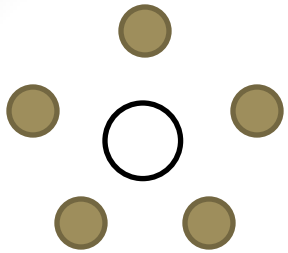


Accelerated k-means Clustering on Multi-Core CPU & GPGPU

Mordechai (“Moti”) Butrashvily
Schools of Geophysics, Physics & Astrophysics, Applied Math.
Tel-Aviv University (TAU), Israel

List of additional contributors follows…

Brief History of Project

- Started during PRACE Summer of HPC 2013
- Hosted here @ NBI

- <u>PRACE Acknowledgement:</u>

  "The project was supported through the PRACE-3IP Summer of HPC programme under EC grant agreement number RI-283493"

Contributors

- Lukas Maly
  - VŠB-TUO: Technical University of Ostrava, Czech-Republic
- Dr. Jacob Trier Frederiksen
  - Niels Bohr Institute (NBI), Copenhagen, Denmark
- Dr. Mads Ruben Burgdorff Kristensen
  - Niels Bohr Institute (eScience@NBI), Copenhagen, Denmark

Agenda

- Simulation Challenges in Plasma Physics (PIC Codes)
- Introduction to k-means Clustering

- Accelerating k-means
  - CPU & GPU
- Performance Results & Conclusions
- Future Work

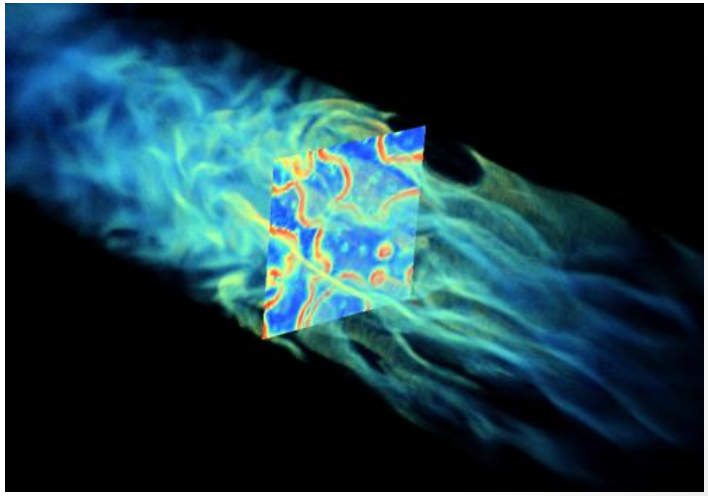
- Few Side Notes (if time permits)
  - Fortran & C/C++ Interoperability
  - K-means Convergence Testing

PLASMA PHYSICS & CHALLENGES

Simulating Plasma Processes

- Accurate physical modeling =>
  - Represent system as close to reality

- Intractable for many-particle systems:
  1. Number of initial particles is large (e.g. sun ~ 10^{57})
  2. Collision/scattering generates more particles with time



CUOS @ UMICH

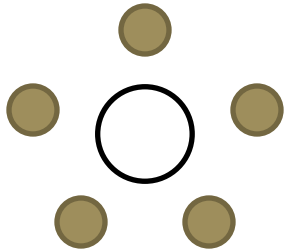
Particle-in-Cell Codes

- A common tool
- In simple words:
  - Decompose space into “independent” volumes (Debye length…)
  - Apply dynamics and advance simulation
- Physical reasoning:
  - Effect of distant charges is shielded by localized charges
- Still, cannot accommodate for fast (exponential) particle increase

Challenges Summary

- Accurately model systems with many particles
- Handle fast particle increase

- <u>Remedy:</u>
  - Represent many particles compactly
  - In other words: **compression/clustering**
  - Assign *weight* to new “mega-particles”



- **Upside** – saves memory + computation time
- **Downside** – information is lost + statistical properties

- Hopefully keep physics as reliable and close to reality

K-MEANS CLUSTERING

K-means in Simple Words

- A class of clustering methods (many varieties & many others)
- Very intuitive geometrical interpretation and understanding
- Similar but slightly reduced version of k-nearest neighbors

- Typically:
  - Take an initial set of N points = dataset
  - Group them into **k < N** clusters = centers / “mega-particles”

- The cluster’s center represents all member points (mean)
- Member point = minimum distance to this center

- One has freedom to choose different distance metrics ($L_{1,2\ldots}$)

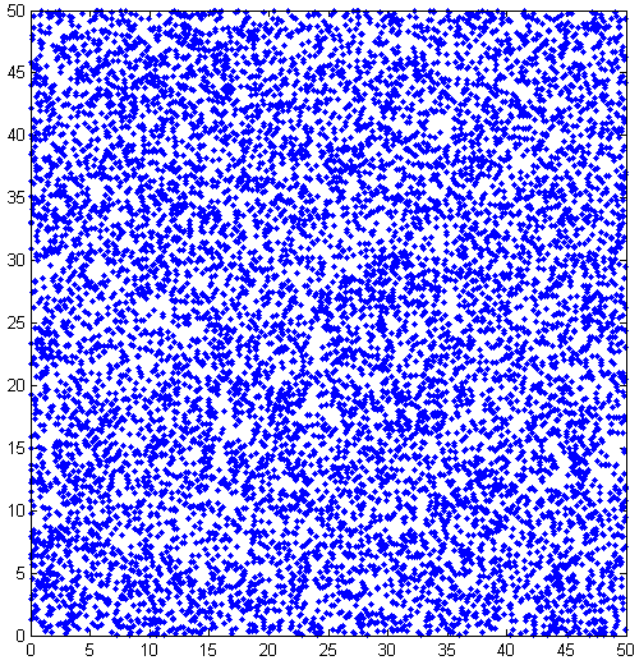
K-means Formally

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \sum_{\vec{x} \in S_i} \|\vec{x} - \mu_i\|^2$$

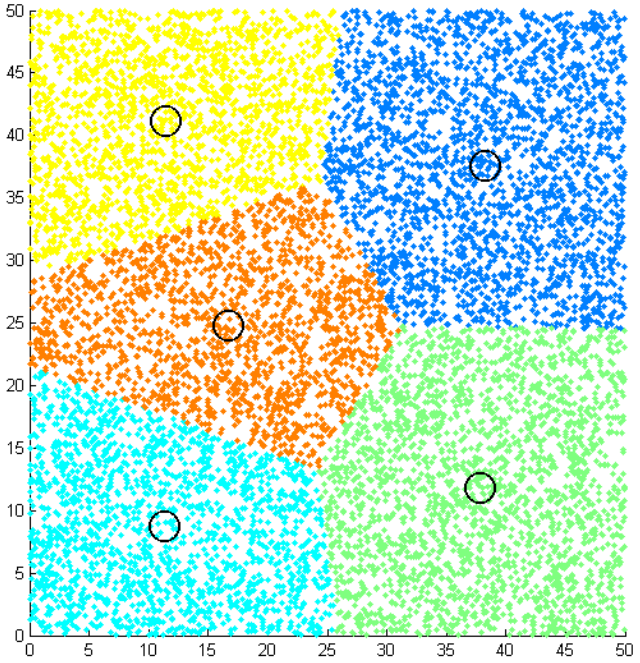
- $\boldsymbol{S} = \{S_1, S_2, \ldots, S_3\}$
- μ_i is mean of points in S_i
- $\vec{x}$ is any point belonging to dataset $\boldsymbol{S}$
- Taking L_2 norm (Euclidean metric)

Clustering Example

Original Dataset (10K pts)



Using 5 Clusters



Problems With k-means

- Managed with physics, but **work only begins**

- K-means clustering is NP-hard to solve exactly
  - Finding an optimum is not trivial (non-convex problem)
  - x100 more points => $100^k \cdot \log 100$ more work
  - Exponential increase ☹

- Alternative, use heuristic algorithms:
  - Lloyd's Algorithm (naïve)
  - Improve by KD-Tree Decomposition
- Faster computation time

Lloyd's Algorithm

1. Heuristic:
   - Pick **k** points at random from dataset => *potential clusters*

2. For every point in dataset:
   1. Compute distance to each cluster
   2. Assign to cluster with minimal distance
3. Compute new clusters' mean => *updated clusters*

4. Did clusters change significantly? (See side note #2)
   1. Yes → go to step 2 for refinement
   2. Otherwise → stop

Step 2 is the **most** intensive

Lloyd's Algorithm (Cont.)

- Advantages:
  - Good and converges fast (10's of iterations normally suffice)
- Pitfalls:
  - High complexity $O(N \cdot k \cdot d)$
    - N = # dataset points, k = # clusters and d = vector dimension
  - Has to traverse all points and clusters in each iteration
- Key observations:
  - Computing distances is a **dot product** and **independent**
  - Can vectorize products using SIMD
  - Can parallelize distance computations (OpenMP etc.)

Short reassessment break #1

- Consider a PIC setting:
  - $N \sim$ 1M particles
  - $k \sim 0.7 * N$ (700,000 clusters / ”mega-particles”)
  - $d = 6$ (3 space, 3 momentum)

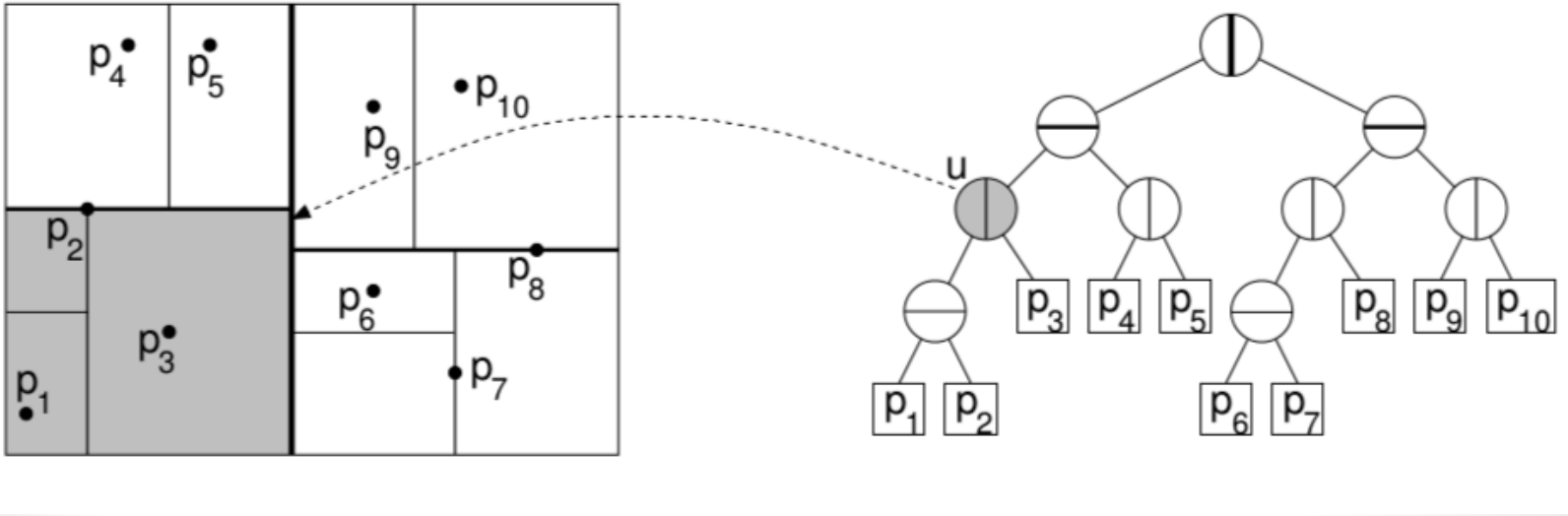
- Computing distance between a single particle-cluster pair:
  - Roughly 18 FLOPS (excluding memory I/O etc.)
- For 1 optimization iteration:
  - Bounded below by ~37.8 TFLOPS (1 CPU core ~ 40 GFLOPS)
  - But usually x10 worse due to inherent overhead

- **Conclusion:** clustering takes HOURS
- Simply parallelizing & SIMD Lloyd’s on CPU is not enough

KD-Tree Approach

- Special data structure based on a binary tree
- Each dimension's/coordinate median helps organize points in subdomain

- Extra time is necessary to sort and build the tree
- But tree doesn't change between iterations!



KD-Tree Approach (Cont.)

- Algorithm remains similar to Lloyd's
- **But**, in step 2 we don't go over all points
- A KD-Tree space decomposition eliminates farther points

- Now, complexity reduces to about $k \cdot d \cdot \log N$
- Major improvement!

- **NOTE:**
  - This is an approximated approach to solve k-means
  - It is possible to miss close points considered to reside in "far" subdomains

Short Performance Analysis

- Runtime has improved by a factor of $\frac{\mathrm{N}}{\log N}$
- Still unsatisfying?...

- Unfortunately:
  - Cannot use SIMD anymore – distance is accumulated during tree traversal
  - Cannot easily parallelize tree traversal using conventional OpenMP
    - Complex data structure representation
    - No guarantee for balanced representation

- However:
  - Can use dynamic task creation in OpenMP as needed ☺
  - Leads to very good results (…)

Novelties

- Most previous works have parallelized tree construction
- In PIC scenarios this is negligible

- First time KD-Tree traversal is parallelized in elegant and satisfying degree

Short reassessment break #2

- Revisiting Lloyd's naïve algorithm
- A rough analysis led to ~37.8 TFLOPS per iteration
- CPUs cannot handle such workloads in reasonable time
  - Solved by employing KD-Tree decomposition

- On the other hand – GPUs can
- A high-end GPU can deliver 5 TFLOPS
  - Or 1.7 TFLOPS back in 2013

- Why wasn't considered earlier?
  - Prevalent hypothesis of CPU unfeasibility
  - Until making simple calculations

GPGPU Environment

- Using NBI Manjula cluster
- GPU: AMD Radeon HD7850 (Consumer)
  - #Cores: 1024
  - RAM: 2 GB
  - Bandwidth: 153.6 GB/s
  - Computation: 1.76 TFLOPS

- Using OpenCL™ 1.2
- Why?
  - Cross-vendor/platform/OS/device
  - AMD GPUs only support OpenCL

Manjula is Apu's wife in "The Simpsons", Image courtesy of https://simpsonswiki.com/

GPGPU Challenges

1. Implementing KD-Tree on GPU is difficult
   - Mostly fits computations with static execution problem size
   - Recent generations allow dynamic work generation
   - Preferred to stay with Lloyd's naïve approach

2. GPUs still have limited RAM capacity
   - COTS/consumer hardware have 2-3 GB
   - High-end devices (e.g. Tesla/FirePro) can have up to 12 GB

   - Can accommodate between 38M – 230M particles
   - Beyond that host I/O increases

GPGPU Implementation

- Based on Lloyd's naïve algorithm:
  1. Initially copying dataset and k clusters into GPU
  2. Using OpenCL for distance computations and comparisons
     - Taking advantage of special GPU features (increased constant memory)
  3. Results are uploaded to CPU for computing new clusters
  4. Repeating 2-3 until convergence is reached

GPGPU Implementation (Cont.)

- Utilizing 1 GPU per instance
- Can solve multiple clustering with OpenMP or MPI

- Support for more particles than GPU RAM accommodates
- Using host shared memory to eliminate data I/O
- Dynamic memory balancing & allocation:
  - Depending on GPU RAM
  - Points / clusters ratio

- Reduction computations performed on CPU

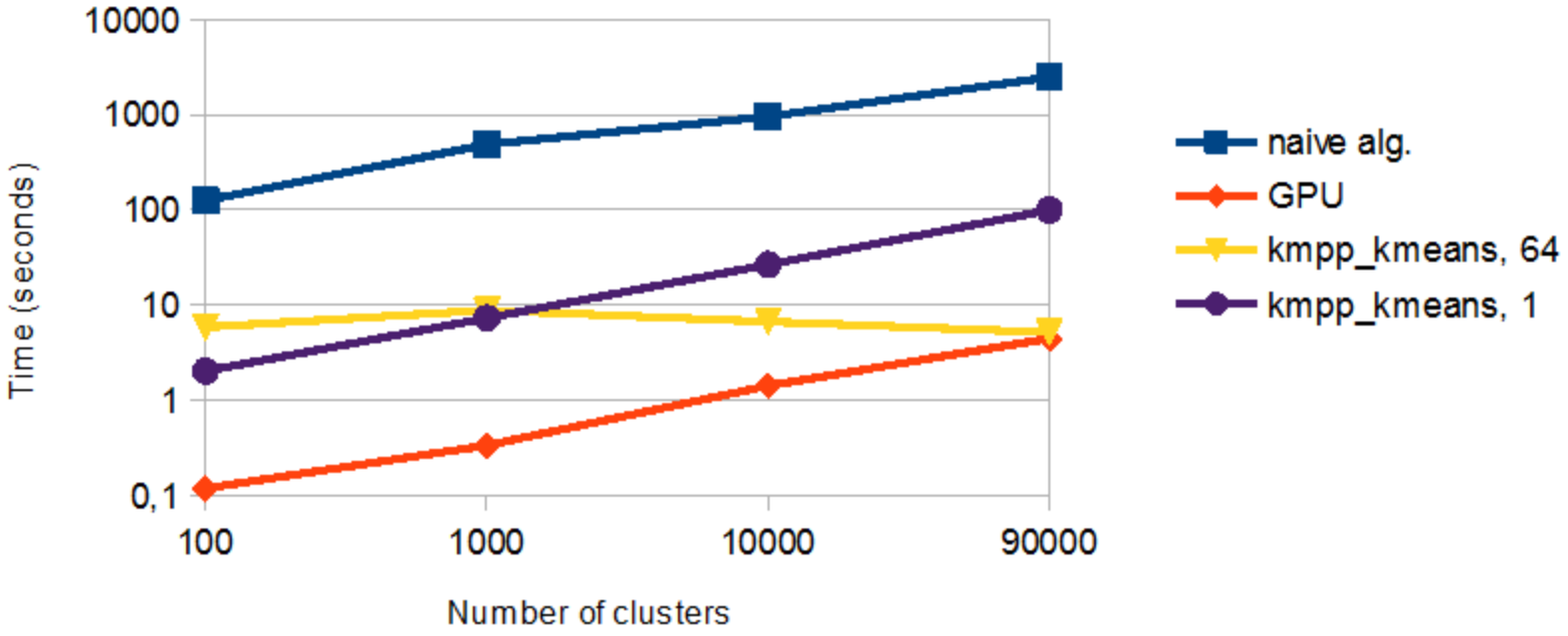
RESULTS & CONCLUSIONS

Benchmark Environment

- Performance metrics were collected using Manjula cluster
- GPU: AMD Radeon HD7850
- CPU: AMD Opteron 6272
  - 16 cores
  - 2.1 GHz
  - Core L2 cache: 1 MB (2 MB shared between 2 cores)
  - CPU L3 cache: 16 MB

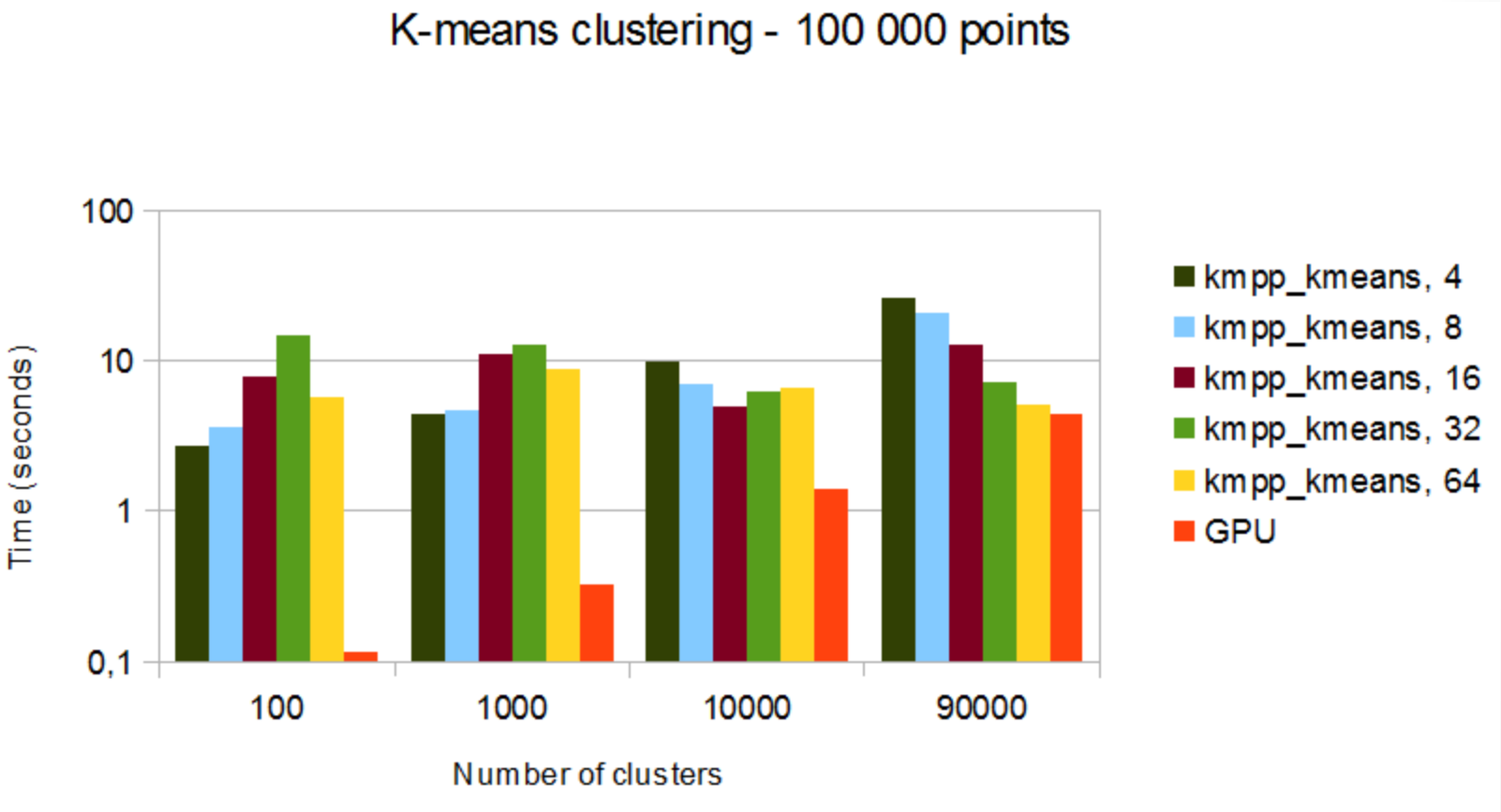
Timing for 100K Points

- Performance graphs and discussion



Logarithmic scale!
Two extreme CPU core setups.

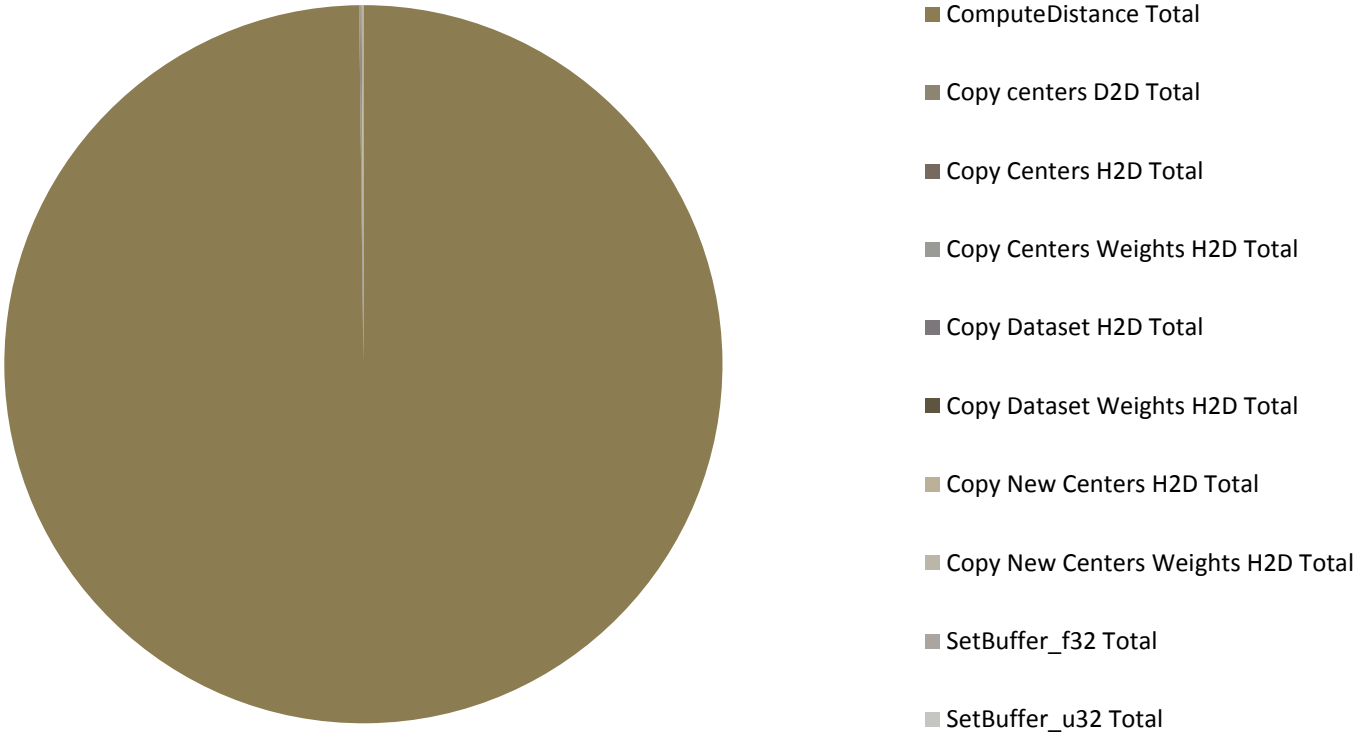
Compare Accelerated Methods



Logarithmic scale!

Varying # of CPU cores in KMPP.

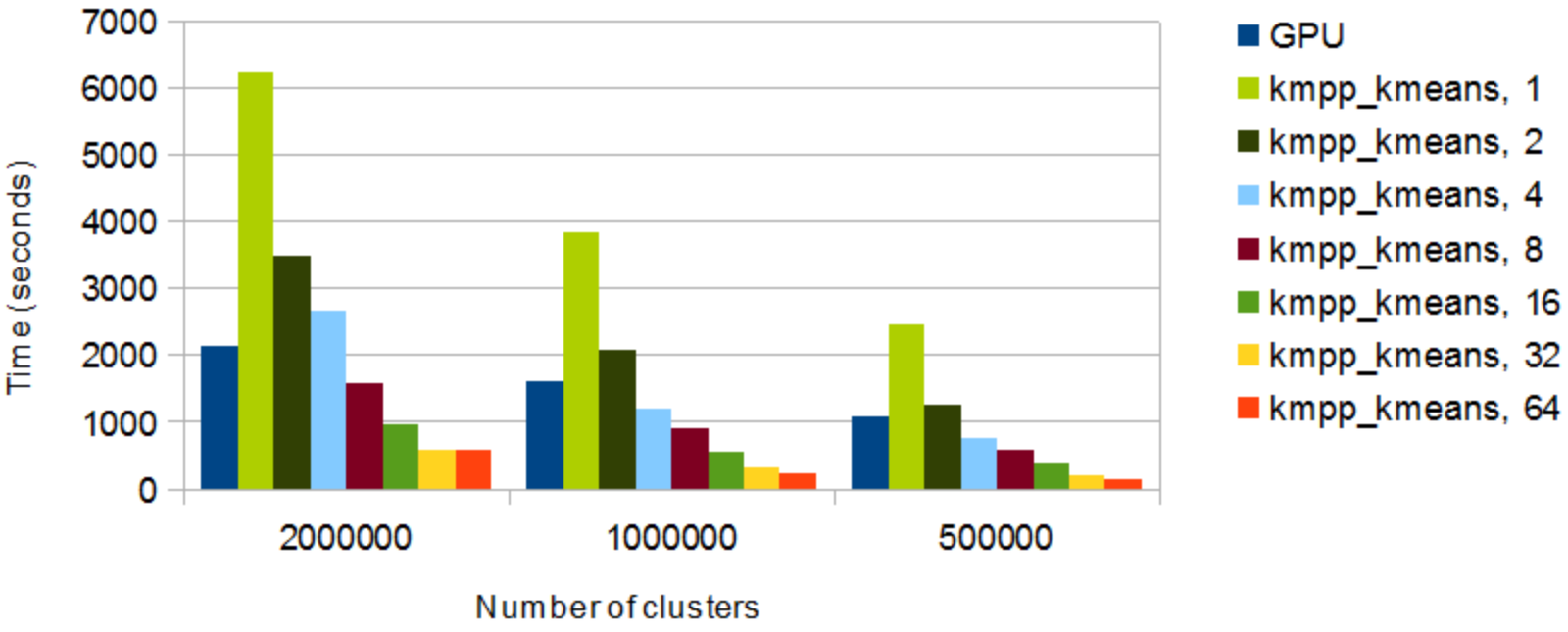
GPU Performance 100K Points



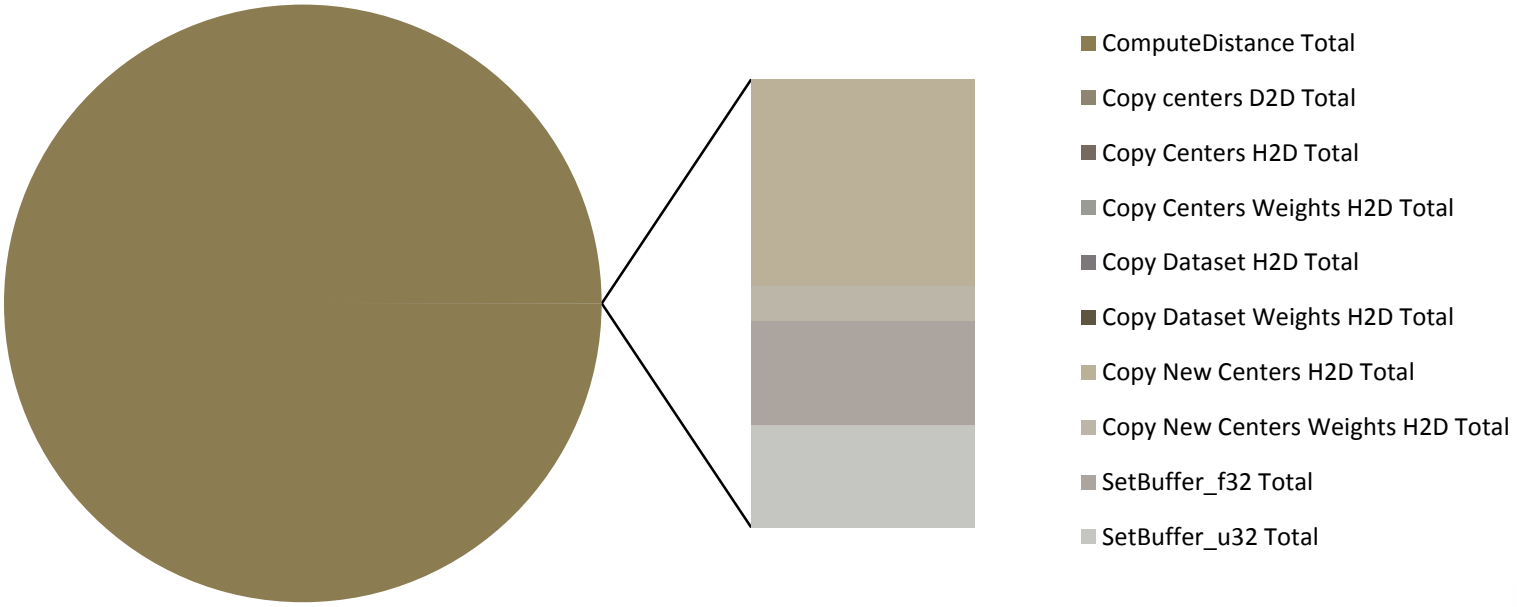
90K Centers!

Time spent on I/O is negligible.

Comparison for 2.4M Points



GPU Performance 2.4M Points



1M Centers!

Time spent on I/O is not even a fraction.

Conclusions

- Both implementations outperform existing algorithms
  - x100 – x1000 speedup
- If GPU exists, can consider Lloyd's naïve algorithm
  - Simpler to maintain
  - Can perform even better than KD-Tree, depending on problem configuration
  - I/O is surprisingly not an issue
- KD-tree is preferred with larger datasets
  - Benefits higher CPU cache
  - And more CPU cores

Conclusions (Cont.)

- The GPU can outperform CPU even in non-trivial cases
- HW is improving both for GPUs and CPUs

- Though the GPU is not good for solving every problem
  - Convergence tests are better to implement on CPU
  - Increased I/O didn’t add much

Implications

1. From 300,000 processors to ~300
2. Increase physics accuracy using similar resources
   - E.g. by adding much more particles

Future Work

- Add support for double precision (currently)
- Collect updated performance data on recent HW
- Better accounting for particle weights

- Which platform would win the CPU-GPU rival?

Side Note #1: Fortran & C/C++

- For many years Fortran & C/C++ could not interface in standard ways

- With ISO_C_BINDING extension of Fortran 2003/2008 it is now possible
- Mainly:
  1. Define a Fortran function that binds to a C function
  2. Better type matching/conversion

- Helps integrate C and Fortran, cross-platform/compiler
- Especially when complex libraries and data structures are easier to implement in C (e.g. KD-trees)

Side Note #1: Example 1

- Define Fortran signature for function implemented in C
  - Taken from OpenCL API

```
integer(c_int32_t) function clGetPlatformIDs(num_entries, &
                    platforms, num_platforms) &
        BIND(C, NAME='clGetPlatformIDs')
        USE ISO_C_BINDING

        integer(c_int32_t), value, intent(in)   :: num_entries
        type(c_ptr), value, intent(in)          :: platforms
        integer(c_int32_t), intent(out)         :: num_platforms
end function
```

Side Note #1: Example 2

- Scalar and structure definitions:

```
integer(c_int64_t)  :: num = 1
integer(c_size_t)   :: ptr = 12

type, BIND(C) :: particle
      integer(c_int32_t) :: id
      real(c_float)      :: coords(3)
      real(c_float)      :: momentum(3)
      real(c_float)      :: weight
end type
```

- And there are much more standard definitions to ease life

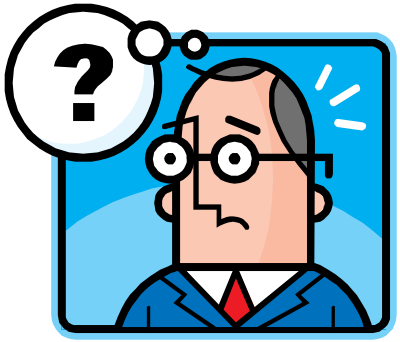
Side Note #2: Convergence

- Convergence for k-means clustering iterations
- Using Loss Quality Error

- An acceptable measure in clustering theory
  1. Normalize the sum of distances from a center by the number of points that belong to it (denoted N_k)
  2. Then sum normalized valued over all centers

$$\underbrace{\sum_{k} \underbrace{\frac{1}{|N_k|} \cdot \sum_{N_k} distance(pt, k)}_{1}}_{2}$$

Questions?



Thank You ☺