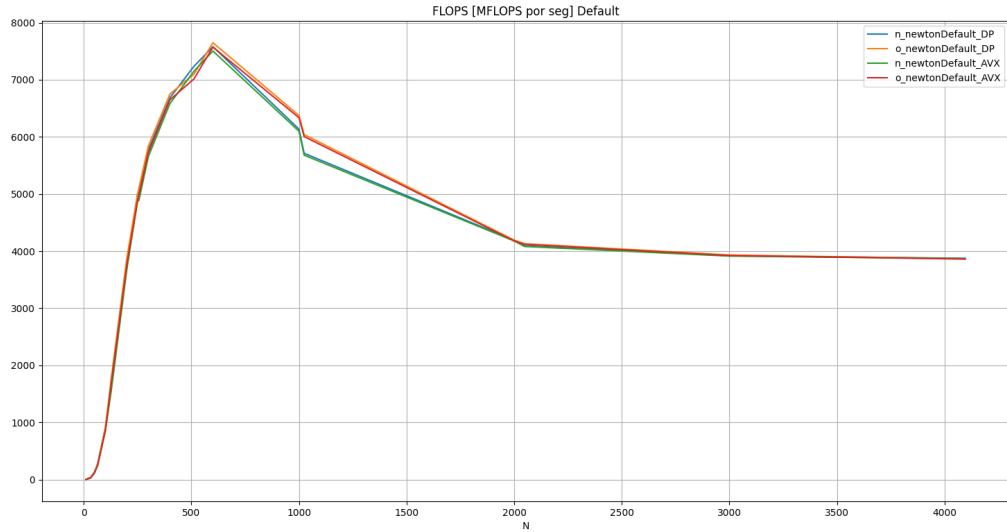


Sobre o tempo total dos métodos: as alterações realizadas culminaram nos gráficos abaixo. Vemos que, principalmente, o tempo de execução do método de Newton Inexato ficou consideravelmente menor, enquanto o tempo total no método de Newton Padrão não teve tanta alteração apesar das melhorias visíveis de acesso à cache e utilização de SIMD. Especulamos que esse efeito se deve à falta de otimizações na eliminação de Gauss e retrosubstituição, que não realizamos devido à perda de performance que observamos quando fizemos, de fato, as alterações: consideramos que possivelmente as demais alterações nas demais funções teriam um impacto significativo no tempo total do método, o que não foi verdade, como mostra o experimento, tendo aumentado somente as operações AVX e Flops total por segundo.









Conclusão

Acreditamos que, conforme demonstram os gráficos, o comportamento dos métodos são muito mais influenciados pelo solver que por qualquer um dos outros cálculos e, por isso, infelizmente não obtivemos melhores resultados no código do método exato.

Contudo, ainda assim acreditamos que o experimento foi um sucesso, visto que conseguimos aplicar várias melhorias vistas em sala, como *unroll & jam* e *loop blocking*, que resultaram em códigos mais rápidos em várias partes dos métodos, além de já ter estruturas condizentes com os conteúdos, como a utilização de matrizes contíguas na memória e utilização de funções, como memset e memcpy, mais eficientes para inicializar e copiar estruturas complexas.