Exercises for the Lecture: "Architecture and Programming Models for GPUs and Coprocessors" Exercise Sheet № 1

Priv.-Doz. Dr. rer. nat. Stefan Zellmann

WS 2021/22

1 Cache-Aware Programming

Ex. 1.1

Given the matrix A with:

$$A_{i,j} = (i, j), i, j = 0, 1, .., N - 1. \tag{1}$$

Visualize A for $N = 8$ using a table. Given the binary representation of the matrix elements, perform the *in shuffle* operation for all pairs of numbers (i, j):

$$k(i, j) = (k_5, .., k_2, k_1, k_0) := (i_2, j_2, .., i_0, j_0). \tag{2}$$

The binary representation of the resulting integer k is an alternating sequence of the bit representation of the pair (i, j). In the table of size 8×8, enter the result in both their binary and decimal form. Connect the table entries with a polyline and in ascending order.

Ex. 1.2

a.)
Compile the file `morton.cpp` that accompanies this exercise sheet with a C++11 compatible compiler. Configure the compiler to use the highest optimization level w.r.t. the expected program execution speed. With `gcc` that can e.g. be achieved like this:

```
g++ morton.cpp -std=c++11 -O3 -o morton
```

The template class `Grid` implements a 2D dynamic array whose elements can be accessed using `operator()`:

```
Grid<float> grid(W, H); // 2D array of size W x H
grid(1, 2) = 3.14f; // write to array element [1,2]
```

The function `main()` will first generate a 2D floating point array and fills it with random numbers. It will then run a so called image processing filter on the array. Filter application consists of a row-wise iteration, followed by a column-wise iteration through the array.

Execute the program that you just compiled. What are your observations? Do you have an explanation for the behavior that you can observe?

b.)
Extend the class `Grid` with an alternative implementation of `operator()`. You can switch between the existing and your new implementation using the already defined compile time flag `MORTON`. For your implementation you will use the *space-filling curves* that we derived in Ex. 1.1. Implement a helper function `expand_bits()`. This will allow you to perform the *in shuffle* operation:

```
int z = expand_bits(x) | (expand_bits(y) << 1)
```

The resulting *"Morton Codes"* can be used as a (linear) index into the data array maintained by the `Grid` class. `expand_bits()` will successively pull the x/y coordinates apart using bit operations and masking:

1. `x = --------.--------.FEDCBA98.76543210`
2. `x = --------.FEDCBA98.--------.76543210`
3. `x = ----FEDC.----BA98.----7654.----3210`
4. `x = --FE--DC.--BA--98.--76--54.--32--10`
5. `x = -F-E-D-C.-B-A-9-8.-7-6-5-4.-3-2-1-0`

(Assume that the − bits are 0-valued, that will allow you to perform the *in shuffle* operation using bitwise `or`.)

Execute the modified program and compare the execution times of the two implementations. What do you observe and what is your explanation?

c.)
The previous part of the exercise was comprised of a cache optimization for 2D data structures. In the particular case, can you imagine a more pragmatic solution to speed up the program execution? (Note: you are allowed to change the order of the data items that are stored in memory.) Can you still imagine circumstances where the cache optimization from before is beneficial? (That might e.g. have to do with the memory access patterns that come to use.)

The exercise sheet will be discussed on Nov 10, 2021.