Introduction: General Purpose GPU 101

Minh-Tue Vo
(mvo @ Tool and Frameworks)

More reasons to buy your dream gaming laptop/desktop this Christmas*

Before that

- Sign up for ESPP!

- Dev VPN: get **Junos Pulse** to replace Network Connect
  - Faster, better than Network Connect.
  - https://confluence.inside-box.net/display/BOX/Junos+Pulse+VPN+Instructions

Wow! Look at that graphics!

Once upon a time…

- S. A. Shalom, M. Dash, and M. Tue, "*Efficient K-Means Clustering Using Accelerated Graphics Processors*". pp. 166-175, 2008.

- S. A. A. Shalom, M. Dash, and M. Tue, "*Graphics hardware based efficient and scalable fuzzy c-means clustering*," AusDM, ser. CRPIT, JF Roddick, J. Li, P. Christen, and PJ Kennedy, Eds, vol. 87, pp. 179-186, 2008.

A brief history of GPGPU

- Very fast at certain floating point calculations.
- By the early 2000s vendors were starting to support **programmable** features such as lighting, shading calculations, texture mapping.
- By the mid 2000s some were **"hacking"** OpenGL and other programming tools to do numeric computations on GPU devices.
- In effect, one had to **map a mathematical computation** onto a graphics computation, do the graphics, and extract the result from the graphics memory.

A brief history of GPGPU

- During the late 2000 NVIDIA introduced proprietary **CUDA** ("**C**ompute **U**nified **D**evice **A**rchitecture") as a general purpose tool to make GPUs available for numerical computation.

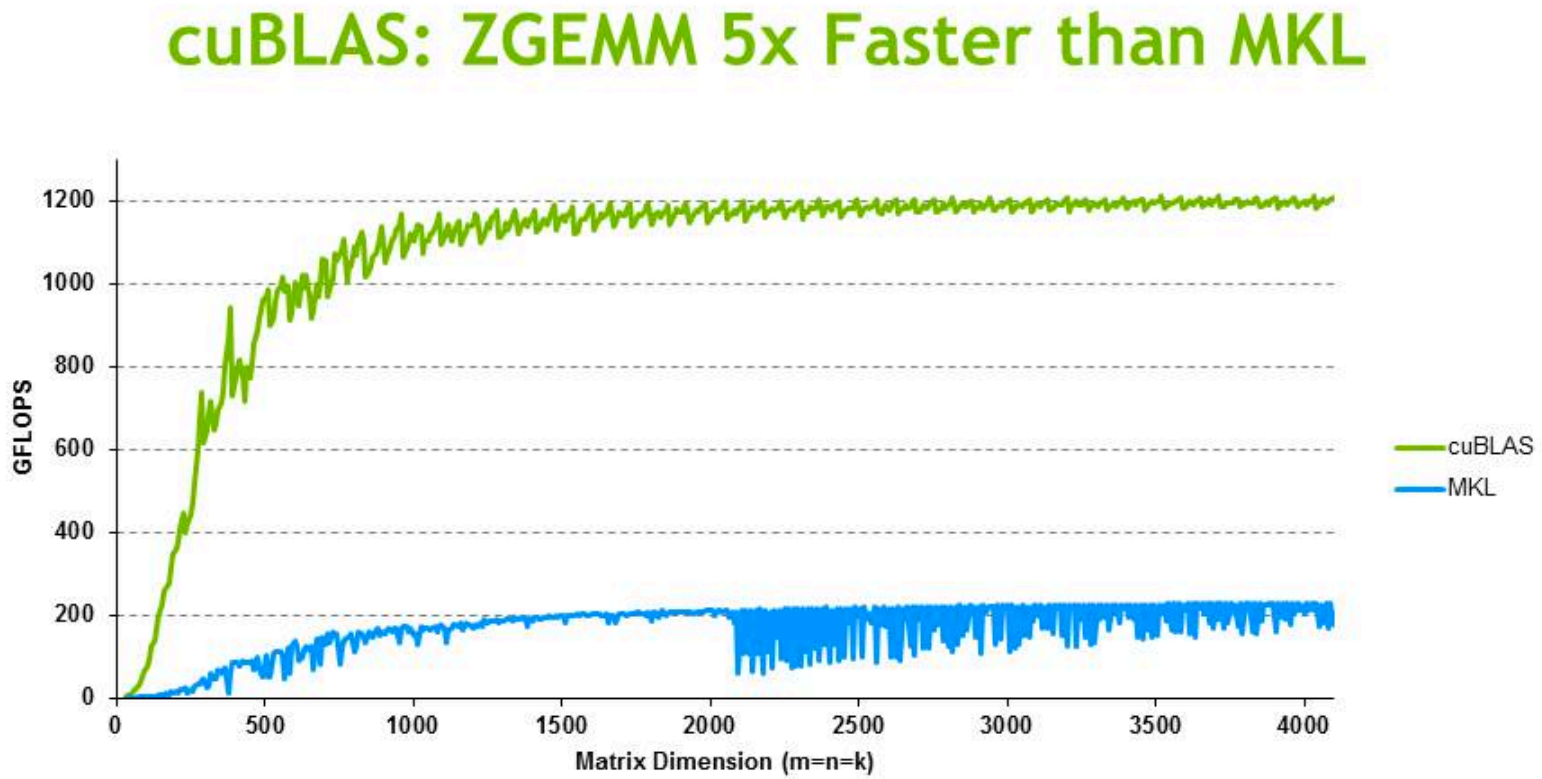
- The OpenCL standard was introduced by a consortium of vendors.

- In 2012 Intel released its Phi accelerator, similar to GPU hardware but without the graphics heritage.

Three main reasons for ML@GPGPU

- Excellent performance per cost (huge demand!)
- Native support for Linear Algebra functions.
- Excellent integration with standard programming languages (C++, Python, Java*).

GPU Matrix Multiplication is fast !



Performance may vary based on OS version and motherboard configuration

- cuBLAS 6.0 on K40m, ECC ON, input and output data on device
- MKL 11.0.4 on Intel IvyBridge 12-core E5-2697 v2 @ 2.70GHz

Samples

- push-relabel maximum flow algorithm
- fast sort algorithms of large lists
- two-dimensional fast wavelet transform
- molecular dynamics simulations

NVIDIA's CUDA is easy*

- Native support in C++.
- PyCUDA
- Java
- Matlab
- Mathematica

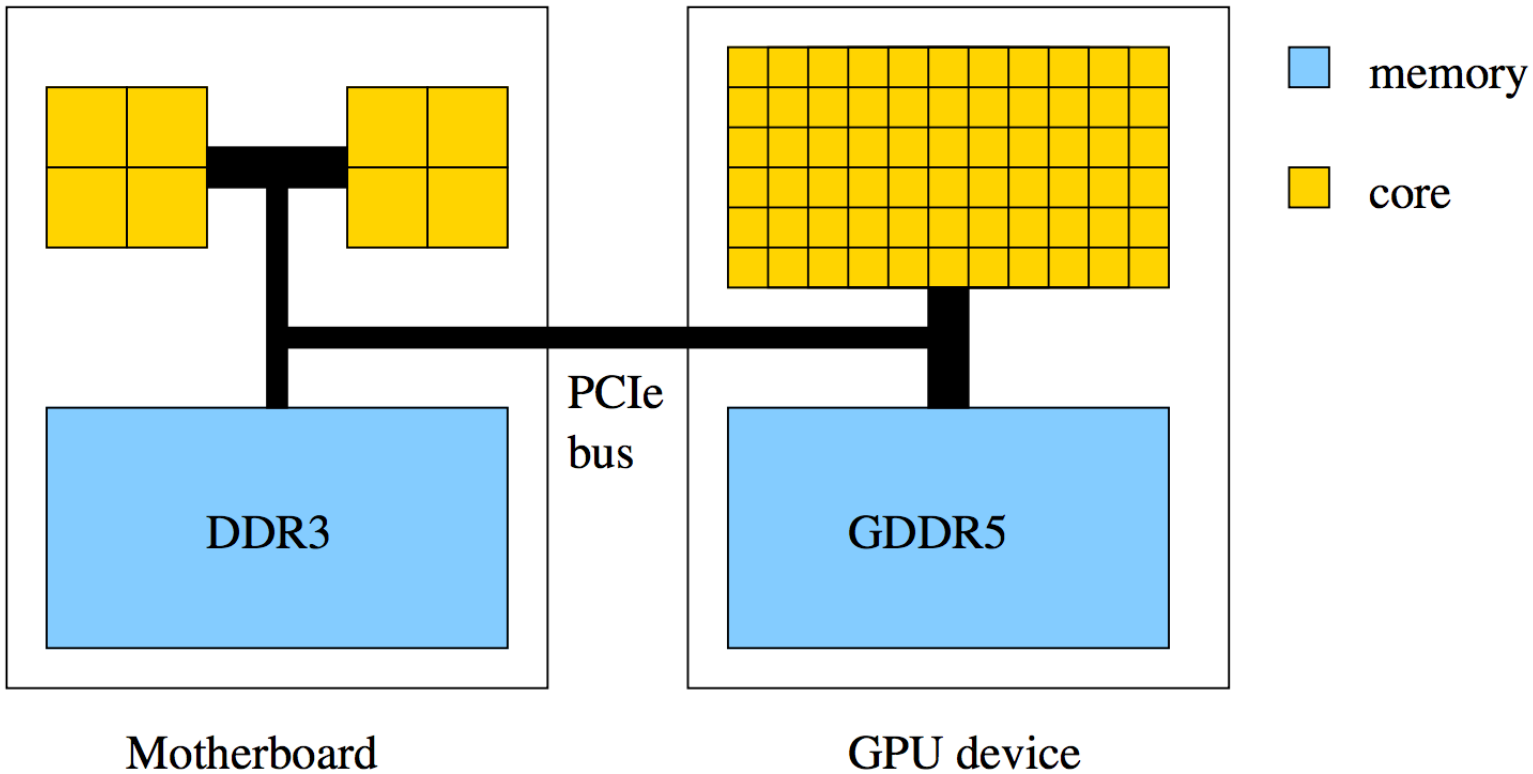
…

*only works with NVIDIA's GPUs

cuBLAS

- CUDA Basic Linear Algebra Subroutines (cuBLAS)
- Deliver 6x to 17x faster performance than the latest MKL BLAS*.
- CUDA 6.0: multi-GPU support in cuBLAS-XT.

Hardware Overview



What does GPU have?

- 192 cores and 64K registers
- 64KB of shared memory / L1 cache (user configurable)
- 8KB of cache for constants
- 64KB of texture cache for read-only arrays
- up to 2K threads per Stream Processor

Stream processor (SMX): Single Instruction, Multiple Thread

- All cores execute the same instructions (kernel) simultaneously, but with different data.
- Minimum of 32 threads (a warp) all doing same thing at the same time.
- No “context switching;” each thread has its own registers (limits the number of active threads).
- Each thread in a warp executes exactly the same instruction, or waits while others in the warp execute instructions note - want to avoid branches within a warp as the branches are executed sequentially

So what's stream processor good for?

- Task parallel
  - Independent processes with little communication
- Data parallel
  - Lots of data on which the same computation is being executed
  - No dependencies between data elements in each step in the computation

What does a program look like?

1) Host (CPU) and device (GPU) are initialized when program starts.

2) Separate memory for data is allocated on host and on device.

3) Host data is initialized.

4) Data copied from host to device. Slow ☹

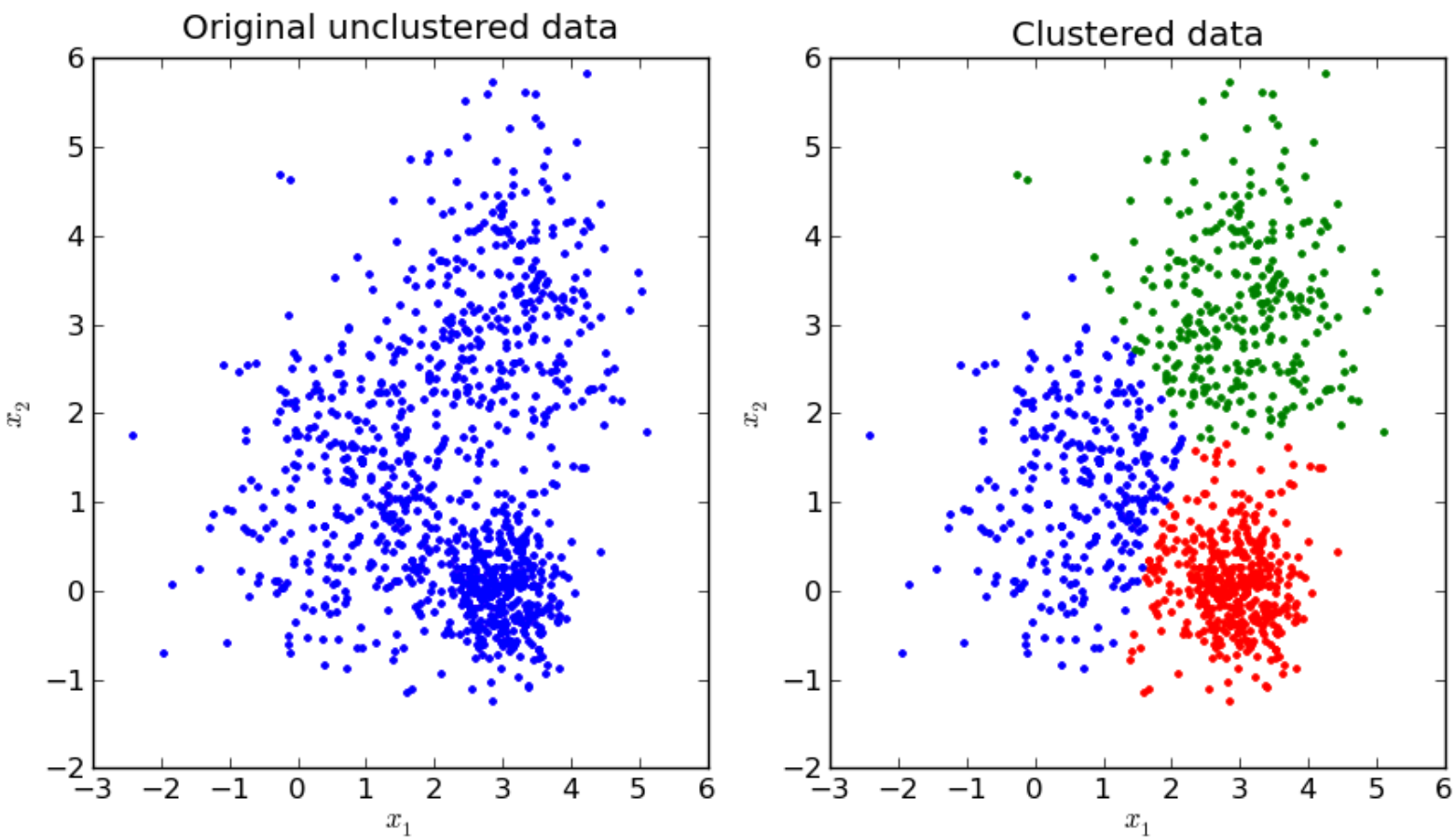
5) Host launches multiple instances of execution “kernel” on device. Fast! ☺

6) Data copied from device to host. Slow ☹

7) Repeat steps 4 – 6.

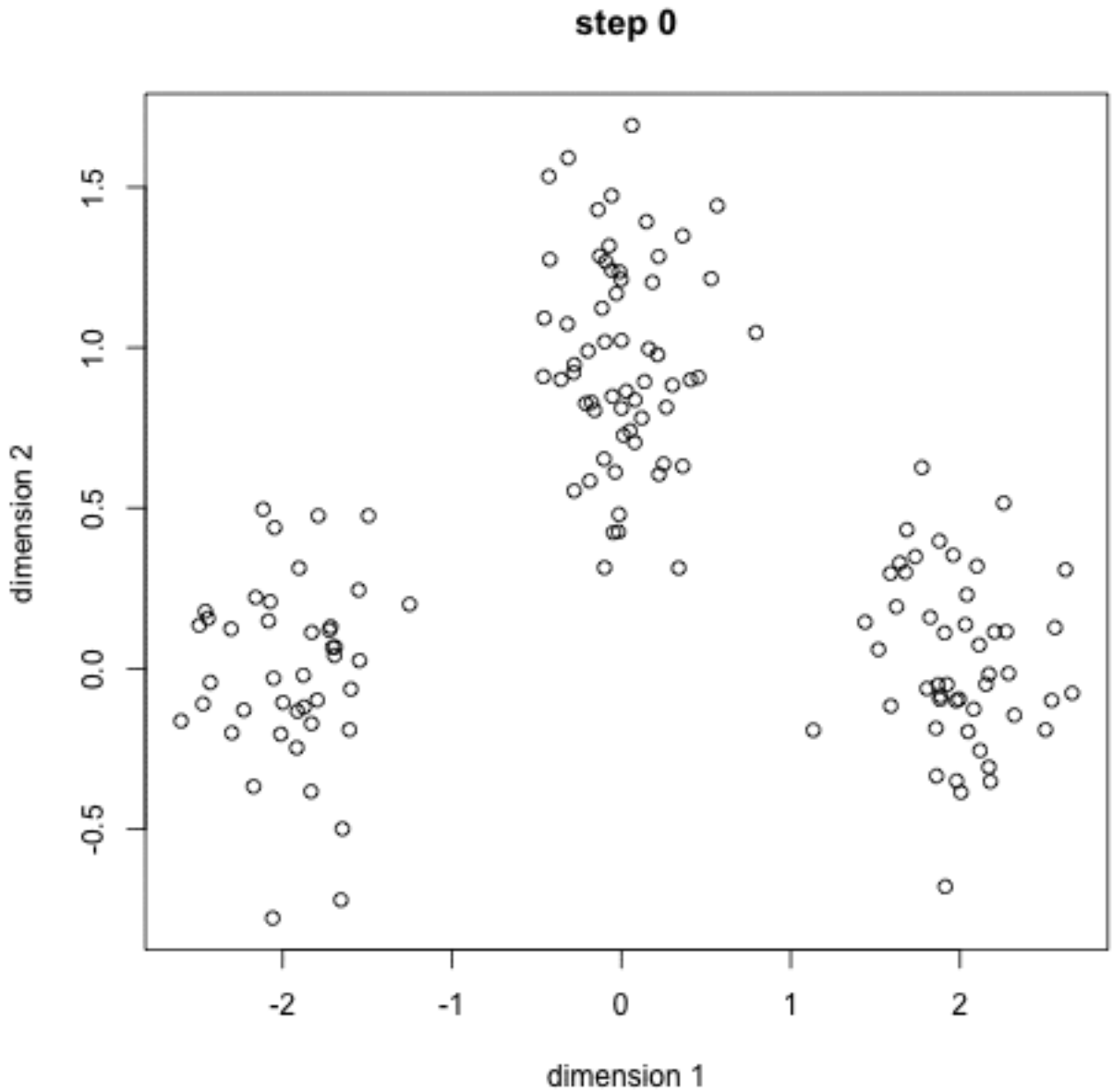
8) Host deallocates memory on device and host and terminates.

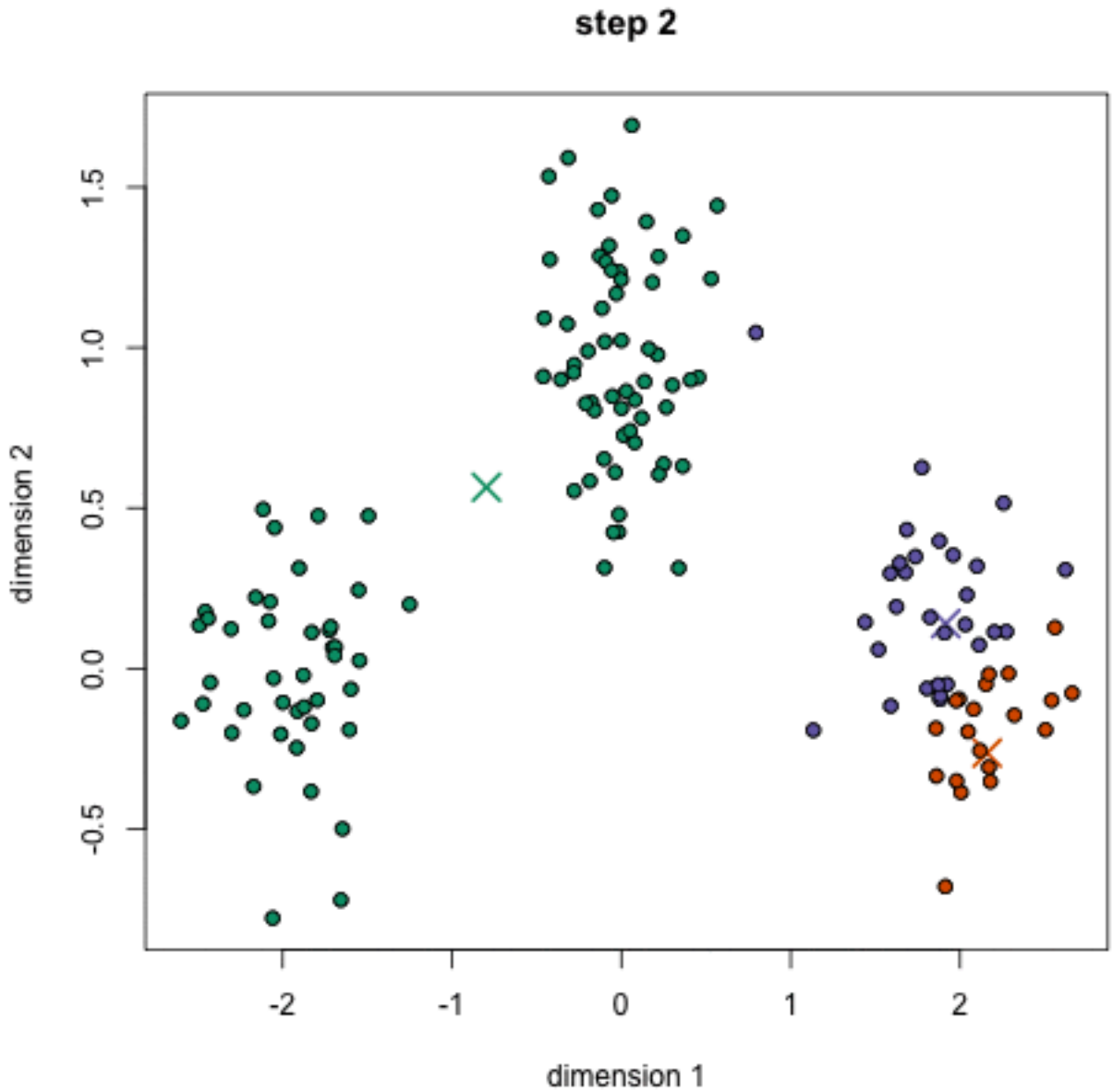
Example: Clustering

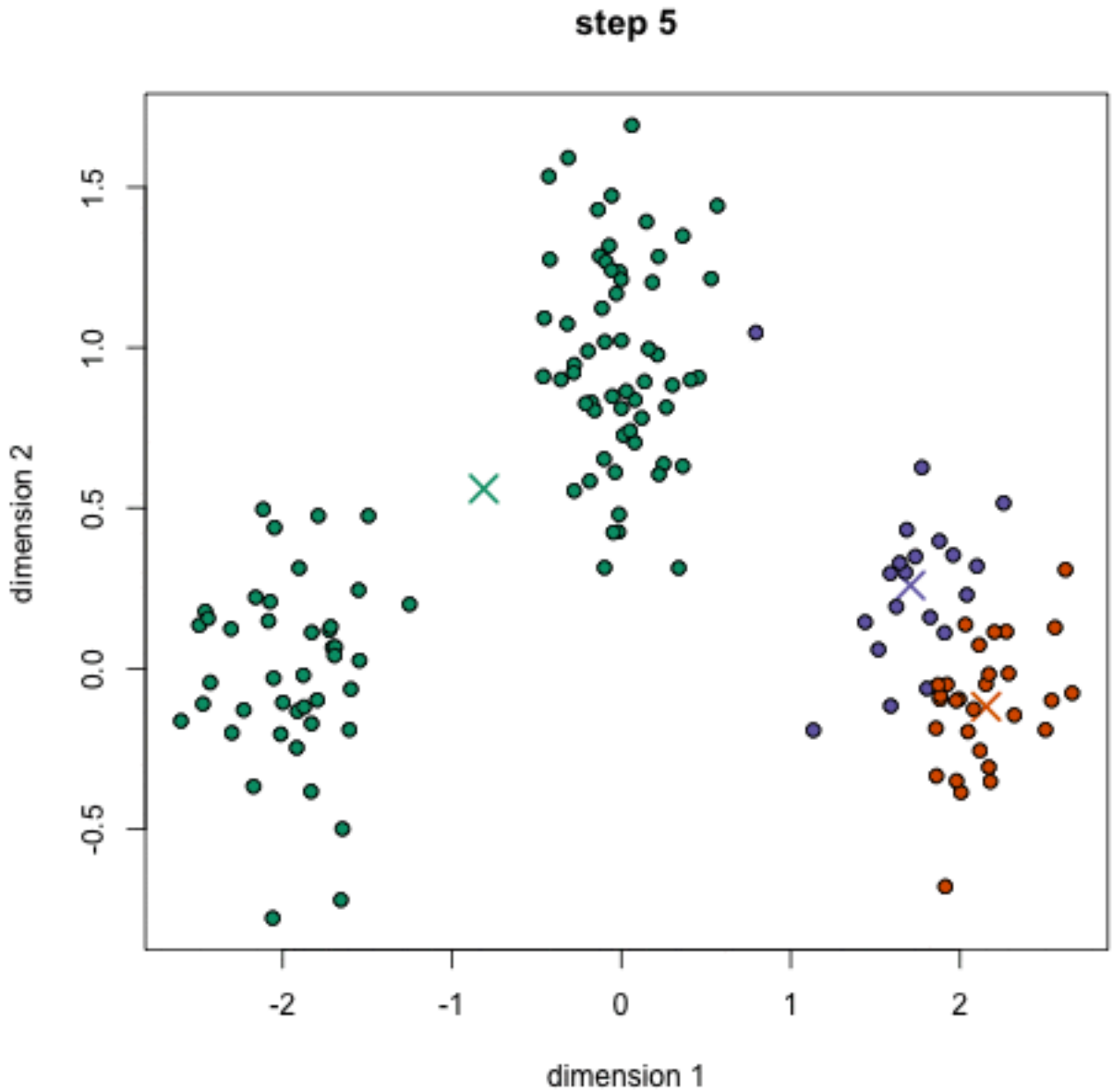


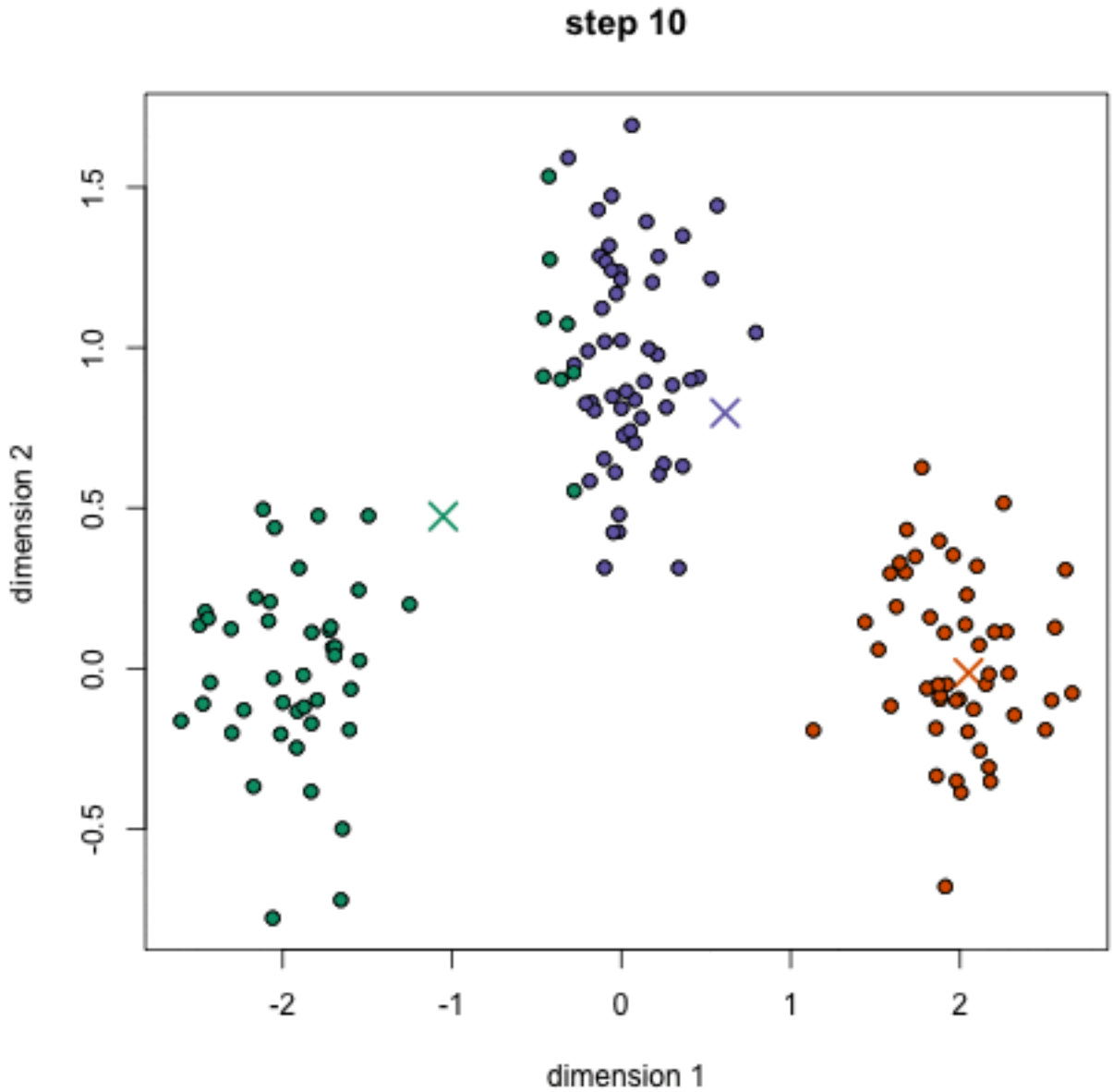
Example: K-means algorithm

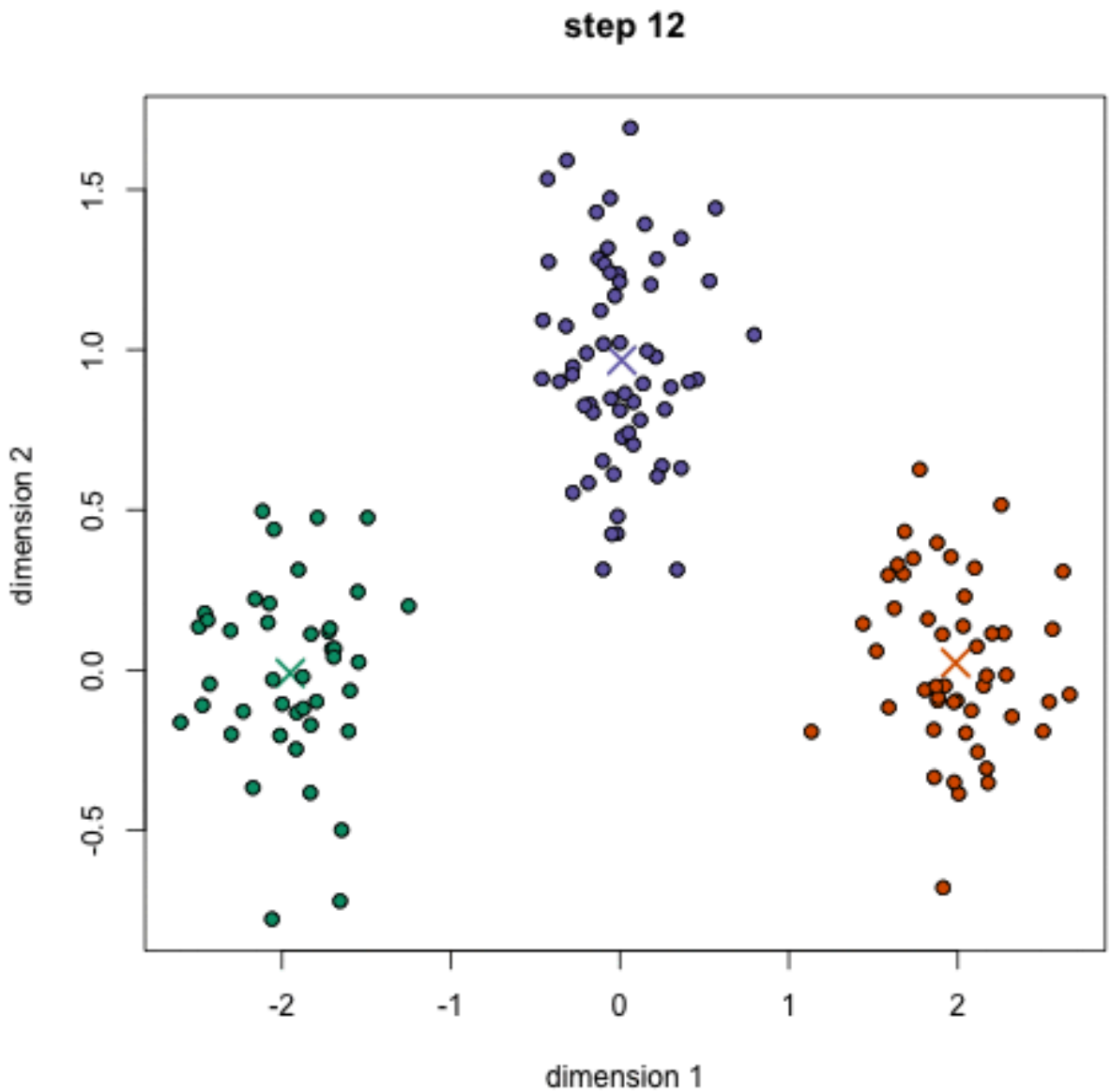
- *N* data points, *m* clusters, N >>> m
- Output: *m* cluster centers (centroids)


step 0
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 2
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 5
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 10
dimension 2
dimension 1
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2


step 12
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1

Example: K-means algorithm

- *N* data points, *m* clusters, N >>> m

- Randomly assign *m* cluster centers (centroids).
- For each data point:
  - Compute the distance to every centroid.
  - Find the centroid with minimum distance.
- For each centroid:
  - Compute the sum of all cluster members.
  - Average to compute the new centroid.
- Check for convergence.

K-means on GPU

- Host:
  - Randomize cluster centers (centroids).
  - Transfer data points from host (CPU) to device (GPU) **(exp, read only**).
  - Transfer cluster centers from host to device **(exp, read only)**.
- Device: For each data point:
  - Compute the distance to every centroid (**cheap**)
  - Find the closest centroid (**cheap**).
- Host:
  - Transfer data from device to host (**exp**).
  - Remap data, transfer from host to device (**exp**)
- Device: for each centroid:
  - Compute the sum of all cluster members (**cheap**).
  - Average to compute the new centroid (**cheap**).
- Host:
  - Transfer data from device to host (**exp**)
  - Check for convergence.

Summary

- Good: Awesome computation.
- Bad: Terrible data bandwidth.
  - Careful data transfer
  - Careful data memory usages (shared, read-only, …)
- Further optimization: GPU architecture (thread granularity, benchmarking transferring rate, …)

Getting started (Mac OS X)

- http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-mac-os-x

- a CUDA-capable GPU
- Mac OS X 10.8 or later
- the gcc or Clang compiler and toolchain installed using Xcode
- the NVIDIA CUDA Toolkit (available from the CUDA Download page)

ML @ GPGPU

- http://www.nvidia.com/object/machine-learning.html

- http://gpgpu.org/

Until next time

- Coding tutorial with CUDA in Python/C++
- Benchmarking: comparing apples & oranges
- Papers!

Reference

- Presentation adapted from:
  - http://www.math-cs.gordon.edu/courses/cps343/presentations/Intro_to_GPGPU.pdf
  - www.gpgpu.org