Flood Maps, dRMSD Computation

Jiří Filipovič

Fall 2022

Introduction

Artificial problem, was solved as a project PV197 in 2010

- various formulations of algorithm possible with very different performance

The input of the problem is height map, source of water and elevation of the source. The output is the map of flooded area.

- the water source has infinite amount of the water, the ground is waterproof
- so, we need to found a continuous region containing the water source, where the ground is under the water source level
- this is analogy to searching for a connected component in special form of a graph

Naive Algorithm

Probably the simplest version to implement

- a thread is created for each point in the map
- each thread periodically checks its neighbourhood points, if there is a water and thread's point is under water source level, thread puts water into its point
- algorithm iterates until no points are flooded

Extremely inefficient

- let's consider trivial example – the whole map will be flooded
- in an iteration, we perform n^2 operations, we need $2n$ iterations
- complexity is $\mathcal{O}(n^3)$, but sequential algorithm has $\mathcal{O}(n^2)$

Moving Line

How to distribute more water within a single iteration?

- we will create a "moving line"(each line point is processed by a thread)
- the line is moved through a picture horizontaly and vertically in both directions
- if a thread hits a water source, it inserts water until the heigh of processed points is higher than the water source level
- the line iterativelly passes the map until no water is written in all directions of line processing

Moving Line

Efficiency

- we perform n^2 steps in one iteration, where we can flood at most n^2 points
- low performance if the height map is complicated (the water is spread by river basins)
- we need at least 2 iterations, each consisting of 4 line movements (left, right, up, down), but often we need more
- complexity is $\mathcal{O}(i \cdot n^2)$, where i is number of iterations

The second stage of the project requires 250Mpix/s performance

- if the line access memory coalesced (vertical line needs to use shared memory), we overcome the required performance

Tiled Access

Issues with the moving line

- we are processing already flooded points many times
- no temporal locality in data access

Tiled Access

Issues with the moving line

- we are processing already flooded points many times
- no temporal locality in data access

We can process the tiles which fits into the shared memory

- the tile is loaded into shared memory and the moving line is processing the tile iterativelly
- we can do more work without accessing the global memory
- if some tile do not have newly flooded neighbours in an iteration, it can be ommited in the iteration
- can suffer from insufficient parallelism

Tiled Access

Better efficiency

- much less replications of global memory accesses
- replicated accesses mainly into fast shared memory

Some tricks

- blocks completely under/above vater source level handled by special code

Construction of Regions

In nature, the water is moved sequentially

- but do we need to respect it in our algorithm?

Construction of Regions

In nature, the water is moved sequentially

- but do we need to respect it in our algorithm?

We can found continuous regions under the water source level in parallel

- but we do not know, which are to be flooded
- we need to connect those regions with the one containing water source

So we have transformed sequential spread of the water to a problem of connecting regions

- the number of steps is logarithmic with respect to map size

Evaluation

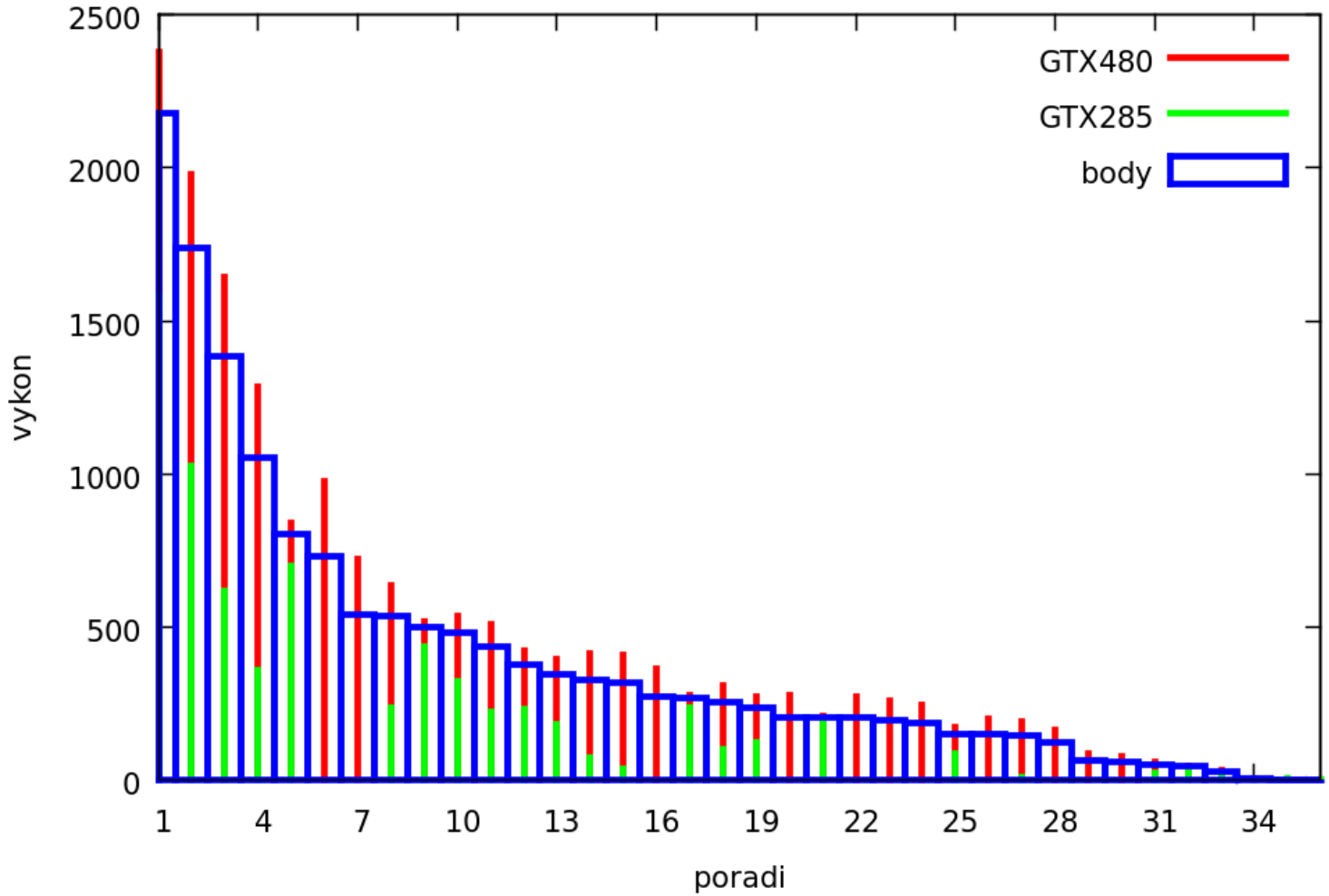
What we get from students

- various implementations, various algorithms (only main ideas presented here)

Performance

- 4 orders of magnitude difference between the fastest and the slowest implementation
  - mainly by using different algorithm
- CUDA optimizations and various tricks creates high spread of performance (so there is no clear distinction of algorithms seen in performance)

Evaluation


2500
2000
1500
1000
500
0
vykon
1
4
7
10
13
16
19
22
25
28
31
34
poradi
GTX480
GTX285
body

RMSD

RMSD (root-mean-square deviation) is used in computational chemistry to compare different structures of a molecule

- tell us how similar are two structures of a molecule (i.e., we have two structures with the same atoms, but different spatial organization)
- used to cluster similar molecules (many outputs of a simulation, some of them very similar)
- used to trace convergence of simulations

dRMSD

dRMSD is one possible variant of RMSD computation, it measures average difference between corresponding pairs of atoms of two molecules

$$dRMSD = \sqrt{\frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d_{ij}^{A} - d_{ij}^{B})^2}$$

where d_{ij}^{A} is the Euclidean distance between i-th and j-th atom in molecule A.
GPU acceleration is part of publication:

- Jiří Filipovič, Jan Plhák, David Střelák. Acceleration of dRMSD Calculation and Efficient Usage of GPU Caches. In *Proceedings of IEEE International Conference on High Performance Computing & Simulation.* 2015.

Memory locality

Flop-to-word ratio

- an iteration of inner sum performs 21 flops and transfers 6 numbers
- flop-to-word ratio 0.875 for single precision, so memory locality needs to be exploited

Improving memory locality

- we store n atoms in registers of threads within a thread block
- when new atom is loaded into shared memory, it can be used to compute n distances
- compute intensity required to saturate GPU can be easily reached

Parallelization

1D parallelization

- only outer sum is parallelized: $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d_{ij}^{A} - d_{ij}^{B})^2$
- straightforward naive implementation (inner sum transformed into kernel code)
  - quite good cache locality, so even naive algorithm works well
- when shared memory is utilized
  - sliding window is not efficient
  - threads within the block can be synchronized to access the same atom in the same time

Parallelization

2D parallelization

- both outer and inner sums are parallelized:
$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d_{ij}^{A} - d_{ij}^{B})^2$$
- better strong scaling
- higher memory workload

Optimizations

When shared memory is utilized, the algorithm is instruction-bound.

- we will optimize instructions

Block size and loop unrolling factor balance

- balance latency hiding and instruction efficiency
- when properly set, we reach 95% occupancy of compute units (according to profiler at Fermi and Maxwell)

Optimizations

SFU instructions optimization

- with `-use_fast_math` compiler flag, `sqrtf(x)` is transformed to `1/rsqrtf(x)`
- it may generate too high workload for SFU units (both reciprocal square root and reciprocal are computed on SFU)
- can be manually optimized by using `x * rsqrtf(x)`
  - speed up computation on Fermi, no effect on Maxwell (it has 2× higher speed of SFUs relative to SPs)

Optimizations

Overhead instructions

- after loop unrolling and SFU optimizations, about 90% of instructions are FP32
- most of instructions in remaining 10% are shared memory loads (6 128-bit loads per 4 iterations)
- to reduce load instructions, we can unroll outer loop

Outer loop unrolling

- unrolling factor u means thread process u iterations of outer sum and thus reduces number of load instructions to $\frac{1}{u}$
- u has to be small to keep registers consumption at reasonable level
  - we set u to 4 for Maxwell and 2 for Fermi
- reduces strong scaling

Performance

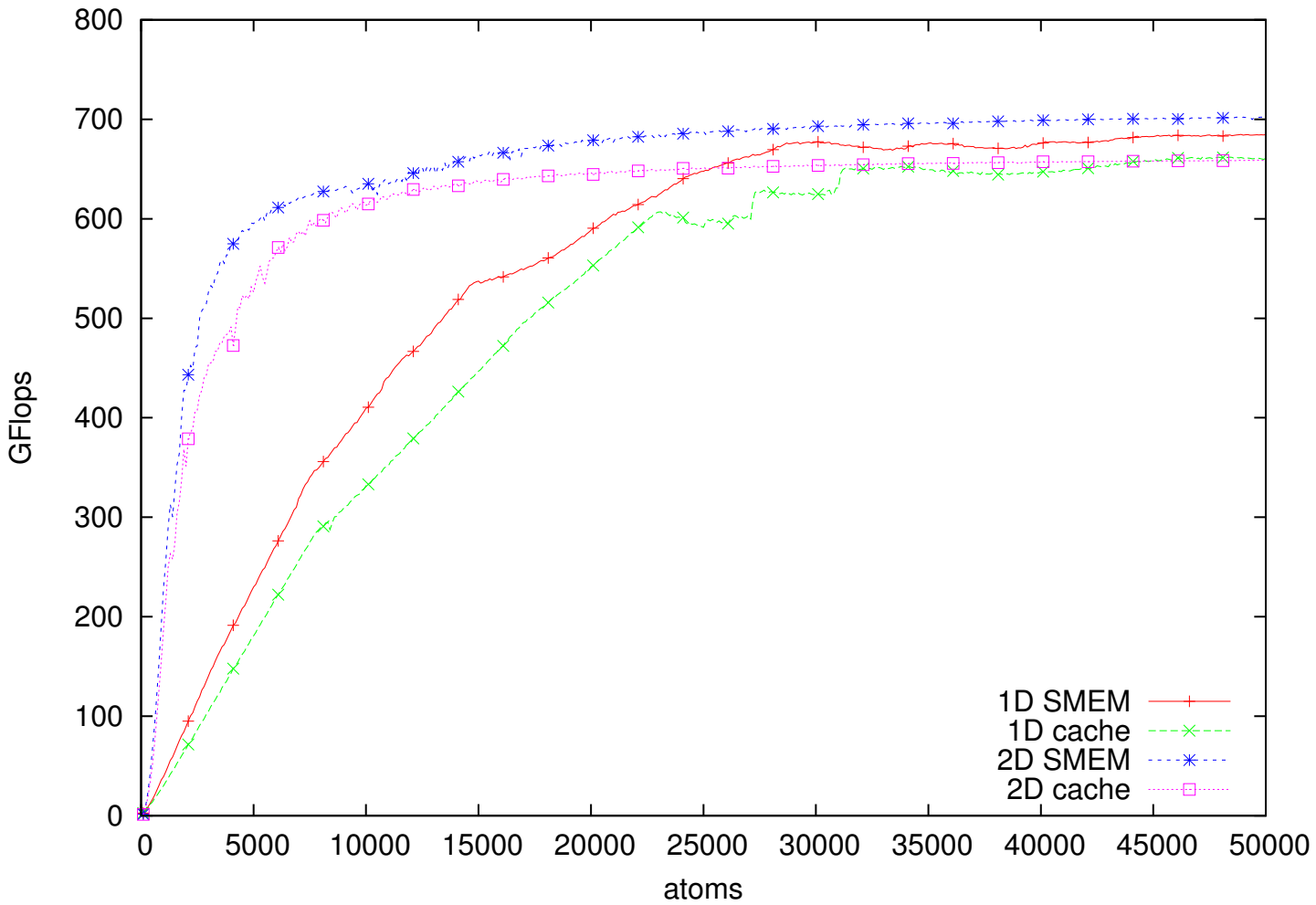
GPU Utilization

- we have tested the performance on GeForce GTX480 (Fermi) and GeForce GTX750 (Maxwell)
- we are able to reach 95 % utilization of ALUs, where 95 % of executed instructions are floating point ones

Comparison with CPU

- dRMSD computation 62.7× faster than ClusCo computation on Core i7-3820
- 13.4× faster than clustering in ClusCo running on Core i7-3820
- performance boost not proportional to theoretical performance difference – the main reason is poor vectorization of ClusCo

Scaling with Fermi



Scaling with Maxwell