Módulo 11

OpenMP 2 : GEMM

GEMM

Copie o ficheiro `/share/acomp/GEMM-P11.zip` para a sua directoria e `faça` o `unzip`; será criada a directoria de trabalho para este módulo, designada de `P11`. Visualize a função `gemm10()`, cujo código se repete a seguir:

```
void gemm10 (float *__restrict__ a, float *__restrict__ b, float *__restrict__ c,
int n) {
        #pragma omp parallel
        {
          int i, j, k;
          float aik;

          MYPAPI_thread_start(omp_get_thread_num());
          #pragma omp for
          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];
              }
            }
          }
          MYPAPI_thread_stop(omp_get_thread_num());
        }
}
```

Note que o código é equivalente ao de `gemm3()`, com as seguintes diferenças:

- é criado um bloco paralelo – a partir deste ponto, e até ao fim deste bloco, o código respectivo é executado em paralelo por um determinado número de *threads*;
- há um conjunto de variáveis declaradas dentro do contexto do bloco paralelo, garantindo assim que estas são **locais** a cada uma das *threads*;
- As funções `MYPAPI_thread_start()` e `MYPAPI_thread_stop()` são invocadas antes e depois da execução dos ciclos *for*, mas dentro do bloco paralelo, para instrumentar cada uma das *threads*, lendo os respectivos contadores de eventos;
- a directiva `#pragma omp for` é colocada antes do ciclo *for* mais externo, para que o seu espaço de iteração seja distribuído pelas várias *threads*.

Exercício 1 – Construa o executável e execute `gemm3()`:

```
sbatch gemm.sh 1024 3
```

Preencha a respectiva linha na Tabela 1.

No próximo exercício iremos executar `gemm10()`. Note que:

- para executar versões do `gemm` com ID igual a superior a 10 (isto é, que usam OpenMP) é necessário indicar o número de *threads* a utilizar. Para usar 8 *threads* escreva:
  ```
  sbatch gemm.sh 1024 10 8
  ```
  A omissão do 3º parâmetro resulta na criação de apenas 1 *thread*.
- o tempo de execução de um programa paralelo, conforme percepcionado pelo utilizador, corresponde ao tempo do último processador a terminar. Admitindo que a frequência do *clock* do processador é fixa, então tal corresponde ao processador que usa mais *clock cycles* para executar a sua *thread*. O CPI percepcionado corresponde portanto ao número de ciclos que decorrem desde que a função inicia até que termina, dividido pelo número total de instruções executado por **todas as threads**. Esta métrica é reportada pelo programa para versões do `gemm()` maiores ou iguais a 10.

Exercício 2 – Execute `gemm10()` usando 1, 2, 4, 8, 16 e 32 *threads* e preencha as respectivas linhas na Tabela 1. As 2 colunas da esquerda (SpeedUp e Eficiência) serão preenchidas no próximo exercício.

NOTA: A máquina usada neste exercício tem 16 *cores* físicos. Cada um deles suporta *Simultaneous Multithreading* (*HyperThreading* na terminologia da Intel), pelo que o Sistema Operativo reporta 32 *cores* lógicos.

O ***SpeedUp*** exprime o ganho, em tempo de execução, quando são usados *p* processadores relativamente a uma versão de referência. Esta versão de referência é, frequentemente, a versão sequencial (1 processador apenas) que usa o mesmo algoritmo que a versão paralela, resultando naquilo que na literatura se designa por *relative speedup*. Esta é a definição que usaremos para medir o ganho, resultando na expressão

$$S_p = \frac{T_1}{T_p}$$

onde S_p é o *speedup* com *p* processadores, T_p o tempo de execução com *p* processadores e T_1 o tempo de execução com 1 processador.

A **Eficiência** indica quão longe o *speedup* obtido está do *speedup* ideal. No caso em que todos os *p* processadores são idênticos o *speedup* ideal é igual ao número de processadores. Isto é, se usamos 2 processadores então esperamos um ganho de 2, se usarmos 16 processadores esperamos um ganho de 16 – este resultado é designado por *speedup linear*. A eficiência indica em que percentagem atingimos este objectivo e é dada por:

$$E_p = \frac{S_p}{p} * 100$$

Exercício 3 – Preencha agora as colunas de SpeedUp e Eficiência da Tabela 1 para `gemm10()` usando como tempo de referência, T_1, o tempo de `gemm3()`.
Como evolui a eficiência com o aumento do número de processadores? Porquê?

Exercício 4 – Pretende-se que `gemm11()` combine o *multithreading* com a vectorização. Escreva o código de `gemm11()`, usando auto-vectorização, tendo em consideração que:

- o código é em tudo idêntico ao de `gemm10()`;
- é necessário activar a opção `tree-vectorize`, já incluída no código fornecido;
- é necessário preceder o ciclo a vectorizar (isto é, o ciclo *for* mais aninhado – índice `j`) com a directiva `#pragma omp simd` (na ausência desta directiva o compilador não vectorizará o código deste ciclo enquanto usa simultaneamente o OpenMP).

Construa o executável. Preencha a linha correspondente a `gemm6()` na Tabela 1. Esta é a versão sequencial equivalente a `gemm11()`. Preencha as linhas correspondentes a `gemm11()` na mesma tabela. Para o cálculo do *speedup* e eficiência use como tempo de referência, T_1, o tempo de `gemm6()`.

RESOLUÇÃO:

```
#pragma GCC optimize("tree-vectorize")
void gemm11  (float *__restrict__ a, float *__restrict__ b, float *__restrict__ c, int n)
{
        #pragma omp parallel
        {
          int i, j, k;
          float aik;

          MYPAPI_thread_start(omp_get_thread_num());
          #pragma omp for
          for (i = 0; i < n; ++i) {
            for(k = 0; k < n; k++ ) {
              aik = a[i*n+k];
              #pragma omp simd
              for (j = 0; j < n; j++) {
                /* c[i][j] += a[i][k]*b[k][j] */
                c[i*n+j] += aik * b[k*n+j];
              }
            }
          }
          MYPAPI_thread_stop(omp_get_thread_num());
        }
}
#pragma GCC reset_options
```

Exercício 5 – Desenvolva o código de `gemm12()`, baseado em `gemm8()` e `gemm10()`, usando *compiler intrinsics* para vectorizar.

Construa o executável. Preencha a linha correspondente a `gemm8()` na Tabela 1. Esta é a versão sequencial equivalente a `gemm12()`. Preencha as linhas correspondentes a `gemm12()` na mesma tabela. Para o cálculo do *speedup* e eficiência use como tempo de referência, T_1, o tempo de `gemm8()`.

RESOLUÇÃO:

```
void gemm12  (float *__restrict__ a, float *__restrict__ b, float *__restrict__ c, int n)
{
  #pragma omp parallel
  {
    int i, j, k;
    __m256 aik, bkj, cij, prod;

    MYPAPI_thread_start(omp_get_thread_num());
    #pragma omp for
    for (i = 0; i < n; ++i) {

      for(k = 0; k < n; k++ ) {
        // set all 8 elements of aik with a[i,k]
        aik = _mm256_broadcast_ss (&a[i*n+k]);
        // add 8 to j since 8 SPFP are processed per iteration
        for (j = 0; j < n; j+=8) {
          // c[i][j] += a[i][k]*b[k][j]
          bkj = _mm256_load_ps (&b[k*n+j]);
          cij = _mm256_load_ps (&c[i*n+j]);
          prod = _mm256_mul_ps (aik, bkj);
          cij = _mm256_add_ps (cij, prod);
          _mm256_store_ps (&c[i*n+j], cij);
        }
      }
    }
    MYPAPI_thread_stop(omp_get_thread_num());
  }
}
```

Exercício 6 – Iniciamos o semestre com a versão mais ineficiente da multiplicação de matrizes, nomeadamente, `gemm1()`.Neste caminho quantas vezes mais rápida ficou a nossa função? Preencha o quadro abaixo, usando o melhor resultado que obteve (terá sido com `gemm11()` ou `gemm12()`).

Versão	Threads	T (ms)	Ganho
gemm1()	1	7653	----------
gemm11()	32	15	**510**

Exercício 7 – De facto, a primeira versão de `gemm1()` foi compilada sem optimizações `(-O0)`. Use a directiva `#pragma GCC optimize("O0")` para gerar essa primeira versão e calcule o ganho total que obtivemos.

gemm1 (-O0) T (ms)	gemm11 T (ms)	Ganho
21656	15	**1444**

Tabela 1 - Resultados GEMM (n=1024)

Ver.	Obs	#threads	T(ms)	#I(M)	CPI	SpeedUp	Eficiência
`gemm3()`	Versão base	1	649	6450	0.340	---	---
`gemm10()`	Versão base + OpenMP	1	651	6450	0.341	1,0	100%
		2	357	6450	0.173	1,8	91%
		4	209	6450	0.086	3,1	78%
		8	106	6451	0.043	6,1	77%
		16	47	6461	0.022	13,9	97%
		32	53	6461	0.021	12,3	60%
`gemm6()`	Compiler vectorization	1	269	1625	0.430	---	---
`gemm11()`	Compiler vectorization + OpenMP	1	203	1626	0.414	1,0	100%
		2	133	1627	0.207	2,0	101%
		4	66	1627	0.104	4,0	101%
		8	33	1627	0.052	8,2	101%
		16	17	1652	0.026	15,8	99%
		32	15	1635	0.023	17,9	56%
`gemm8()`	Intrinsics vectorization	1	280	813	0.812	---	---
`gemm12()`	Intrinsics vectorization + OpenMP	1	216	824	0.787	1,3	130%
		2	128	824	0.394	2,2	109%
		4	63	824	0.198	4,4	111%
		8	32	825	0.099	8,8	111%
		16	16	825	0.051	17,5	109%
		32	18	873	0.051	15,5	49%