

Módulo 8

GEMM: Vectorização

Contextualização

Em sessões anteriores o código da função `gemm()` foi otimizado, no sentido de i) explorar a hierarquia de memória e ii) através do *loop unrolling*, explorar a superescalaridade e reduzir o número de instruções executadas.

Exercício 1 – Copie o ficheiro o ficheiro `/share/acomp/GEMM-P08.zip` para a sua directoria e extraia os respectivos ficheiros.

Modifique o ficheiro `main.c` para que sejam monitorizados 4 eventos, sendo o novo evento `PAPI_VEC_SP`:

```
#define NUM_EVENTS 4
```

```
int Events[NUM_EVENTS] = { PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_VEC_SP , PAPI_L1_DCM};
```

Este evento reporta o número de operações em vírgula flutuante, precisão simples, realizadas pelas unidades funcionais vectoriais associadas às extensões SSE e AVX. Note que:

- conta operações, e não instruções, portanto uma instrução SSE corresponde a 4 operações e uma instrução AVX corresponde a 8;
- as operações contabilizadas incluem, além de operações aritméticas ou lógicas, qualquer outro tipo de instrução vectorial precisão simples, tais como *loads*, *stores* e movimentos de dados envolvendo registos `%xmm` ou `%ymm`.

Construa o executável e preencha as linhas correspondentes a `gemm3()` e `gemm5()` na Tabela 1, escrevendo:

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

```
> sbatch gemm.sh 1024 5
```

Auto-vectorização

Examine agora o código de `gemm6()`. Este é exactamente o mesmo que `gemm3()` com as seguintes excepções:

1. Os apontadores `*a`, `*b` e `*c` estão qualificados como `__restrict__`. Isto garante ao compilador que cada um destes apontadores é o único apontador usado para aceder ao respectivo objecto (matriz, neste caso). O compilador pode ignorar a possibilidade de *aliasing*, que o programador lhe indica não existir, e aplicar optimizações agressivas, tais como a vectorização;
2. `#pragma GCC optimize("tree-vectorize")` indica ao compilador que deve tentar vectorizar o código de `gemm6()`.

Exercício 2 – Execute `gemmm6()` (`sbatch gemm.sh 1024 6`) e complete a linha relativa a `gemmm6()` na Tabela 1. Compare com os resultados de `gemmm3()`, respondendo às questões abaixo:

- Qual o ganho no tempo de execução?
 - $648 / 197 \sim 3.3$
- Há algum ganho na exploração da hierarquia da memória?
 - Não
- Qual o ganho em termos do número absoluto de instruções executadas?
 - $6450 / 1625 = 4$
- Tendo em consideração os 3 pontos anteriores de que resulta o ganho no desempenho?
 - Claramente da diminuição do número de instruções. Há um pequeno aumento do CPI que implica que o ganho no desempenho seja inferior ao ganho em #I
- Como explica que `gemmm6()` realize as mesmas operações que `gemmm3()` com um número muito mais reduzido de instruções?
 - A primeira usa instruções vectoriais AVX. Cada uma destas opera sobre 8 elementos de dados precisão simples, logo são necessárias menos instruções do que na versão escalar usada em `gemmm3()`.
- Sabendo que a largura das unidades vectoriais AVX é 8, discuta razões para que o ganho relativamente a `gemmm3()`, em tempo e em número de instruções, seja inferior a 8.
 - O ganho em número de instruções é menor do que 8 porque nem todas as instruções operam sobre valores em vírgula flutuante (e estas não são vectorizadas) e porque o compilador gera, para cada instrução SPFP de acesso à memória, várias instruções vectoriais – creio que serão estas que permitem manter um CPI tão baixo
 - o custo de acesso à memória é significativo comparado como o cálculo de operações aritméticas como a multiplicação e adição. Este algoritmo exhibe uma baixa intensidade computacional (número de operações matemáticas por acesso à memória) pelo que não consegue explorar o total potencial do processamento vectorial.

Exercício 3 – O compilador pode tentar vectorizar (para este algoritmo) todas as operações que envolvam valores em vírgula flutuante, precisão simples. Olhando para o algoritmo de `gemmm6()` podemos identificar:

- n^2 leituras (*load*) da memória para `a[i][k]`;
- n^3 leituras (*load*) da memória de `b[k][j]`;
- n^3 multiplicações de `a[i][k]` por `b[k][j]`;
- n^3 leituras (*load*) da memória de `c[i][j]`;

- n^3 adições de `c[i][j]` com `aik*[k][j]`;
- n^3 escritas (*store*) na memória de `c[i][j]`;

Totalizando $n^2 + 5 * n^3$ potenciais operações em vírgula flutuante. Para $n = 1024$, temos então potencial para aproximadamente 5 000 Mega operações realizadas recorrendo às unidades vectoriais. Verifique na tabela 1 o valor de `PAPI_VEC_SP` para `gemmm6()`. O compilador terá optado por vectorizar todas as operações possíveis?

Claramente: NÃO!

Exercício 4 – `gemmm7()` é exactamente igual a `gemmm6()`, mas com a opção de *loop unrolling* activa. Execute-a (`sbatch gemm.sh 1024 7`) e complete a respectiva linha na Tabela 1. O tempo de execução diminuiu? Diminuiu mais ou menos do que o número de instruções executadas? Qual o outro factor do nosso modelo de desempenho que aumentou para que o ganho não seja proporcional à diminuição do número de instruções?

A redução do número de instruções é acompanhada de um aumento semelhante do CPI.

Vectorização explícita

O programador pode ter maior controlo sobre o código vectorial se utilizar as pseudo-funções designadas por *compiler intrinsics*. Para este caso de estudo (GEMM) usaremos as funções associadas ao AVX. Edite o código de `gemmm8()`, tendo em atenção as seguintes recomendações:

- inclua o ficheiro que contém os protótipos das funções intrínsecas necessárias, nomeadamente:
`#include <immintrin.h>`
- `__m256` permite declarar variáveis de 256 *bits* que correspondem ao formato com que 8 valores em vírgula flutuante SP são guardados nos registos `%ymm`; Precisaremos de 4; declare-as como locais a `gemmm8()`, conforme segue:
`__m256 aik, cij, bkj, prod;`
- `_mm256_broadcast_ss (float *)` permite carregar para todos os 8 elementos de uma variável `__m256` o mesmo valor. Será útil para ler o valor de `a[i][k]` que é usado em todas as iterações do ciclo `j`; Adicione no local apropriado a leitura de `aik`, usando esta função.
- `_mm256_load_ps (float *)` permite carregar 8 valores SP da memória para uma variável `__m256`. Note que os 8 valores a carregar **TÊM QUE ESTAR** em endereços consecutivos na memória e o endereço passado como parâmetro **TEM** que ser alinhado a múltiplos de 32; Adicione no local apropriado a leitura de `bkj` e `cij` usando esta função.

- `_mm256_mul_ps (__m256, __m256)` e `_mm256_add_ps (__m256, __m256)` calculam, respectivamente, o produto e a adição de duas variáveis `__m256` (8 valores SP). Ambas devolvem uma variável `__m256` com o resultado. Calcule o produto de `aik` com `bkj` e adicione-o a `cij`.
- `_mm256_store_ps (float *, __m256)` permite guardar 8 valores SP de uma variável `__m256` na memória. Note que os 8 valores a guardar **TÊM QUE ESTAR** em endereços consecutivos na memória e o endereço passado como parâmetro **TEM** que ser alinhado a múltiplos de 32; Adicione esta função para guardar `cij`.

Código (não fornecido aos alunos):

```
#include <immintrin.h>

void gemm8 (float *__restrict__ a, float *__restrict__ b, float *__restrict__ c, int n) {
    int i, j, k;
    __m256 aik, bkj, cij, prod;

    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);
            }
        }
    }
}
```

Exercício 5 – Execute `gemm8()` (`sbatch gemm.sh 1024 8`) e complete a respectiva linha na Tabela 1.

Compare com os resultados de `gemm6()`, respondendo às seguintes questões:

- Existe um ganho significativo no número total de instruções executadas?
 - Sim, são aproximadamente metade (ganho =2)
- Tendo em conta a resposta anterior, como justifica o tempo de execução observado?
 - O CPI também duplica
- Esta versão vectoriza mais ou menos operações do que a auto-vectorizada?
 - 1.5 vezes mais.
- Qual o ganho em instruções relativamente a `gemm3()` ?
 - $6450 / 813 = 7.9$ – semelhante à largura das unidades AVX.

Exercício 6 – Copie `gemm8()` para `gemm9()`. Note que esta última activa a opção de *unrolling* pelo compilador. Execute-a (`sbatch gemm.sh 1024 9`) e complete a respectiva linha na Tabela 1. Compare com os resultados de `gemm7()`, respondendo às seguintes questões:

- Existe um ganho significativo no número total de instruções executadas?
 - $1272 / 501 = 2.5$
- Tendo em conta a resposta anterior, como justifica o tempo de execução observado?
 - Aumento do CPI
- Qual o ganho em instruções relativamente a `gemm5()`?
 - $3647 / 501 = 7.3$
- Qual o ganho em instruções relativamente a `gemm8()`? Então porquê o ganho marginal no tempo de execução?
 - $813 / 501 = 1.6$ para um ganho em tempo de 1.05. O aumento do CPI (1.5 vezes maior) neutraliza o ganho no número de instruções.

Tabela 1 - Medições GEMM

n=1024						
Versão	Obs	T (ms)	CPI	#I (M)	VEC_SP (M)	L1_DCM (M)
<code>gemm3()</code>	Variável local: <code>aik</code>	648	0.3	6450	0	67
<code>gemm5()</code>	<i>Loop unrolling</i> compilador	523	0.5	3647	0	67
<code>gemm6()</code>	Auto-vectorização	197	0.4	1625	2148	67
<code>gemm7()</code>	compiler Vect + <i>unroll</i>	182	0.5	1272	2148	67
<code>gemm8()</code>	Vectorização <i>intrinsics</i>	192	0.8	813	3235	67
<code>gemm9()</code>	vect <i>intrinsics</i> + compiler unroll	179	1.2	501	3039	67

Um efeito, talvez surpreendente numa primeira análise, da vectorização é o aumento do CPI. A que se deverá? O nosso modelo de desempenho diz-nos que:

$$T_{exec} = \frac{\#I * (CPI_{CPU} + CPI_{MEM})}{f}$$

O CPI_{CPU} não deverá aumentar. As unidades funcionais vectoriais efectivamente realizam todas as operações (8, no caso de AVX vírgula flutuante precisão simples) em paralelo, pelo que este parâmetro deverá manter-se aproximadamente constante relativamente à versão escalar.

Já o CPI_{MEM} aumenta porque a versão vectorizada acede a muitos mais dados em memória por unidade de tempo (largura de banda) do que a versão escalar; a largura de banda entre os vários níveis da hierarquia da memória pode ser insuficiente para satisfazer os pedidos do CPU, resultando em mais ciclos de espera (*stall*). Este fenómeno pode ser verificando medindo o número de *cache misses* nos vários níveis de memória. O processador especula sobre o padrão de acesso aos dados exibido pelo programa e tenta ler para a *cache* os

dados necessários antes de estes serem solicitados, usando um mecanismo designado por *pre-fetching*. Se, devido a limitações na largura de banda, o *pre-fetching* é incapaz de aceder aos blocos de dados atempadamente, então as instruções de *load* observam uma *cache miss*.

Exercício 7 – Modifique o ficheiro `main.c` para que sejam monitorizados 5 eventos:

```
#define NUM_EVENTS 4
int Events[NUM_EVENTS] = { PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_L2_TCM , PAPI_L3_TCM
};
```

`PAPI_L2_TCM` e `PAPI_L3_TCM` reportam o número total de *misses* em L2 e L3. Preencha agora a Tabela 2. Que conclui sobre o comportamento dos vários níveis de *cache* e respectivo impacto no CPI?

Sabendo que a *cache* L3 dos processadores usados tem 20 MiBytes e que cada matriz ocupa $4 * 1024 * 1024 = 4$ MiB, como explica as *miss rates* observadas na L3?

(LPS): O aumento no CPI deve-se de facto a um aumento muito significativo da *miss rate* na L2. As matrizes basicamente cabem na L3.

Versão	Obs	T (ms)	CPI	L2_TCM (M)	L3_TCM (M)
gemm3 ()	Variável local: aik	648	0.3	3.42	0.010
gemm6 ()	Auto-vectorização	196	0.4	2.37	0.026
gemm7 ()	compiler Vect + unroll	184	0.5	12.90	0.029
gemm8 ()	Vectorização <i>intrinsic</i> s	192	0.8	10.54	0.029
gemm9 ()	vect <i>intrinsic</i> s + compiler unroll	179	1.2	20.54	0.032

Tabela 2 - Cache misses

ADVANCED NOTE:

Se o padrão de acesso aos dados é idêntico em todas as versões acima (não há alterações na ordem de acesso aos vários elementos das matrizes), como se compreende que o número de *misses* na L2 aumente?

A resposta está no mecanismo de *prefetching*. Os processadores modernos possuem mecanismos de *prefetching* de dados que detectam padrões de acesso regulares à memória e tentam carregar os dados que se prevê serem necessários no futuro ANTES das respectivas instruções de *load* serem executadas.

O impacto do *prefetching* no desempenho é tudo menos linear. O respectivo sucesso depende de vários factores, incluindo:

- Interações entre padrões de acesso à memória e cache L3 pelos vários *cores*;
- Sucesso na previsão dos dados a carregar em antecipação;
- Distância do *prefetching* – isto é, quantos acessos o mecanismo de *prefetching* se antecipa aos *loads* – uma distância muito curta pode não permitir esconder a latência de acesso aos níveis mais elevados da hierarquia (isto é, a instrução de *load* ocorre antes de o carregamento dos dados para a cache L1

ou L2 estar terminada, originando uma *miss*); uma distância muito grande, pode resultar em invalidação de linhas da *cache* que viriam ainda a ser usadas e em erros de previsão.

Neste caso, tudo parece indicar que no caso das versões vectorizadas (e em particular com *loop unrolling*) a distância de *prefetch* para a L2 não seja suficiente para mascarar os acessos à L3, resultando num número mais elevado de *misses* – relembre que com as instruções vectoriais o processador consome mais dados por ciclo do relógio.

Detalhes sobre o mecanismo de prefetching da arquitectura Sandy Bridge e Ivy Bridge podem ser encontrados nas secções 2.3.5.4 e 2.3.7 de "*Intel® 64 and IA-32 Architectures Optimization Reference Manual*" (Order Number: 248966-033; June 2016), disponível em <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>