



GPU Teaching Kit
Accelerated Computing



Module 17 – Computational Thinking

Lecture 17.1 – Introduction to Computational Thinking

Objective

- To provide you with a framework for further studies on
 - Thinking about the problems of parallel programming
 - Discussing your work with others
 - Approaching complex parallel programming problems
 - Using or building useful tools and environments

Fundamentals of Parallel Computing

- Parallel computing requires that
 - The problem can be decomposed into sub-problems that can be safely solved at the same time
 - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
 - To solve problems in less time (strong scaling), and/or
 - To solve bigger problems (weak scaling), and/or
 - To achieve better solutions (advancing science)

The problems must be large enough to *justify* parallel computing and to exhibit *exploitable concurrency*.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
 - This is what CUDA (and OpenMP, OpenCL) is based on
 - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
 - However, you will find parallels for almost every technique you learned in this course
 - Need to be aware of space-time constraints

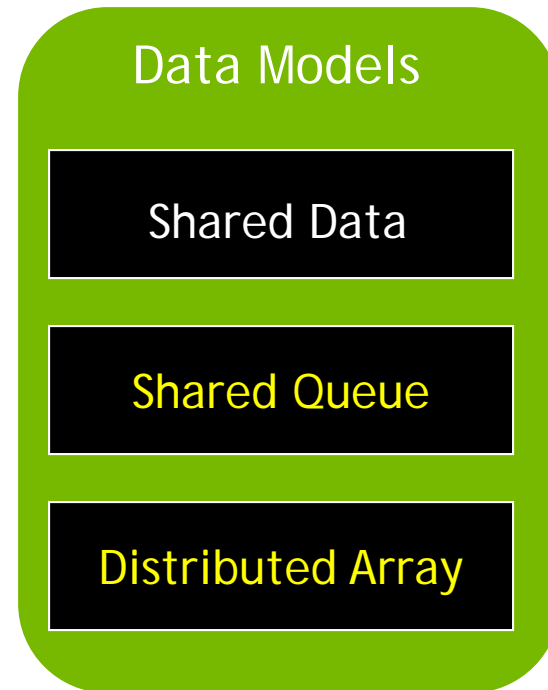
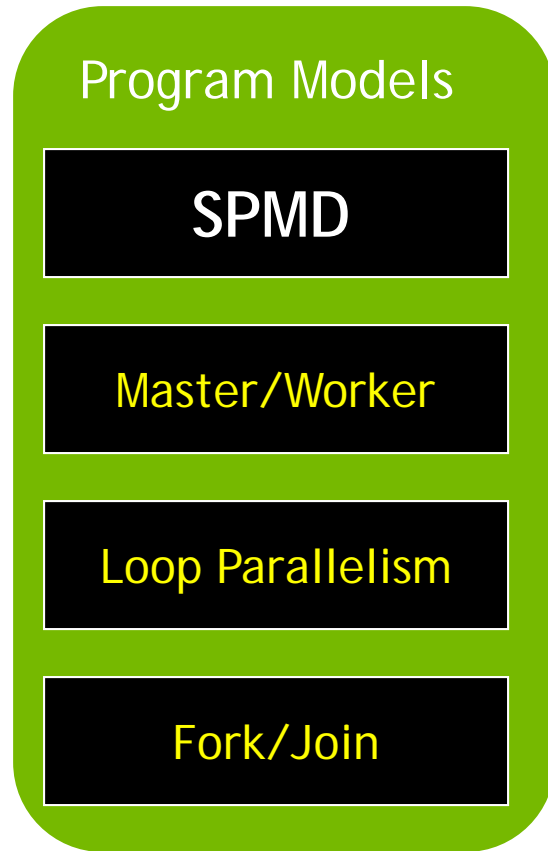
Data Sharing

- Data sharing can be a double-edged sword
 - Excessive data sharing drastically reduces advantage of parallel execution
 - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
 - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- **Many:Many**, **One:Many**, **Many:One**, **One:One**

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
 - Watch out for serialization
- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles – Program and Data Models



These are not necessarily mutually exclusive.

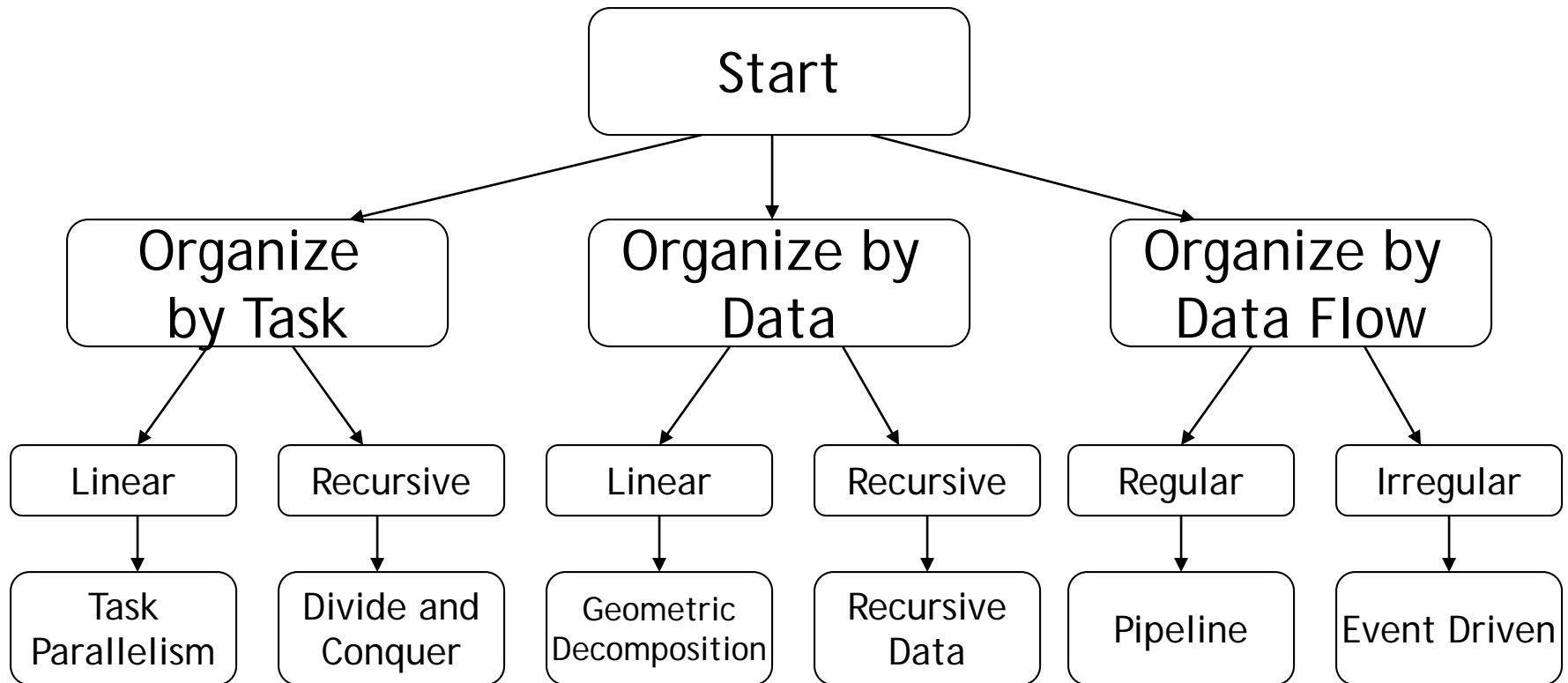
Program Models

- SPMD (Single Program, Multiple Data)
 - All PE's (Processor Elements) execute the same program in parallel, but has its own data
 - Each PE uses a unique ID to access its portion of data
 - Different PE can follow different paths through the same code
 - This is essentially the CUDA Grid model (also OpenCL, MPI)
 - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
 - A Master thread sets up a pool of worker threads and a bag of tasks
 - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
 - Loop iterations execute in parallel
 - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
 - Most general, generic way of creation of threads

Algorithm Structure



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

More on SPMD

- Dominant coding style of scalable parallel computing
 - MPI code is mostly developed in SPMD style
 - Many OpenMP code is also in SPMD (next to loop parallelism)
 - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
 - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
 - Establish localized data structure and communication channels
- Obtain a unique identifier
 - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
 - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
 - Decompose global data into chunks and localize them, or
 - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
 - More details in next slide...
- Finalize
 - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
 - Use thread ID in loop index calculations to split loop iterations among threads
 - Potential for memory/data divergence
 - Use thread ID or conditions based on thread ID to branch to their specific actions
 - Potential for instruction/execution divergence

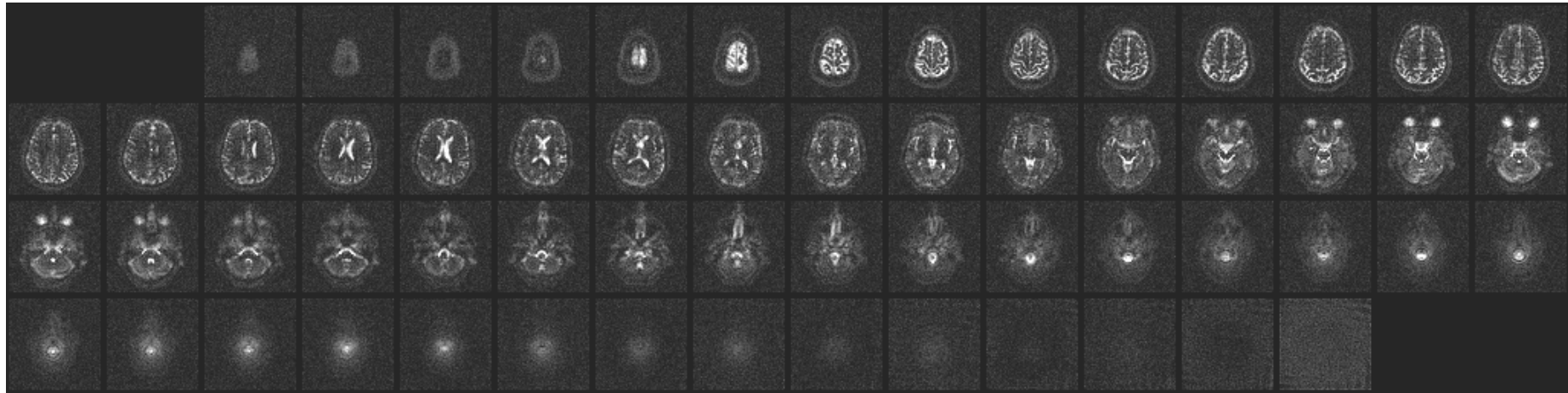
Both can have very different performance results and code complexity depending on the way they are done.

Making Science Better, not just Faster

or... in other words:

There will be no Nobel Prizes or Turing Awards awarded for “just recompile” or using more threads

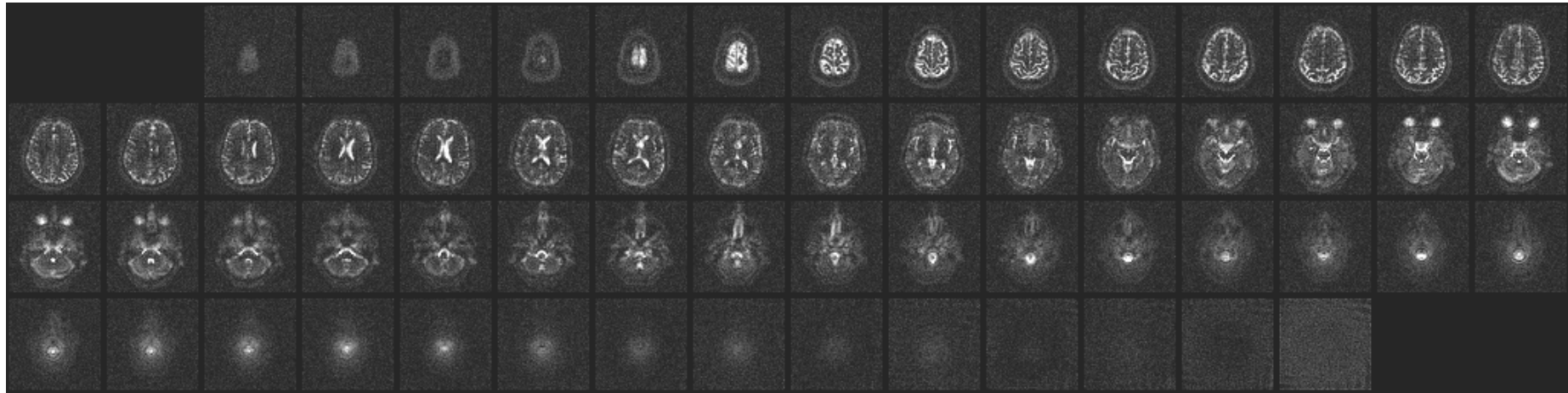
A Revolution - Sodium Map of the Brain



- Images of sodium in the brain
 - Sodium is one of the most regulated substance in human tissues
 - Any significant shift in sodium concentration signals cell death
 - Much less abundant than water in human tissues, about 1/2000
 - Very large number of samples are needed for good SNR
 - Requires high-quality reconstruction, currently considered impractical

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

A Revolution - Sodium Map of the Brain

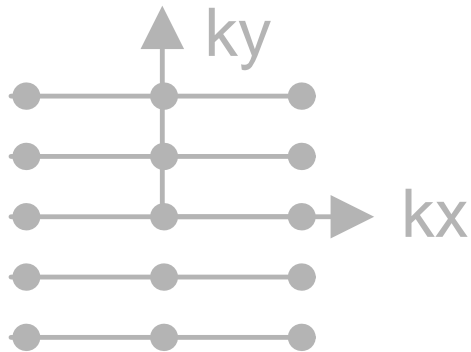


- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment.
 - Drastic improvement of timeliness of treatment decision
 - Minutes for stroke and days for oncology.

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

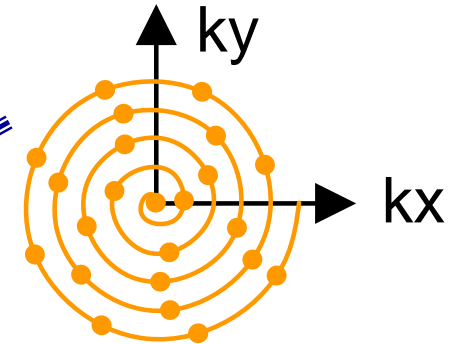
Reconstructing MR Images

Cartesian Scan Data



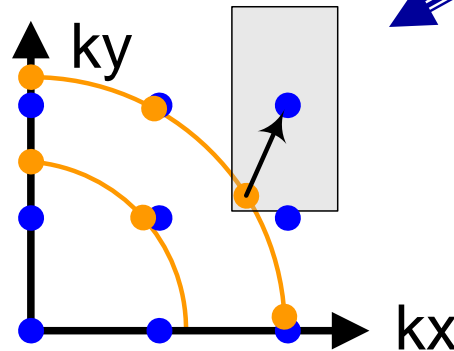
FFT

Spiral Scan Data



LS

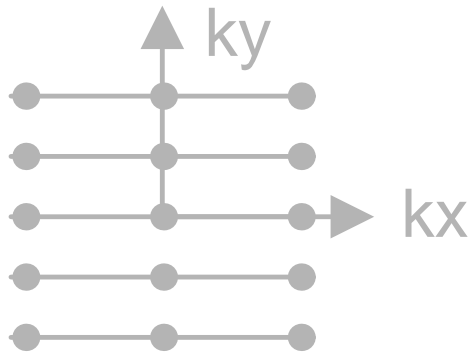
Gridding¹



Spiral scan data + Gridding + FFT:
Fast scan, fast reconstruction, good images
Can become realtime with about 10X speedup.

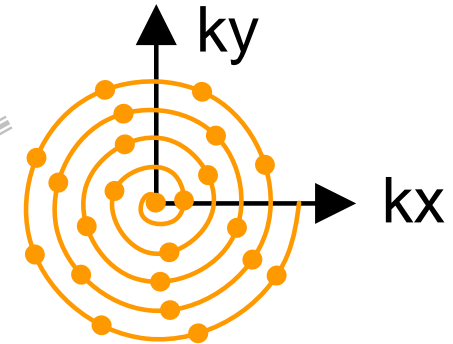
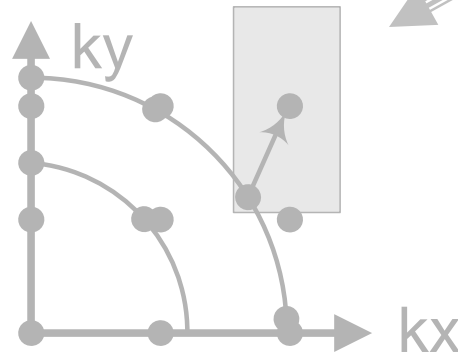
Reconstructing MR Images

Cartesian Scan Data



FFT

Gridding



Least-Squares (LS)

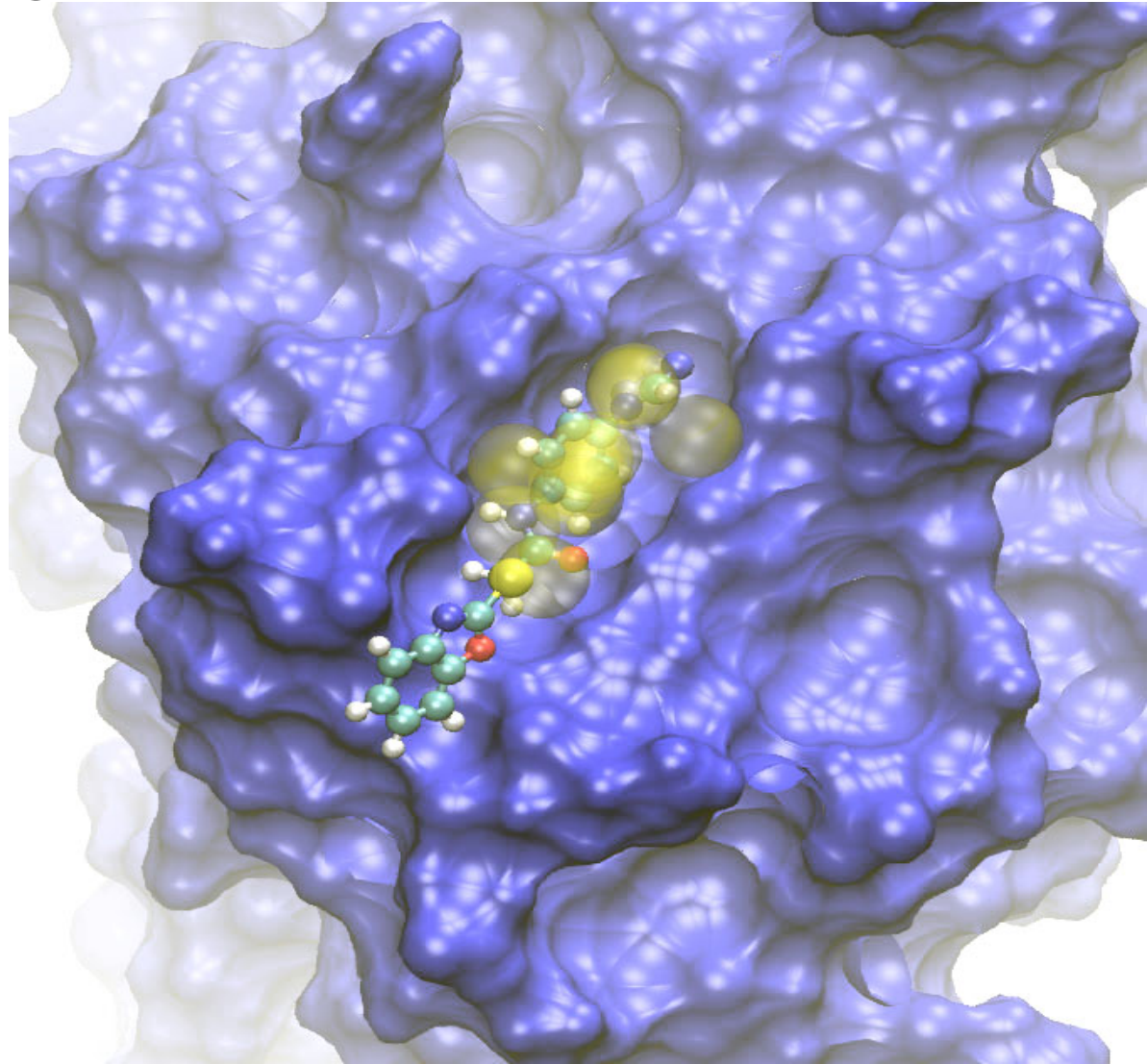
Spiral scan data + LS

Superior images at expense of significantly more computation;
several hundred times slower than gridding.

Traditionally considered impractical!

High-Throughput Computing = Futuristic Biology

- in-silico screening of drugs
- mastering diseases
- personalized medicine
- Thanks: Lorena Barba



In-silico Drug Screening

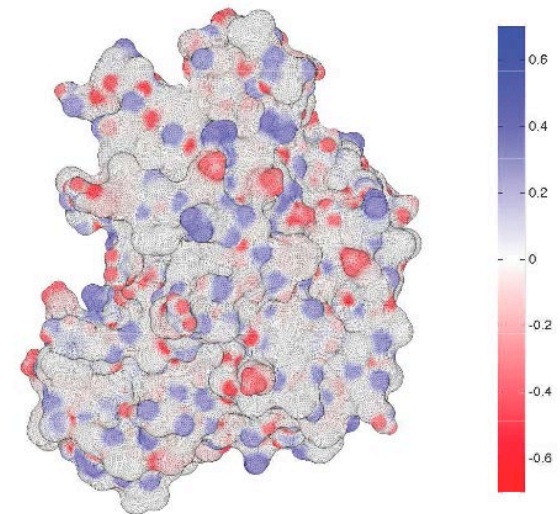
- Weed out inactive compounds
- Rank “drug candidates” for given targets
- Example:
 - CERN grid — 300,000 potential drugs against avian flu screened
 - 2000 computers, 4 weeks!
 - 4 years cpu-time

Protein Aggregation

- a process critical in
 - some degenerative diseases (e.g., Parkinson’s): aggregates abnormal
 - drug production: aggregates undesirable
- time scale of the process:
 - in vitro: up to days!
 - impossible for molecular dynamics

Electrostatic Interactions Play a Crucial Role

- Classical molecular dynamics:
 - very detailed ... but too expensive at large scale!
- Alternative: continuum model of surrounding water
 - don't care what the H₂O molecules do
 - model as a continuum dielectric
 - leads to a boundary integral equation (BIE) problem
- Fast algorithm, well-suited for GPU:
 - fast multipole method, solves BIE in $O(N)$ ops






WIKIPEDIA
The Free Encyclopedia

[Main page](#)
[Contents](#)
[Featured content](#)
[Current events](#)
[Random article](#)
[Donate to Wikipedia](#)
[Wikimedia Shop](#)


Interaction

- [Help](#)
- [About Wikipedia](#)
- [Community portal](#)
- [Recent changes](#)
- [Contact Wikipedia](#)

[Toolbox](#)
[Print/export](#)

Languages 

[Français](#)

 [Edit links](#)

Article [Talk](#)

Read [Edit](#) [View history](#)

Search



Fast multipole method

From Wikipedia, the free encyclopedia

The **fast multipole method (FMM)** is a [mathematical](#) technique that was developed to speed up the calculation of long-ranged forces in the [n-body problem](#). It does this by expanding the system [Green's function](#) using a [multipole expansion](#), which allows one to group sources that lie close together and treat them as if they are a single source.^[1]

The FMM has also been applied in accelerating the iterative solver in the [method of moments](#) (MOM) as applied to [computational electromagnetics](#) problems.^[2] The FMM was first introduced in this manner by [Greengard](#) and [Rokhlin](#)^[3] and is based on the [multipole expansion](#) of the vector [Helmholtz equation](#). By treating the interactions between far-away basis functions using the FMM, the corresponding matrix elements do not need to be explicitly stored, resulting in a significant reduction in required memory. If the FMM is then applied in a hierarchical manner, it can improve the complexity of matrix-vector products in an iterative solver from $O(N^2)$ to $O(N)$. This has expanded the area of applicability of the MOM to far greater problems than were previously possible.

The FMM, introduced by Rokhlin and Greengard, has been acclaimed as one of the top ten [algorithms](#) of the 20th century.^[4] The FMM algorithm dramatically reduces the complexity of matrix-vector multiplication involving a certain type of dense matrix which can arise out of many physical systems.

The FMM has also been applied for efficiently treating the Coulomb interaction in [Hartree–Fock](#) and [density functional theory](#) calculations in [quantum chemistry](#).

See also [\[edit\]](#)

Portals

Access related topics



[Mathematics portal](#)



[Physics portal](#)



[Astronomy portal](#)

References [\[edit\]](#)

- ↑ Rokhlin, Vladimir (1985). "Rapid Solution of Integral Equations of Classic Potential Theory." J. Computational Physics Vol. 60, pp. 187-207.
- ↑ Nader Engheta, William D. Murphy, Vladimir Rokhlin, and Marius Vassiliou (1992), "The Fast Multipole Method for Electromagnetic Scattering Computation." IEEE Transactions on Antennas and Propagation 40, 634-641

12 Steps to a Fast Multipole Method on GPUs

by Dr Rio Yokota
Boston University

Why would I want to learn FMM?

The combination of algorithmic acceleration and hardware acceleration can have tremendous impact. The FMM is a fast algorithm for calculating matrix vector multiplications in $O(N)$ time, and it runs very fast on GPUs. Its combination of high degree of parallelism and $O(N)$ complexity make it an attractive solver for the Peta-scale and Exa-scale era. It has a wide range of applications, e.g. quantum mechanics, molecular dynamics, electrostatics, acoustics, structural mechanics, fluid mechanics, and astrophysics.

What are the prerequisites?

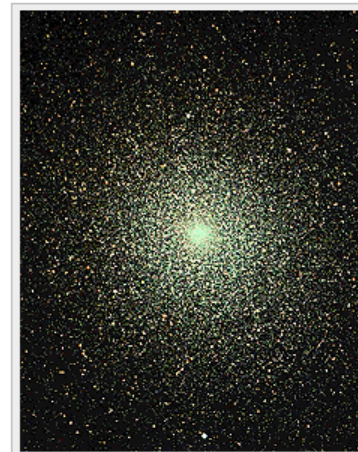
- Basic understanding of linear algebra and calculus
- Some experience with C or C++

The aim of this course

This course will provide a hands-on tutorial with simple exercises that will lead to a full FMM code that runs on GPUs. Each step is carefully planned so that there is no sudden jump in the difficulty. During the tutorial each attendee will have remote access to the GPU cluster in Nagasaki Japan. There will be Teaching Assistants walking around to assist you, so that no one is left behind.

The 12 steps

- step 1 : direct N-body
- step 2 : multipole expansion



Fast multipole methods are used for gravitational N-body problems

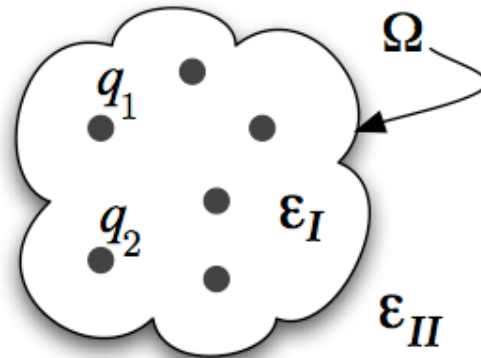
ABOUT

COURSES

- GPU computing and programming
- Building robust scientific codes
- Parallel performance and parallel algorithms
- Python for parallel scientific computing
- GPU programming with PyOpenCL and PyCUDA
- Introduction to numerical linear algebra in parallel
- Quarks, GPUs and Multigrid
- Advanced algorithmic techniques for GPUs
- Iterative methods for sparse linear systems on GPU
- Building & maintaining a cluster of GPUs
- **12 Steps to a Fast Multipole Method on GPUs**
- Advanced computing in solid-earth dynamics
- Boundary-Integral methods in molecular science and engineering
- Computational methods for oil recovery

As in Many Computation-hungry Applications

- Three-step approach:
 - Restructure the mathematical formulation
 - Innovate at the algorithm level
 - Tune core software for hardware architecture



Conclusion: Three Options

- **Good:** “Accelerate” Legacy Codes
 - Recompile/Run
 - Call CUBLAS/CUFFT/thrust/matlab/PGI pragmas/etc.
 - => good work for domain scientists (minimal CS required)
- **Better:** Rewrite / Create new codes
 - Opportunity for clever algorithmic thinking
 - => good work for computer scientists (minimal domain knowledge required)
- **Best:** Rethink Numerical Methods & Algorithms
 - Potential for biggest performance advantage
 - => Interdisciplinary: requires CS and domain insight
 - => Exciting time to be a computational scientist

Think, Understand... then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
 - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
 - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part 😊)

Future Studies

- More complex data structures
- More scalable algorithms and building blocks
- More scalable math models

- Thread-aware approaches
 - More available parallelism
- Locality-aware approaches
 - Computing is becoming bigger, and everything is further away



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

