Pontificia Universidad JAVERIANA Cali

Facultad de Ingeniería y Ciencias
Carrera de Ingeniería de
Sistemas y Computación

PROGRAMACIÓN PARALELA
Matrix-Matrix Multiplication Tutorial

A matrix is a rectangular two-dimensional array of numbers. Matrix operations are key to many important HPC applications. Besides, many important applications can be “reduced” to operations on matrices, including (but not limited to) searching and sorting, numerical simulation of physical processes, and optimisation.

It is natural to use a 2D array to store a dense or banded matrix. C (and C++, Java, Python) stores matrix elements using a row-major (also known as row-wise) pattern. Thus, `M[row][col]` and `M[row][col+1]` are adjacent in memory:

The matrix

0	1	2	3
4	5	6	7
8	9	10	11

will be stored in memory as

0	1	2	3	4	5	6	7	8	9	10	11

The *stride* between adjacent elements in the same row is 1. The stride between adjacent elements in the same column is the row length (4 in this example).

In contrast to this, Fortran stores 2D arrays in column-major form. Many important numerical libraries (e.g. LAPACK) are written in Fortran, however. Hence, to use them in C, we need to work with a transposed matrix.

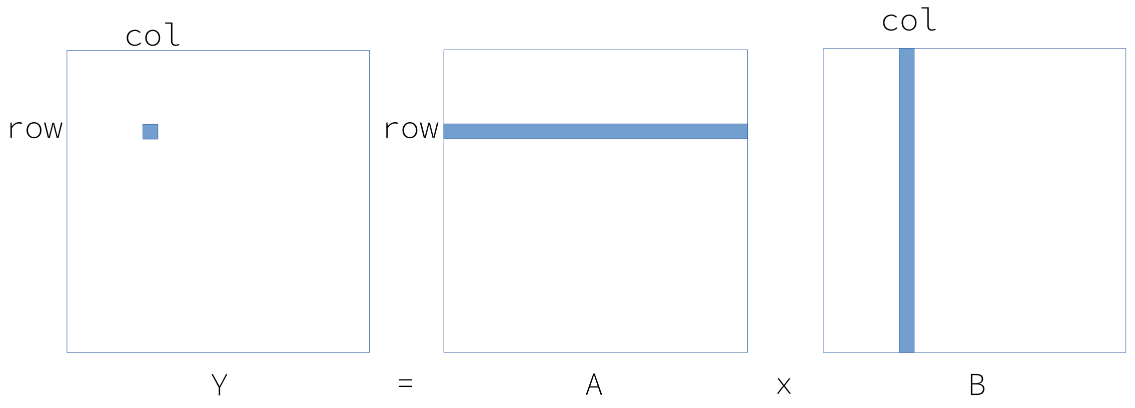
In the accompanying archive, there is a compressed directory called `Lab9`. Once the archive is uncompressed change to that directory. Inside it there is one directory: `06-MatMatMult`. Inside it there are three directories: `0-C`; `1-OpenMP`; `2-CUDA`. We will start with the `0-C` directory.

The `MatMatMult0.c` program multiplies two matrices (`matA` and `matB`) of size `NxN`. The resulting matrix is `matY` and is set to `0`. Notice that when passing an array as a function parameter, the size of the last dimension must be used. The `MatrixMatrixMult` function can be used to multiply two square matrices of any size (as given by the `N` manifest constant). The approach used here is to calculate the dot product as per the expression `MY [row][col] += MA [row][i] * MB [i][col]`, where `i` goes from `0` to `N-1`, for each column of every row of `MY`. The rationale behind the use of the intermediate variables `a`, `b`, `sum`, is to reinforce the fact that the accumulation is faster when the processor uses registers. An optimising compiler however should be able to detect this.

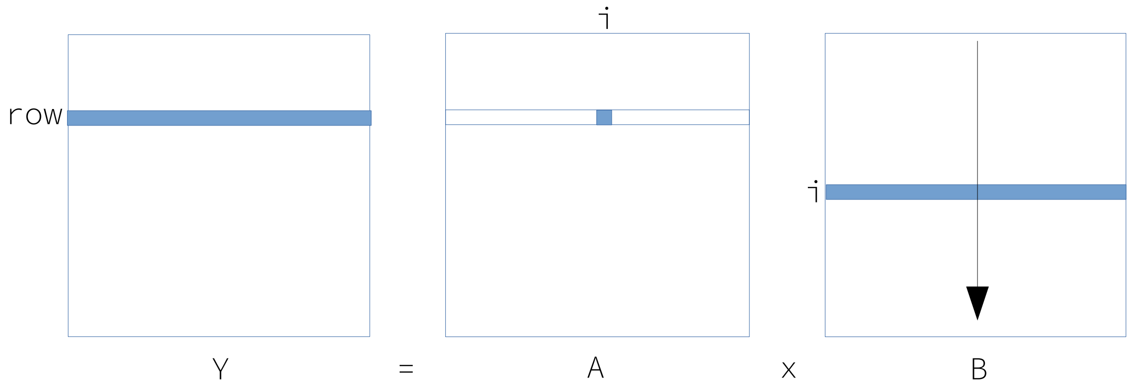
The `MatMatMult1.c` program just maps the 2D-array notation to the corresponding 1D-array. This mapping allows us to use pointers as arguments to the functions.

Memory access patterns can have a dramatic impact on performance, especially on contemporary computers with a complicated memory hierarchy. This is the purpose of the following programs. `MatMatMult2.c` and `MatMatMult3.c` are identical except that the order of the two inner loops is swapped as shown below. These code segments access the same elements of an array, but the order of accesses is different:

```
for row = 1..N
  for col = 1..N
    for i = 1..N
       Y[row,col] += A[row,i] * B[i,col]
```



```
for row = 1..N
  for i = 1..N
    for col = 1..N
       Y[row,col] += A[row,i] * B[i,col]
```



For square matrices of size `NxN` both programs require $2N^3$ operations. In the first case, matrix `A` is accessed row-by-row but matrix `B` is accessed column-by-column, which leads to different spatial and temporal locality properties.

In `MatMatMult4.c` and `MatMatMult5.c` we switch to dynamically allocated memory, use of pointers, mapping from 2D to 1D, and matrices storing `double` precision elements with values ranging between `-1.0` and `1.0`. Notice that the `printf` messages are used only if the matrix sizes are at most `32x32`. You can compare the results of both programs by redirecting each output to a file:

```
MatMatMult4 > out4
MatMatMult5 > out5
```

and then applying `diff`:

```
diff out4 out5
```

`MatMatMult6.c` and `MatMatMult7.c` work with bigger matrices so that we can play with matrices that will not fit inside different cache memory levels, so we can compare execution times. Notice that the value passed in the command line is the square root of the size (in Ki) you want to use. To collect some results, you can use the `collect_data.sh` shell script. It just sweeps through a range of matrix dimensions (via `s`) and then generates an output file for each variant (via `variants`). The nomenclature of the output file name corresponds to `MMMv-hname-proc.out`, where `MMMv` corresponds to the executed program (`v` denotes the program name suffix; i.e., `6` or `7`), `hname` corresponds to the name of the host where the program ran; and, `proc` denotes the processor used. Each entry in the output file has the matrix size and its corresponding execution time using a `.csv` format.

Finally, `MatMatMult8.c` and `MatMatMult9.c` illustrate a *tiled* (blocked) approach to do the matrix multiplication. Both need two arguments: the matrix side, `N`, and the tile (block) side `B`. Both assume that the matrices are `NxN` in size and that `N` is divisible by `B`.

The `1-OpenMP` directory has two programs: `MatMatMult0.c` and `MatMatMult1.c`. The first one uses two independent loops while the second one uses nested loops. Is there a better way to write the code?

The `2-CUDA` directory has two programs: `MatMatMult_threads0.cu` and `MatMatMult_threads1.cu`.

The first one uses `1` block and `NxN` threads; each thread computes one element of the block sub-matrix; that is, the corresponding dot product. Note that now we need to calculate two indices, one in each dimension (`x`, `y`) and the `blockId`, `blockDim`, and `threadIdx` identifiers in the expression to denote the indices. Notice also that in this case, we use the `dim3` type to set the configuration parameters required for kernel launching. The `dim3` type is CUDA's way to specify dimensions (`x`, `y`, and `z`); any component left unspecified is initialised to `1`.

The second program attempts to generalise the computation, so that the total amount of threads per dimension as given by `MAX_THREADS` will not be exceeded.