TP4 - Tiled Matrix-Matrix Multiplication

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

Benchmarking the machine

The goal of this exercise is to write a small code that benchmarks two major machine parameters regarding performance; namely peak floating-point operations per second (flops/s) and memory copy bandwidth in (bytes/s).

Ex. 1

a) Write a test program that measures the peak compute performance (flops/s) of a single core of your CPU in single precision (FP32) by calling multiple dummy FMA instructions in a sufficiently unrolled loop (to hide the loop overhead) that runs for a sufficiently long time. You can assing the FMA results into `volatile` variables to prevent the compiler from erasing all the meaningless FMA computation in the loop. Count the total number of operations performed, then divide by the time passed, and finally print the performance (in flops/s or Gflops/s). Next, find the theoretical peak performance of by hand, by using the formula:

$$peakCompute = frequency \times numVectorUnits \times vectorUnitFP32Size \times 2$$

where the final multiplier of 2 corresponds to the FMA support. You need to search these parameters for your processor model. Does the theoretical peak correspond to the measured peak?

b) Write a second test program that measures the peak memory copy bandwidth of your machine. To do this, you can first allocate two sufficiently `float` large arrays A and B aligned to 32 bytes, initialize A, then make an unrolled copy using AVX load/store instructions in an unrolled loop. Compute the amount of bytes copied while measuring the time passed, then print the copy bandwidth (in bytes/s or GB/s). Next, find the theoretical bandwidth of your memory using the formula:

$$peakBandwidth = (\#memoryChannels \times frequency \times 64 \text{ bits}/8) \text{ bytes/s}$$

where the $frequency$ is your DDR RAM frequency (e.g. for DDR4 2400, we perform 2400 megatransfers/s) and 64 bits (or 8 bytes) is the size of each individual memory transfer. Typically, if you have a laptop or standard PC, you will have two memory channels which effectively doubles the bandwidth thanks to parallel operation. Workstations and servers tend to have 4 to 8 memory channels per CPU socket. How much of a gap do you have between the theoretical and experimental bandwidth?

Part 2

Matrix-matrix multiplication with cache blocking

The goal of this exercise is to perform a fast multiplication of two square matrices $C = AB$ of size $A, B, C \in \mathbb{R}^{N \times N}$ such that

$$C(i,j) = \sum_{k=1}^{N} A(i,k)B(k,j).$$

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]`)

Ex. 2

a) Implement a sequential matrix-matrix multiplication that computes $C = AB$ using three nested loops in order $i \to j \to k$. Measure the performance in $Gflops/s$ (an $N \times N$ matrix-matrix multiplication takes $2N^2(N-1)$ flops).

b) Implement a second version that rather uses the loop order $i \to k \to j$. Does it go faster? Why?

c) Implement a third version that uses tiling, with tiles of size $B_1 \times B_1$ where B_1 is a small constant and a power of 2. The goal is to have a sufficiently small tile size so that all $B_1 \times B_1$ elements of A, B, C fits into the L1 cache and effectively reused, yielding much better memory performance. This time, you will have six loops, first three iterating over tile indices of size N/B_1 each, the next three will iterate within each tile with a loop variable of size B_1. Verify the computation using the version without tiling. Test the code performance for all power-of-two tile sizes ($B_1 = 1, 2, 4, 8, 16, \ldots, N$). Which one gives the best performance?

d) Implement a fourth version that uses double tiling, one tiling of size B_1 for the L1 cache, and another tiling of size $B_2 = cB_1$ with c being a positive integer for the L2/L3 cache. You will need 9 nested loops for this version; first three iterating over L2/L3 tile indices, the next three iterating over L1 tile indices, and the final three iterating within the tile and performing the multiplication.

e) Vectorize the computation in the innermost three loops by employing a vectorized matrix-matrix multiplication kernel of size 8×8 using AVX2. To do this, you will do the computation of each line of a tile of C as follows:

$$C_{tile}(i,:) = \sum_{k=0}^{7} A_{tile}(i,k) * B_{tile}(k,:)$$

To do this efficiently, you need to first load all 8 lines of B_{tile} into 8 separate vector variables with `_mm256_load_ps(...)`. Then, for each value of i, you should first load the line i of C_{tile} into a vector to get the previous value, then for each k broadcast the value $A_{tile}(i,k)$ into a vector using `_mm256_broadcast_ss (float const * mem_addr)`. Finally, you should perform an FMA (`_mm256_fmadd_ps(...)`) using this vector and the vector containing line k of B_{tile} with accumulate over the vector containing the line i of C_{tile}. Note that for computing the next line $i+1$ of C_{tile}, you should not need to reload the matrix B_{tile}. In each B_1 tile, you need to call this vectorized kernel $(B_1/8)^2$ times. Make sure to rearrange the instructions within the vectorized kernel to hide the instruction latency as much as possible (e.g., by unrolling loops interleaving instructions with no dependencies for example).

f) Finally, parallelize this second version by adding OpenMP task parallelism, by having a single thread that iterates over tiles to generate tile multiplications with dependencies, and available threads executing those tasks. Once this works, parallelize the generation of tasks as well, by executing the k-loop on all threads and using `omp for collapse(2)` on the following i and j loops. This is because for each value of k for tile multiplication $C_{tile}(i,j) += A_{tile}(i,k) * B_{tile}(k,j)$ are independent, hence the corresponding tasks can be created in parallel.

g) Display your final speedup with respect to the first naive version of the code. Display your final performance (Gflops/s) for increasing values of N. What percentage of your machine's peak performance are you able to achieve at maximum?