Lab Guide 1

Instruction Level Parallelism

Objectives:

- Review of the basic processor performance equation
  - Execution time = Instructions x Cycles Per Instruction x Clock Cycle Time (Texe = #I x CPI x Tcc)
- Identify instruction level parallelism (ILP) and its limitations on the matrix multiplication case study

Introduction

This lab session uses the `perf` tool to profile the execution of the Matrix Multiplication (MM) case study (both were introduced in lab session 0). The main goal is to collect performance events counts of the basic performance equation, as well as other performance counters in order to understand ILP and its limitations.

The students are encouraged to use their own code, developed in lab session 0, but a simple MM code can be downloaded from the course eLearning page. Additionally, for students that don't have the `perf` tool installed on their personal machine, the Annex 1 contains instruction for remote access and execution of these exercises on the Search cluster, using the password received in the university email account.

To obtain the application profile with `perf` run the application with `perf record ./a.out` to sample the execution data at fixed time intervals and use `perf report` to generate a [flat] view profile with the application hotspots. In simple cases, these two steps can be replaced by `perf stat -e` *`XXX`* `./a.out` which gets the performance counter *XXX* over the entire program (even better, use the `-r X` option to run multiple times and collect an average number. Why is this important?).

To get a better profile and more accuracy, the code (C program) should be compiled with `-g -fno-omit-frame-pointer`.

Exercise 1 - The basic processor performance equation

a) Performance estimation: What is the complexity of the MM (e.g., in big O notation, where NxN is the matrix size)? What increase in execution time is expected when the N value doubles? Which component of the performance equation is affected by this increase (e.g., #I, CPI or Tcc)?

b) Instructions estimation**:** Look at the MM assembly code that is generated (i.e., `gcc -O2 -S …`) with and without optimisation (-O2 versus -O0). Can you estimate the number of instructions executed for a NxN matrix, on each case? (Note: if you have difficulty you can use the perf annotation tool to identify the instructions inside the MM hotspot). What is the expected gain from the compiler optimised version?

c) Measure**:** The number of instructions executed (#I) and clock cycles (#CC) can be directly measured with `perf stat -e cycles,instructions ./a.out` (note: this takes overall metrics, not metrics by function as `perf record`, but it is enough for this case. Why?). Fill the following table for matrix sizes of 128x128, 256x256 and 512x512 using the -O2 optimisation level. Measure the time without optimisation just for a 512x512 matrix size. Also, fill the matrix by calculating the average CPI column for the 512x512 matrix size (see lecture notes for CPI definition).

	size	Texe	#CC	#I	#I Estimated	Average CPI (calculated)
-O2	128					
	256					
	512					
-O0	512					

d) Accuracy of the #I estimation: Compare the measured #I against the estimative from 1b). Comment the difference. Does it increases at the expected rate when the N value doubles?

e) #I vs CPI tradeoff: What is the gain obtained with the O2 optimisation level on a 512x512 matrix? What component of the performance equation is responsible for this gain? Why it is lower than expected (estimated in 1b)?

Exercise 2 - Instruction level parallelism and its limits

a) Look again at the previous table. Why is the CPI less that one in some cases? What is the ideal CPI on this machine?

b) Explain why the CPI value is lower (i.e., better!) with the -O0 optimisation level (there are two important factors; one of them is in the answer to the next question).

c) Look at the assembly of the inner loop of the code generated with -O2 level. In this code identify the Read After Write (RAW) data dependencies. (note: your generated assembly code should be similar to the following assembly code for double data type: SD is single double):

```
.L12:
     movsd (%rdx), %xmm0  ;move SD from memory to register %xmm0
     mulsd (%rax), %xmm0  ;move SD from memory and mul with %xmm0
     addq  $4096, %rax    ; add 4096 to %rax
     addq  $8, %rdx
     addsd %xmm0, %xmm1
     movsd %xmm1, (%rcx)
     cmpq  %rax, %rsi
     jne   .L12
```

d) Draw an instruction dependency graph (i.e., task dependency graph, each instruction is a task) where each box in an assembly instruction (e.g., primitive operation) and each arrow represents a data dependency (one of these instructions should be represented by two boxes, why?).

e) Look at the final graph, without further improvements, what is the effective CPI of this code on an ideal machine (e.g., a processing unit with infinite resources, and 1 cycle operations: memory load, add, multiplication, …) (suggestion: focus on the longest dependency chain).

f) (*) Suggest code improvements to increase the ILP available on this code and compute the potential gain of that code on an ideal machine by drawing a new dependency graph.

g) (*) Implement the optimised code and test it on small matrices (e.g., 128x128). Compare the performance with and without this optimisation.

Annex 1: (simple) Instructions for using the search cluster (in portuguese)

Nesta disciplina irá ser usado o cluster computacional SeARCH, mais especificamente dois nós com 20 núcleos computacionais, na partição "`-cpar`". Para executar o código deste módulo (e dos seguintes), deve aceder à máquina `s7edu.di.uminho.pt` por `ssh`, usando as credenciais recebidas por email. O código da multiplicação de matrizes pode ser executado num dos nós disponíveis com o comando `srun`.
Na página de *elearning* da disciplina está disponível uma implementação muito simples da multiplicação de matrizes que poderá ser usada nesta aula. O ficheiro pode ser copiado (com `scp`) para a máquina `s7edu`. A edição e compilação do código deve ser realizada na máquina `s7edu`, mas a execução deverá ser realizada num nó de computação da partição "`-cpar`", usando o comando `srun` já referido.

a) **Copy local file to remote machine :** `scp <local file name> <student_id>@s7edu.di.uminho.pt:`

b) **Login:** `ssh <student_id>@s7edu.di.uminho.pt`

c) **Load the gcc environment:** `module load gcc/9.3.0`

d) **Compile:** `gcc …`

e) **Run:** `srun --partition=cpar perf stat -e instructions,cycles <<full_path>>/a.out`

Annex 2: Matrix Multiplication Algorithms (in Portuguese)

(see https://en.wikipedia.org/wiki/Matrix_multiplication for more information)
Uma multiplicação de matrizes pode ser implementada de várias formas. Neste conjunto de módulos vamos utilizar, como base, uma implementação com três ciclos aninhados, começando pela variante mais comum, designada por ijk (posteriormente será identificada por DOT), para matrizes A, B e C quadradas com dimensão NxN:

```
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
    for(int k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j]; // nota: assume matriz C inicializada com 0s
```

Numa execução sequencial qualquer ordem dos ciclos i, j e k é correta, mas essa ordem origina diferenças significativas de desempenho que serão analisadas nas próximas aulas. Esta primeira variante (ijk) será designada por DOT, pelo facto de cada elemento da matriz C resultar do produto interno de uma linha da matriz A com uma coluna da matriz B, tal como ilustram a expressão/algoritmo e a figura seguintes:

$$C_{ij} = DOT_{linha\_A_i, coluna\_B_j} = \sum_{k=0}^{n-1} (A_{ik} * B_{kj})$$

```
Para cada linha de A
  Para cada coluna de B
    Clinha,coluna = DOTlinha de A,coluna de B
```