Progetto ACA – CUDA Kernel Implementation

- Ghitti Marco

Project Description

- The algorithm search for the best rotation for a structure of 64 atoms by evaluating the score of all the possible rotations, with a given precision, and select the rotation with higher score.
- Given a rotation its score is the sum of the scores of the rotated atoms inside the poket.
- The goal of the project is to implement a CUDA kernel to accelerate the algorithm by exploiting data parallelism.

CUDA Kernel implementation

The Kernel can be divided into 4 steps:

1. Calculate the score of all atoms for every considered rotation.
2. Find the score of each rotation by summing the scores of all 64 atoms.
3. Find the rotation with higher score value.
4. Calculate the output atoms by appling the found rotation.

CUDA Kernel implementation

Score of each atom

- For each atom we calculate and apply the rotation matrix before selecting it’s score inside the poket
- The rotation matrix is calculated multiple times (once for each atom), that has been done to avoid the overhead of memory synchronization and multiple kernel launches.
- The score of the atom is saved in a __shared__ array of float ready for the sum reduction.

CUDA Kernel implementation

Scores sum reduction

- To efficently reduce the time required to sum the score of all 64 atoms we can perform a parallel reduction.
- Parallel reuctions are efficient ways to exploit SIMD instructions to reduce the number of required operations from O(n) to O(log(n)) for certain types of linear algorithms (such as find the maximum value or sum of big arrays).
- Since we already know that the number of atoms is 64 we can use 32 threads to perform 5 add with different offsets to assign the sum of all the elements of the array in its first position

CUDA Kernel implementation

Scores max reduction

- The max reduction find the rotation with the highest overall score value.
- Use the same idea of the sum reduction but introduce the blockSize as a template unsigned int value; that allow to completely unroll the reduction at compile time and still have a variable block size.

CUDA Kernel implementation

Output atoms

- The output atoms are calculated with the same method used for the first step: To avoid memory overhead we calculate the rotation matrix with higher score multiple times (once for each atom) and we apply the rotation to the single atom.
- The output atoms are saved in global memory ready to be retrieved from the device memory as the result.

Optimizations

Contant memory

- The number of rotations to be considered and the precision of the rotations need to be used multiple times and do not change during the execution of the kernel
- These constants can be saved in constant memory to allow the device to optimize the access by exploiting aggressive caching and allow the compiler to optimize the code by consider those values as constants.

Optimizations

Poket score access

- The access to the poket score value require three clamp operations (one for each dimension) and one non coalesced memory access.
- The clamp operation can be obtained by combining min and max operations.
- A faster way to access the poket score is to exploit a 3D texture which use a read only memory area and use hardware to automatically perform the clamp operations.

Optimizations

Poket score access

- Texture memory use a dedicated memory area that is separated from global memory.
- Texture memory is also cached which reduces the non coaleshed access problem.
- Even if the clamp operation use min and max hardware operations the usage of texture memory improve the execution time of the kernel since the texture can perform the operation by only using hardware.

Optimizations

Parallel reductions

- The reductions use sequential adressing to reduce bank conflicts.
- When the number of elements to be reduced is lower than the warp size we use a warp reduce function which allow to avoid the usage of the __syncthreads operation that is useless inside the same warp.
- The reductions are completely unrolled, that allow the compiler to better optimize the code.