Lab Guide 3

Vector Processing and Roofline Model

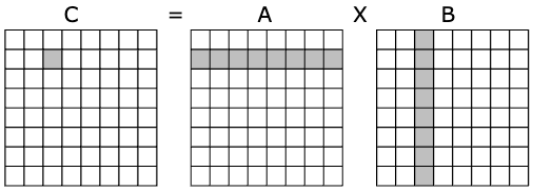
Objectives:

- Review of loop-unrolling optimisation
- Understand performance benefits and limitations of vector processing
- Use of the roofline model as a tool to identify hardware limitations and potential benefits of optimisations

Introduction

The peak performance of modern machines can only be attained when vector processing is explored. However, vector processing can only be applied to certain kinds of applications (e.g., data parallel) and requires *regular* data structures. Moreover, applications using vector processing consume more data per second, becoming more constrained by memory bandwidth. The roofline model is a visual tool that shows the peak performance and how memory bandwidth constrains the system's performance for a particular algorithm.

(*) Exercise 1 - Loop-Unrolling Review




```
/* Cij =0 */
for i=0 to N-1
      for j=0 to N-1
           for k=0 to N-1
                C[i][j] += A[i][k] * B[k][j]
```

a) (*) Estimation of the impact of loop unrolling: Loop unrolling can be enabled in `gcc` with the option `-funroll-loops`. Look at the assembly generated with this option and compare with the assembly of the base matrix multiplication (e.g., code in lab guide 1, exercise 2c). How many times was the loop unrolled? What is the estimated performance gain of this optimisation?

b) (*) Measure and discuss the results: Fill the table with performance data collected with `perf stat -e L1-dcache-load-misses -M cpi ./a.out`. Calculate the performance gain between *base* and *unroll* and comment the results.

N	Version	Time	CPI	#I (inst_retired.any)	L1_DMiss	Miss/#I
512	base()					
	unroll()					

Exercise 2 - Vectorisation

a) Identification of limitations**:** The compiler cannot vectorise the "base" DOT matrix multiplication code. Identify two reasons for this (suggestion: look at the matrix data access patterns in the DOT code, namely, reads from A; reads from B and writes to C).

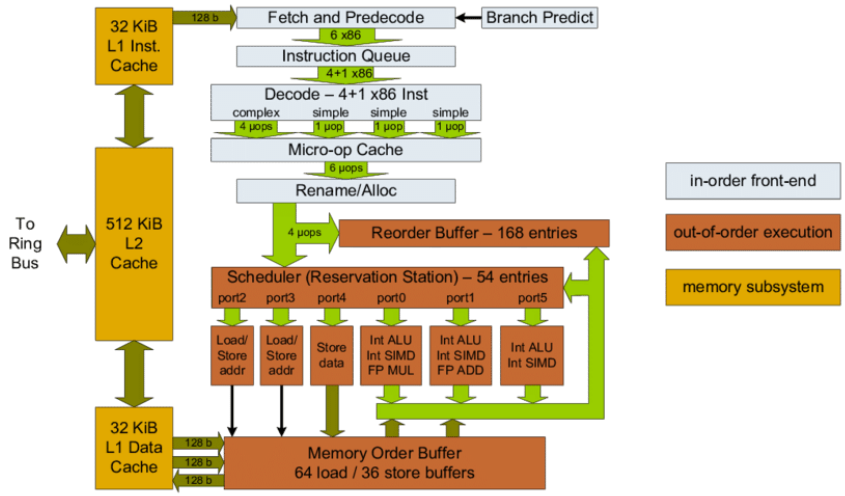
b) Vectorisation: The order of the loops can be changed without affecting the correctness of the results. Identify one (or more) order of the loops that support vectorisation.

c) Measure and discuss the results: Implement the version that you selected and compile the code adding the options `-ftree-vectorize -msse4` (the second flag generates SSE4 vector instructions which supports vector operations of 128bits). How many elements are processed at once? Estimate the number of instructions executed with and without vectorisation, by comparing the assembly with and without vectorisation. Complete the table and comment the results (is the performance gain what was expected? Why?).

d) (*) Vectorisation fine-tuning: In order to further improve vectorisation you can compile with `-mavx` to use vector operations on 256bit vectors. However to take advantage of this vector size the data should be aligned at 32-byte memory addresses (Why?). On way to enforce this alignment in `gcc` is to declare the vector statically with the aligned attribute (e.g., `double A[size*size] __attribute__ ((aligned (32)));` etc.).

N	Version	Time	CPI	#I (inst_retired.any)
512	base_v()			
	vect()			

Exercise 3 - Roofline Model



a) Peak Performance: Look at the figure above. How many operations can be performed at each cycle in this architecture? Identify, from the figure, constraints among operations that can be performed simultaneously. What is the floating point peak performance (with double data type), in GFlop per second, if the machine runs at 2.5 GHz and vector size is 256bits. Draw a graph with this GFlop/s in the Y-axis and plot a horizontal line representing this limit (suggestion: use a logarithmic scale with base 2).

b) Memory Bandwidth Limitation: Assuming a machine with a memory bandwidth 20 GB/s how many bytes can be used per floating point operation without being constrained by the memory bandwidth? Write this point in the X-axis. What happens if the application reads more bytes per floating point operation (e.g., two bytes per operation)? Plot a line in the graph showing this limit. The X-axis contains the number floating point operations per each byte read from memory, i.e., arithmetic (or operational) intensity .

c) Arithmetic Intensity: Look at the code of the `ikj` variant of the matrix multiplication. How many floating point operations are performed per byte read from memory ? Draw this point in the figure in order to estimate the attainable performance of this implementation.

```
/* Cij =0 */
for i=0 to N-1
     for k=0 to N-1
          for j=0 to N-1
              C[i][j] += A[i][k] * B[k][j]
```

d) (*) Draw the roofline model for the matrix multiplication execution on your personal computer (vectorised variant). You can use the STREAM benchmark the get the memory bandwidth (or get it from the machine's specification). You can also apply the *parallel* loop unrolling technique (see lab 2, exercise 2a)) to avoid the memory bottleneck and measure the performance gain on a large matrix (e.g., of 1024x1024 or larger).