Introduction

Part 1 - Architecture-Aware Programming

Four specially compiled binaries for matrix multiplication were provided. These all provided the same function but with various optimisations that leverage the specific architecture of the system, i.e. architecture-aware programming.

Four arguments can be provided to the program to optimise performance: square dimensions, if the second operand should be transposed or not, tile size and number of threads.

Transposing the second operand makes a difference since matrix multiplication computes the products of rows and columns. When reading an element, the system will presumably cache contiguous row data, but not column data. Transposing means that in both cases row data will be read, leading to fewer cache misses.

The dimension of the matrix will determine the row length, and the optimal size will most likely correspond to the various cache sizes. Due to the time needed for simulations, a dimension in excess of 256 bytes is impractical to model, however.

a)

Running version 1 with and without transposition yielded a significant speedup in the former case. Examining the stats files shows that the version without transposition has an order of magnitude more D-cache misses, just as expected. `gemm` version 2 has tiling capabilities and, assuming the tiles are sized so that they correspond to the cache line size in the L1 D-cache, we expect to see some performance gains. Running the experiments and looking at the CPI proves this assumption correct.

b)

The L1 and L2 caches were changed to 64kB and 256kB, respectively. Five cases for the tile size were observed, with CPI as the main metric.

As we can see, the CPI increases marginally as the tile size increases.

Table 1: The tile size for the `gemm` was varied in 5 different cases and the CPI observed. The L1 and L2 caches were changed to 64kB and 256kB, respectively.

Case	**CPI**
tile size<L1 cache	2,756
tile size=L1 cache	2,831
L1<tile size<L2	2,870
tile size=L2	2,897
tile size>L2	2,929

Part 2 - ARMv8 big.LITTLE Architecture

a) Roofline Model

First, we try and plot the bound given by the DRAM. According to the output `config.ini` file, the memory controller used in the ARMv8 experiment is of an an abstract type called `SimpleMemory`. As we did not parameterise this is in any script, examination of the source code[1] reveals that the default memory bandwidth is 12.8 GB/s, representative of a x64 DDR3-1600 channel.

Next, we give an estimated theoretical upper bound for the Flop/s of the system using a generic equation for multicore processors. That is, we ignore all latencies incurred by the memory subsystem.

Peak GFlop/s = CPU Frequency (GHz) x Core Count x Threads Per Core x 2 (for FMA[2] unit)

$\implies 2 \cdot 4 \cdot 1 \cdot 2 = 16$ GFlop/s

The values of the two memory-bound points were found in the next task and are presented in table 2.

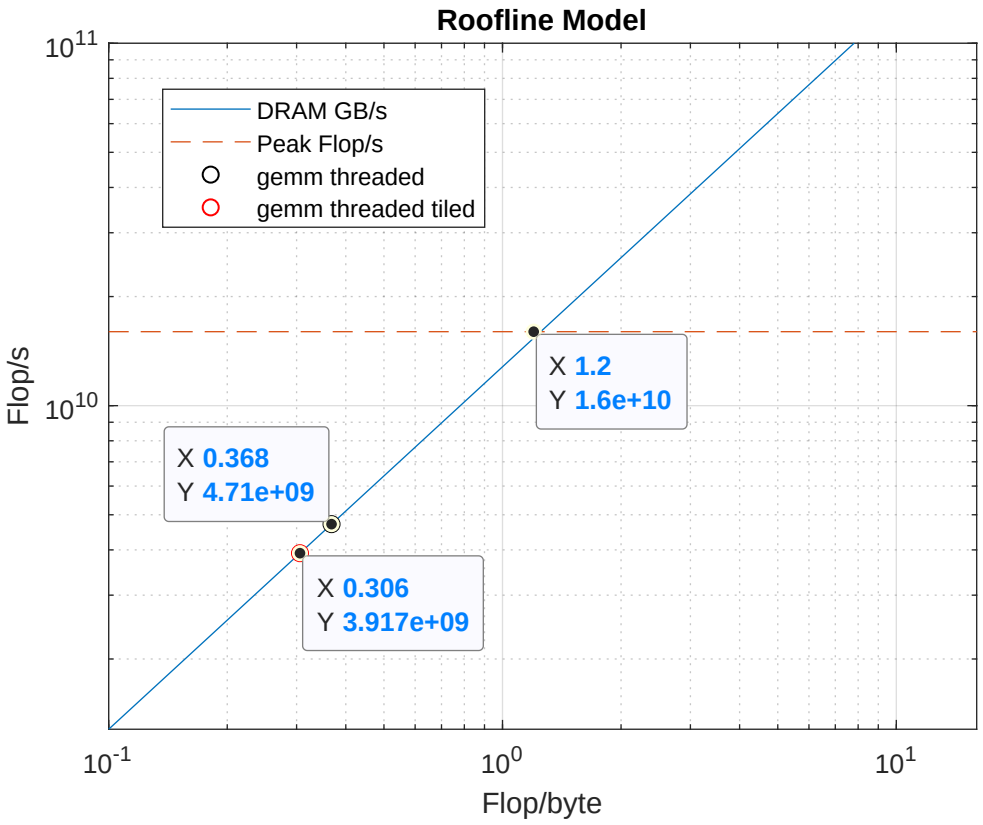

Figure 1: Roofline model for the `gemm` (general Matrix Multiplication) test on a full-system ARMv8 big.LITTLE gem5 simulation.

[1] `http://grok.gem5.org/source/xref/gem5/src/mem/SimpleMemory.py`
[2] Fused Multiply-Add

b) + c) Plot and Observations

In this experiment, versions 3 (threaded) and 4 (threaded and tiled) of the matrix multiplication (`gemm`) binaries from Part 1 were run in a full-system ARMv8 simulation.

Arithmetic Intensity (AI) is a measure of Flops per byte of DRAM traffic (reads and writes). To estimate the AI of the programs in question, the stats files were examined. The number of bytes read and written by the memory controller were found directly. The number of floating point operations was estimated using the `FloatMultAcc` parameter in the stats file. This was chosen as it is consistent across all four cores in the system, whereas various other floating point operation were only performed on a single core. The values can be seen in table. 2. The AI for the programs is plotted relative to the roofline model in fig. 1.

Not all DRAM traffic will be relevant to the floating point operations that we consider and so the AI estimate will be pessimistic. Additionally, according to the ARMv8 ISA, the `FloatMultAcc` instruction is fused, meaning that each such instruction performs (up to) two floating point operations. This is reflected in the AI result.

Table 2: Arithmetic Intensity for two versions of the `gemm` program.

Experiment	**DRAM Traffic**	**FMA Instructions**	**AI**
gemm_threaded	3213248	589824	0.368
gemm_threaded_tiled	3851008	589824	0,306