POLITECNICO
MILANO 1863

COMPUTER SCIENCE AND ENGINEERING

COURSE ON "ADVANCED COMPUTER ARCHITECTURES"

A.Y. 2018/19

CUDA Kernel Implementation for a Chemical Engineering application

Giacomo Astolfi
Computer Science & Engineering MSc student

Marco Fringuelli
Computer Science & Engineering MSc student

ABSTRACT

In this **presentation**, after a brief discussion about the **domain problem** and the **algorithm** to be optimized, we show the technical specifications of the technology we have used. After that, we list the **CUDA C programming constructs** we considered and those advantages that the CUDA architecture provided us. We then describe the **logic** of our optimization, along with the headers of the **implemented kernels**. In conclusion, we show the **output** of our code with its **benchmarks**.

DOMAIN PROBLEM

A molecule is composed of **`N_FRAGS`** fragments (corresponding to as many degrees of freedom) each of them composed of **`N_ATOMS`** atoms, whose coordinates are in the 3D euclidean space. The objective of the algorithm is to find the optimal shape for the molecule given a 3D pocket (whose size is **`VOLUMESIZE`**) determining the shotguns for each atom position and a mask (whose size is **`MASKSIZE`**) determining which atom is occupying a certain position in the space.

The goal of our project is to *heavily* optimize the execution of this algorithm solving this problem.

THE ALGORITHM

DATA STRUCTURES

- ***float* in*** - *initialization of the atom positions (of size* INSIZE*)*
- ***float* out*** - *a data structure where to put the final result*
- ***float precision*** - *describes how many angles are to be evaluated*
- ***float* score_pos*** - *a data structure for the 3D pocket*
- ***int* start*** - *a data structure describing the extreme points of each fragment*
- ***int* stop*** - *a data structure describing the extreme points of each fragment*
- ***int* mask*** - *a data structure for the 3D mask*

MAIN FUNCTIONS

- **void ps_check(…)** - *implementation of the algorithm for CPU execution*
- **void ps_kern(…)** - *implementation of the algorithm for GPU execution*

USED TECHNOLOGIES & SPECS

By using the **NVIDIA CUDA®** platform, we are able to use parallel computing in solving the problem. So, we used a CUDA-enabled general purpose GPU, the **NVIDIA GEFORCE GTX 970®** (logging into the host via SSH).

The GPU we have used has the following technical properties (obtainable by properly calling **`cudaGetDeviceProperties(&prop)`**)

```
 --- General Information for device 0 ---
Name: GeForce GTX 970
Compute capability: 5.2
Clock rate: 1215500
Device copy overlap: Enabled
Kernel execition timeout : Enabled
   --- Memory Information for device 0 ---
Total global mem:  4238999552
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  13
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

USED CUDA PROGRAMMING CONSTRUCTS

The **CUDA API** allows to specify inline which functions (**kernels**) are to be executed on the GPU (**device**). The computation can be organized into **blocks** where one or more **threads** perform, possibly cooperatively, all the operations needed on the data structures stored in the **device memory**.

Below we list all the CUDA constructs we referred to when implementing the kernels and the device functions used inside **`ps_kern`**.

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

- a simple example of device execution environment

- **CUDA Kernels.** By adding the qualifier `__global__` to a function we specify that that function (which becomes a kernel) is to be executed on the device. When calling it, we pass the runtime parameters to the device with the angle brackets `<<< , >>>`. The threads within a block are synchronized via calling `__syncthreads()`. (example: `eval_angles<<<1,ceil(MAX_ANGLE/precision),0,s1>>>`).

- **CUDA Device functions.** By adding the qualifier `__device__` to a function we specify that that function is to be executed on the device (without parallelism).

- **Global Memory.** A read-write memory in the GPU concurrently accessible by all the threads of any block. Space in this memory is allocated via calling `cudaMalloc(…)` from the host and data structures are transferred via calling `cudaMemcpy(…)`. When the work is done, the space must be freed with `cudaFree()`.

- **Shared Memory.** A read-write memory in the GPU concurrently accessible by all the threads of one block, allowing thread cooperation. Each block will have its own copy of the data allocated in the shared memory (also called cache). To specify that a data structure has to be in shared memory, we add the qualifier `__shared__`.

- **Texture Memory.** A read-only cached on chip memory devised to provide higher effective bandwidth when there are many reads and the memory access patterns exhibit a great deal of spatial locality. It is needed to declare inputs as texture reference before using them: example: **texture<int, 1, cudaReadModeElementType> texMask**. After that, using Texture Objects (created with **cudaCreateTextureObject(…)**) it is possible to pass texture memory-referenced stored data structures to functions and kernel as if they were pointers to global memory (type **cudaTextureObject_t**). Reading from texture memory must be performed via calling **tex1Dfetch<type>(array,index)** (Also **tex2Dfetch** and **tex3Dfetch** exist to read from higher dimensionality data structures). As explained below, with texture objects it is also possible to create ad hoc texture data structures that can have different addressing modes.

- **CUDA Streams.** A stream is a sequence of operations that execute in issue-order on the GPU. By identifying which kernels perform independent loads of computation (no hazards) we can make them run concurrently on different streams (default stream is 0). Asynchronous streams are created via **cudaStreamCreate(…)**, and passed as 4th runtime parameter inside the kernel call angle brackets (e.g. **<<<blocks,threads,0,stream1>>>**) . To synchronize, **cudaStreamSynchronize(…)** must be called before and after the kernel call.

- **CUDA Warps.** At Hardware level, a GPU executes groups of 32 parallel threads known as *warps* in a SIMT (Single Instruction, Multiple Threads) fashion. By using, software-side, warp-level primitives, allowing to organize thread operations per warps, higher performance could be achieved. By knowing the value of **`warpSize`** for the used GPU and then setting, for each thread, the parameters **`wid`** (warp ID) and **`lane`** (thread ID in its warp) reductions can be performed warp-wise using primitives like **`__shfl_down_sync(…)`**.

- **Timing.** It is possible to check the execution times of specific fragments of code by creating two objects **`cudaEvent_t`** (start and stop) and then properly calling **`cudaEventRecord(…)`** on each of them.

OUR OPTIMIZATION

After having understood the **logic** of the algorithm, we passed to “translate” the C++ code we were given into CUDA code, with **kernels** performing massive parallel work and efficiently using **device memory**.

We list below the kernels we implemented, explaining the way they are to improve performances. Finally, we show the output of the CUDA implementation of the algorithm along with the profiling results got with **`nvprof`**.

KERNELS

- **`__global__ void rotate(float* in, cudaTextureObject_t mask, int iter, float precision, int* start, int* stop) {…}`** This kernel performs the rotation of one molecule fragment (given by "iter"), whose atoms coordinates are in the "in" array and its extreme points are in the "start" and "stop" arrays.
  REPLACES: **`inline void rotate( float* in, int* mask, const free_rotation::value_type &rotation_matrix)`**.
  PARALLELISM: Number of blocks = Number of angles; Number of threads per block = Numbers of atoms per fragment.
  STREAM: Stream 1.

- **`__global__ void measure_shotgun(float* in, cudaTextureObject_t scores, int* shotgun, float precision, int iter) {…}`** This kernel computes the fragment ("iter") shotgun for a given angle of rotation, referring to the score grid "scores" and the atom coordinates contained in "in".
  REPLACES: **`int measure_shotgun (float* atoms, float* pocket) {…}`**
  PARALLELISM: Number of blocks = Number of angles; Number of threads per block = Numbers of atoms per fragment.
  STREAM: Stream 2.

- **`__global__ void fragment_is_bumping(float* in, cudaTextureObject_t mask, int* is_bumping_p, int iter, float precision, int* is_bumping) {…}`** This kernel, for the fragment "iter", checks whether its considered configuration is legal, that is, if it is bumping with parts of other fragments of the molecule. The "mask" data structure allows to determine which atom occupies which position (if any), while the arrays "is_bumping_p" and "is_bumping" serve as boolean masks in the computation to store partial bumping results (is bumping? yes/no).
  REPLACES: **`inline bool fragment_is_bumping( const float* in, const int* mask) {…}`**
  PARALLELISM: Number of blocks = **`bump_blocks`**; Number of threads per block = Numbers of atoms per fragment.
  STREAM: Stream 1.

- **`__global__ void eval_angles(float* in, int* shotgun, int* bumping) {…}`** This kernel serves to perform angle evaluation in parallel, starting from fragment atom positions contained in "in" and updating the values of "shotgun" and "bumping".

  REPLACES: **`for ( int j = 0 ; j < 256; j += precision ) {…}`**.

  PARALLELISM: Number of blocks = **1**; Number of threads per block = Numbers of angles.

  STREAM: Stream 1.

DEVICE FUNCTIONS

These are the functions called inside the kernels to perform the necessary computations on the data structures. **`__inline__`** serves to force the compiler to inline the functions code avoiding stack calls.

- **`__inline__ __device__ int warpReduce(int val) {…}`**. This function serves to sum all the different values of the thread "val" variables (one copy per thread) belonging to the same warp. **`__shfl_down_sync(…)`** allows to perform this reduction in a recursive fashion.

- **`__inline__ __device__ int blockReduce(int val) {…}`**. This function serves to sum all the different values of the thread "val" variables (one copy per thread) belonging to the same block. **`warpReduce(val)`** is therefore called once per warp, and the outputs of it are then summed using a shared cache.

- **`__device__ void compute_matrix( const int rotation_angle, const float x_orig, const float y_orig, const float z_orig, const float x_vector, const float y_vector, const float z_vector, float* matrix){…}`**. This function is totally identical to that used in the CPU implementation of the algorithm, computing the rotation matrix given a certain angle, all the necessary points and data structures.

- **`__inline__ __device__ void warpReduce(int ind, int sho, int bum, int &ret1, int &ret2, int &ret3) {…}`**. Given a warp of threads, this function serves to find the best rotation angle for a fragment, that is, the one with the highest shotgun value among the non-bumping ones. The results for each angle are stored in the variables "ind" (which angle is this), "sho" (the shotgun) and "bum" (is it bumping? yes/no). The reduction is performed symmetrically on the three groups of values with **`__shfl_down_sync(…)`**.

- **`__inline__ __device__ int find_best(int* shotgun, int* bumping, int index){…}`**. This function uses the computations of **`warpReduce(…)`** on each warp to get the block overall result, therefore giving the final result. In this case, three shared caches are used to store partial shotgun, bumping and best_angle results.

TEXTURE OBJECTS

Texture memory can bring an important speedup when dealing with large data structures accessed in a "chain" pattern by the threads of a block. So we used **texture objects**, that are directly passable to functions and kernels as arguments of type **`cudaTextureObject_t`**.

Objects are initialized with a **resource descriptor** and a **texture descriptor**, which describes their properties and access modes. Via calling **`cudaCreateTextureObject(…)`** the object is finally created.

The two texture objects we use are:

- **`texScore_pos`** - *texture memory version of "score_pos": a 3D object that is accessed when measuring the shotgun of a rotation angle. We use* `cudaAddressModeClamp` *In order to perform hardware-side the boundaries control so to get the best efficiency*

- **`texMask`** - *texture memory version of "mask": an 1D object that is accessed inside "rotate(…)" and "fragment_is_bumping(…)"*

CODE OUTPUT

Here is a chunk of the output of the **GPU** implementation (first column) compared to that of the **CPU** implementation (second column)

```
Kernels executed in 0.660288 milliseconds
best angle is: 184
best angle is: 212
best angle is: 213
best angle is: 27
0:      16.2419 16.2418    | 64:   20.2790 20.2788    | 128:  30.4126 30.4126
1:      16.6925 16.6924    | 65:   20.2761 20.2759    | 129:  29.5199 29.5198
2:      17.1432 17.1430    | 66:   20.2732 20.2730    | 130:  28.6272 28.6271
3:      17.5938 17.5936    | 67:   20.2702 20.2700    | 131:  27.7345 27.7345
4:      18.0444 18.0443    | 68:   20.2673 20.2671    | 132:  26.8418 26.8417
5:      18.4950 18.4949    | 69:   20.2643 20.2641    | 133:  25.9491 25.9490
6:      18.9457 18.9455    | 70:   20.2614 20.2612    | 134:  25.0564 25.0563
7:      19.3963 19.3961    | 71:   20.2585 20.2582    | 135:  24.1637 24.1636
8:      19.8469 19.8467    | 72:   20.2555 20.2553    | 136:  23.2709 23.2709
9:      20.2975 20.2973    | 73:   20.2526 20.2524    | 137:  22.3782 22.3782
10:     20.7482 20.7479    | 74:   20.2497 20.2494    | 138:  21.4855 21.4855
11:     21.1988 21.1985    | 75:   20.2467 20.2465    | 139:  20.5928 20.5928
12:     21.6494 21.6491    | 76:   20.2438 20.2434    | 140:  19.7001 19.7000
13:     22.1000 22.0997    | 77:   20.2408 20.2405    | 141:  18.8074 18.8073
14:     22.5506 22.5503    | 78:   20.2379 20.2375    | 142:  17.9147 17.9146
15:     23.0013 23.0011    | 79:   20.2350 20.2348    | 143:  17.0220 17.0219
16:     24.6089 24.6087    | 80:   21.7132 21.7129    | 144:  16.8204 16.8203
17:     25.2632 25.2628    | 81:   22.2875 22.2870    | 145:  16.3285 16.3284
18:     25.9175 25.9172    | 82:   22.8618 22.8614    | 146:  15.8366 15.8364
19:     26.5718 26.5715    | 83:   23.4361 23.4357    | 147:  15.3446 15.3445
20:     27.2262 27.2258    | 84:   24.0105 24.0100    | 148:  14.8527 14.8525
21:     27.8805 27.8801    | 85:   24.5848 24.5844    | 149:  14.3607 14.3607
22:     28.5348 28.5345    | 86:   25.1591 25.1587    | 150:  13.8688 13.8687
23:     29.1891 29.1888    | 87:   25.7335 25.7330    | 151:  13.3769 13.3768
24:     29.8435 29.8431    | 88:   26.3078 26.3073    | 152:  12.8849 12.8848
25:     30.4978 30.4974    | 89:   26.8821 26.8816    | 153:  12.3930 12.3929
```

PROFILING RESULTS (WITH NVPROF)

```
==10784== Profiling application: ./hellocuda
==10784== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.02%  1.3174ms         5  263.49us     672ns  1.3142ms  [CUDA memcpy HtoD]
                   27.73%  561.81us         4  140.45us  140.00us  141.18us  fragment_is_bumping(float*, __int64, int*, int, float, int*)
                    3.01%  60.959us         1  60.959us  60.959us  60.959us  [CUDA memcpy DtoA]
                    2.19%  44.288us         4  11.072us  9.9840us  14.112us  rotate(float*, __int64, int, float, int*, int*)
                    1.37%  27.680us         4  6.9200us  6.6240us  7.1040us  measure_shotgun(float*, __int64, int*, float, int)
                    0.60%  12.192us         4  3.0480us  2.9440us  3.3280us  eval_angles(float*, int*, int*)
                    0.09%  1.9200us         1  1.9200us  1.9200us  1.9200us  [CUDA memcpy DtoH]
      API calls:   95.21%  75.502ms         8  9.4377ms  2.4010us  75.413ms  cudaMalloc
                    1.27%  1.0040ms         6  167.33us  3.0750us  967.50us  cudaMemcpy
                    0.99%  787.09us         4  196.77us  190.09us  202.01us  cudaGetDeviceProperties
                    0.82%  650.54us        13  50.041us     857ns  138.90us  cudaStreamSynchronize
                    0.64%  505.39us         1  505.39us  505.39us  505.39us  cudaMalloc3DArray
                    0.42%  336.91us         1  336.91us  336.91us  336.91us  cuDeviceTotalMem
                    0.30%  239.06us        97  2.4640us     291ns  100.58us  cuDeviceGetAttribute
                    0.12%  91.942us        16  5.7460us  4.5400us  13.457us  cudaLaunchKernel
                    0.06%  48.551us         1  48.551us  48.551us  48.551us  cuDeviceGetName
                    0.05%  37.427us         1  37.427us  37.427us  37.427us  cudaMemcpy3D
                    0.04%  30.859us         6  5.1430us  3.4820us  9.4730us  cudaFree
                    0.02%  14.034us         2  7.0170us     830ns  13.204us  cudaDestroyTextureObject
                    0.01%  8.7300us         2  4.3650us  1.2950us  7.4350us  cudaStreamCreate
                    0.01%  8.4180us         2  4.2090us  2.0200us  6.3980us  cudaCreateTextureObject
                    0.01%  6.2890us         2  3.1440us  1.6330us  4.6560us  cudaStreamDestroy
                    0.01%  5.3730us         2  2.6860us  1.8620us  3.5110us  cudaEventRecord
                    0.01%  4.8580us         1  4.8580us  4.8580us  4.8580us  cudaSetDevice
                    0.01%  4.2620us         1  4.2620us  4.2620us  4.2620us  cudaEventSynchronize
                    0.00%  3.1330us         1  3.1330us  3.1330us  3.1330us  cuDeviceGetPCIBusId
                    0.00%  2.8290us         2  1.4140us     527ns  2.3020us  cudaEventCreate
                    0.00%  2.7030us         3     901ns     413ns  1.8540us  cuDeviceGetCount
                    0.00%  2.2050us         1  2.2050us  2.2050us  2.2050us  cudaGetDevice
                    0.00%  1.9900us         2     995ns     349ns  1.6410us  cuDeviceGet
                    0.00%  1.5750us         2     787ns     252ns  1.3230us  cudaGetDeviceCount
                    0.00%  1.5500us         2     775ns     401ns  1.1490us  cudaEventDestroy
                    0.00%  1.4250us         1  1.4250us  1.4250us  1.4250us  cudaEventElapsedTime
                    0.00%     569ns         1     569ns     569ns     569ns  cuDeviceGetUuid
                    0.00%     217ns         1     217ns     217ns     217ns  cudaCreateChannelDesc
```