University of Minho
School of Engineering

Perf

Utilização da ferramenta *Perf* para a análise de aplicações

Hugo Afonso Da Gião
PG41073

12 de Julho de 2020

Resumo

Este trabalho consiste na analise diferentes algoritmos utilizando a ferramenta *Perf*.Estes algoritmos consistem numa primeira fase em vários algoritmos de ordenação e numa fase posterior em vários algoritmos de multiplicação de matrizes. Na análise destes algoritmos é também utilizada a ferramenta *FlameGraph*.

Palavras-chave:*Perf*,*Sort*,*Matrix multiplication*,*FlameGraphs*

Conteúdo

Capítulo 1

Introdução

Este documento relata o trabalho realizado na análise de vários algoritmos utilizando a ferramenta *Perf*, a analise dos resultados obtidos e também a visualização dos mesmos com recurso a *FlameGraphs*.

No primeiro capítulo é detalhado o trabalho realizado para a analise de vários algoritmos de ordenação de modo a obter possíveis explicações no desempenho dos mesmos, obter e analisar os perfis de execução dos vários algoritmos, visualizar os seus comportamentos com o recurso a *FlameGraphs* e analisar o perfil de alguns dos algoritmos a nível *assembly*.

O segundo capítulo relata o trabalho realizado para a analise de duas versões do algoritmo de multiplicação de matrizes, este consiste inicialmente na análise e realização de *benchmarks* da versão *naive* do algoritmo para diferentes tamanhos e posteriormente na comparação entre os dois algoritmos e visualização do comportamento dos algoritmos com o auxilio dos *FlameGraphs*.

Posteriormente são apresentadas algumas conclusões relacionadas com os resultados obtidos, a aprendizagem realizada e trabalho futuro.

Capítulo 2

Analise de diferentes algoritmos de ordenação

2.1 Hardware de teste

L1 data cache	32 *KiB*
L1 instructions cache	32 *KiB*
L2 data cache	256 *KiB*
L3 data cache	12228 *KiB*
Sockets	2
CPU cores per socket	6
SMT	*yes*

Tabela 2.1: Especificações de *hardware* da maquina r431 utilizada para os testes realizados

Para os testes realizados nesta secção foram utilizadas maquinas do *rack r431* do *cluster SeARCH*.O uso destas maquinas para a resolução deste trabalho deu-se pela sua conveniência por possuir previamente uma instalação do *Perf*,pela sua disponibilidade e por possuir vários contadores que permitam a avaliação dos vários algoritmos.

2.2 Uso de *Perf* para explicar as diferenças no desempenho dos diferentes algoritmos

Metric/Algorithm	1	2	3	4
instructions	79,142,070,310	83,795,234,588	136,598,913,137	177,008,053,513
cache-misses	15,953,867	31,073,839	1,049,451,527	45,791,788
branch-misses	1,267,781,147	446,627	1,441,193,590	1,459,663,522
cpu-migration	2	0	0	6
branches	13,100,153,040	7,018,274,144	14,995,852,221	24,794,572,013
time	27.156050625	23.550020823	57.118593935	47.881616187

Tabela 2.2: Especificações de *hardware* da maquina r432 utilizada para os testes realizados

A partir da tabela acima podemos verificar que o algoritmo 2 tem os tempos de execução mais baixos seguido do algoritmo 1 ambos com tempos de execução entre 20 e 30 segundos e posteriormente os 3 e 4 com tempos significativamente piores do que os anteriores sendo estes entre 2 e 3 vezes piores que os dois melhores.

Estes tempos justificam-se parcialmente pelo número de instruções sendo que os algoritmos 1 e 2 apresentam consideravelmente menos instruções e tempos de execução comparativamente aos 3 e 4.No entanto o algoritmo 2 apresenta tempos de execução inferiores apesar de o seu numero de instruções ser superior ao algoritmo 1.O numero de cache misses similarmente justifica algumas das diferenças de performance nomeadamente justifica a pior performance da versão 3 relativamente a 4 dada apesar da primeira possuir menos instruções.

Algo que justifica a melhor performance do algoritmo 2 relativamente ao 1 é o numero de *branches* e *branch-misses* sendo que ambos são inferiores para o algoritmo 2 especialmente o numero de *branch-misses*.A existência de um menor numero de *branches* e *branch-misses* permite que o processador faça um menor uso da *cache*,como os dados carregados são os utilizados em vez de serem descartados. Algo similar acontece com instruções em que as instruções necessárias são carregadas mais eficientemente.

2.3 Analise dos perfis de execução dos diferentes algoritmos

```
> perf record -F 99 ./sort algorithm 1 100000000
> perf report -n --stdio
```

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 80447717144
#
# Overhead       Samples  Command      Shared Object                          Symbol
# ........  ............  .......  .................  ...............................
#
    93.68%          2559     sort  sort               [.] sort1(int*, int, int)
     1.97%            90     sort  sort               [.] ini_vector(int**, int)
     1.16%            58     sort  sort               [.] copy_vector(int*, int*, int)
     0.88%            35     sort  libc-2.12.so       [.] __random_r
     0.80%            40     sort  libc-2.12.so       [.] __random
     0.36%            18     sort  libc-2.12.so       [.] rand
     0.31%             8     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.28%            14     sort  [kernel.kallsyms]  [k] clear_page_c
     0.15%             1     sort  [kernel.kallsyms]  [k] __d_lookup
     0.12%             1     sort  [kernel.kallsyms]  [k] page_fault
     0.10%             5     sort  sort               [.] rand@plt
     0.04%             1     sort  [kernel.kallsyms]  [k] rcu_process_gp_end
     0.04%             1     sort  [kernel.kallsyms]  [k] physflat_send_IPI_mask
     0.04%             1     sort  [kernel.kallsyms]  [k] _spin_lock_irqsave
     0.02%             1     sort  [kernel.kallsyms]  [k] idle_cpu
     0.02%             1     sort  [kernel.kallsyms]  [k] __mem_cgroup_commit_charge
     0.02%             1     sort  [kernel.kallsyms]  [k] ____pagevec_lru_add
     0.02%             1     sort  [kernel.kallsyms]  [k] __rcu_process_callbacks
     0.00%            20     sort  [kernel.kallsyms]  [k] native_write_msr_safe
```

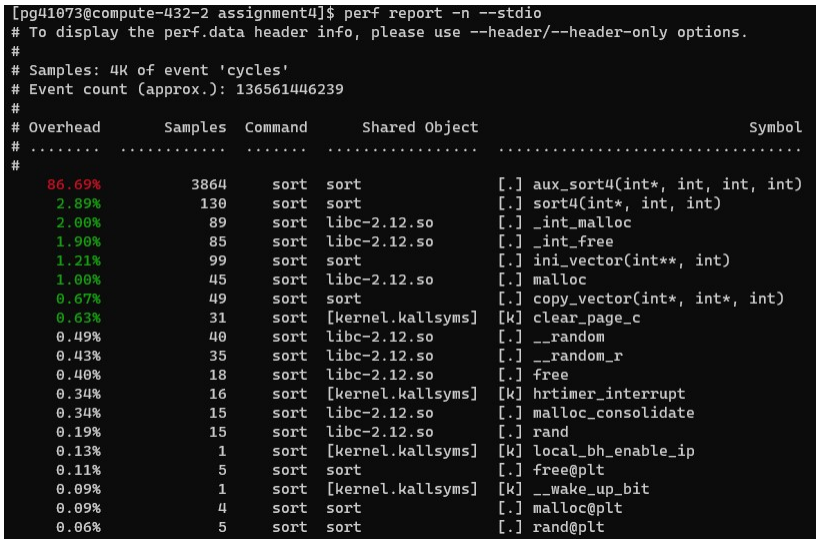
Figura 2.1: resultados para um array de inteiros de tamanho 100,000,000 para o algoritmo 1

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 72382577977
#
# Overhead       Samples  Command      Shared Object                          Symbol
# ........  ............  .......  .................  ...............................
#
    92.63%          2396     sort  sort               [.] sort2(int*, int)
     2.17%            96     sort  sort               [.] ini_vector(int**, int)
     1.32%            51     sort  sort               [.] copy_vector(int*, int*, int)
     1.18%            44     sort  libc-2.12.so       [.] __random
     0.63%             9     sort  sort               [.] rand@plt
     0.54%            24     sort  libc-2.12.so       [.] __random_r
     0.44%            17     sort  [kernel.kallsyms]  [k] clear_page_c
     0.27%             7     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.20%             1     sort  [kernel.kallsyms]  [k] page_fault
     0.18%             1     sort  [kernel.kallsyms]  [k] audit_putname
     0.16%             7     sort  libc-2.12.so       [.] rand
     0.09%             2     sort  [kernel.kallsyms]  [k] idle_cpu
     0.06%             2     sort  [kernel.kallsyms]  [k] irq_exit
     0.04%             1     sort  [kernel.kallsyms]  [k] rcu_process_dyntick
     0.04%             1     sort  [kernel.kallsyms]  [k] tick_program_event
     0.03%             1     sort  [kernel.kallsyms]  [k] get_page_from_freelist
     0.02%             1     sort  [kernel.kallsyms]  [k] ktime_get
     0.00%            20     sort  [kernel.kallsyms]  [k] native_write_msr_safe
```

Figura 2.2: Resultados para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 2

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 5K of event 'cycles'
# Event count (approx.): 174903982073
#
# Overhead       Samples  Command      Shared Object                          Symbol
# ........  ............  .......  .................  ...............................
#
    96.74%          5483     sort  sort               [.] sort3(int*, int)
     0.94%           102     sort  sort               [.] ini_vector(int**, int)
     0.52%            37     sort  sort               [.] copy_vector(int*, int*, int)
     0.47%            51     sort  libc-2.12.so       [.] __random
     0.35%            21     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.26%            28     sort  libc-2.12.so       [.] __random_r
     0.19%            16     sort  [kernel.kallsyms]  [k] clear_page_c
     0.13%            14     sort  libc-2.12.so       [.] rand
     0.09%             1     sort  [kernel.kallsyms]  [k] security_socket_sendmsg
     0.09%             2     sort  [kernel.kallsyms]  [k] _spin_lock
     0.05%             5     sort  sort               [.] rand@plt
     0.04%             2     sort  [kernel.kallsyms]  [k] __percpu_counter_add
     0.04%             2     sort  [kernel.kallsyms]  [k] account_user_time
     0.02%             1     sort  [kernel.kallsyms]  [k] run_timer_softirq
     0.02%             1     sort  [kernel.kallsyms]  [k] rcu_process_gp_end
     0.02%             1     sort  [kernel.kallsyms]  [k] scheduler_tick
     0.02%             1     sort  [kernel.kallsyms]  [k] update_cfs_shares
     0.02%             1     sort  [kernel.kallsyms]  [k] native_read_tsc
     0.02%             1     sort  [kernel.kallsyms]  [k] irq_exit
     0.00%             8     sort  [kernel.kallsyms]  [k] native_write_msr_safe
```

Figura 2.3: Resultados para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 3



```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 4K of event 'cycles'
# Event count (approx.): 136561446239
#
# Overhead       Samples  Command      Shared Object                              Symbol
# ........  ............  .......  .................  ...................................
#
    86.69%          3864     sort  sort               [.] aux_sort4(int*, int, int, int)
     2.89%           130     sort  sort               [.] sort4(int*, int, int)
     2.00%            89     sort  libc-2.12.so       [.] _int_malloc
     1.90%            85     sort  libc-2.12.so       [.] _int_free
     1.21%            99     sort  sort               [.] ini_vector(int**, int)
     1.00%            45     sort  libc-2.12.so       [.] malloc
     0.67%            49     sort  sort               [.] copy_vector(int*, int*, int)
     0.63%            31     sort  [kernel.kallsyms]  [k] clear_page_c
     0.49%            40     sort  libc-2.12.so       [.] __random
     0.43%            35     sort  libc-2.12.so       [.] __random_r
     0.40%            18     sort  libc-2.12.so       [.] free
     0.34%            16     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.34%            15     sort  libc-2.12.so       [.] malloc_consolidate
     0.19%            15     sort  libc-2.12.so       [.] rand
     0.13%             1     sort  [kernel.kallsyms]  [k] local_bh_enable_ip
     0.11%             5     sort  sort               [.] free@plt
     0.09%             1     sort  [kernel.kallsyms]  [k] __wake_up_bit
     0.09%             4     sort  sort               [.] malloc@plt
     0.06%             5     sort  sort               [.] rand@plt
```

Figura 2.4: Resultados para um array de inteiros de tamanho 100,000,000 para o algoritmo 4

A partir dos testes realizados pude verificar que a maioria do tempo gasto de execução dos vários algoritmos é gasto em funções de utilizador mais precisamente nas várias funções de ordenação e auxiliares.

Mais precisamente pude verificar que a percentagem do tempo gasto nas funções de utilizador é superior para os 3 primeiro algoritmo, especialmente para o segundo algoritmo e consideravelmente inferior para o quarto algoritmo, sendo que neste os testes evidenciam que um tempo considerável da execução do mesmo é gasto em funções de alocação e libertação estáticas de memória.

Denotasse também que a percentagem do tempo despendido nas funções de inicialização é superior para o primeiro e segundo algoritmo. Isto deve-se por estes possuírem tempos de execução consideravelmente inferiores como visto anteriormente.

2.4 Uso de *Flamegraphs* para analisar as chamadas dos algoritmos

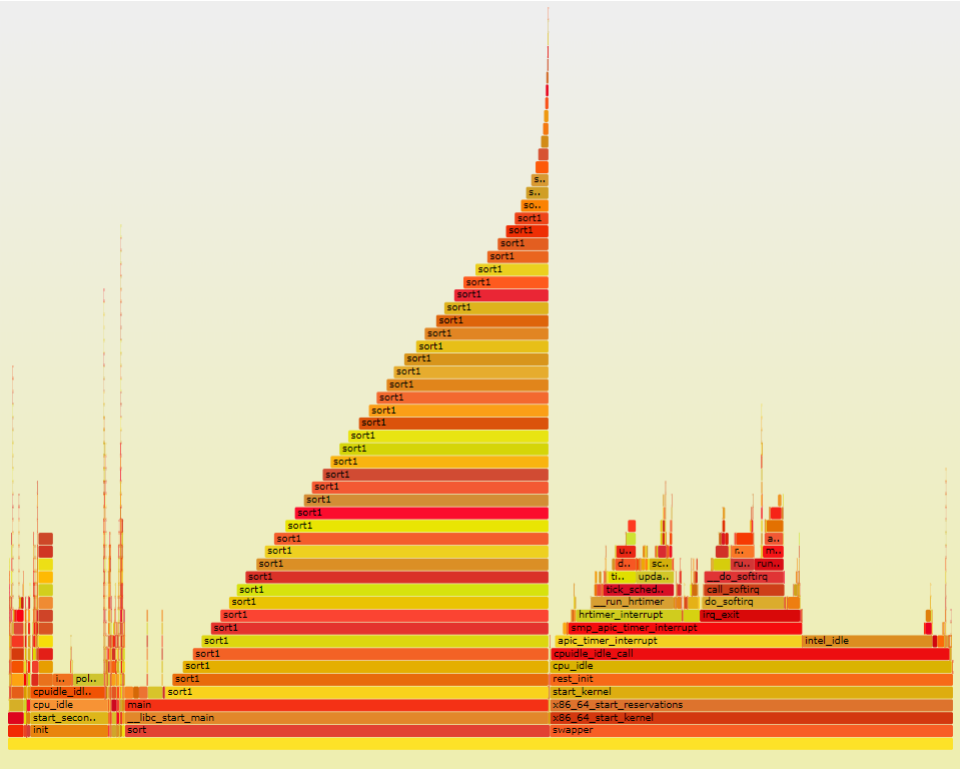


Figura 2.5: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 1

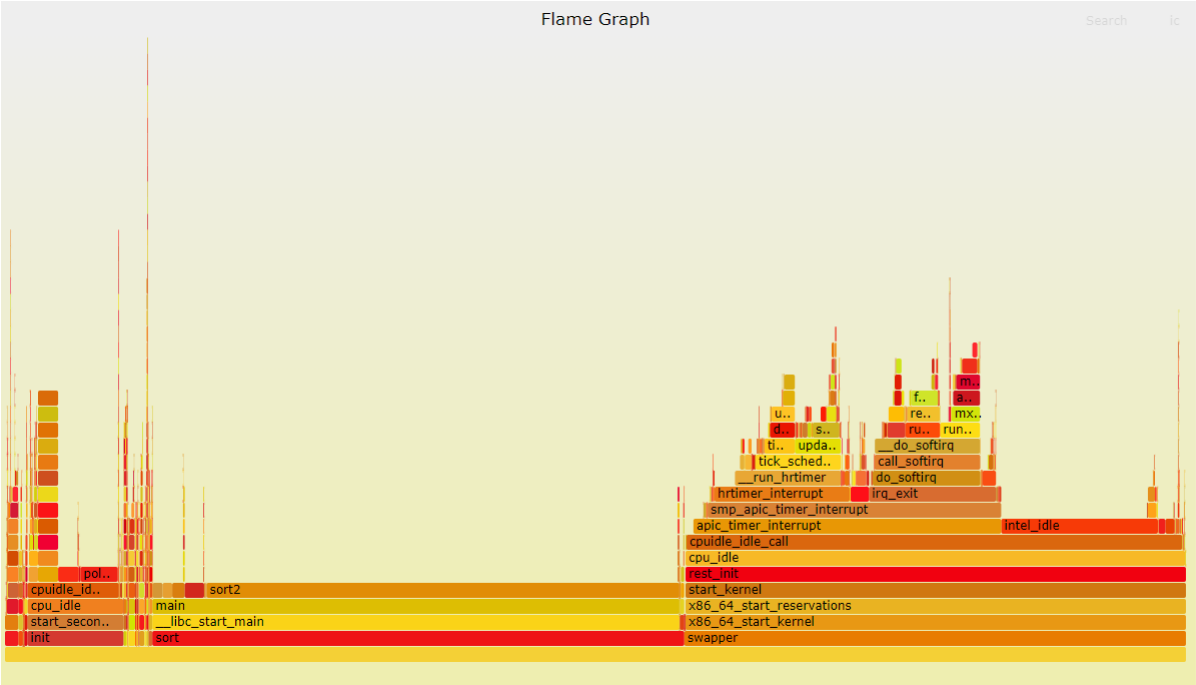


Figura 2.6: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 2

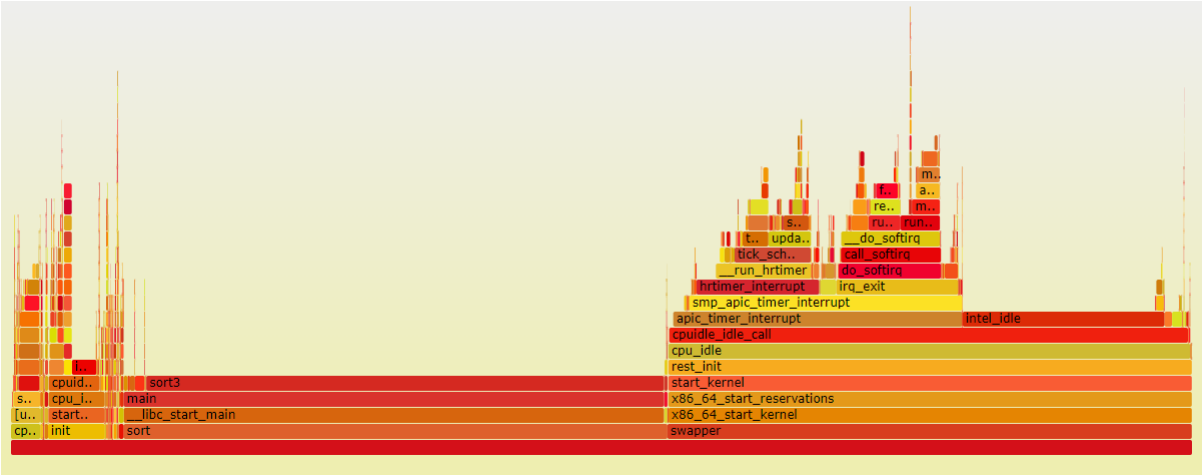


Figura 2.7: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 3

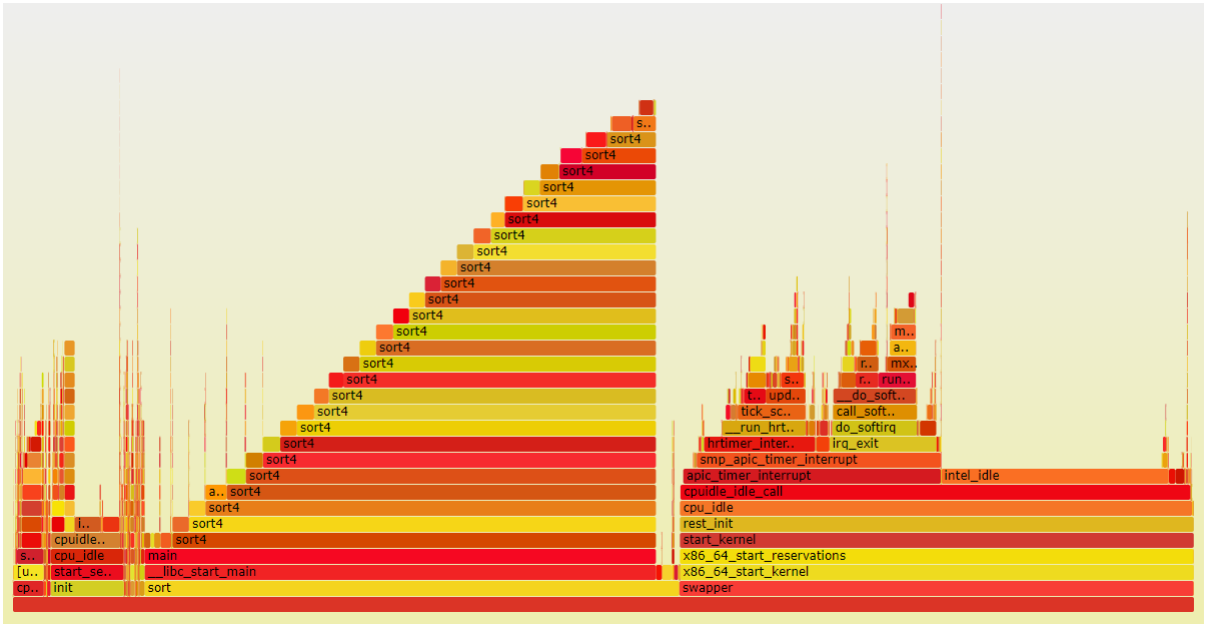


Figura 2.8: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 4

Com base no aninhamento das chamadas pude verificar que os algoritmos 1 e 4 tem um comportamento similar em termos de chamadas recursivas. Podemos verificar também a partir dos gráficos que no algoritmo 4 as funções de ordenação fazem chamadas a uma função auxiliar que tem um tempo de execução relativamente constante relativamente a chamada recursiva da mesma função a medida que são realizadas as chamadas. Verifica-se também que são realizadas significativamente menos chamadas para o algoritmo 4 do para o 1.Visto que no algoritmo de *merge sort* em todos os níveis das chamadas recursivas é realizado o *merge* de todos os elementos e este poderá ter tempos de execução semelhantes para cada um dos níveis podemos assumir que a função auxiliar ao algoritmo 4 realiza esta tarefa e o gráfico do algoritmo gráfico 4 demonstra que durante as chamadas as função auxiliar do algoritmo 4 são invocadas varias funções relacionadas com alocações de memória, por estas razões podemos assumir que o algoritmo 4 é o algoritmo *merge sort*.Como o algoritmo possui um comportamento similar em termos de chamadas recursivas possuindo mais níveis do que o anterior, dá-se a possibilidade de este ser o algoritmo de *quick sort*,o que justifica a existência de um maior numero de níveis das chamadas recursivas por este algoritmo não ser estável.

A partir dos gráficos é similarmente verificável que ambos os algoritmos 2 e 3 não realizam chamadas recursivas. Sendo que os comportamentos destas funções são similares a nível de chamadas pode-se verificar a partir das medições realizadas anteriormente em que realizamos que existe um numero de cache-misses bastante inferior para o algoritmo 2 e que este possui tempos de execução também significativamente inferiores aos vários outros assumir que este algoritmo seja o *radix sort* e por possuir um numero elevado quando comparado de *cache-misses* e tempos comparativamente aos outros algoritmos podemos assumir que o algoritmo 3 é o *heap sort*,visto que este algoritmo apresenta fraca localidade temporal e espacial nos acessos aos dados.

2.5 Analise do segundo algoritmo a nível *assembly*

```
> perf record -g ./bin/sort 2 1 100000000
> perf annotate --stdio
```

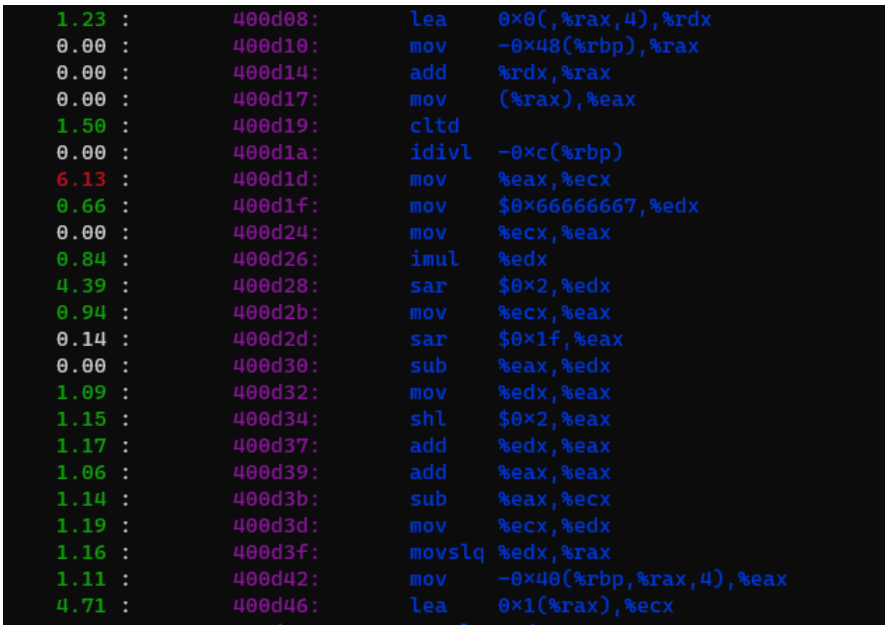


Figura 2.9: *assembly* anotado para o algoritmo 2 parte 1

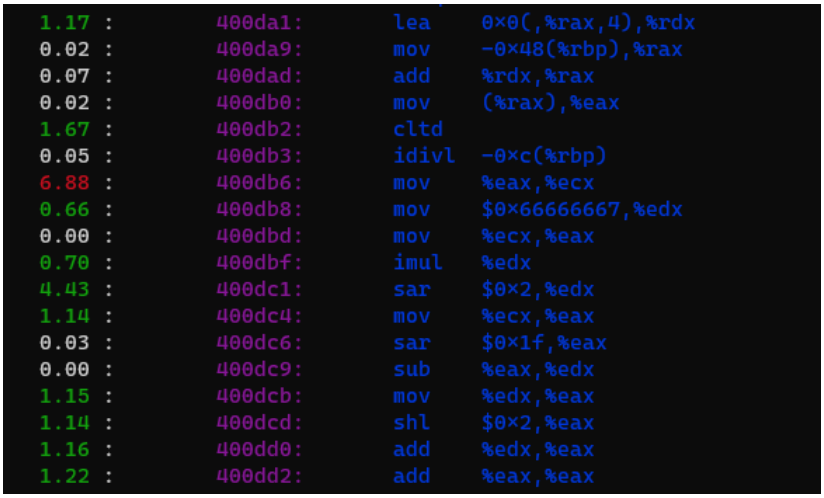


Figura 2.10: *assembly* anotado para o algoritmo 2 parte 2

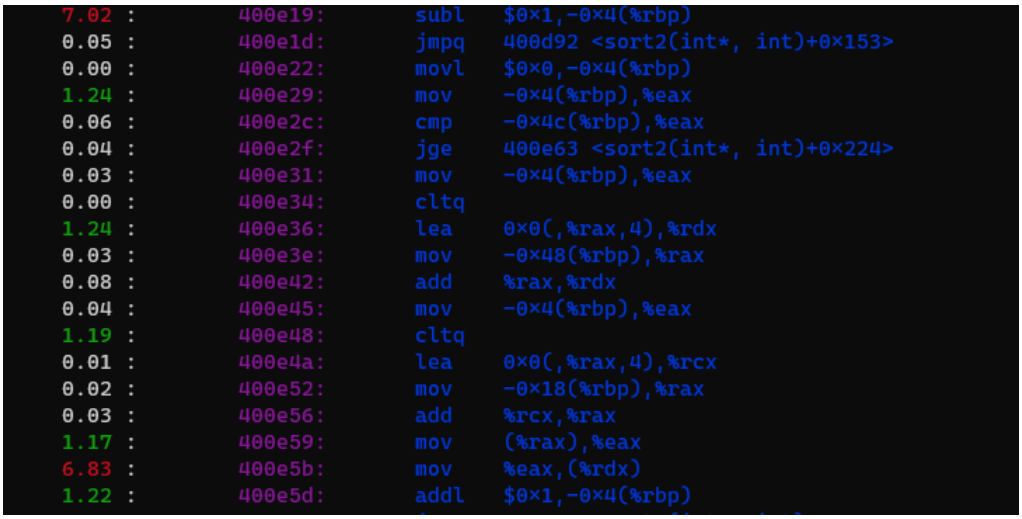


Figura 2.11: *assembly* anotado para o algoritmo 2 parte 3

A partir dos resultados deste comando pude verificar que grande parte do tempo de execução é gasto no carregamento e escrita de dados e no cálculo das posições do *array* onde inserir e retirar dados. Tendo isto em conta uma maneira de melhorar a performance deste algoritmo passaria por aumentar a localidade dos dados por exemplo utilizar a variante *MSD* em vez de *LSD* como está a utilizar o algoritmo, visto que esta possui melhor localidade temporal. Algo que também poderia melhorar a performance deste algoritmo passaria por realizar um alinhamento do *array* dos dígitos e dos *arrays* temporários de modo a diminuir *cache misses* e fazer uso da vectorização.

Capítulo 3

Analise de algoritmos de multiplicação de matrizes com uso do *Perf*

3.1 Algoritmos de multiplicação de matrizes utilizados

Para o trabalho realizado nesta secção foi utilizado um implementação do algoritmo *naive* de multiplicação de matrizes encontrado em [2].Para alguns dos testes realizados foi também implementado um algoritmo de multiplicação de matrizes alternativo, este algoritmo modifica o ordem dos ciclos k e j, o que garante uma maior localidade temporal no acesso aos dados o que permite obter melhor usos de cache e da memoria e das extensões vetoriais.

Para facilitar a analise e os testes realizados o programa foi modificado para aceitar o tamanho das matrizes e o algoritmo a utilizar como argumento. Foi também utilizada a diretiva *#pragma GCC ivdep*, como as matrizes são passadas como argumento e para garantir que os resultados são mais semelhantes aos da versão anterior em que as matrizes são alocadas dinamicamente e consistem em variáveis globais, o uso desta diretiva permite que o compilador assuma que não existem dependências entre as matrizes.

Algorithm 1 Naive matrix multiplication algorithm

```
1: for (i ← 0; i < M_size; i++) do
2:     for (j ← 0; j < M_size; j++) do
3:         sum ← 0
4:         for (k ← 0; k < M_size; k++) do
5:             sum+ = M_a[i][k] * M_b[k][j]
6:         end for
7:         M_r[i][j] ← sum
8:     end for
9: end for
```

Algorithm 2 Loop nest interchange matrix multiplication algorithm

```
1: for (i ← 0; i < M_size; i++) do
2:     for (k ← 0; k < M_size; k++) do
3:         for (j ← 0; j < M_size; j++) do
4:             M_r[i][j]+ = M_a[i][k] * M_b[k][j]
5:         end for
6:     end for
7: end for
```

3.2 *Hardware* de teste

L1 data cache	32 *KiB*
L1 instructions cache	32 *KiB*
L2 data cache	256 *KiB*
L3 data cache	20480 *KiB*
Sockets	2
CPU cores per socket	8
SMT	*yes*

Tabela 3.1: Especificações de *hardware* da maquina r431 utilizada para os testes realizados

Para efeitos de teste foram utilizadas como para o exercício anterior maquinas do *rack* $r431$ do *cluster SeARCH* estas maquinas foram utilizadas por terem o *software* necessário a resolução destes exercícios previamente instalados e por possuírem contadores de *hardware* e eventos similares aos do *Hardware* similar aos das maquinas utilizadas nos tutoriais utilizados como base para este exercício.

3.3 Tamanhos do problema

Os testes foram realizados com 4 tamanhos distintos estes foram escolhidas de modo as estruturas de dados utilizadas pelo programa caberem em *cache* nível 1,*cache* nível 2,*cache* nível 3 e em memoria *RAM*.

fits in L1	32
fits in L2	128
fits in L3	512
fits in RAM	2048

Tabela 3.2: Especificações de *hardware* da maquina r432 utilizada para os testes realizados

3.4 Estabelecer uma *beline*

Para estabelecer a *baseline* foram medidos os evento $cpu - clocks$ e $faults$ decorridos durante a execução de uma multiplicação de matrizes utilizando o algoritmo *naive*, para os diferentes tamanhos utilizando o primeiro algoritmo. Para tal foram utilizados os comandos seguintes.

```
> perf stat -e cpu-clock ./bin/naive 1 size
> perf stat -e cpu-clock,faults ./bin/naive 1 size
```

size	**cpu-clocks events in milliseconds**	**faults**	**time in seconds**
32	2.322247	308	0.003654498
128	7.190261	355	0.008417665
512	314.661839	1081	0.317850716
2048	47168.633150	12650	47.341694693

Tabela 3.3: medições do evento $cpu-clock$ e tempos de execução do primeiro algoritmo para diferentes tamanhos

O cálculo da *baseline* permite avaliar que o número de eventos de $cpu - clocks$ é aproximado do número real sendo que os valores coincidem com os que seriam de esperar de acordo com o tempo de execução registado. Apesar disso encontram-se algumas discrepâncias para os dois tamanhos inferiores.

3.5 Encontro de pontos quentes

A fim de analisar possíveis *bottlenecks* e possibilidades de otimização no programa foi utilizado o comando abaixo para poder visualizar as funções que mais registaram os eventos $cpu - clock$ e $faults$ para a versão *naive* para os diversos tamanhos.

```
> perf record -e cpu-clock,faults ./bin/naive 1 size
```

```
> perf report --stdio --sort comm,dso
```

```
[pg41073@compute-432-2 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 167K of event 'cpu-clock'
# Event count (approx.): 167323
#
# Overhead  Command      Shared Object
# ........  .......  ..................
#
    99.40%    naive  naive
     0.37%    naive  [kernel.kallsyms]
     0.23%    naive  libc-2.12.so
     0.00%    naive  ld-2.12.so
     0.00%    naive  [nfs]


# Samples: 356  of event 'faults'
# Event count (approx.): 12772
#
# Overhead  Command      Shared Object
# ........  .......  ..................
#
    48.90%    naive  libc-2.12.so
    48.22%    naive  naive
     2.86%    naive  ld-2.12.so
     0.02%    naive  [kernel.kallsyms]
```

Figura 3.1: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```
[pg41073@compute-431-5 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 1K of event 'cpu-clock'
# Event count (approx.): 1277
#
# Overhead  Command      Shared Object
# ........  .......  ..................
#
    96.48%    naive  naive
     2.19%    naive  libc-2.12.so
     0.78%    naive  [kernel.kallsyms]
     0.55%    naive  ld-2.12.so


# Samples: 11  of event 'faults'
# Event count (approx.): 1100
#
# Overhead  Command      Shared Object
# ........  .......  ..................
#
    70.73%    naive  libc-2.12.so
    29.00%    naive  ld-2.12.so
     0.27%    naive  [kernel.kallsyms]
```

Figura 3.2: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.

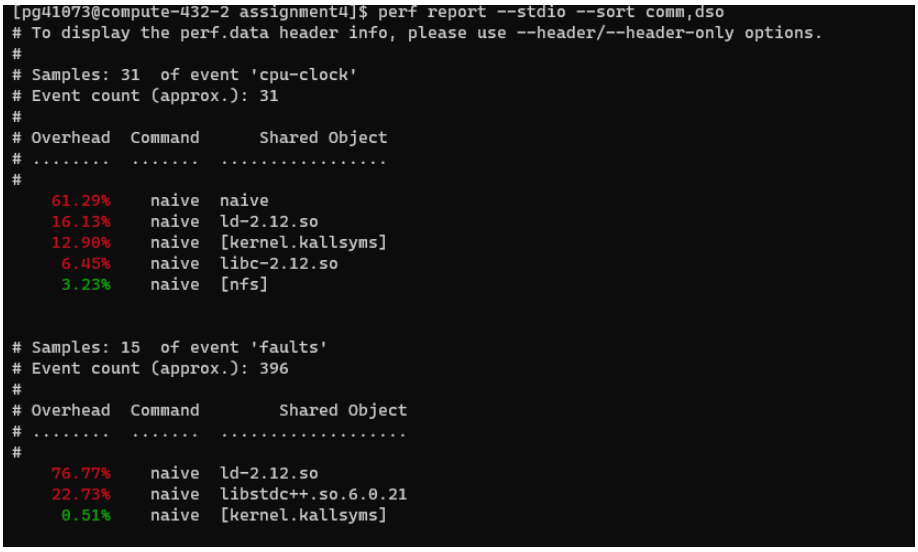


Figura 3.3: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

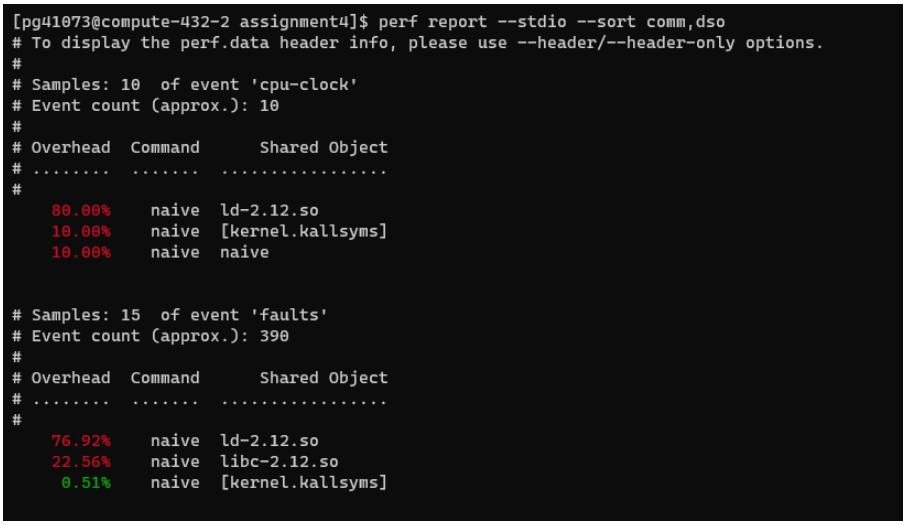


Figura 3.4: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Em termos de $cpu - clock$ é possível verificar que a maioria do tempo gasto pelo programa, para os 3 tamanhos maiores, para o menor tamanho verifica-se que o tempo gasto maioritariamente no *Dynamic linker* e funções de *kernel*.

Em termos de $fault$ verifica-se que estas acontecem para o maior tamanho maioritariamente em *system calls* e no programa. Para os outros tamanhos estas acontecem maioritariamente em *system calls*.

Para restringir os resultados as funções do programa e da biblioteca de *runtime* do *c* foi utilizado o comando abaixo.

```
> perf report --stdio --dsos=naive,libc-2.12.so,ld-2.12.so,libstdc++.so.6.0.21
´
```

```
#
# Samples: 162K of event 'cpu-clock'
# Event count (approx.): 162429
#
# Overhead  Command        Shared Object                                                Symbol
# ........  .......  ...................  ....................................................
#
    99.60%    naive  naive                [.] multiply_matrices(float**, float**, float**, int)
     0.07%    naive  naive                [.] initialize_matrices(float**, float**, float**, int)
     0.05%    naive  libc-2.12.so         [.] __random
     0.05%    naive  libc-2.12.so         [.] __random_r
     0.02%    naive  naive                [.] rand@plt
     0.01%    naive  libc-2.12.so         [.] rand
     0.00%    naive  libc-2.12.so         [.] _int_malloc
     0.00%    naive  ld-2.12.so           [.] _dl_lookup_symbol_x
     0.00%    naive  ld-2.12.so           [.] _dl_relocate_object
     0.00%    naive  ld-2.12.so           [.] do_lookup_x
     0.00%    naive  libc-2.12.so         [.] __brk
     0.00%    naive  libstdc++.so.6.0.21  [.] _GLOBAL__sub_I_locale_inst.cc


# Samples: 300  of event 'faults'
# Event count (approx.): 12687
#
# Overhead  Command        Shared Object                                                Symbol
# ........  .......  ...................  ....................................................
#
    48.48%    naive  libc-2.12.so         [.] _int_malloc
    48.44%    naive  naive                [.] initialize_matrices(float**, float**, float**, int)
     0.76%    naive  ld-2.12.so           [.] match_symbol
     0.76%    naive  ld-2.12.so           [.] strcmp
     0.69%    naive  libstdc++.so.6.0.21  [.] call_gmon_start
     0.31%    naive  ld-2.12.so           [.] _dl_sysdep_start
     0.18%    naive  naive                [.] multiply_matrices(float**, float**, float**, int)
```

Figura 3.5: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```
# Event count (approx.): 1268
#
# Overhead  Command  Shared Object                                                  Symbol
# ........  .......  .............  ....................................................
#
    95.43%    naive  naive          [.] multiply_matrices(float**, float**, float**, int)
     1.18%    naive  naive          [.] initialize_matrices(float**, float**, float**, int)
     0.63%    naive  libc-2.12.so   [.] __random_r
     0.32%    naive  libc-2.12.so   [.] __random
     0.32%    naive  libc-2.12.so   [.] rand
     0.08%    naive  ld-2.12.so     [.] _dl_lookup_symbol_x
     0.08%    naive  ld-2.12.so     [.] _dl_map_object
     0.08%    naive  ld-2.12.so     [.] _dl_relocate_object
     0.08%    naive  ld-2.12.so     [.] dl_main
     0.08%    naive  ld-2.12.so     [.] do_lookup_x
     0.08%    naive  libc-2.12.so   [.] _int_malloc
     0.08%    naive  naive          [.] rand@plt


# Samples: 20  of event 'faults'
# Event count (approx.): 1117
#
# Overhead  Command  Shared Object                      Symbol
     3.67%    naive  ld-2.12.so     [.] memset
     1.07%    naive  ld-2.12.so     [.] _dl_new_object
     0.72%    naive  ld-2.12.so     [.] _dl_load_cache_lookup
     0.63%    naive  ld-2.12.so     [.] _dl_sysdep_start
     0.63%    naive  ld-2.12.so     [.] _start
     0.54%    naive  ld-2.12.so     [.] _dl_map_object_from_fd
     0.09%    naive  ld-2.12.so     [.] _dl_next_tls_modid
```

Figura 3.6: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 22  of event 'cpu-clock'
# Event count (approx.): 22
#
# Overhead  Command  Shared Object                                                Symbol
# ........  .......  .............  ..................................................
#
    50.00%    naive  naive          [.] multiply_matrices(float**, float**, float**, int)
    13.64%    naive  ld-2.12.so     [.] strcmp
     9.09%    naive  libc-2.12.so   [.] rand
     4.55%    naive  ld-2.12.so     [.] do_lookup_x


# Samples: 15  of event 'faults'
# Event count (approx.): 556
#
# Overhead  Command        Shared Object                                        Symbol
# ........  .......  ...................  ...............................................
#
    39.39%    naive  ld-2.12.so           [.] _dl_lookup_symbol_x
    28.96%    naive  libstdc++.so.6.0.21  [.] _GLOBAL__sub_I_compatibility_thread_c__0x.cc
    18.88%    naive  ld-2.12.so           [.] _dl_map_object
     4.86%    naive  ld-2.12.so           [.] _dl_cache_libcmp
     3.24%    naive  ld-2.12.so           [.] dl_main
     1.26%    naive  ld-2.12.so           [.] _dl_start
     1.08%    naive  ld-2.12.so           [.] _dl_sysdep_read_whole_file
     0.90%    naive  ld-2.12.so           [.] _dl_map_object_from_fd
     0.54%    naive  ld-2.12.so           [.] _start
     0.18%    naive  ld-2.12.so           [.] _dl_next_tls_modid
     0.18%    naive  ld-2.12.so           [.] memset
```

Figura 3.7: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 8  of event 'cpu-clock'
# Event count (approx.): 8
#
# Overhead  Command  Shared Object                  Symbol
# ........  .......  .............  ......................
#
    37.50%    naive  ld-2.12.so     [.] do_lookup_x
    12.50%    naive  ld-2.12.so     [.] _dl_lookup_symbol_x
    12.50%    naive  ld-2.12.so     [.] _dl_map_object_deps
    12.50%    naive  ld-2.12.so     [.] check_match.12442


# Samples: 15  of event 'faults'
# Event count (approx.): 421
#
# Overhead  Command  Shared Object                         Symbol
# ........  .......  .............  ............................
#
    25.18%    naive  ld-2.12.so     [.] match_symbol
    25.18%    naive  ld-2.12.so     [.] strcmp
    25.18%    naive  libc-2.12.so   [.] malloc
    13.54%    naive  ld-2.12.so     [.] dl_main
     4.99%    naive  ld-2.12.so     [.] _dl_cache_libcmp
     1.43%    naive  ld-2.12.so     [.] _dl_setup_hash
     1.43%    naive  ld-2.12.so     [.] _dl_sysdep_read_whole_file
     1.19%    naive  ld-2.12.so     [.] _dl_map_object_from_fd
     0.71%    naive  ld-2.12.so     [.] _dl_start
     0.24%    naive  ld-2.12.so     [.] _dl_next_tls_modid
     0.24%    naive  ld-2.12.so     [.] _start
     0.24%    naive  ld-2.12.so     [.] memset
:
```

Figura 3.8: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Os resultados obtidos com estas opções são similares aos anteriores. Com este comando pode-se verificar que a função *multiply_matrices* é a onde é gasto a maioria do tempo para os 3 tamanhos maiores. Verifica-se também que algum tempo é gasto na função de *initialize_matrices* para os dois maiores tamanhos. Em termos de *faults* verifica-se que estes estão relacionados com alocação de memória para o tamanho maior para os restantes tamanhos varias *system calls* estão associadas aos

mesmos.

3.6 Analise do programa ao nível *assembly*

De modo a poder analisar as secções da função de multiplicação de matrizes *naive* que consomem mais tempo de execução foi utilizado o comando *perf annotate* para essa função.

```
> perf annotate --stdio --dsos=naive
--symbol="multiply_matrices(float**, float**, float**, int)"
```

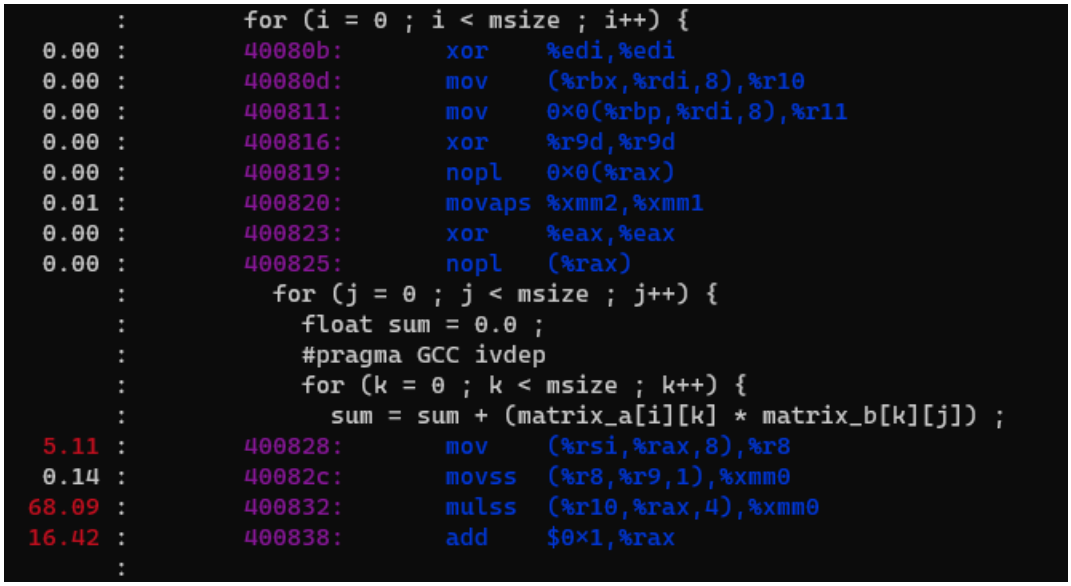


Figura 3.9: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048 parte 1.

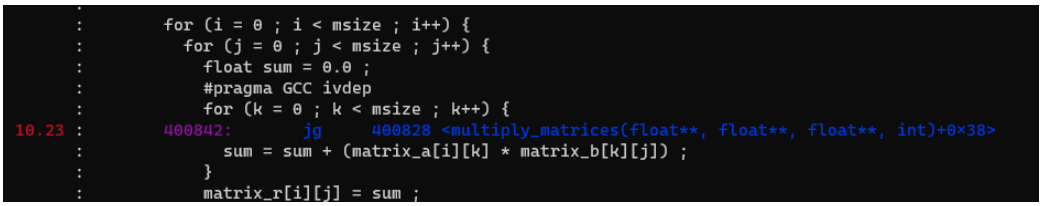


Figura 3.10: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048 parte 2.

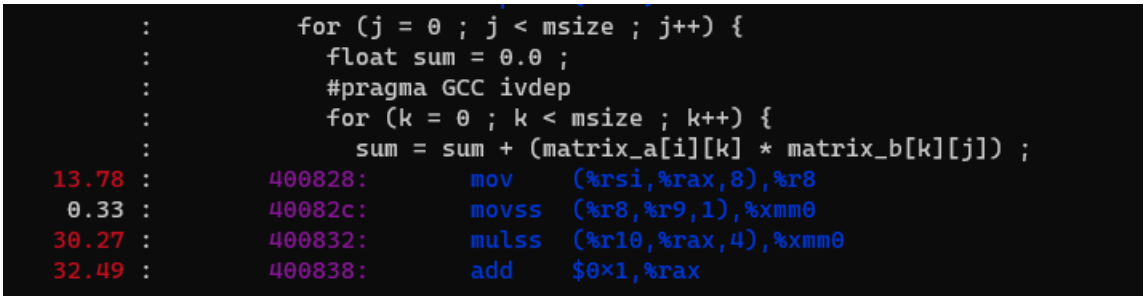


Figura 3.11: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512 parte 1.

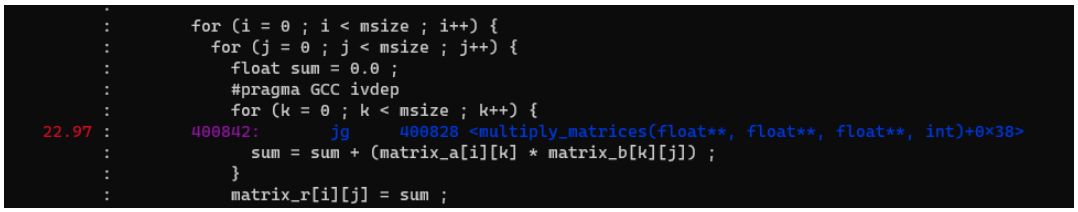


Figura 3.12: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512 parte 2.

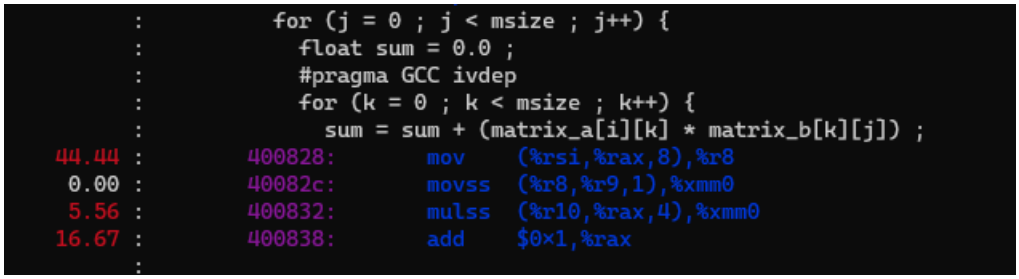


Figura 3.13: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128 parte 1.

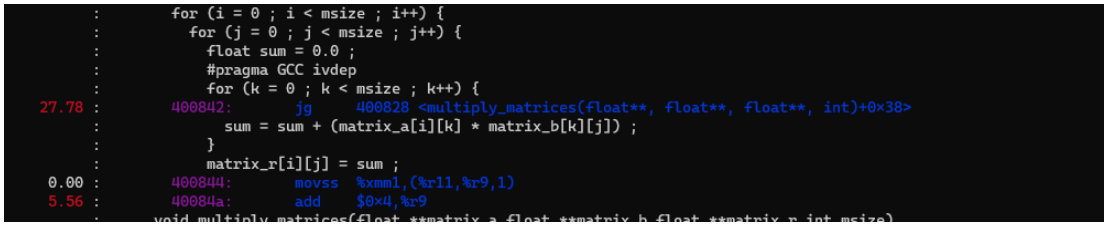


Figura 3.14: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128 parte 2.

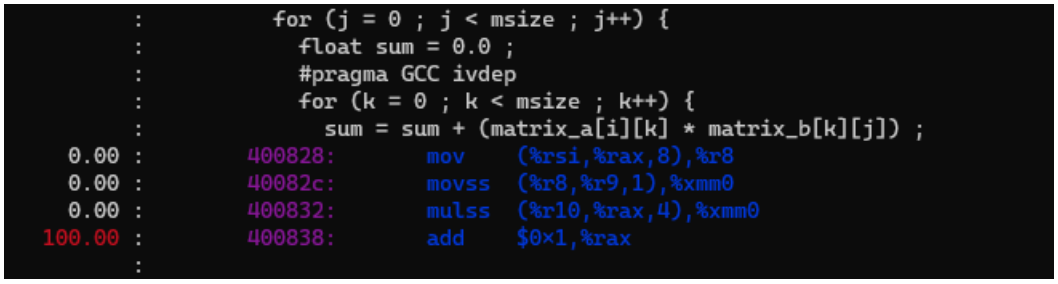


Figura 3.15: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

De acordo com os resultados obtidos podemos verificar que grande parte do tempo do programa é gasto em instruções relacionadas com as somas e multiplicações necessárias para calcular a matriz resultado. Estas instruções consistem em instruções de multiplicação, de movimentação de dados e de adição. A percentagem das mesmas varia consideravelmente com os tamanhos. Verifica-se também para os vários tamanhos que é gasto um tempo considerável na operação de salto condicional associado ao *loop* mais interno.

Como existem complicações na interpretação do resultado do comando anterior devido a otimizações realizadas pelo compilador e dificuldades em associar o output em *assembly* ao respetivo código foi também utilizado este comando com a opção $--no-source$ para obter apenas o *dissambly* anotado.

```
> perf annotate --stdio --dsos=naive
--symbol="multiply_matrices(float**, float**, float**, int)" --no-source
```

```
         :      Disassembly of section .text:
         :
         :      00000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
    0.00 :        4007f0:       test   %ecx,%ecx
    0.00 :        4007f2:       jle    40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
    0.00 :        4007f4:       lea    -0x1(%rcx),%eax
    0.00 :        4007f7:       pxor   %xmm2,%xmm2
    0.00 :        4007fb:       push   %rbp
    0.00 :        4007fc:       mov    %rdx,%rbp
    0.00 :        4007ff:       push   %rbx
    0.00 :        400800:       lea    0x4(,%rax,4),%rdx
    0.00 :        400808:       mov    %rdi,%rbx
    0.00 :        40080b:       xor    %edi,%edi
    0.00 :        40080d:       mov    (%rbx,%rdi,8),%r10
    0.00 :        400811:       mov    0x0(%rbp,%rdi,8),%r11
    0.00 :        400816:       xor    %r9d,%r9d
    0.00 :        400819:       nopl   0x0(%rax)
    0.01 :        400820:       movaps %xmm2,%xmm1
    0.00 :        400823:       xor    %eax,%eax
    0.00 :        400825:       nopl   (%rax)
    5.11 :        400828:       mov    (%rsi,%rax,8),%r8
    0.14 :        40082c:       movss  (%r8,%r9,1),%xmm0
   68.09 :        400832:       mulss  (%r10,%rax,4),%xmm0
   16.42 :        400838:       add    $0x1,%rax
    0.01 :        40083c:       cmp    %eax,%ecx
    0.00 :        40083e:       addss  %xmm0,%xmm1
   10.23 :        400842:       jg     400828 <multiply_matrices(float**, float**, float**, int)+0x38>
    0.00 :        400844:       movss  %xmm1,(%r11,%r9,1)
    0.01 :        40084a:       add    $0x4,%r9
    0.00 :        40084e:       cmp    %rdx,%r9
    0.00 :        400851:       jne    400820 <multiply_matrices(float**, float**, float**, int)+0x30>
    0.00 :        400853:       add    $0x1,%rdi
    0.00 :        400857:       cmp    %edi,%ecx
    0.00 :        400859:       jg     40080d <multiply_matrices(float**, float**, float**, int)+0x1d>
    0.00 :        40085b:       pop    %rbx
    0.00 :        40085c:       pop    %rbp
    0.00 :        40085d:       repz retq
```

Figura 3.16: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```
         :      Disassembly of section .text:
         :
         :      00000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
    0.00 :        4007f0:       test   %ecx,%ecx
    0.00 :        4007f2:       jle    40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
    0.00 :        4007f4:       lea    -0x1(%rcx),%eax
    0.00 :        4007f7:       pxor   %xmm2,%xmm2
    0.00 :        4007fb:       push   %rbp
    0.00 :        4007fc:       mov    %rdx,%rbp
    0.00 :        4007ff:       push   %rbx
    0.00 :        400800:       lea    0x4(,%rax,4),%rdx
    0.00 :        400808:       mov    %rdi,%rbx
    0.00 :        40080b:       xor    %edi,%edi
    0.00 :        40080d:       mov    (%rbx,%rdi,8),%r10
    0.00 :        400811:       mov    0x0(%rbp,%rdi,8),%r11
    0.00 :        400816:       xor    %r9d,%r9d
    0.00 :        400819:       nopl   0x0(%rax)
    0.08 :        400820:       movaps %xmm2,%xmm1
    0.00 :        400823:       xor    %eax,%eax
    0.00 :        400825:       nopl   (%rax)
   13.78 :        400828:       mov    (%rsi,%rax,8),%r8
    0.33 :        40082c:       movss  (%r8,%r9,1),%xmm0
   30.27 :        400832:       mulss  (%r10,%rax,4),%xmm0
   32.49 :        400838:       add    $0x1,%rax
    0.00 :        40083c:       cmp    %eax,%ecx
    0.00 :        40083e:       addss  %xmm0,%xmm1
   22.97 :        400842:       jg     400828 <multiply_matrices(float**, float**, float**, int)+0x38>
    0.00 :        400844:       movss  %xmm1,(%r11,%r9,1)
    0.08 :        40084a:       add    $0x4,%r9
    0.00 :        40084e:       cmp    %rdx,%r9
    0.00 :        400851:       jne    400820 <multiply_matrices(float**, float**, float**, int)+0x30>
    0.00 :        400853:       add    $0x1,%rdi
    0.00 :        400857:       cmp    %edi,%ecx
    0.00 :        400859:       jg     40080d <multiply_matrices(float**, float**, float**, int)+0x1d>
:
```

Figura 3.17: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.

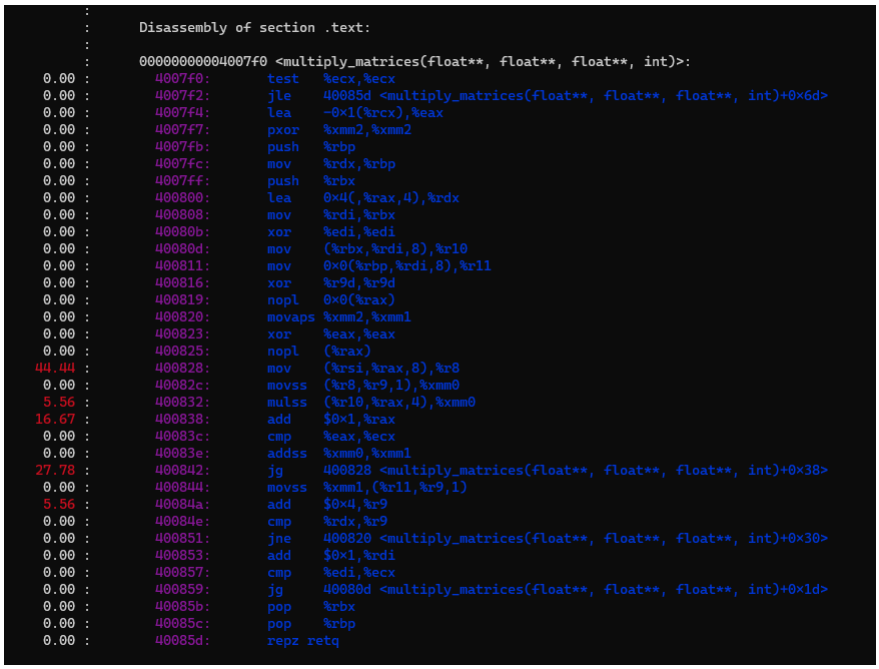


Figura 3.18: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

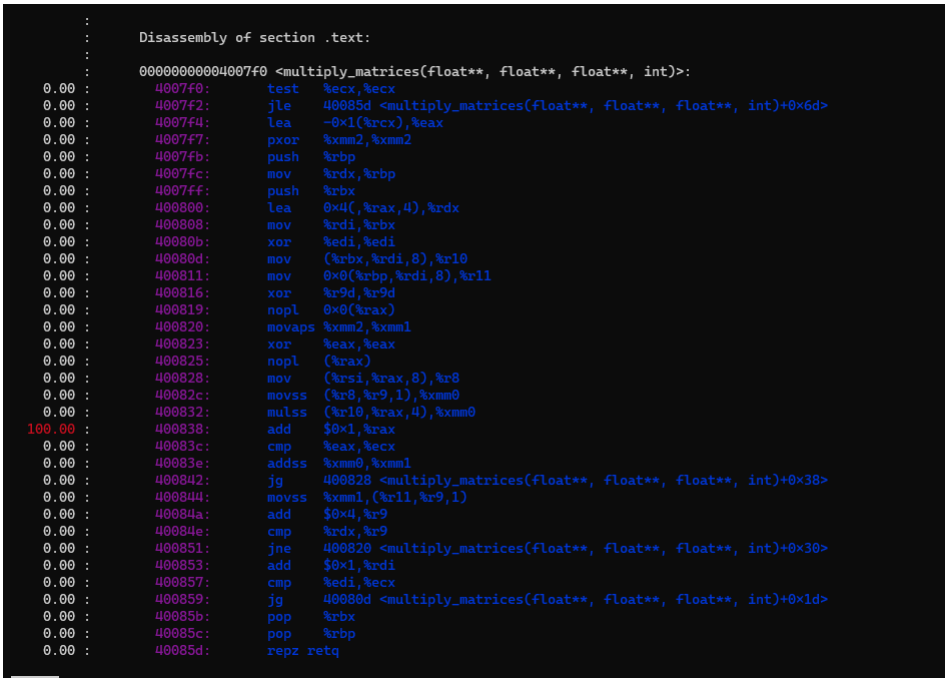


Figura 3.19: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Como seria de esperar os resultados obtidos foram semelhantes aos obtidos anteriormente.

3.7 Incrementar a frequência das amostras

Dado que as medições do *Perf* são realizadas estatisticamente medi o número de amostras obtido para cada um dos eventos $clock - rate$ e $faults$ para vários de tamanhos para poder visualizar o impacto resultante do aumento da frequência no número de amostras recolhidas. Utilizei o comando abaixo para obter a frequência inicial.

```
>  perf evlist -F
```

Inicialmente a frequência era 4000.Com o comando seguinte medi o numero de amostras utilizando as frequências de 8000 e 4000.

```
> perf record -e cpu-clock --freq=freq ./bin/naive 1 size
```

Sample frequency	**clock-rate**	32 **faults**
4000	115471	12663
8000	242611	12654

Tabela 3.4: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 2048x2048

Sample frequency	**clock-rate**	32 **faults**
4000	9952	3416
8000	22132	3409

Tabela 3.5: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 512x512

Sample frequency	**clock-rate**	32 **faults**
4000	36	469
8000	64	423

Tabela 3.6: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 128x128

Sample frequency	**clock-rate**	32 **faults**
4000	13	713
8000	25	313

Tabela 3.7: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 32x32

Os resultados obtidos mostram que para o evento $cpu - clock$ o número de amostras duplica com a duplicação do tamanho da frequência utilizada. Para o evento $faults$ verifica-se que os resultados são similares para os 3 maiores tamanhos tendo apenas alguma variação para o menor tamanho.

3.8 Eventos utilizados

Foi utilizado o comando abaixo a fim de verificar quais os eventos disponíveis no sistema de teste. A partir da análise dos eventos disponíveis verifiquei que a maioria dos eventos utilizados para os testes subsequentes estão disponíveis na máquina em questão.

```
> perf list
```

```
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                                [Hardware event]
  instructions                                        [Hardware event]
  cache-references                                    [Hardware event]
  cache-misses                                        [Hardware event]
  branch-instructions OR branches                     [Hardware event]
  branch-misses                                       [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend     [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend       [Hardware event]

  cpu-clock                                           [Software event]
  task-clock                                          [Software event]
  page-faults OR faults                               [Software event]
  context-switches OR cs                              [Software event]
  cpu-migrations OR migrations                        [Software event]
  minor-faults                                        [Software event]
  major-faults                                        [Software event]
  alignment-faults                                    [Software event]
  emulation-faults                                    [Software event]

  L1-dcache-loads                                     [Hardware cache event]
  L1-dcache-load-misses                               [Hardware cache event]
  L1-dcache-stores                                    [Hardware cache event]
  L1-dcache-store-misses                              [Hardware cache event]
  L1-dcache-prefetches                                [Hardware cache event]
  cpu-clock                                           [Software event]
  task-clock                                          [Software event]
  page-faults OR faults                               [Software event]
:
  LLC-load-misses                                     [Hardware cache event]
  LLC-stores                                          [Hardware cache event]
  LLC-store-misses                                    [Hardware cache event]
  LLC-prefetches                                      [Hardware cache event]
  LLC-prefetch-misses                                 [Hardware cache event]
  dTLB-loads                                          [Hardware cache event]
  dTLB-load-misses                                    [Hardware cache event]
  dTLB-stores                                         [Hardware cache event]
  dTLB-store-misses                                   [Hardware cache event]
  iTLB-loads                                          [Hardware cache event]
  iTLB-load-misses                                    [Hardware cache event]
  branch-loads                                        [Hardware cache event]
  branch-load-misses                                  [Hardware cache event]
```

Figura 3.20: Eventos disponíveis na maquina de teste

3.9 Analise das diferentes versões do algoritmo e tamanhos

Foram medidos para os vários tamanhos e para os dois algoritmos os valores de vários eventos e estes foram subsequentemente utilizados para calcular um conjunto de métricas.Para facilitar os testes e evitar que os vários contadores interfiram nos valores dos outros foi utilizado uma *shell script* que permita automatizar as medições e medir cada evento independentemente.

```
> perf record -e $metric ./bin/naive 1 2048
> perf report --stdio --show-nr-samples --dsos=naive | grep "Event"
```

Event	naive	interchange
cpu-cycles	153579835565	18872420661
cpu-clock	172034	18524
L1-dcache-load-misses	11676490203	553111515
L1-dcache-loads	25998720494	6631326934
L1-dcache-store-misses	7039985	1391208
instructions	60876847831	22086792694
cache-misses	439586633	198526441
branch-misses	4542740	4312800
cpu-migrations	-	-
branches	8768057460	2311156792
L1-dcache-loads	25997343784	6631155117
L1-dcache-load-misses	11569448154	553059738
L1-dcache-stores	124533957	2246988346
L1-dcache-store-misses	6739880	1387443
L1-icache-loads	492611767	491902057
LLC-loads	1769729847	103888720
LLC-load-misses	441026564	91188369
LLC-store-misses	1124507	871207
dTLB-load-misses	3211089527	13032985
iTLB-load-misses	6492	2513
branch-loads	8768875350	2309001206
branch-load-misses	212944277	192790716

Tabela 3.8: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 2048x2048

Metric	naive	interchange
Instructions per cycle	1.035	2.07
L1 cache miss ratio	1074	980
L1 cache miss RATE PTI	9.2	1.09
Data TLB miss rate PTI	9.2	1.09
Branch mispredicted ratio	0.002	0.006
Branch mispredict rate PTI	0.303	0.76

Tabela 3.9: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 2048x2048

Event	naive	interchange
cpu-cycles	946050773	182284638
cpu-clock	1313	267
L1-dcache-load-misses	9187142	221568
L1-dcache-loads	414816403	113304751
L1-dcache-store-misses	385986	115571
instructions	979881005	377382887
cache-misses	77272	73946
branch-misses	297242	288866
cpu-migrations	-	-
branches	144118235	44374194
L1-dcache-loads	414672586	113348901
L1-dcache-load-misses	161378094	9193286
L1-dcache-stores	6969231	40054004
L1-dcache-store-misses	386466	117283
L1-icache-loads	27057094	27434975
LLC-loads	8790308	1804307
LLC-load-misses	13608	13006
LLC-store-misses	56781	52756
dTLB-load-misses	9015504	414743
iTLB-load-misses	1253	1425
branch-loads	144090002	44342765
branch-load-misses	11276553	11293725

Tabela 3.10: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 512x512

Metric	naive	interchange
Instructions per cycle	0.39	1.17
L1 cache miss ratio	2.25	11.99
L1 cache miss RATE PTI	190	25
Data TLB miss rate PTI	52.7	0.59
Branch mispredicted ratio	0.00051	0.086
Branch mispredict rate PTI	0.07	0.20

Tabela 3.11: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 2048x2048

Event	naive	interchange
cpu-cycles	12772194	8018015
cpu-clock	30	17
L1-dcache-load-misses	221568	235263
L1-dcache-loads	8037070	3538381
L1-dcache-store-misses	1184906	1083633
instructions	19396	17889
cache-misses	22026	18639
branch-misses	40373	25559
cpu-migrations	-	-
branches	3559816	2054391
L1-dcache-loads	8165605	3352742
L1-dcache-load-misses	224487	233555
L1-dcache-loads	242611	12654
L1-dcache-stores	824538	1298700
L1-dcache-store-misses	19087	17995
L1-icache-loads	3404898	4275438
LLC-loads	27576	24772
LLC-load-misses	10052	9254
LLC-store-misses	9196	9481
dTLB-load-misses	12436	11190
iTLB-load-misses	1910	1257
branch-loads	3479978	2040020
branch-load-misses	1290122	1474193

Tabela 3.12: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 128x128

Metric	naive	interchange
Instructions per cycle	0.0015	0.0022
L1 cache miss ratio	36.27	15.04
L1 cache miss RATE PTI	414367.4	197796.47
Data TLB miss rate PTI	641.16	625.52
Branch mispredicted ratio	0.011	0.012
Branch mispredict rate PTI	2081.5	1428

Tabela 3.13: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 128x128

Event	naive	interchange
cpu-cycles	5186860	4707675
cpu-clock	10	9
L1-dcache-load-misses	64811	68625
L1-dcache-loads	1184906	1083633
L1-dcache-store-misses	9973	10821
instructions	4991671	4656035
cache-misses	18196	16385
branch-misses	25175	24280
cpu-migrations	-	-
branches	923332	881412
L1-dcache-loads	1253916	1150298
L1-dcache-load-misses	66573	68785
L1-dcache-stores	424239	436621
L1-dcache-store-misses	11366	11349
L1-icache-loads	1988026	2019354
LLC-loads	26817	23894
LLC-load-misses	8581	8700
LLC-store-misses	4663	4433
dTLB-load-misses	10223	8816
iTLB-load-misses	1429	1213
branch-loads	914612	899932
branch-load-misses	674893	659745

Tabela 3.14: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive	interchange
Instructions per cycle	499167.1	517337.22
L1 cache miss ratio	18.28	15.79
L1 cache miss RATE PTI	237.4	232.74
Data TLB miss rate PTI	2.048	1.89
Branch mispredicted ratio	0.02	1
Branch mispredict rate PTI	5.04	5.2

Tabela 3.15: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 32x32

Em termos de instruções por ciclo podemos verificar que para os vários tamanhos a performance da versão *interchange* é superior. Em termos de *miss rates* para o maior tamanho encontrei o *cache miss ratio* é inferior para a versão *interchange*,para o segundo maior tamanho verifica-se o oposto ,no entanto para ambas estas versões ambos os *L1 cache miss RATE PTI* e *Data TLB miss rate PTI* são consideravelmente inferiores para a versão *interchange* isto justifica o maior numero de instruções por ciclo desta versão por perder menos tempo em acessos a memoria. Os resultados para os dois menores tamanhos foram inconclusivos e poderão ser atributivos a falta de dados.

3.10 Comparação entre *Counting mode* e *Sampling*

O *perf* suporta a realização das medições dos vários eventos utilizando *sampling*, esta técnica de medição consiste em recolher várias amostras durante a execução do programa e posteriormente

juntá-las de modo a obter informação necessária à sua avaliação. Foram comparados os resultados das medições realizadas com os dois modos para os dois tamanhos maiores e foram também calculadas métricas utilizando esses resultados. As métricas utilizando o *Sampling mode* foram recolhidas da mesma forma do que as métricas anteriores utilizando *shell scripts* e o comando abaixo. O período utilizado foi 100000, este período foi utilizado por ser o mesmo do que o utilizado no tutorial.

```
> perf record -e metric -c period ./bin/naive 1 size
> perf report --stdio --show-nr-samples --dsos=naive | grep "Event"
```

Event	**naive 2048x2048**	**interchange 2048x2048**	**naive 512x512**	**interchange 512x512**
cpu-cycles	142389883755	13261216763	957819437	184828719
cpu-clock	158948	17344	1277	260
cache-references	1925133684	103747696	8977771	1852816
cache-misses	431869403	85727780	49341	29002
instructions	61058300000	22171400000	982800000	378200000
LLC-loads	1988954894	101944331	8748003	1818427
LLC-load-misses	432113925	83135408	11994	9720
dTLB-load-misses	2328224247	12988596	9015770	412392
branches	8763879676	2309228272	144096534	44367590
branch-misses	4519347	4293829	288968	288711

Tabela 3.16: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Event	**naive 2048x2048**	**interchange 2048x2048**	**naive 512x512**	**interchange 512x512**
cpu-cycles	145812500000	16049600000	953000000	183900000
cpu-clock	42867400000	4434900000	321800000	69600000
cache-references	1531700000	191000000	8900000	1800000
cache-misses	438200000	81200000	-	-
instructions	60861147730	22086293233	980082628	377841466
LLC-loads	2144100000	97500000	8700000	1700000
LLC-load-misses	437700000	80900000	-	-
dTLB-load-misses	2421900000	12800000	9000000	400000
branches	8756400000	2309400000	143900000	44100000
branch-misses	4400000	4200000	200000	200000

Tabela 3.17: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
Instructions per cycle	0.429	1.672	1.026	2.046
Cache miss ratio	4.458	1.21	181.954	63.886
Cache miss rate PTI	31.529	4.679	9.135	4.899
LLC load miss ratio	4.603	1.226	729.365	187.081
LLC load miss rate PTI	7.077	3.75	0.012	0.026
dTLB load miss rate PTI	38.131	0.586	9.174	1.09
Branch mispredict ratio	0.0015	0.0025	0.0013	0.005
Branch mispredict rate PTI	0.076	0.22	0.210	0.550

Tabela 3.18: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
Instructions per cycle	0.417	1.376	1.028	2.055
Cache miss ratio	3.495	2.352	-	-
Cache miss rate PTI	25.167	8.648	9.081	4.764
LLC load miss ratio	4.899	1.205	-	-
LLC load miss rate PTI	7.192	3.663	-	-
dTLB load miss rate PTI	39.794	0.58	9.183	1.059
Branch mispredict ratio	0.001	0.002	0.001	0.005
Branch mispredict rate PTI	0.072	0.19	0.204	0.529

Tabela 3.19: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Com a exceção das medições relacionadas com o numero de *cache misses* que não foram obtidas para o tamanho $512x512$,a não existência dessas medições poderá ser atribuída ao tamanho insuficiente do tempo de amostra, as medições apresentam-se similares entre os dois métodos utilizados.

3.11 Utilização de *FlameGraphs* para analise dos algoritmos

Foram também utilizados *FlameGraphs* para avaliar as chamadas ao sistema dos vários algoritmos para os dois maior tamanhos.

```
> perf record -F 99 -ag ./bin/naive version size
> perf script | ./FlameGraph/stackcollapse-perf.pl
| ./FlameGraph/flamegraph.pl > tests/name.svg
```

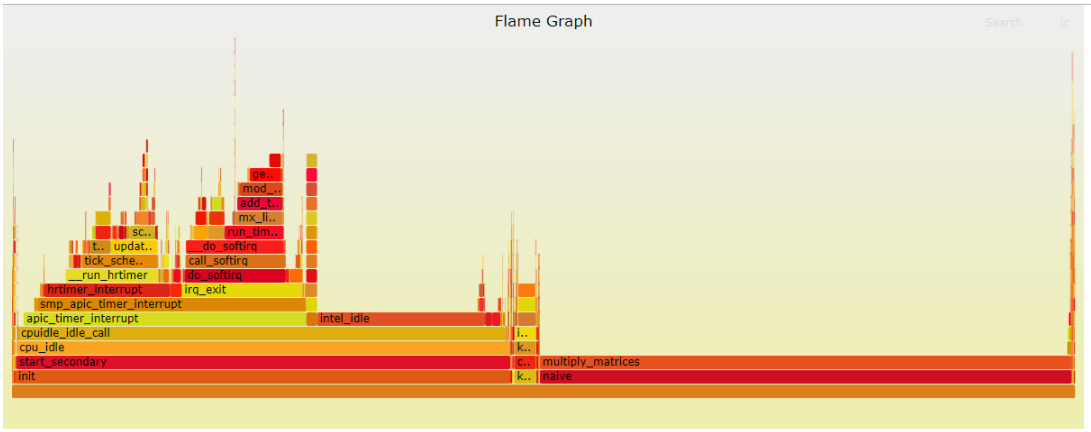


Figura 3.21: *FlameGraph* para o algoritmo *naive* e matrizes de tamanho 2048x2048.

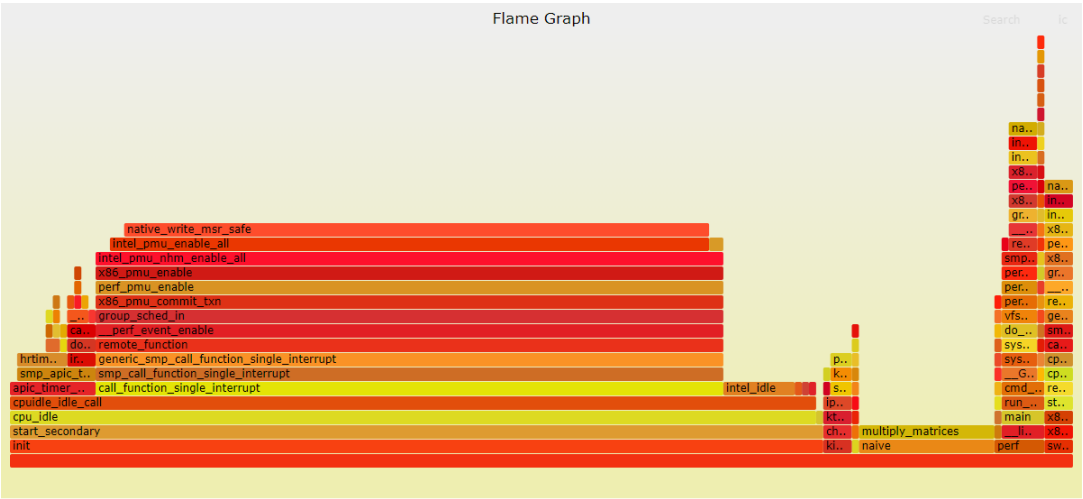


Figura 3.22: *FlameGraph* para o algoritmo *naive* e matrizes de tamanho 512x512.

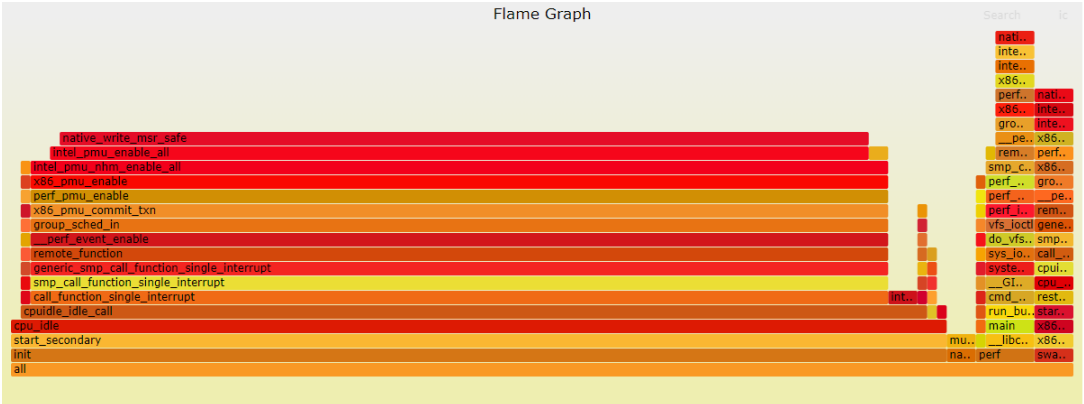


Figura 3.23: *FlameGraph* para o algoritmo *interchange* e matrizes de tamanho 2048x2048.

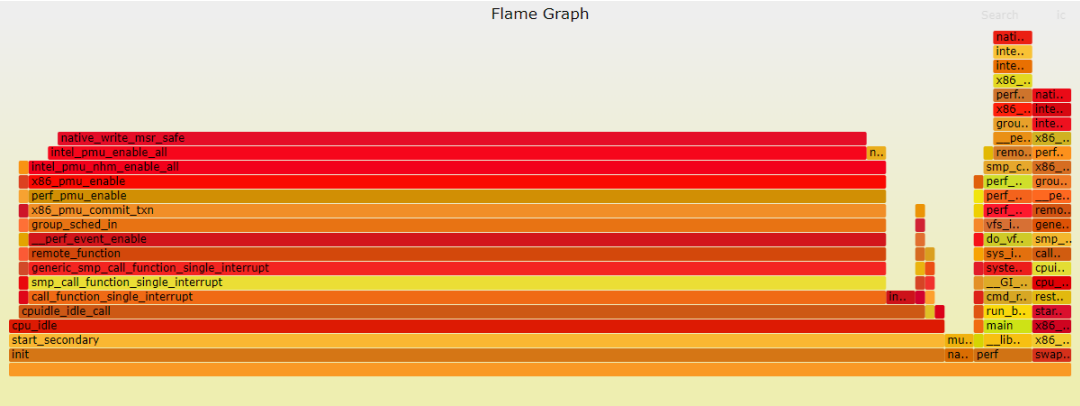


Figura 3.24: *FlameGraph* para o algoritmo *interchange* e matrizes de tamanho 512x512.

Os *FlameGraphs* permitem verificar para o tamanho maior que a versão *naive* ocupa uma percentagem maior do tempo no seu nível da *stack* relativamente a versão *interchange* isto acontece porque esta função é menos eficiente.

Capítulo 4

Conclusões e trabalho futuro

Este trabalho permitiu ganhar familiaridade com as ferramentas *Perf* e *FlameGraph*,permitiu também estudar diferentes algoritmos e perceber algumas das motivações por detrás das diferenças de performance.

Em termos de algoritmos de ordenação pude verificar que o algoritmo *radix-sort LSD* apresenta melhor desempenho relativamente aos outros e que estas diferenças assentam principalmente no menor numero de *branches* e *branch-misses* mas também pela sua complexidade e comportamento no que toca a chamadas recursivas. Encontrei também alguns *bottlenecks* no desempenho da aplicação este encontra-se principalmente no acesso aos dados.

No que toca aos algoritmos de multiplicação de matrizes o trabalho realizado permitiu visualizar diferenças em ambos os algoritmos, sendo que estas assentam principalmente em acessos a memoria, permitiu também visualizar os pontos quentes da aplicação.

Em termos de trabalho futuro, as ferramentas *Perf* e *FlameGraphs* poderão ser utilizadas em vários outros programas para medir, estudar e melhor o desempenho de diversas aplicações.

Bibliografia

[1] Paul J. Drongowski. *PERF tutorial: Counting hardware performance events*. 2015. URL: `http://sandsoftwaresound.net/perf/perf-tut-count-hw-events/`.

[2] Paul J. Drongowski. *PERF tutorial: Finding execution hot spots*. 2015. URL: `http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/`.

[3] Paul J. Drongowski. *PERF tutorial: Profiling hardware events*. 2015. URL: `http://sandsoftwaresound.net/perf/perf-tut-profile-hw-events/`.

[4] Brendan Gregg. *Cpuu FlameGraphs*. 2015. URL: `http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html`.