Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação da Universidade de São Paulo
Av. Trab. São Carlense, 400 - Centro, São Carlos - SP, 13566-590 – Fone (16) 3373-9700

Atividade 1

SSC0951 - Desenvolvimento de Código Otimizado

Alunos:
Felipe Guilermmo Santuche Moleiro - 10724010
Matheus Tomieiro de Oliveira - 10734630

OUTUBRO/2021

Introdução

Neste trabalho nós iremos testar dois métodos diferentes de multiplicação de matriz e analisar algumas métricas sobre essas execuções utilizando o profiler perf. Os métodos analisados serão o loop unrolling e o loop interchange. Já as métricas analisadas com o perf serão o L1-dcache-loads (número de acessos de memória na cache L1), L1-dcache-load-misses (número de vezes em que um acesso a cache L1 resultou em miss), branch-instructions (número de instruções de desvio) e branch-misses (número de vezes que a execução errou a predição do desvio).

Desenvolvimento

Para as análises foi escrito um código em C com as duas formas de multiplicação e a alocação das matrizes foi feita da seguinte forma: alocado um vetor de ponteiros, em que cada posição é um ponteiro para um vetor alocado que representa a linha da matriz.

Em seguida foram executadas 10 vezes o programa para cada configuração. Primeiro para o modelo de loop interchange, de complexidade O(n^3), com n=100 e n=1000; após isso um modelo loop unrolling com 4 operações, também O(n^3), com n=100 e n=1000.

Para rodar todos os teste, foi utilizado um script em bash, que executa o perf 10 vezes para cada configuração de testes; são quatro testes por iteração, 100/LI, 1000/LI, 100/LU e 1000/LU, sendo o primeiro argumento o número o tamanho N da matriz quadrada N x N e o segundo, LI e LU indicam o modelo de loop (interchange ou unrolling, respectivamente).

Resultados

L1-dcache-loads

Vamos analisar o número de cache loads na L1 para todos os experimentos. A seguir, há um gráfico em escala logarítmica para demonstrar a quantidade de acessos médio à memória L1 para cada configuração ocorrida durante a execução do programa. Foram adicionados os intervalos de confiança nos gráficos, entretanto, eles foram tão pequenos que não é possível ver com clareza nos plots. Os valores reais do intervalo são possíveis de serem observados na tabela após o gráfico.

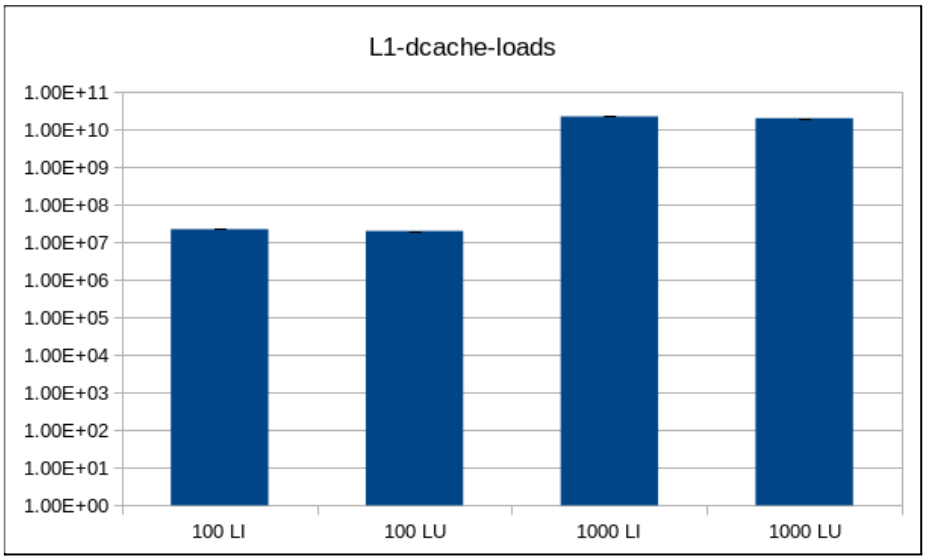

Figura [1.0] (Escala log)

Configuração	Média de acessos a cache nas 10 execuções	Intervalo de confiança 95%(+/-)
100 LI	22118998.9	134.70479786747
100 LU	19869068.4	489.947108398183
1000 LI	22005826937.1	72605.717222712
1000 LU	19756379242.3	211168.160998031

Tabela [2.0]

É possível visualizar nessa análise que o número de loads na cache é praticamente o mesmo, independente do método utilizado para multiplicação das matrizes, mas possui significativa dependência do tamanho da matriz. Isso faz sentido, pois os dois métodos de multiplicação realizam a mesma quantidade de acessos à memória, apenas alterando a ordem desses acessos, o que deve influenciar o número de misses na cache, como será analisado na próxima métrica, mas não influencia o número de acessos à cache.

Ou seja, podemos ver que a maior influência para esta métrica é o tamanho da matriz de entrada, já que o algoritmo terá que percorrer uma matriz muito maior e fazer mais acessos de memória. É interessante notar também que ao aumentarmos o n em 10 vezes, a média de acessos a memória aumentou em torno de 1000 vezes, o que faz todo sentido devido a complexidade desses códigos, de $O(n^3)$, ou seja, um aumento de 10 vezes em n causa, aproximadamente, n^3 mais operações de acesso à memória.

Aqui, temos uma análise a partir dos gráficos de qual fator influenciou mais essa métrica, entretanto podemos utilizar ferramentas estatísticas para esse experimento fatorial 2^2 para obter porcentagens da influência de cada configuração no resultado final.

Com esses cálculos nós temos:

Fatores	Influência
Tipo de Multiplicação	0.29%
Tamanho da Matriz	99.42%
Ambos	0.29%

Tabela [2.1]

(Obs: Pequenos erros de arredondamento durante os cálculos fazem com que a soma dos 3 não somem a exatamente 100%).

Analisando estatisticamente, concluímos também que o tamanho da matriz é o que tem o maior impacto, sendo equivalente a 99.42% da influência de acordo com os cálculos.

L1-dcache-load-misses

Agora, será analisado o número de misses que obtivemos em cada execução. O gráfico a seguir mostra a quantidade de misses médios que cada configuração obteve ao longo das 10 execuções. Além disso, o intervalo de confiança foi adicionado, mas novamente ele apresentou um valor muito pequeno e se tornou imperceptível no gráfico, por esse motivo, adicionamos novamente uma tabela apresentando os valores médios e os de erro.

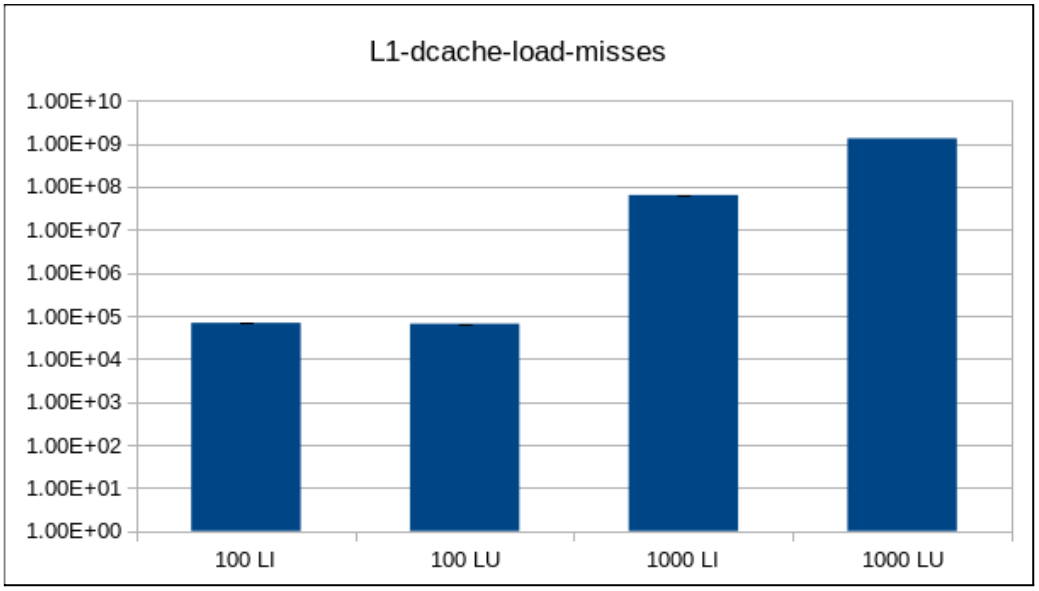

Figura [1.1] (Escala log)

Configuração	Média de misses na cache nas 10 execuções	Intervalo de confiança 95%(+/-)
100 LI	67882.8	87.0609442220623
100 LU	65106	375.175766889787
1000 LI	63352309.1	35396.0189581067
1000 LU	1312784902.6	642457.838080838

Tabela [2.2]

Aqui podemos notar algo muito interessante, para as execuções com n=100, os dois métodos de multiplicação tiveram a mesma performance em questão de misses na cache L1, entretanto, quando vamos para as execuções com n=1000, temos uma disparidade bem grande entre os métodos, de aproximadamente 20 vezes. Isto se dá pela forma que as matrizes estão alocadas na memória e na forma que esse acesso é feito por cada um dos algoritmos.

Neste ponto, vale a pena notar que os códigos foram executados em um processador i5-3470, em que se tem 4 cores, cada um com 128 KiB de cache L1. Portanto, no caso em que temos n=100, o tamanho das 3 matrizes na memória seria de 100*100*3*sizeof(int) = 120000 Bytes ~ 120KiB, portanto todas as matrizes cabem diretamente na cache, quando n=100. Por esse motivo os acessos de certas regiões de memória resultaram em miss apenas na primeira vez em que são acessadas, e posteriormente sempre resultaram em hit, já que temos memória suficiente na cache para armazenar todas as matrizes. Já no caso em que temos n=1000, certas regiões da memória carregadas previamente podem ser retiradas da cache, sendo substituídas por outras informações, e desta forma aumentando o número de misses quando essas regiões foram utilizadas novamente pelo programa.

No nosso caso, as matrizes foram alocadas em blocos por linhas, então cada linha é um pedaço de memória contínuo, enquanto os itens por colunas estão em regiões de memória completamente diferentes. Por esse motivo, acessar a matriz percorrendo linhas

ao invés de colunas é muito mais eficiente para a cache quando temos o problema de dados maiores do que a cache tem capacidade de armazenar. Isso é o que acontece no Loop Interchange, pois percorremos as matrizes por linha, onde a memória é contínua, e depois não voltaremos para essa região de memória; já no Loop Unrolling percorremos a matriz por colunas, ou seja, carregamos a região de memória da primeira linha, segunda, terceira, e assim por diante, até a última linha, e ao chegar ao final da coluna, a região de memória da primeira linha provavelmente já foi deslocada da cache por falta de memória, resultando em um miss ao acessar a primeira linha da próxima coluna, o que teria sido evitado se percorrermos a matriz por linhas.

Podemos ver que para N pequeno, o método de multiplicação não importa, mas para N grande, o método é muito mais relevante. E claro, o que parece ter mais influência independente do método ainda é o tamanho da matriz.

Fazendo a análise estatística, temos:

Fatores	Influência
Tipo de Multiplicação	20.87%
Tamanho da Matriz	58.34%
Ambos	20.79%

Tabela [2.3]

(Obs: Pequenos erros de arredondamento durante o cálculo fazem com que a soma dos 3 não somem a exatamente 100%).

Analisando estatisticamente chegamos a conclusão de 58% é influenciado pelo tamanho da matriz, ou seja, ainda é o fator mais influente, como deduzido pelos gráficos. O tipo de multiplicação tem 21% de influência , o que é bastante relevante, e os dois fatores juntos têm 21% de influência.

branch-instructions

Nesta etapa, analisaremos os resultados do número de instruções de branch executadas em média de 10 execuções. O gráfico a seguir, mostra a quantidade de instruções de branch que cada configuração executou em média. Novamente tivemos uma variação muito pequena e o intervalo de confiança é imperceptível, portanto será disponibilizada uma tabela com os valores numéricos.

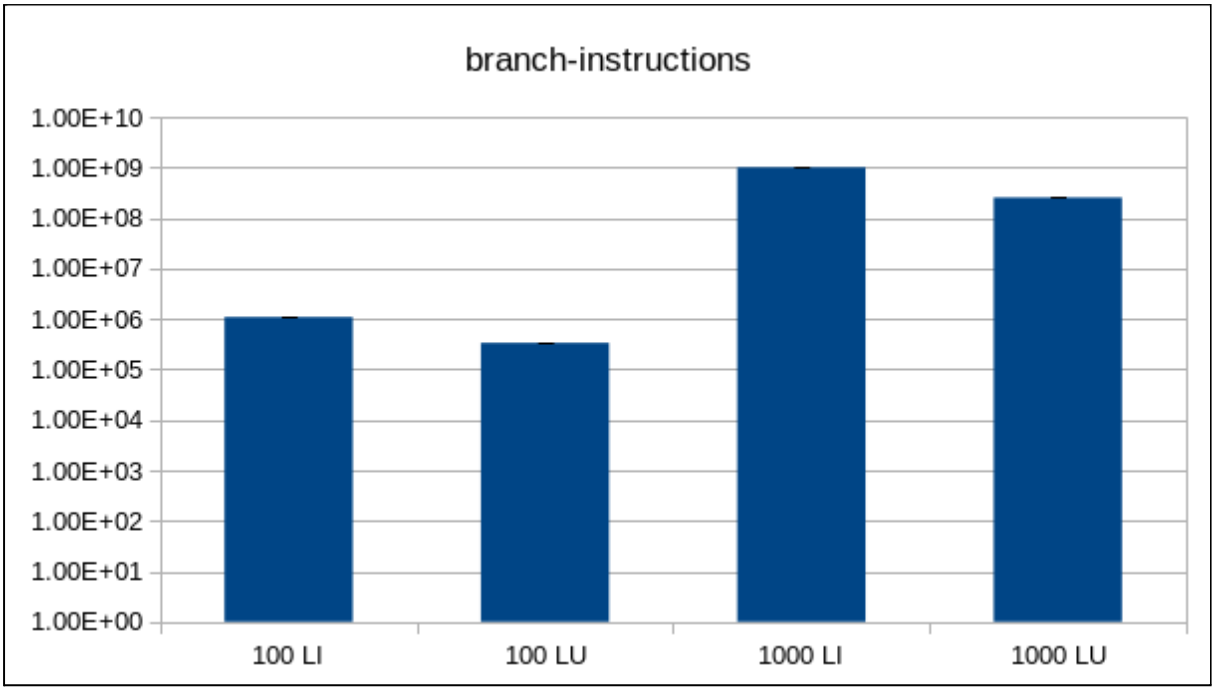

Figura [1.2] (Escala log)

Configuração	Média de instruções de branch nas 10 execuções	Intervalo de confiança 95%(+/-)
100 LI	1084987.7	2.81838307965396
100 LU	334998.2	2.06184733388164
1000 LI	1003294667.7	421.091018516725
1000 LU	253297478.5	1131.32412836419

Tabela [2.4]

Podemos observar que o número de instruções de branch cresce de forma notável com o tamanho da matriz, e segue a proporção de n^3, o que é esperado levando em conta a natureza $O(n^3)$ e a forma como esses algoritmos se comportam, ou seja, um aumento de 10 vezes em N, resultará em um número 10000 vezes maior de instruções.

Entretanto uma característica interessante aqui é a diferença entre os métodos de multiplicação de matrizes. Apesar de não ser tão claro no gráfico por conta da escala log é muito perceptível observando os valores brutos que em geral o número de instruções de branch no método Loop Unrolling é 4 vezes menor que no Loop Interchange. Isso faz sentido, já que no código é utilizado um loop unrolling que faz operações de 4 em 4 posições de memória, ou seja, ele reduz quatro instruções do *for* em uma, e como cada *for* corresponde a uma instrução de branch, nós temos aproximadamente 4 vezes menos instruções desse tipo. Então, há uma queda de 1x10^6 para 3x10^5 instruções e de 1x10^9 para 2,5x10^8 instruções.

Podemos ver que o tamanho da matriz tem um impacto enorme que aumenta conforme o tamanho de N, enquanto o método de multiplicação loop unrolling sempre tem algo em torno de 4 vezes menos instruções comparado ao o loop interchange.

Fazendo a análise estatística, temos uma as seguintes influências:

Fatores	Influência
Tipo de Multiplicação	31.12%
Tamanho da Matriz	37.75%
Os Dois Fatores juntos	31.12%

Tabela [2.5]

(Obs: Pequenos erros de arredondamento durante os cálculo fazem com que a soma dos 3 não somem a exatamente 100%)

Podemos notar que nesses experimentos, que a influência do tamanho da matriz foi de 38% enquanto o tipo de multiplicação foi 31% e os dois juntos, 31%.

branch-misses

Agora vamos analisar a métrica de erros de predição de desvio. Nesta análise verificamos a média de erros de predição de desvio em 10 execuções e plotamos em um gráfico. Novamente há uma variação muito pequena e o intervalo de confiança é imperceptível, portanto teremos uma tabela com os valores numéricos.

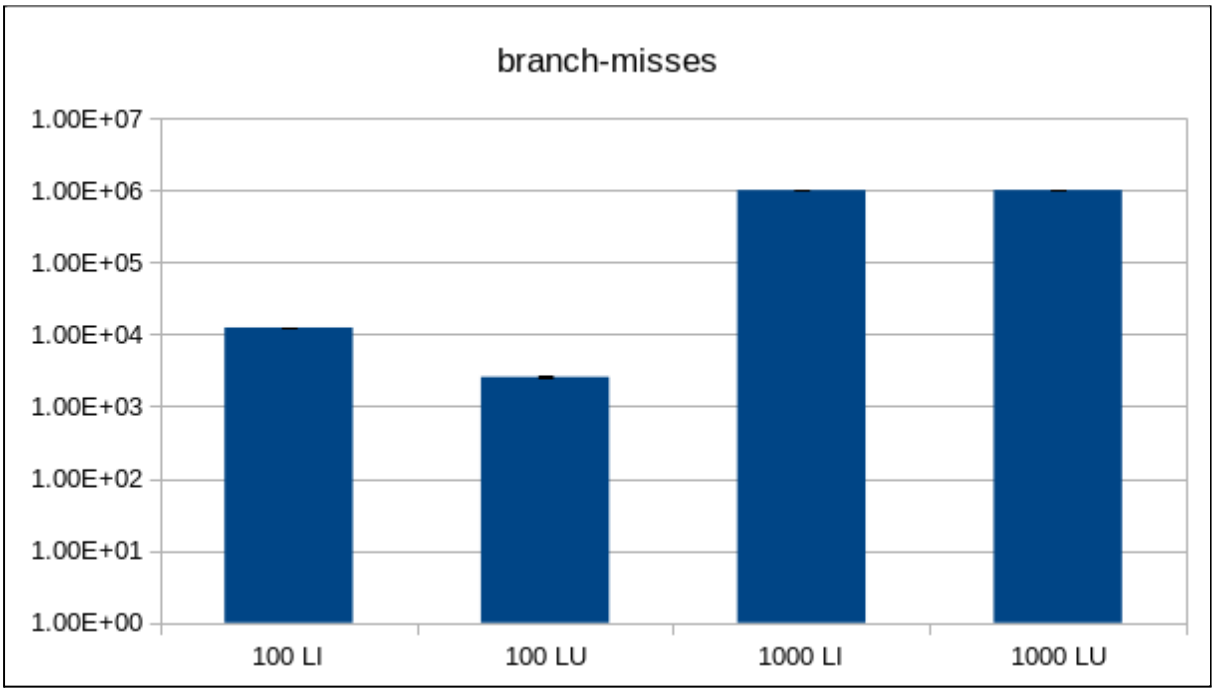

Figura [1.3] (Escala log)

Configuração	Média de erros de predição de desvio nas 10 execuções	Intervalo de confiança 95%(+/-)
100 LI	12433.6	37.2945205559248
100 LU	2564.2	120.937807325751
1000 LI	1004633.8	430.548550022225
1000 LU	1007156.5	1230.98218842596

Tabela [2.4]

Podemos perceber que com n=100, temos uma grande diferença entre erros de predição nos diferentes métodos de multiplicação de matriz, já com n=1000, temos uma quantidade de erros muito próximas. Para o caso do n=100 temos que a proporção de branch-instructions e branch-misses entre os métodos é igual, ou seja, no método LU, há 4 vezes menos instruções e 4 vezes menos erros de predição. Entretanto, é interessante notar que conforme aumentamos o tamanho da matriz, essa diferença de erro se perde, ou seja, erramos o mesmo tanto de predições nos dois métodos com um valor de n grande. A nossa hipótese sobre o motivo disso acontecer é que o algoritmo de predição de desvio tenha uma razoável taxa de acertos de desvios conforme a quantidade de exemplos, ou seja, quando aumentamos o N o algoritmo de predição de desvio começa a acertar mais previsões do *for* que seriam mitigadas pelo loop unrolling indo de 4 em 4, então, ir de 4 em 4 não faz diferença se todos os desvios forem bem preditos.

Essa hipótese se confirma se levarmos em conta que ao aumentarmos o valor de N em 10 vezes, o número de instruções de branch aumentou 1000 vezes, já o número de erros de predição aumentou somente 100 vezes, o que mostra que muito provavelmente os primeiros dois *for* alinhados estão gerando erros de desvio sempre(n^2), já o último *for* alinhado sempre acerta. Por esse motivo, as mudanças no *for* mais interno e na forma como ele faz os desvios (Loop Unrolling de 4 em 4) não faz muita diferença, pois este *for* que modificamos não é o *for* que causa os erros de predição (pelo menos não com n suficientemente grande).

Agora calculando a influência de cada fator no resultado final temos:

Fatores	Influência
Tipo de Multiplicação	1.354E-03 %
Tamanho da Matriz	99.99%
Os Dois Fatores juntos	3.851E-03 %

Tabela [2.5]

(Obs: Pequenos erros de arredondamento durante os cálculo fazem com que a soma dos 3 não somem a exatamente 100%)

Podemos ver que claramente o único fator que realmente afeta o branch-miss é o tamanho da matriz. Pois apesar de para um n pequeno o método de multiplicação ter tido

uma certa quantidade de influência, para o n maior o método não fez a mínima diferença, e em geral o que causou o aumento de branch-misses foi o tamanho da matriz.

seconds time elapsed

Finalmente, achamos interessante incluir mais uma métrica somente para termos uma ideia do tempo de execução de cada programa. Passando rapidamente por ela, já que não foi pedida na especificação do trabalho, mas acreditamos que ela gere informações importantes sobre os algoritmos.

No gráfico a seguir, temos informações dos tempos de execução e em seguida uma tabela com os valores.

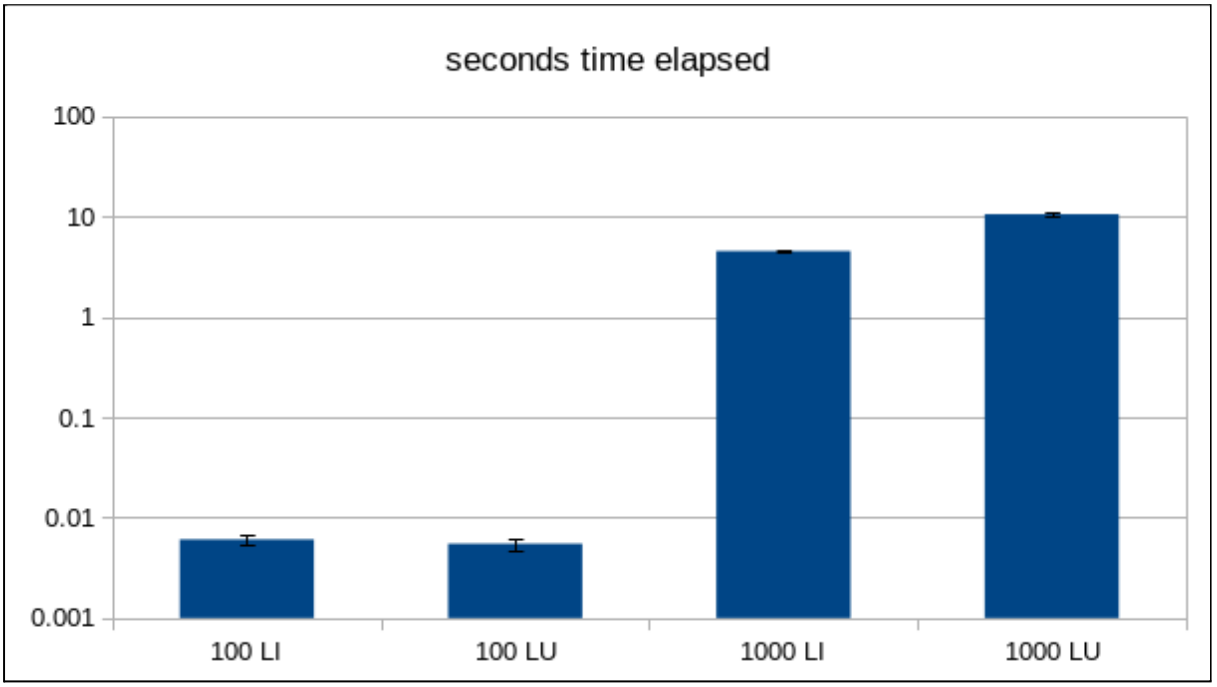

Figura [1.4] (Escala Log)

Configuração	tempo medio nas 10 execuções	Intervalo de confiança 95%(+/-)
100 LI	0.0060603162 s	0.000644202805109223 s
100 LU	0.005495879 s	0.000771276882788573 s
1000 LI	4.5453025364 s	0.04893147516676 s
1000 LU	10.5958115443 s	0.647088614681434 s

Tabela [2.6]

Podemos observar aqui como para valores pequenos de N nós tivemos tempos experimentalmente iguais para os dois métodos, já que eles se interceptam dentro da margem de confiança de 95%, apesar de aparentar ser um pouco menor para o LU . Já para valores grandes de n nós temos uma diferença considerável entre o tempo de execução, de quase duas vezes mais tempo dependendo do método de multiplicação utilizado.

Com isso podemos ver que na prática para n pequenos, não há problemas de miss cache pois temos pouquíssimos dados, e conseguimos carregar tudo de uma vez na cache- Os problemas que temos de branch miss não afetam muito, pois temos poucos branchs, e em geral esse tamanho da matriz é tão pequeno que provavelmente a maior parte do tempo de execução vem do SO preparando o ambiente de execução do que do programa executando em si, mas ainda sim, essa pequena diferença provavelmente explique o porque o LU parece rodar um pouco mais rápido que o LI em média. Agora para n grandes, podemos ver que o número de instruções de branch para o método LI é maior que para o método LU, mas no fundo, o número de branch-misses é o mesmo devido a boa predição do algoritmo de desvio, então na prática o que realmente influencia no tempo de execução entre os algoritmos é a questão do acesso à memória não fazer bom uso da cache.

Podemos ver ent qual fator influenciou mais o tempo de execução do programa:

Fatores	Influência
Tipo de Multiplicação	12.11 %
Tamanho da Matriz	75.77 %
Os Dois Fatores juntos	12.12 %

Tabela [2.7]

(Obs: Pequenos erros de arredondamento durante os cálculo fazem com que a soma dos 3 não somem a exatamente 100%)

E como esperado o que mais influencia é o tamanho da matriz, mas ainda temos 12% no tipo da multiplicação e 12% nos dois fatores variando conjuntamente.

Conclusão

Após a análise de todas essas métricas geradas pelo perf e utilizando conhecimentos de arquitetura e organização de computadores, podemos ter uma boa ideia de como certos algoritmos se relacionam com os outros no desempenho. O que podemos concluir é que antes de tudo, o principal fator que influencia sempre é o tamanho da matriz. Entretanto, essa é uma característica que muitas vezes não podemos mudar, já que os dados muitas vezes tem um certo tamanho e não podem ser divididos.

Apesar disso, podemos comparar os dois algoritmos de multiplicação de matriz, Loop Interchange e Loop Unrolling, e verificar qual tem melhor desempenho em diversas métricas. Observamos que os dois algoritmos fazem o mesmo número de acessos a cache, variando apenas com o valor de n. Percebemos também que apesar de realizarem o mesmo número de acessos a cache, para valores maiores de n, o Loop Interchange faz um acesso mais inteligente da memória(para matrizes alocadas por linha), o que faz com que esse algoritmo tenha muito menos misses de cache em geral.

Também percebemos que na questão de instruções de branch, o Loop Unrolling consegue diminuir muito o número de instruções de desvio executadas, e para n pequenos ele consegue ter menos erros de desvio, entretanto para n maiores isso não faz muita diferença pois a cpu consegue predizer bem os desvios do último *for* alinhado(o *for* que o loop unrolling otimiza) nos dois algoritmos, e na prática os dois tem o mesmo número de erros de desvio.

Finalmente, analisamos também o tempo de execução, e junto com as informações que tivemos dessas outras métricas concluir que na prática, para valores bem pequenos de n, o método Loop Unrolling pode ser melhor, mas para N maior, o Loop Interchange é claramente melhor, pois os problemas de branch são lidados pelo preditor de desvio muito bem, mas o acesso a memória é muito pior no Loop Unrolling, fazendo com que muitos misses ocorram e o tempo de execução seja maior.

Obs: Todo o código utilizado encontra-se alocado no GitHub: **https://github.com/FelipeMoleiro/MateriaCodigoOtimizado**

A seguir, encontra-se a tabela com os resultados das execuções.

INPUT FILE	L1-DCACHE-LOADS	L1-DCACHE-LOAD-MISSES	BRANCH-INSTRUCTIONS	BRANCH-MISSES	SECONDS TIME ELAPSED
./results/100li1.out	22118961	67934	1084986	12462	0,006520055
./results/100li2.out	22119026	68019	1084986	12542	0,006473777
./results/100li3.out	22118895	67763	1084992	12434	0,005566125
./results/100li4.out	22118861	67798	1084991	12471	0,005300968
./results/100li5.out	22119270	68127	1084989	12468	0,005530406
./results/100li6.out	22118536	67737	1084982	12393	0,005225338
./results/100li7.out	22119270	67982	1084986	12421	0,006158773
./results/100li8.out	22118954	67951	1084984	12315	0,00530858
./results/100li9.out	22119163	67693	1084997	12394	0,008689683
./results/100li10.out	22119053	67824	1084984	12436	0,005829457
AVG	22118998,9	67882,8	1084987,7	12433,6	0,0060603162
STDEV	217,3376533	140,4673153	4,547282461	60,17234507	0,001039380394
CONFIDENCE 0.95	134,7047979	87,06094422	2,81838308	37,29452056	0,0006442028051
./results/100lu1.out	19868770	65341	334997	2523	0,005097737
./results/100lu2.out	19868708	64778	334996	2516	0,004800038
./results/100lu3.out	19868879	64737	334997	2574	0,005717767
./results/100lu4.out	19868629	64909	334995	2517	0,004891376
./results/100lu5.out	19868599	64556	334995	2427	0,004946432
./results/100lu6.out	19868559	64606	335003	3037	0,004966569
./results/100lu7.out	19869168	65262	334997	2521	0,005034442
./results/100lu8.out	19869019	64795	334998	2319	0,004780836
./results/100lu9.out	19869121	65514	334999	2724	0,008880017
./results/100lu10.out	19871232	66562	335005	2484	0,005843576
AVG	19869068,4	65106	334998,2	2564,2	0,005495879
STDEV	790,498605	605,32232	3,326659987	195,1254867	0,001244406364
CONFIDENCE 95	489,9471084	375,1757669	2,061847334	120,9378073	0,0007712768828
./results/1000li1.out	22005764112	63276648	1003294483	1004657	4,414300023
./results/1000li2.out	22005812834	63295339	1003294728	1005052	4,434211775
./results/1000li3.out	22005776694	63346739	1003294678	1006078	4,4985823
./results/1000li4.out	22005713584	63326586	1003293991	1004654	4,5217493
./results/1000li5.out	22005853528	63436567	1003294784	1004347	4,59099514
./results/1000li6.out	22005853747	63365650	1003294763	1004115	4,577263132
./results/1000li7.out	22005746399	63312634	1003293976	1003769	4,542755858
./results/1000li8.out	22005751308	63330549	1003294137	1004056	4,588755696
./results/1000li9.out	22006123943	63449385	1003296360	1005348	4,648119904
./results/1000li10.out	22005873222	63382994	1003294777	1004262	4,636292236
AVG	22005826937	63352309,1	1003294668	1004633,8	4,545302536
STDEV	117144,7228	57109,23307	679,4036683	694,6627959	0,07894783375
CONFIDENCE 95	72605,71722	35396,01896	421,0910185	430,54855	0,04893147517
./results/1000lu1.out	19755958651	1310718675	253295963	1006430	8,331753403
./results/1000lu2.out	19756267371	1313927331	253297269	1007700	9,035299501
./results/1000lu3.out	19756511907	1313371190	253298175	1008036	10,53406068
./results/1000lu4.out	19756606253	1313306119	253298680	1010275	11,27421245
./results/1000lu5.out	19756200100	1311691446	253296392	1005868	11,15033855
./results/1000lu6.out	19757194812	1313212629	253302024	1010461	11,29538668
./results/1000lu7.out	19756200952	1313291887	253296268	1004771	10,99824644
./results/1000lu8.out	19756216290	1313870575	253296399	1005228	11,16564976
./results/1000lu9.out	19756210562	1312072586	253296393	1005677	11,185632
./results/1000lu10.out	19756425525	1312386588	253297222	1007119	10,98753599
AVG	19756379242	1312784903	253297478,5	1007156,5	10,59581154
STDEV	340706,4432	1036565,001	1825,319774	1986,111737	1,044036465
CONFIDENCE 95	211168,161	642457,8381	1131,324128	1230,982188	0,6470886147