Introdução ao PERF
Determinação de Hotspots de execução de kernels recorrendo a PERF (linux-tools)

Filipe Oliveira
Departamento de Informática
Universidade do Minho
Email: a57816@alunos.uminho.pt

1. Introdução – Contextualização da utilização da ferramenta PERF

O presente trabalho prático surge como um ”diário de bordo”relativo ao acompanhamento de um conjunto de 3 parte do tutorial: **”PERF tutorial: Finding execution hot spots”**.[1]

A parte 1 descreve a utilização da ferramenta PERF como meio de identificação e análise de hotspots de execução em kernels. Cobre maioritariamente os comandos PERF mais básico, e respectivas opções e eventos de medição de performance relativa.

A parte 2 introduz eventos que permitem a medição de contadores de hardware e demonstra a utilização do PERF nesse sentido. Discute ainda a importância de certas métricas de performance e a sua influência relativa na performance geral de uma aplicação.

A parte 3 recorre aos eventos de contadores de hardware para, uma vez mais e desta vez de uma forma mais aprofundada, analisar os hotspots de um kernel/aplicação.

Todo o tutorial assenta em duas versões de uma aplicação de multiplicação de matrizes:

- **Naive:** Algoritmo de multiplicação de matrizes com acessos a memória ”column-wise”.
- **Interchange:** Algoritmo de multiplicação de matrizes com acessos a memória ”row-wise”.

A primeira versão serão, com seria de esperar, apresentará problemas de performance relacionados com o acesso à memória. É importante também realçar que as aplicações são ambas compiladas com a versão de compilador GCC 4.9.0., com as seguintes flags de compilação -O2 -ggdb -g”. Denote que a flag de otimização -O3, que activa um conjunto de otimizações de código[2] está desativada, muito pela necessidade de manter o código (apesar de pouco eficiente) legível e com capacidade de ser compreendido pelo programador.

1. http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/

2. -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize, -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone options

2. Caracterização do Hardware do ambiente de testes

Tão importante com especificar as versões de código e as propriedades é especificar o ambiente de teste nos quais pretendemos realizar as benchmarks.

Através da análise do hardware disponível no Search6 [3], uma das nossas plataformas de teste, foi seleccionado o nó do tipo compute-431, sendo a disponibilidade global do mesmo e o correcto funcionamento do perf os principais factores. Na tabela 1 encontram-se especificadas as principais características dos sistemas em teste.

Tabela 1: Características de Hardware do nó 431

Sistema	compute-431
# CPUs	2
CPU	Intel® Xeon® X5650
Arquitectura de Processador	Nehalem
# Cores por CPU	6
# Threads por CPU	12
Freq. Clock	2.66 GHz
Cache L1	192KB (32KB por Core)
Cache L2	1536KB (256KB por Core)
Cache L3	12288KB (partilhada)
Ext. Inst. Set	SSE4.2
#Memory Channels	3
Memória Ram Disponível	48GB
Peak Memory BW Fab. CPU	32 GB/s

Tal como referido iremos recorrer à ferramenta PERF por formar a completarmos o conjunto de tutoriais disponibilizados, sendo a versão utilizada a mais recente disponível no Search6 – versão 4.0.0.

3. Determinação do tamanho dos datasets

Talvez dos pontos mais importantes presente na tabela de caracterização de hardware seja o tamanho da Cache L3, dado que ambos os algoritmos dependem massivamente de leitura/escrita na memória principal. Assim sendo, se pretendemos realçar as penalizações resultantes de diferentes tipos de acesso aos dados devemos realizar dois tipos distintos de testes:

3. Services and Advanced Research Computing with HTC/HPC clusters

- Matrizes contidas na LLC:
  Sabemos que as 3 matrizes terão de ter uma tamanho de coluna/linha inferior a:

$$lado\ matriz \leq \frac{\sqrt{(\frac{12288KB}{4\ Bytes\ p/float}}}{3\ matrizes} \leq 1011$$

  para que os datasets estejam completamente contidos em cache. Foi escolhido o lado da matriz de 512 elementos.
- Matrizes contidas na memória principal:
  Sabemos que as 3 matrizes terão de ter uma tamanho de coluna/linha superior a 1011 elemento por linha/coluna. Foi escolhido o lado da matriz de 2048 elementos.

Doravante, todos os comandos apresentados terão por standard o dataset menor (matrizes de 512*512), e o kernel naive. Em caso de serem utilizados outro kernel ou dataset será feita a referência antes da apresentação dos resultados.

4. Parte 1

4.1. Passo 0 – uma análise aos software e hardware events disponíveis na máquina

Recorrendo ao seguinte commando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf list | wc -l
```

Podemos ter uma noção em ternos de eventos de hardware e software disponíveis na máquina de teste:

```
784
```

4.2. 1º Passo – o encontrar dos tempos base sem overhead de medição para o menor dataset (dataset em LLCACHE)

Por forma a podermos estabelecer um limite dentro do aceitável das possíveis alterações à performance das versões do kernel necessitamos de primeiramente ter uma medição com o mínimo de overhead possível, para o menor dataset em profiling (matriz 512*512), para o caso **naive**. Recorrendo ao seguinte comando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf stat -e cpu-clock ./naive
```

```
 Performance counter stats for './naive':

        201.696023      cpu-clock (msec)

       0.206153405 seconds time elapsed
```

Teremos uma base de referência futura, para podermos validar as nossas medições com overhead.

Para além do tempo de execução é ainda demonstrado o número de cpu-clocks necessários para a execução.

Como seria de esperar, é possível num única medição registar vários eventos. Recorrendo ao seguinte comando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf stat -e cpu-clock,faults ./naive
```

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf stat -e cpu-clock,faults ./naive

 Performance counter stats for './naive':

        196.395754      cpu-clock (msec)
               844      faults

       0.200874285 seconds time elapsed
```

4.3. O processo de procura de hotpsots de uma aplicação

O processo de procura de hotpsots de uma aplicação pode ser traduzido em 2 passos simples:

- Passo 1: Recolha de dados para profiling da aplicação
- Passo 2 : Tratamento e apresentação da informação recolhida.

4.3.1. Passo 1. Podemos recolher informação de profiling simplesmente correndo o seguinte comando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf record -e cpu-clock,faults ./naive
```

Neste caso específico o PERF corre a aplicação naive e recolhe dados relativos aos eventos: cpu-clock e page-faults.

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf record -e cpu-clock,faults ./naive
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.046 MB perf.data (830 samples) ]
```

4.3.2. Passo 2. Relativamente ao passo 2 - Tratamento e apresentação da informação recolhida - o seguinte comando permite visualizar a informação presente no ficheiro prof.data (ficheiro default no qual é gravada a informação):

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio
```

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 808  of event 'cpu-clock'
# Event count (approx.): 808
#
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ..............................
#
    94.68%  naive    naive              [.] multiply_matrices
     2.10%  naive    naive              [.] initialize_matrices
     1.36%  naive    libc-2.12.so       [.] __random
     0.87%  naive    libc-2.12.so       [.] __random_r
     0.25%  naive    [kernel.kallsyms]  [k] __mem_cgroup_commit_charge
```

```
     0.25%  naive    naive              [.] rand@plt
     0.12%  naive    [kernel.kallsyms]  [k] __call_rcu
     0.12%  naive    [kernel.kallsyms]  [k] ↩
         __mem_cgroup_uncharge_common
     0.12%  naive    libc-2.12.so       [.] malloc
     0.12%  naive    libc-2.12.so       [.] rand


# Samples: 22  of event 'faults'
# Event count (approx.): 1157
#
# Overhead  Command  Shared Object       Symbol
# ........  .......  ..................  ↩
    ......................
#
    63.61%  naive    naive              [.] ↩
        initialize_matrices
    29.56%  naive    libc-2.12.so       [.] ↩
        __init_cpu_features
     5.62%  naive    ld-2.12.so         [.] dl_main
     0.52%  naive    ld-2.12.so         [.] ↩
        _dl_setup_hash
     0.26%  naive    ld-2.12.so         [.] _dl_start
     0.17%  naive    [kernel.kallsyms]  [k] ↩
        __clear_user
     0.17%  naive    libc-2.12.so       [.] ↩
        __strstr_sse2
     0.09%  naive    ld-2.12.so         [.] _start
```

Existem outras alternativas a esse mesmo comando sendo esta:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --↩
    tui
```

que apresenta o Terminal-based User Interface (TUI).

Voltemos ao output apresentado pelo comando com a opção stdio. Denote no seguinte detalhe:

```
Samples: 808  of event 'cpu-clock'
Event count (approx.): 808
```

Como poderá confirmar foram registadas 808 medições do evento ”cpu-clock”.

Adicionando a opção - -sort conseguimos uma visualização agregada. Em conjunção com a opção **comm** e **dso** obtemos o seguinte resultado.

```
[a57816@search6 ESC_FLAME_GRAPH]$ perf report --stdio ↩
    --sort comm,dso
```

```
[a57816@search6 ESC_FLAME_GRAPH]$ perf report --stdio ↩
    --sort comm,dso
[kernel.kallsyms] with build id 09↩
    b53ce5dfe09c7b1ad9473e1356d7d922512e27 not found, ↩
    continuing without symbols
# To display the perf.data header info, please use --↩
    header/--header-only options.
#
# Samples: 808  of event 'cpu-clock'
# Event count (approx.): 808
#
# Overhead  Command  Shared Object
# ........  .......  ..................
#
    97.03%  naive    naive
     2.48%  naive    libc-2.12.so
     0.50%  naive    [kernel.kallsyms]


# Samples: 22  of event 'faults'
# Event count (approx.): 1157
#
# Overhead  Command  Shared Object
# ........  .......  ..................
#
    63.61%  naive    naive
    29.73%  naive    libc-2.12.so
     6.48%  naive    ld-2.12.so
     0.17%  naive    [kernel.kallsyms]
```

Como pode confirmar pelo output, as duas opções no comando permitem agregar os resultados por comando e shared object.

Por forma a simplificar o processo de catalogação e posterior tratamento de dados é possível aceder a metadados das medições, através da junção da opção **–header** ao comando anterior. Desta forma temos acesso a por exemplo, os dados de hardware do ambiente de teste, o comando perf utilizado e informação relativa aos eventos medidos. Esta funcionalidade torna-se bastante útil dada a tremenda facilidade que temos hoje em dia de gerar dados de teste. A consequência da facilidade de tal geração é a posterior necessidade de tratamento correcto dos mesmos.

```
[a57816@search6 ESC_FLAME_GRAPH]$ perf report --stdio ↩
    --sort comm,dso --header
```

```
[a57816@search6 ESC_FLAME_GRAPH]$ perf report --stdio ↩
    --sort comm,dso --header
[kernel.kallsyms] with build id 09↩
    b53ce5dfe09c7b1ad9473e1356d7d922512e27 not found, ↩
    continuing without symbols
# ========
# captured on: Mon May 23 23:00:56 2016
# hostname : compute-431-1.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 12319324 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-↩
    clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, ↩
    config1 = 0x0, config2 = 0x0, excl_usr = 0, ↩
    excl_kern = 0, excl_host = 0, excl_guest = 1, ↩
    precise_ip = 0, attr_mmap2 = 0, attr
# event : name = faults, type = 1, config = 0x2, ↩
    config1 = 0x0, config2 = 0x0, excl_usr = 0, ↩
    excl_kern = 0, excl_host = 0, excl_guest = 1, ↩
    precise_ip = 0, attr_mmap2 = 0, attr_mm
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to ↩
    display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# ========
#
# Samples: 808  of event 'cpu-clock'
# Event count (approx.): 808
#
# Overhead  Command  Shared Object
# ........  .......  ..................
#
    97.03%  naive    naive
     2.48%  naive    libc-2.12.so
     0.50%  naive    [kernel.kallsyms]


# Samples: 22  of event 'faults'
# Event count (approx.): 1157
#
# Overhead  Command  Shared Object
# ........  .......  ..................
#
    63.61%  naive    naive
    29.73%  naive    libc-2.12.so
```

```
     6.48%  naive    ld-2.12.so
     0.17%  naive    [kernel.kallsyms]
```

4.3.3. Aumentar a granularidade do profilling – fase I. O exemplo dado anteriormente apresenta os dados agregados relativos a toda a aplicação. Ora, daí podemos observar que **97.03%** do tempo de execução é despendido em métodos da própria aplicação, enquanto que os restantes **2.48%** são despendidos na biblioteca do sistema **libc-2.12.so**.

O próximo passo será portanto aumentar a granularidade do profiling nestes dois "shared objects".

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio --dsos=naive,libc-2.12.so
```

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio --dsos=naive,libc-2.12.so
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 808  of event 'cpu-clock'
# Event count (approx.): 808
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......................
#
    94.68%  naive    naive          [.] multiply_matrices
     2.10%  naive    naive          [.] initialize_matrices
     1.36%  naive    libc-2.12.so   [.] __random
     0.87%  naive    libc-2.12.so   [.] __random_r
     0.25%  naive    naive          [.] rand@plt
     0.12%  naive    libc-2.12.so   [.] malloc
     0.12%  naive    libc-2.12.so   [.] rand


# Samples: 22  of event 'faults'
# Event count (approx.): 1157
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ......................
#
    63.61%  naive    naive          [.] initialize_matrices
    29.56%  naive    libc-2.12.so   [.] __init_cpu_features
     0.17%  naive    libc-2.12.so   [.] __strstr_sse2
```

O método **multiply_matrices()** é o responsável pela maioria do tempo de computação, sendo aproximadamente 94.86%.

A segunda função, em termos de tempo de computação é a **initialize_matrices** com 2.10%.

Podemos ainda analisar as **page faults** da aplicação, sendo que a maioria ocorre no método **initialize_matrices** – 63.61%, como seria de esperar. Este comportamento é o esperado, e não é responsável por qualquer problema de performance da aplicação, dado que esse mesmo método apenas ocupa 2.10% do tempo total da aplicação, não afectando o processo de multiplicação de matrizes, que é o responsável pela maior porção do mesmo.

Podemos ainda "validar"o número de page faults ocorrido, associando-o ao tamanho da matriz em uso. Ora, 512*512 floats representa um total de $512*512*(4\ bytes) = 1024KB$, multiplicado por 3 (3 matrizes), representa um total de 3072 KB, que representa um total de páginas $\frac{3072KB}{4KB\ por\ pag.} = 768$. Ora, 63.61% do total de **page faults** contabilizadas representa $0.6361 * 1157 = 735,9677 = 736\ pags.$, como espectado.

4.3.4. Aumentar a granularidade do profilling – fase II. Agora que temos a completa percepção de qual o método que está a consumir maioritariamente o tempo de execução, podemos realizar um análise não por biblioteca, mas por método, associando então ao método **multiply_matrices()** o respectivo código máquina e posição relativa ao início do método, recorrendo ao comando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf annotate --stdio --dsos=naive --symbol=multiply_matrices
```

Produzindo o seguinte output:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf annotate --stdio --dsos=naive --symbol=multiply_matrices

Percent | Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------------------------------------

         :
         :
         :
         : Disassembly of section .text:
         :
         : 0000000000400810 <multiply_matrices>:
         : multiply_matrices():
         :                 }
         :         }
         : }
         :
         : void multiply_matrices()
         : {
    0.00 :   400810:       pxor   %xmm2,%xmm2
    0.00 :   400814:       mov    $0x7e97c0,%edi
    0.00 :   400819:       mov    %rdi,%r8
    0.00 :   40081c:       xor    %esi,%esi
    0.00 :   40081e:       sub    $0x7e97c0,%r8
    0.00 :   400825:       nopl   (%rax)
    0.13 :   400828:       lea    0x6f5580(%rsi),%rax
    0.00 :   40082f:       lea    0x7e97c0(%rsi),%rcx
    0.00 :   400836:       mov    %rdi,%rdx
    0.00 :   400839:       movaps %xmm2,%xmm1
    0.00 :   40083c:       nopl   0x0(%rax)
         :
         :         for (i = 0 ; i < MSIZE ; i++) {
         :                 for (j = 0 ; j < MSIZE ; j++) {
         :                         float sum = 0.0 ;
         :                         for (k = 0 ; k < MSIZE ; k++) {
         :                                 sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
   24.97 :   400840:       movss  (%rdx),%xmm0
    0.00 :   400844:       add    $0x7d0,%rax
    0.00 :   40084a:       mulss  -0x7d0(%rax),%xmm0
   18.56 :   400852:       add    $0x4,%rdx
         :         int i, j, k ;
         :
         :         for (i = 0 ; i < MSIZE ; i++) {
         :                 for (j = 0 ; j < MSIZE ; j++) {
         :                         float sum = 0.0 ;
         :                         for (k = 0 ; k < MSIZE ; k++) {
   22.35 :   400856:       cmp    %rcx,%rax
         :                                 sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
    0.00 :   400859:       addss  %xmm0,%xmm1
         :         int i, j, k ;
         :
```

```
50          :          for (i = 0 ; i < MSIZE ; i++) {
51          :                  for (j = 0 ; j < MSIZE ; j++) {
52          :                          float sum = 0.0 ;
53          :                          for (k = 0 ; k < MSIZE ; k++) {
54    33.86 :   40085d:       jne    400840 <multiply_matrices+0x30>
55          :                                  sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
56          :                          }
57          :                          matrix_r[i][j] = sum;
58     0.00 :   40085f:       movss  %xmm1,0x601340(%r8,%rsi,1)
59     0.13 :   400869:       add    $0x4,%rsi
60          : void multiply_matrices()
61          : {
62          :          int i, j, k ;
63          :
64          :          for (i = 0 ; i < MSIZE ; i++) {
65          :                  for (j = 0 ; j < MSIZE ; j++) {
66     0.00 :   40086d:       cmp    $0x7d0,%rsi
67     0.00 :   400874:       jne    400828 <multiply_matrices+0x18>
68     0.00 :   400876:       add    $0x7d0,%rdi
69          :
70          : void multiply_matrices()
71          : {
72          :          int i, j, k ;
73          :
74          :          for (i = 0 ; i < MSIZE ; i++) {
75     0.00 :   40087d:       cmp    $0x8dda00,%rdi
76     0.00 :   400884:       jne    400819 <multiply_matrices+0x9>
77     0.00 :   400886:       repz retq
```

Caso pretendamos retirar o código na linguagem de alto nível, mantendo apenas a associação a código máquina com percentagem de tempo de execução, podemos recorrer ao seguinte comando:

```
Percent | Source code & Disassembly of naive for cpu-clock
```

```
1  Percent | Source code & Disassembly of naive for cpu-clock
2  ----------------------------------------------------------
3           :
4           :
5           :
6           : Disassembly of section .text:
7           :
8           : 0000000000400810 <multiply_matrices>:
9           : multiply_matrices():
10    0.00 :   400810:       pxor   %xmm2,%xmm2
11    0.00 :   400814:       mov    $0x7e97c0,%edi
12    0.00 :   400819:       mov    %rdi,%r8
13    0.00 :   40081c:       xor    %esi,%esi
14    0.00 :   40081e:       sub    $0x7e97c0,%r8
15    0.00 :   400825:       nopl   (%rax)
16    0.13 :   400828:       lea    0x6f5580(%rsi),%rax
17    0.00 :   40082f:       lea    0x7e97c0(%rsi),%rcx
18    0.00 :   400836:       mov    %rdi,%rdx
19    0.00 :   400839:       movaps %xmm2,%xmm1
20    0.00 :   40083c:       nopl   0x0(%rax)
21   24.97 :   400840:       movss  (%rdx),%xmm0
22    0.00 :   400844:       add    $0x7d0,%rax
23    0.00 :   40084a:       mulss  -0x7d0(%rax),%xmm0
24   18.56 :   400852:       add    $0x4,%rdx
25   22.35 :   400856:       cmp    %rcx,%rax
26    0.00 :   400859:       addss  %xmm0,%xmm1
27   33.86 :   40085d:       jne    400840 <multiply_matrices+0x30>
28    0.00 :   40085f:       movss  %xmm1,0x601340(%r8,%rsi,1)
29    0.13 :   400869:       add    $0x4,%rsi
30    0.00 :   40086d:       cmp    $0x7d0,%rsi
31    0.00 :   400874:       jne    400828 <multiply_matrices+0x18>
32    0.00 :   400876:       add    $0x7d0,%rdi
33    0.00 :   40087d:       cmp    $0x8dda00,%rdi
34    0.00 :   400884:       jne    400819 <multiply_matrices+0x9>
35    0.00 :   400886:       repz retq
```

Pela análise da execução dos dois comandos anteriores, temos então a resposta há muito aguardada. O nosso primeiro trabalho de detective dá-se por terminado com descoberta da "principal responsável":

```
sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
```

4.4. Método de profiling

Recolhidos os dados e descoberto o "culpado"resta-nos saber se o "modus operandi"da recursão ao comando PERF foi o mais indicado.

O PERF recorre a sampling estatístico para recolher os dados de profiling. Cada amostra é recolhida com base num tempo constante, tendo por base o **Linux time clock** para medir também a passagem do mesmo. Quando termina o intervalo de tempo constante é gerada uma interrupção e o PERF determina o trabalho que estava a ser realizado pelo CPU no momento da interrupção, registando por exemplo metadados do CPU, program counter, e software, produzindo aquilo que é chamado de um **sample**, sendo este escrito num buffer, posteriormente escrito no ficheiro **perf.data**.

Dado que nos iremos basear no sampling estatístico para inferir conclusões acerca do comportamento de um kernel necessitamos de ser cautelosos quanto ao número de samples a recolher. **Se por um lado o excesso de amostras levará a uma interferência nos dados a serem medidos, a falta delas levará à incerteza.** Poderemos estar a concluir erradamente que uma porção de código é a responsável pela degradação de performance devido à falta de amostras que nos apontem noutro sentido.

Uma forma de aumentar o número de amostras é especificando a frequência sobre a qual as mesmas serão recolhidas, através do seguinte comando:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf record -e cpu-clock --freq=8000 ./naive
```

```
1  [a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf record -e cpu-clock --freq=8000 ./naive
2  [ perf record: Woken up 1 times to write data ]
3  [ perf record: Captured and wrote 0.070 MB perf.data (1636 samples) ]
```

A execução do comando anterior resultou na recolha do dobro de amostras (1636 amostras vs as anteriores 830) em comparação com a medição efectuada em 4.3.1, devido também ao aumento em dobro da frequência – 8000 vs 4000, tal como pode ser confirmado pela execução do comando seguinte:

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio --show-nr-samples --dsos=naive
```

```
[a57816@compute-431-1 ESC_FLAME_GRAPH]$ perf report --stdio --show-nr-samples --dsos=naive
# To display the perf.data header info, please use --header/--header-only options.
#
# dso: naive
# Samples: 1K of event 'cpu-clock'
# Event count (approx.): 1636
#
# Overhead       Samples  Command  Symbol
# ........  ............  .......  ..........................
#
     94.74%          1550  naive    [.] multiply_matrices
      1.47%            24  naive    [.] initialize_matrices
      0.06%             1  naive    [.] rand@plt
```

Confirmamos portanto que se mantém o ponto crítico da nossa aplicação. 94.74% do tempo da mesma foi despendido no método **multiply_matrices**, tal como pretendíamos confirmar.

5. Parte 2

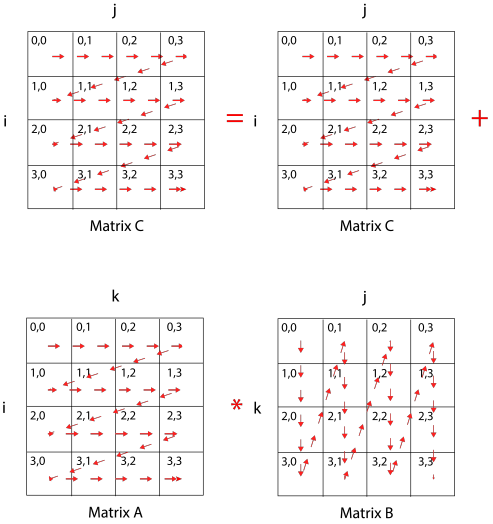
Encontrado o ponto alvo na "investigação"anterior, restanos resolver o problema. Atente nas seguintes transcrições de código, pertencentes nomeadamente à versão naive e interchange:

- Naive:

```
void multiply_matrices()
{
  int i, j, k ;

  // Textbook algorithm
  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      float sum = 0.0 ;
      for (k = 0 ; k < MSIZE ; k++) {
        sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      }
      matrix_r[i][j] = sum ;
    }
  }
}
```

Ilustração do acesso às posições da matriz via ordem IJK:

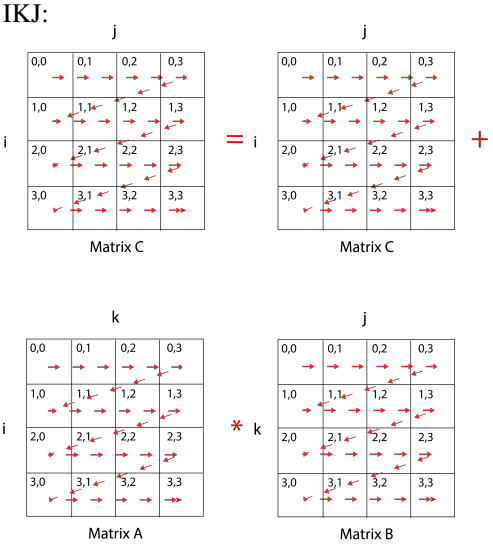


- Interchage:

```
void multiply_matrices()
{
  int i, j, k ;

  // Loop nest interchange algorithm
  for (i = 0 ; i < MSIZE ; i++) {
    for (k = 0 ; k < MSIZE ; k++) {
      for (j = 0 ; j < MSIZE ; j++) {
        matrix_r[i][j] = matrix_r[i][j] +
          (matrix_a[i][k] * matrix_b[k][j]) ;
      }
    }
  }
}
```

Ilustração do acesso às posições da matriz via ordem IKJ:



A ordem dos ciclos I-J-K na versão naive resulta na constante invalidação das linhas de cache que correspondam a dados da matriz B. Ora, este problema irá acentuar-se em caso de necessidade de acesso à memória principal, aumento a latência da recolha dos mesmos e provocando um atraso ainda mais visível quando comparada com a versão I-K-J.

A seguinte tabela resume vários contadores de performance para as duas versões do kernel – naive e interchange,

para o dataset menor em estudo:

Tabela 2: Performance events (naive vs. interchange) para o nó compute-431

# EVENT NAME	NAIVE	INTERCHANGE
cpu-cycles	535187277	399561216
instructions	1044692763	1152237507
cache-references	8196140	429971
cache-misses	36522	43034
branch-instructions	126101720	132065934
branch-misses	258384	249858
bus-cycles	0	0
L1-dcache-loads	246027409	253077242
L1-dcache-load-misses	56436199	7577858
L1-dcache-stores	9973628	128034804
L1-dcache-store-misses	322982	106020
LLC-loads	7391770	262810
LLC-load-misses	2671	1001
LLC-stores	218407	69369
LLC-store-misses	18512	0
dTLB-load-misses	2239	950
dTLB-store-misses	446	9
iTLB-load-misses	0	0
branch-loads	129163483	129898962
branch-load-misses	5688441	5560030

Da análise da tabela 2 podemos confirmar que:

- O número de CPU cycles é menor para a versão interchange, reflectindo-se num menor tempo de solução.
- O número de instruções é aproximadamente o mesmo dado para este dataset a invalidação dos dados não ser reflectida em leitura da memória principal mas apenas da LLCACHE (mais à frente neste relatório iremos analisar a influência da leitura da memória principal no tempo de execução).
- O número de "cache-references"é muito maior para o caso naive (tal com esperado pela invalidação dos dados de matrizes).
- O número de LLC loads é também muito maior para o caso naive (pela razão enumerada anteriormente).

Podemos então concluir que é este problema de padrão de acesso aos dados que resulta na degradação de performance da versão naive quando comparada com a versão interchange.

Analisemos agora as métricas compostas, obtidas com base nos valores apresentados:

Tabela 3: Performance rates (naive vs. interchange) para o nó compute-431

RATIO OR RATE	NAIVE	INTERCHANGE
Elapsed time (seconds)	0.2041	0.1597
Instructions per cycle	1.95 IPC	2.88 IPC
L1 cache miss ratio	22,9389 %	2,9942 %
L1 cache miss rate PTI	54,0218	6,5766
L3 cache miss ratio	0,0361 %	0,3808 %
Data TLB miss ratio	0,00027	0,0022
Data TLB miss rate PTI	0,0021	0,0008
Branch mispredict ratio	0,002	0,0019
Branch mispredict rate PTI	0,2473	0,2168

A degradação de performance das versões, para o caso do dataset menor, não é traduzida numa exponenciação dos tempos totais para a solução, muito devido aos dados apesar de estarem a ser acedidos de forma ineficiente, manterem-se contidos na LLCACHE. Focaremos a nossa análise doravante no dataset maior.

6. Parte 3

6.1. Período de amostragem e frequência de amostragem

Dado que pretendemos encontrar os hotspots de ambas as versões, iremos usar a amostragem baseada em cpu-cycles. Desta forma, porções da aplicação que consumam mais tempo terão um maior número de amostras registadas. No entanto, este tipo de amostragem tem um preço – pode ser demasiado pesada na avaliação de desempenho da aplicação.

Temos que ter em conta que cada máquina tem um número de contadores máximos e fixados a certo tipo de eventos específicos. Ora, quando requeremos um grande número de eventos no profiling, é necessária a recorrência a multiplexagem, que gasta tempo de computação. Essa mesma multiplexagem poderá ter um peso demasiado elevado na avaliação de desempenho.

Analisemos o tempo que demoram as versões large_naive e large_interchange sem qualquer tipo de ferramenta de amostragem por forma a podermos validar os resultados da avaliação de desempenho futura. Denote que por forma a validar os resultados foram realizadas 50 medições, sendo o valor apresentado o K Best (sendo K = 3) :

EVENT NAME	L. NAIVE	L. INTERCHANGE
Elapsed time (seconds)	65.61	10.43

Temos portanto agora uma base de referência também para estas versões que recorrem ao maior dataset em estudo. Voltemos portanto ao profiling das versões com os maiores datasets.

6.2. Análise comparativa dos evento de hardware para as versões large_naive e large_interchange, para o maior dataset em teste

Da análise de execução para as versões large_naive e large_interchange, foi produzida a tabela 4, que resume vários contadores de performance para as duas versões do kernel, para o dataset maior em estudo.

Denote que o overhead do processo de profiling não foi abusivo, dado que em comparação com os tempos sem qualquer tipo de profiling o aumento do tempo total da aplicação para ambos os casos rondou os 5%. Esta validação era necessária por forma a garantirmos uma comparação justa entre ambas as versões.

Tabela 4: Performance events (naive vs. interchange) para o nó compute-431

# EVENT NAME	NAIVE	INTERCHANGE
Elapsed time	68.119634290	11.053004702
Measurement Overhead	$\frac{68.12}{65.61} - 1 = 4\%$	$\frac{11.05}{10.43} - 1 = 6\%$
instructions	38 376 000 000	41 776 400 000
cycles	78390700000	12 384 800 000
cache-references	4 744 200 000	14 400 000
cache-misses	4 008 300 000	11 200 000
LLC-loads	4 815 000 000	14 800 000
LLC-load-misses	4 073 600 000	14400000
dTLB-load-misses	1100000	100000
branches	3923900000	3847500000
branch-misses	2200000	1900000

Ambas as versões executam aproximadamente o mesmo número de instruções, sendo que é para a versão mais otimizada que registamos um ligeiro acréscimo no valor medido. Dado que o kernel **large_interchange** executa em metade do tempo de o valor do número de ciclos para esta versão é também $1/2$ do número de ciclos necessários para a execução da versão **large_naive**.

É com base nos valores de cache miss e cache reference que podemos inferir o principal obstáculo da versão **large_naive**. A forma pouco eficiente de acesso aos dados da matriz B reduz drasticamente a perfomance do mesmo. Em comparação, a versão **large_interchange** acede de uma forma correcta aos dados, possibilitando ganhos de performance relativamente à versão original.

A tabela 5 apresenta as métricas compostas com base nos valores apresentados na tabela 4.

Pode comprovar que o número de instruções por segundo para a versão **large_interchange** é sete vezes maior relativamente à versão original e pouco eficiente – **large_naive**.

Tal como poderá constatar a percentagem de cache misses para ambas as versões é extremamente alta, o que é compreensível dado o tamanho dos datasets. Contudo, é no número de **cache-references** que a versão **large_naive** tem valores **330 vezes superiores** aos registados para a versão **large_interchange**.

Tabela 5: Performance rates (large_naive vs. large_interchange) para o nó compute-431

RATIO OR RATE	NAIVE	INTERCHANGE
Elapsed time (seconds)	0.2041	0.1597
Instructions per cycle	1.95 IPC	2.88 IPC
L3 cache miss ratio	84.6023 %	97.2973 %
Data TLB miss ratio	0.0232	0.6944
Data TLB miss rate PTI	0.0287	0.0024
Branch mispredict ratio	0.0561	0.0494
Branch mispredict rate PTI	0.0573	0.0455

6.2.1. Confirmação de metodologia de medição via metadados. Tal como referido na seção 4.3.2 a ferramenta PERF guarda os meta-dados de medição juntamente com o ficheiro de output ”perf.data”. O seguinte comando imprime a informação relativa às métricas recolhidas:

```
perf evlist --input=large_interchange_output
```

```
cpu-cycles
instructions
```

Com a adição da opção -F podemos visualizar a frequência de sampling:

```
perf evlist -F --input=large_interchange_output
```

```
cpu-cycles: sample_freq=100000
instructions: sample_freq=100000
```

Na sequência do já demonstrado na secção 4.3.2 podemos visualizar penas as características nas quais a medição foi realizada através do comando:

```
perf report --header-only --input=↩
    large_interchange_output
```

```
# ========
# captured on: Sun Jun  5 20:57:12 2016
# hostname : compute-431-1.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
# cpuid : GenuineIntel,6,44,2
# total memory : 12319324 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-↩
    cycles,instructions -c 100000 --output=↩
    large_interchange_output ./large_interchange
# event : name = cpu-cycles, type = 0, config = 0x0, ↩
    config1 = 0x0, config2 = 0x0, excl_usr = 0, ↩
    excl_kern = 0, excl_host = 0, excl_guest = 1, ↩
    precise_ip = 0, attr_mmap2 = 0, attr_mmap  = 1, ↩
    attr_mmap_data = 0, id = { 3517, 3518, 3519, 3520,↩
     3521, 3522, 3523, 3524, 3525, 3526, 3527, 3528, ↩
    3529, 3530, 3531, 3532, 3533, 3534, 3535, 3536, ↩
    3537, 3538, 3539, 3540 }
# event : name = instructions, type = 0, config = 0x1, ↩
    config1 = 0x0, config2 = 0x0, excl_usr = 0, ↩
    excl_kern = 0, excl_host = 0, excl_guest = 1, ↩
    precise_ip = 0, attr_mmap2 = 0, attr_mmap  = 0, ↩
    attr_mmap_data = 0, id = { 3541, 3542, 3543, 3544,↩
     3545, 3546, 3547, 3548, 3549, 3550, 3551, 3552, ↩
    3553, 3554, 3555, 3556, 3557, 3558, 3559, 3560, ↩
    3561, 3562, 3563, 3564 }
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to ↩
    display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# ========
#
```

6.3. Análise comparativa do número de amostras por evento de hardware para as versões large_naive e large_interchange

Dado que estamos a recolher amostras estatísticas da execução das duas versões da aplicação, inferindo a partir dessas mesmas amostras conclusões relativas à sua performance, é necessário garantir que o processo de amostragem e recolha é válido para o tamanho do dataset.

Sabemos que o intervalo de confiança dos valores apresentados será tão maior quanto o número de amostras recolhido. Dessa mesma premissa apresenta-se a tabela 6 que relaciona precisamente o número de amostras por métrica.

Tabela 6: Sampling mode: large_naive vs. large_interchange para o nó compute-431

# EVENT NAME	L. NAIVE	L. INTERCHANGE
instructions	383K samples	417K samples
cycles	783K samples	123K samples
cache-references	47K samples	144 samples
cache-misses	40K samples	112 samples
LLC-loads	48K samples	148 samples
LLC-load-misses	40K samples	144 samples
dTLB-load-misses	11 samples	1 samples
branches	39K samples	38K samples
branch-misses	22 samples	19 samples

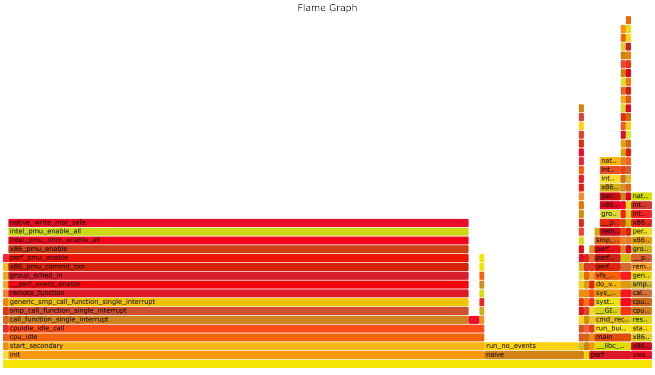
As métricas de maior peso para a nossa análise são precisamente instructions, cycles, cache-references, cache-misses, LLC-loads, e LLC-load-misses, sendo que é nestas onde se registam precisamente um maior número de amostras.

7. Análise gráfica com recurso a Flame Graphs

Os Flame Graphs podem ser produzidos através dos dados recolhidas pela ferramenta PERF, possibilitando a análise gráfica dos mesmos. Cada Rectângulo representa um método, e a largura desse mesmo rectângulo em comparação com a da imagem, demonstra o tempo de computação despendido na mesma. Podemos confirmar a pilha das chamadas também visualmente recorrendo a esta ferramenta. De seguida apresentam-se os Flame Graphs obtidos para as 2 versões da aplicação para os dois datasets em estudo, sendo os mesmos obtidos recorrendo à seguinte sequência de comandos, apenas alterando o executável e nome do ficheiro respectivamente:

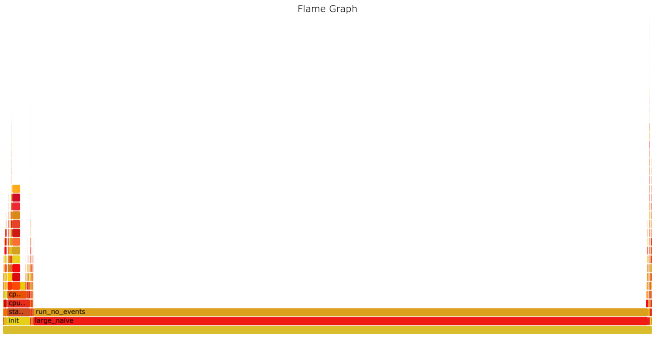
```
perf record -ag -F 99 ./large_naive
perf script | /share/jade/SOFT/FlameGraph/stackcollapse-perf.pl > out.perf-folded
cat out.perf-folded | /share/jade/SOFT/FlameGraph/flamegraph.pl > perf_kernel_large_naive.svg
```

Figura 1: Flame Graph kernel **naive**



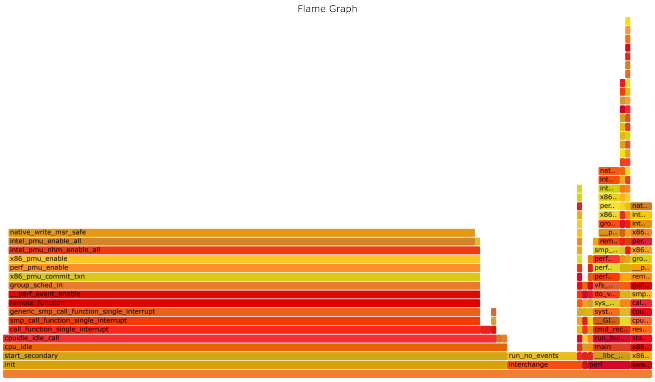
7.0.1. Flame Graph kernel naive.

Figura 2: Flame Graph kernel **large_naive**



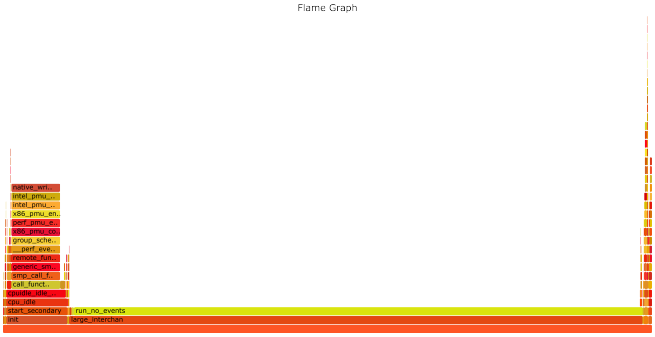
7.0.2. Flame Graph kernel large_naive.

Figura 3: Flame Graph kernel **interchange**



7.0.3. Flame Graph kernel interchange.

Figura 4: Flame Graph kernel **large_interchange**



7.0.4. Flame Graph kernel large_interchange. subsectionAnálise dos resultados gráficos O primeiro comando produziu o profiling de CPU stacks, a uma frequência de 99 Hertz. Ora, o comportamento visual vem confirmar o analisado durante todo este relatório. Dado que abordamos de uma forma introdutória os Flame Graphs será interessante recorrer a este tipo de ferramenta para outros kernels desenvolvidos em outras unidades curriculares.

8. Conclusão

Tal como mencionado no início do presente caso de estudo a ferramenta PERF mostra-se bastante útil e complementar a outras ferramentas já estudadas em termos de funcionalidades quando necessitamos de analisar a perfomance relativa de processos. Ou seja, no contexto da computação paralela será extremamente interessante recorrer a esta ferramenta na execução de algoritmos paralelos, sendo possível recorrendo a scripts como por exemplo o Flame Graph para adicionar funcionalidades à mesma.

Este foi apenas um trabalho introdutório mas permitiu demostrar a capacidade de recolher e ao mesmo tempo tratar dados de todo um sistema extremamente complexo e vasto com apenas uma ferramenta. O caso de estudo ultrapassa portanto os resultados obtidos pelos comandos e aplicações utilizadas, prendendo-se uma vez mais com o desenvolvimento de capacidade prática no uso da ferramenta, e envolvimento com métodos de tratamento de grandes volumes de dados, e análise de métricas de sistemas de computação de alta perfomance.

Retrata sobretudo a capacidade analisar funcionalidades disponibilizadas e a sua correta aplicação na resolução de problemas de computação tendo sempre em conta o mínimo de alteração possível na performance dos kernels/sistemas a analisar.