

## Módulo 2

### Avaliação do Desempenho: contadores de *hardware*

#### Preâmbulo

Este módulo serve vários objetivos:

- introdução à utilização dos nós de computação do *cluster Search*; para esse efeito consultar o “Guia de utilização do Search”;
- familiarização com o caso de estudo a utilizar durante o semestre, nomeadamente o GEMM (*GEneral Matrix Multiply*);
- familiarização com os contadores de eventos dos processadores modernos e com a biblioteca PAPI (*Performance Application Programming Interface*), usada para aceder aos mesmos;
- familiarização com as principais métricas usadas para modelação do desempenho (número de instruções e número de ciclos) e com a estimativa do tempo de execução de um programa.

#### Introdução

A complexidade crescente dos sistemas de computação torna o processo de otimização do tempo de execução das aplicações mais difícil. Para facilitar esta tarefa é necessário medir com exatidão vários aspetos da execução do programa. Neste sentido, os fabricantes de processadores foram introduzindo, ao longo dos últimos anos, **contadores de eventos internos ao processador** que podem ajudar neste processo de otimização. Alguns dos eventos mais frequentes incluem o **número de instruções executadas (#I)**, o **número de ciclos máquina (#CC)** e o **número de acessos à memória**, entre outros.

A biblioteca PAPI (**P**erformance **A**pplication **P**rogramming **I**nterface) apresenta uma abstração sobre estes contadores de eventos, através de uma API que facilita a leitura de um conjunto uniforme de eventos nas diversas arquiteturas.

O comando “`papi_avail`” permite verificar quais os eventos disponíveis numa dada arquitetura.

Exemplos:

- o evento `PAPI_TOT_INS` contabiliza o número total de instruções executadas (#I);
- o evento `PAPI_TOT_CYC` contabiliza o número total de ciclos do relógio (#CC).

O conjunto de eventos disponíveis varia com a arquitectura do processador.

#### Caso de Estudo: Multiplicação de Matrizes

O caso de estudo que iremos seguir é a multiplicação de matrizes, normalmente designada por GEMM (*GEneral Matrix Multiply*).

Relembre que a multiplicação de duas matrizes,  $C = A * B$ , implica calcular o produto interno entre cada linha de A e cada coluna de B. Isto é, cada elemento  $C_{ij}$  (linha i, coluna j) é dado por  $C_{ij} = \sum_{k=0}^{N-1} (A_{ik} * B_{kj})$ .

A Figura 1 ilustra este processo. Neste caso de estudo usaremos matrizes quadradas (número de linhas == número de colunas).

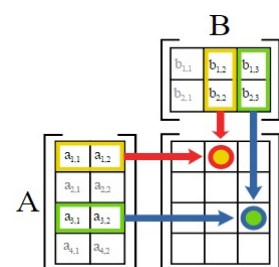


Figura 1 - GEMM

Ligue-se ao *front end* do Search (`ssh <username>@s7edu.di.uminho.pt`), copie o ficheiro `/share/acomp/GEMM-P02.zip` para a sua directoria e extraia os ficheiros usando o comando `unzip` (note que será criada uma pasta `P2` onde encontrará os ficheiros relevantes).

Verifique a função de multiplicação de matrizes em `gemm.c`. Deve examinar o código da função `gemm1()` – as restantes funções destinam-se a versões optimizadas a desenvolver no futuro.

Certifique-se que percebe bem a razão pela qual temos 3 ciclos:

- o mais externo (índice  $j$ ), percorre as colunas de C e B
- o ciclo intermédio (índice  $k$ ), percorre as colunas de A e as linhas de B
- o ciclo mais aninhado (índice  $i$ ), percorre as linhas de C e A

Verifique a função `main()` em `main.c` e note que:

- a função `verify_command_line()` lê e valida os argumentos da linha de comandos. Estes são obrigatórios e incluem o número de linhas (ou colunas) das matrizes quadradas e a versão de `gemm()` a utilizar (apenas a versão 1 está implementada nesta fase). Exemplo: para executar o programa numa matriz com 1024 linhas e usando a versão 1 da função (`gemm1()`) os argumentos são `1024 1` ;
- o PAPI é inicializado; todos os detalhes estão no ficheiro `my_papi.c`; este usa essencialmente as funções associadas à API de alto nível do PAPI, que podem ser consultadas em <https://icl.utk.edu/papi/>;
- inicialização das matrizes A e B com números pseudo-aleatórios;
- inicialização da matriz C a zero;
- a *cache* é aquecida, executando a função 1 vez. Note que `func()` é um apontador para uma das funções `gemm()` e foi inicializado quando da leitura da linha de comandos;
- as medições são efetuadas `NUM_RUNS` vezes para minimizar os efeitos que variações no estado da máquina possam ter no desempenho. São apresentadas as medições da execução que executou em tempo mínimo;
- A função `MYPAPI_start()` inicia a medição do tempo de execução; arranca com os contadores definidos em `Events[]` – nesta primeira versão são os eventos `PAPI_TOT_CYC` e `PAPI_TOT_INS`;
- a função – `func()` – é executada;
- `MYPAPI_stop()` mede o tempo de execução e lê os contadores; adicionalmente vai calculando quais as leituras correspondentes à execução mais rápida;
- `MYPAPI_output()` apresenta os resultados;
- é calculada a multiplicação de matrizes usando uma versão de referência da função `gemm()` e o resultado comparado com o que foi calculado anteriormente para verificar da correção do código.

**Exercício 1** - Construa o executável:

```
> make
```

Verifique o ficheiro `Makefile`. Verá que esta compilação foi feita sem optimizações (`CCFLAGS = -O0`)  
Submeta o programa para execução. A função a usar é `gemm1()` e matrizes com 1024 linhas, isto é:

```
> sbatch gemm.sh 1024 1
```

O `sbatch` indica qual o ID do *job* criado. Aguarde que o ficheiro de output do SLURM (gestor de filas do Search) seja criado; esse ficheiro terá o nome `slurm-<ID do job>.out`. Pode verificar o estado do seu *job* escrevendo

```
> squeue -u <username>
```

Repita a execução algumas vezes e verifique que o tempo de execução, número de instruções e número de ciclos de relógio variam apesar de estarmos a repetir a execução da função `NUM_RUNS` vezes. Isto deve-se a variações no estado da máquina, incluindo a frequência do relógio (que é variável), interrupções para

execução de outros processos e o estado da hierarquia de memória. Se necessário, execute o programa algumas vezes e considere as medições para o menor tempo de execução reportado.

## Avaliação do Desempenho

**Exercício 2** - Anote o tempo de execução, o número de ciclos do relógio e o número de instruções executadas para matrizes com  $n = 256, 512$  e  $1024$ , sendo  $n$  o número de linhas. O tempo de execução e o número de ciclos podem variar para diferentes execuções, mas o número de instruções ( $\#I$ ) executadas só depende do programa e do tamanho do problema ( $n$ ).

Neste exercício é-lhe pedido que vá duplicando o número de linhas. Como é que  $\#I$  varia com  $n$ ? Também duplica ou varia mais rapidamente? Consegue calcular aproximadamente qual a taxa de variação?

$$T_{exec} = \frac{\#I * CPI}{f} = \#cc/f$$

Equação 1 - Modelo de desempenho

**Exercício 3.1** - A Equação 1 apresenta o modelo de desempenho proposto nas aulas. Pretende-se que modifique o seu programa de forma a calcular e reportar o CPI (Ciclos Por Instrução).

Encontrará na função `MYPAPI_output()` (ficheiro `my_papi.c`) um comentário indicando onde fazer esses cálculos. A variável `float CPI` já se encontra declarada. Imprima o CPI com uma única casa decimal (`%.1f`).

**Exercício 3.2** – Sabendo que a frequência do relógio dos processadores das máquinas que está a usar é 2.6 GHz ( $2.6 * 10^9$  Hz) calcule o tempo de execução estimado pela equação 1 para os diferentes tamanhos das matrizes.

Preencha agora a primeira secção da Tabela 1 (linhas correspondentes a `-O0`).

Na secção de conteúdos da plataforma de *elearning* descarregue a folha de cálculo `GEMM-results`. Preencha o `Texec`, `CPI` e `#I` correspondentes a `gemml -O0` para  $n=1024$ . Mantenha esta folha de cálculo para a reutilizar ao longo do semestre.

**Exercício 3.3** – Consultando os valores que preencheu na Tabela 1 responda às seguintes questões:

- a) Que conclui da precisão do modelo teórico usado para estimar o tempo de execução?
- b) Como explica que o CPI possa ser menor do que 1?

**Exercício 4.1** – Modifique o ficheiro `Makefile` de forma a que seja usado o nível de otimização `-O2` (basta retirar o comentário na definição apropriada de `CCFLAGS` e comentar as restantes).

Construa o executável. Note bem que não basta usar o comando `make`; de facto, como os ficheiros de código C não foram alterados desde a última compilação o `make` comunicará que nada há a fazer. É necessário apagar o executável bem como eventuais ficheiros de código objeto que entretanto tenham sido gerados. Use a sequência de comandos:

```
> make clean
> make
```

Preencha agora a segunda secção da mesma tabela, bem com a linha correspondente a `gemml -O2` para  $n=1024$  na folha `GEMM-results`.

	linhas	Tempo medido (usec)	#CC	#I	CPI
-O0	256				
	512				
	1024				
	linhas	Tempo medido (usec)	#CC	#I	CPI
-O2	256				
	512				
	1024				

Tabela 1 - Tabela de medições

**Exercício 4.2** – Comparando os valores que preencheu na Tabela 1 para as duas versões do programa, responda às seguintes questões:

a) Houve ganhos no tempo de execução? A que se devem, isto é, como variam o número de instruções executadas e o CPI?

Uma otimização comum feita pelo compilador é a redução do número de acessos à memória para ler ou escrever dados, recorrendo a uma utilização mais intensiva e eficaz dos registos. O processador usado não permite a contagem direta do número de acessos à memória, mas é possível contar o número de instruções de leitura da memória (*loads* – `PAPI_LD_INS`) e instruções de escrita na memória (*stores* – `PAPI_SR_INS`).

**Exercício 5** – Altere a função `main()` para que passe a medir estes eventos (basta alterar os eventos a medir em `Events[]`).

Execute o programa otimizado e não otimizado para matrizes de 256 linhas e comente o que observa relativamente ao número total de acessos à memória.