TP5 - Memory optimizations

Oguz Kaya, Atte Torri, Matthieu Robeyns
{oguz.kaya,atte.torri,matthieu.robeyns}@universite-paris-saclay.fr

You can consult the Intel's webpage for AVX functions as well as their latencies for each architecture: `http://software.intel.com/sites/landingpage/IntrinsicsGuide`
It would be sufficient to filter for AVX, AVX2 and FMA type instructions on the leftside of the page when searching for an instruction.

To compile the program `program.cpp` with OpenMP and AVX/AVX2 instruction sets and generate the executable `program`, type the following command in the terminal:

```
g++ -O2 -std=c++11 -fopenmp -mavx2 -mfma program.cpp -o program
```

Part 1

False sharing

False sharing is a phenomenon in multi-core cache-coherent architectures that arises when multiple threads modify different data elements belonging to the same `cache line` of 64 bytes. When this happens, even though there is no `race condition` hence no need to perform atomic operations, the performance degregates significantly. This is because when a thrad A modifies this cache line, the same cache line in another thread B gets invalidated. Thread B will then need to re-read the entire cache line next time it uses a data element in that cache line. This phenomenon is called `false sharing`.

The goal of this exercise is to observe the effects of false sharing on an example and analyze a potential solution. You should use the provided skeleton code `false-sharing.cpp`.

Ex. 1

a) The code given in `false-sharing` computes the sum of an array using multiple threads. Each thread has its local sum to compute in in `sum[thid]`. Execute the code for all values of number of threads (1, 2, 4, 8) and `STRIDE` (1, 2, 4, 8, 16, 32, 64) and put them in a matrix.

b) Does the sequential execution get faster with increasing `STRIDE`? Why, why not?

c) Does the parallel execution get faster with increasing `STRIDE`? Why, why not?

d) Starting from which size does `STRIDE` lose its effect? Why?

Part 2

Parallel matrix transposition

In this exercise, we will develop efficient parallel kernels for transposing a dense matrix using AVX and OpenMP parallelism. The transpose of a square $N \times N$ matrix A is a matrix B such that

$$B(i,j) = A(j,i)$$

for all $1 \leq i, j \leq N$.

We assume that N is sufficiently large ($N = 1024, 2048, 4096, \dots$)), a multiple of 32, and the datatype is `float`. The matrices are to be stored in a 1D array of size N^2 using a row-major storage (i.e., $A(i,j) =$ `A[i * N + j]`) You should implement your work in the provided skeleton code `mat-trans.cpp`.

Ex. 2

a) Implement a sequential and scalar matrix transposition that computes Measure the performance in GB/s transposed (an $N \times N$ matrix of floats takes $4N^2$ bytes).

b) Now implement the function `transAVX8x8_ps(_m256 &line[8])` that takes as input a 8×8 matrix stored line by line so that each line is in an AVX variable. The function should transpose this matrix, and overwrite `line[8]` with the transposed matrix. You should do this in three steps as explained in the class, using

- `_mm256_unpacklo_ps` and `_mm256_unpackhi_ps` in the first step
- `_mm256_shuffle_ps` in the second step
- `_mm256_permute2f128_ps` in the final step.

c) Now that you have the vectorized code to transpose a 8×8 matrix, implement a version that makes use of this to compute $B(i,j) = A(i,j)$. Measure the performance.

d) This time, implement an `in-place` transposition that computes the transposition on the matrix A directly, without using another matrix, i.e., $A(j,i) = A(i,j)$. To do this, you will need to load, transpose, and store two 8×8 tiles simultaneously. Measure the performance.

e) Finally, implement a tiled in-place version that transposes two tiles of $B \times B$ each time, where B is a multiple of 8. Each transposition should make multiple calls to `transAVX8x8_ps(...)`. Parallelize this version using OpenMP tasks, measure the performance, and find the optimal tile size B for your machine.