Módulo 8

GEMM: Vectorização

Contextualização

Em sessões anteriores o código da função `gemm()` foi optimizado, 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:

```
> qsub –F "1024 3" gemm.sh
> qsub –F "1024 5" gemm.sh
```

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 `gemm6()` (`qsub -F "1024 6" gemm.sh`) e complete a linha relativa a `gemm6()` na Tabela 1. Compare com os resultados de `gemm3()`, respondendo às questões abaixo:

1. Qual o ganho no tempo de execução?
2. Há algum ganho na exploração da hierarquia da memória?
3. Qual o ganho em termos do número absoluto de instruções executadas?
4. Tendo em consideração os 3 pontos anteriores de que resulta o ganho no desempenho?
5. Como explica que `gemm6()` realize as mesmas operações que `gemm3()` com um número muito mais reduzido de instruções?
6. Sabendo que a largura das unidades vectoriais AVX é 8, discuta razões para que o ganho, quer em tempo, quer em número de instruções, seja significativamente inferior a 8.

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 `gemm6()` podemos identificar:

- n^2leituras (*load*) da memória para `aik`;
- n^3leituras (*load*) da memória de `b[k][j]`;
- n^3multiplicações de `aik` por `b[k][j]`;
- n^3leituras (*load*) da memória de `c[i][j]`;
- n^3adições de `c[i][j]` com `aik*[k][j]`;
- n^3escritas (*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 `gemm6()`. O compilador terá optado por vectorizar todas as operações possíveis?

Exercício 4 – `gemm7()` é exactamente igual a `gemm6()`, mas com a opção de *loop unrolling* activa. Execute-a (`qsub -F "1024 7" gemm.sh`) 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?

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 `gemm8()`, 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 `gemm8()`, 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, __256)` 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`.

Exercício 5 – Execute `gemm8()` (`qsub -F "1024 8" gemm.sh`) 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?
- Tendo em conta a resposta anterior, como justifica o tempo de execução observado?
- Esta versão vectoriza mais ou menos operações do que a auto-vectorizada?
- Qual o ganho em instruções relativamente a `gemm5()`?

Exercício 6 – Copie `gemm8()` para `gemm9()`. Note que esta última activa a opção de *unrolling* pelo compilador. Execute-a (`qsub -F "1024 9" gemm.sh`) 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?
- Tendo em conta a resposta anterior, como justifica o tempo de execução observado?
- Esta versão vectoriza mais ou menos operações do que a auto-vectorizada?
- Qual o ganho em instruções relativamente a `gemm5()`?
- Qual o ganho em instruções relativamente a `gemm8()`? Então porquê o ganho marginal no tempo de execução?

Tabela 1 - Medições GEMM

n=1024

Versão	Obs	T (ms)	CPI	#I (M)	VEC_SP (M)	L1_DCM (M)
`gemm3()`	Variável local: aik					
`gemm5()`	*Loop unrolling* compilador					
`gemm6()`	Auto-vectorização					
`gemm7()`	compiler Vect *+ unroll*					
`gemm8()`	Vectorização *intrinsics*					
`gemm9()`	vect *intrinsics* + compiler unroll					

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 5
int Events[NUM_EVENTS] = { PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_L1_DCM, PAPI_L2_TCM , PAPI_L3_TCM };
```

`PAPI_L1_DCM` reporta o número de misses na *data cache* L1, `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?

n=1024

Versão	Obs	T (ms)	CPI	L1_DCM (M)	L2_TCM (M)	L3_TCM (M)
`gemm3()`	Variável local: aik					
`gemm6()`	Auto-vectorização					
`gemm7()`	compiler Vect *+ unroll*					
`gemm8()`	Vectorização *intrinsics*					
`gemm9()`	vect *intrinsics* + compiler unroll					

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* serrem executadas.

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

- Interacções 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