Módulo 4
Avaliação do Desempenho: contadores de *hardware* e GEMM

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).

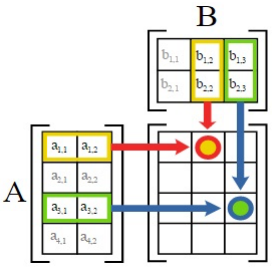

Figura 1 - GEMM

Ligue-se ao *front end* do Search (`ssh <username>@s7edu.di.uminho.pt`), copie o ficheiro `/share/acomp/GEMM-Lab1.zip` para a sua directoria e extraia os ficheiros usando o comando `unzip` (note que será criada uma pa 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 512 linhas, isto é:

```
> sbatch gemm.sh 512 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.

Note que:

- `PAPI_LD_INS` conta o número de instruções de leitura da memória (*loads*) e `PAPI_SR_INS` as instruções de escrita na memória (*stores*). A soma é o número de instruções que acedem à memória.
- `PAPI_L1_DCM` é o número de *misses* de dados na *cache* L1.

Preencha agora a primeira secção da Tabela 1 (linha 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 `gemm1 -O0` para n=1024. Mantenha esta folha de cálculo para a reutilizar ao longo do semestre.

Avaliação do Desempenho

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

Equação 1 - Modelo de desempenho

Exercício 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.
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 3 – Modifique o ficheiro `Makefile` de forma a que seja usado o nível de otimização `-O2` (bastará 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 `gemm1 -O2` para N=512 na folha GEMM-results.

	Tempo (msec)	#CC	#I	CPI	#I$_{MEM}$	L1_DCM
gemm1 -O0						
gemm1 -O2						
gemm2						
gemm3						

Tabela 1 - Tabela de medições (N=512)

Exercício 4 – 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?
b) Como variam o número de instruções executadas e o CPI?
c) Como variou o número de instruções que acedem à memória?

Localidade Espacial

A hierarquia de memória é eficaz na redução dos tempos de acesso à memória quando os padrões de acesso à mesma (dados e instruções) exibem localidade. A localidade espacial caracteriza-se por acessos consecutivos à memória endereçarem células de memória contíguas.

Em C as matrizes são armazenadas em memória em *row major order*, isto é elementos consecutivos da mesma linha são armazenados em posições contíguas de memória.
A função de multiplicação de matrizes usada na última secção, `gemm1()`, utiliza o seguinte algoritmo para calcular C = A * B, sendo A, B e C matrizes quadradas com N linhas (e colunas):

```
for (j = 0; j < n; ++j) {
      for(k = 0; k < n; k++ ) {
            for (i = 0; i < n; ++i) {
                  /* c[i][j] += a[i][k]*b[k][j] */
                  c[i*n+j] += a[i*n+k] * b[k*n+j];
}     }     }
```

Verifique que os acessos às matrizes exibem muito baixa localidade espacial.

A ordem de aninhamento dos 3 ciclos `for` deste código pode ser alterada, mantendo a correcção funcional do programa; no entanto, essa alteração tem um impacto significativo no desempenho.

Exercício 5 – Edite o ficheiro `gemm.c` e copie a função `gemm1()` criando uma nova versão da função `gemm2()`. Nesta última altere a ordem dos ciclos:

```
for (i = 0; i < n; ++i) {
      for(k = 0; k < n; k++ ) {
            for (j = 0; j < n; ++j) {
                  /* c[i][j] += a[i][k]*b[k][j] */
                  c[i*n+j] += a[i*n+k] * b[k*n+j];
}     }     }
```

Construa o executável (`make`), verificando na `Makefile` que estão a ser usadas as opções de optimização:

```
CCFLAGS = -O2 -march=ivybridge
```

Exercício 6 – Execute a multiplicação de matrizes para 512 linhas, usando a versão 2:

```
> sbatch gemm.sh 512 2
```

Preencha a linha da Tabela 1 correspondente a `gemm2()`.

Considerando os valores registados na Tabela 1é fácil concluir que o melhor desempenho da versão 2 se deve a uma redução abrupta do CPI e apenas marginalmente à redução no número de instruções executadas.
No entanto, o número de instruções de acesso à memória não apresenta variações significativas! A que se deverá então a redução do CPI?

Localidade Temporal

Os acessos à matriz A exibem localidade temporal e a hierarquia de memória permite aumentar o desempenho explorando esse facto.

O compilador pode ele próprio explorar a localidade temporal, copiando para um registo o elemento `a[i][k]` e evitando leituras da memória (o tempo de acesso aos registos é inferior ao tempo de acesso à *cache* L1). Esta foi aliás a principal optimização estudada em Sistemas de Computação.

No entanto, na função `gemm2()` o compilador não pode copiar `a[i][k]` para um registo, devido a um bloqueador de optimização designado por *aliasing*. Na verdade, é possível que as matrizes A e C sejam as mesmas, isto é que os apontadores `*a` e `*c` apontem para o mesmo espaço de memória ou para espaços que se intersectam. Como o compilador não verifica se tal acontece, então lê `a[i][k]` de memória sempre que lhe acede.

Exercício 7 – Edite o ficheiro `gemm.c` e copie a função `gemm2()` criando uma nova versão da função `gemm3()`. Nesta última use uma variável local:

```
float aik;
for (i = 0; i < n; ++i) {
      for(k = 0; k < n; k++ ) {
            aik = a[i*n+k];
            for (j = 0; j < n; ++j) {
                  /* c[i][j] += a[i][k]*b[k][j] */
                  c[i*n+j] += aik * b[k*n+j];
}     }     }
```

Execute `gemm3()` para N=512.

Que variações detecta em `PAPI_LD_INS` e `PAPI_SR_INS`? Justifique estas diferenças.

Preencha a linha correspondente a `gemm3()` na Tabela 1 e na folha de cálculo `GEMM-results`.