Parallel Thinking

CSCI 4239/5239
Advanced Computer Graphics
Spring 2013

Objective

- To provide you with a framework based on the techniques and best practices used by experienced parallel programmers for
  - Thinking about the problem of parallel programming
  - Discussing your work with others
  - Addressing performance and functionality issues in your parallel program
  - Using or building useful tools and environments
  - understanding case studies and projects

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, and/or
  - To solve bigger problems, and/or
  - To achieve better solutions

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

A Recommended Reading

Mattson, Sanders, Massingill, *Patterns for Parallel Programming*, Addison Wesley, 2005, ISBN 0-321-22811-1.

- We draw quite a bit from the book
- A good overview of challenges, best practices, and common techniques in all aspects of parallel programming

Key Parallel Programming Steps

- **To find the concurrency in the problem**
- To structure the algorithm so that concurrency can be exploited
- To implement the algorithm in a suitable programming environment
- To execute and tune the performance of the code on a parallel system

Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.

Challenges of Parallel Programming

- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
  - Computational thinking (J. Wing)
- Dependences need to be identified and managed
  - The order of task execution may change the answers
    - Obvious: One step feeds result to the next steps
    - Subtle: numeric accuracy may be affected by ordering steps that are logically parallel with each other
- Performance can be drastically reduced by many factors
  - Overhead of parallel processing
  - Load imbalance among processor elements
  - Inefficient data sharing patterns
  - Saturation of critical resources such as memory bandwidth

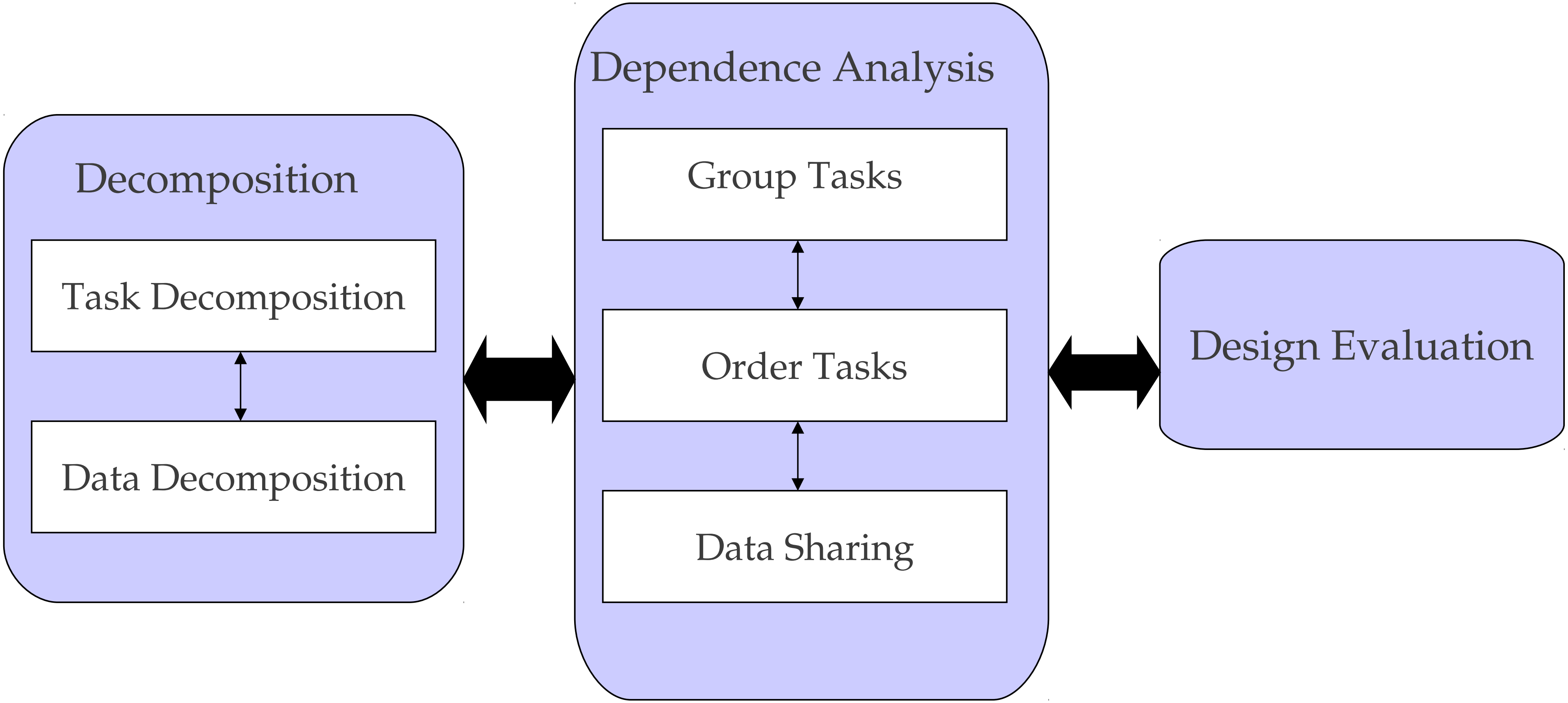
Shared Memory vs. Message Passing

- We will focus on shared memory parallel programming
  - This is what CUDA 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!
  - Look at MPI (Message Passing Interface) and its relatives such as Charm++

Finding Concurrency in Problems

- Identify a decomposition of the problem into sub-problems that can be solved simultaneously
  - A task decomposition that identifies tasks for potential concurrent execution
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering the groups to satisfy temporal constraints
  - An analysis on the data sharing patterns among the concurrent tasks
  - A design evaluation that assesses of the quality the choices made in all the steps

Finding Concurrency – The Process



Decomposition

Task Decomposition

Data Decomposition

Dependence Analysis

Group Tasks

Order Tasks

Data Sharing

Design Evaluation

This is typically a iterative process.
Opportunities exist for dependence analysis to play earlier role in decomposition.

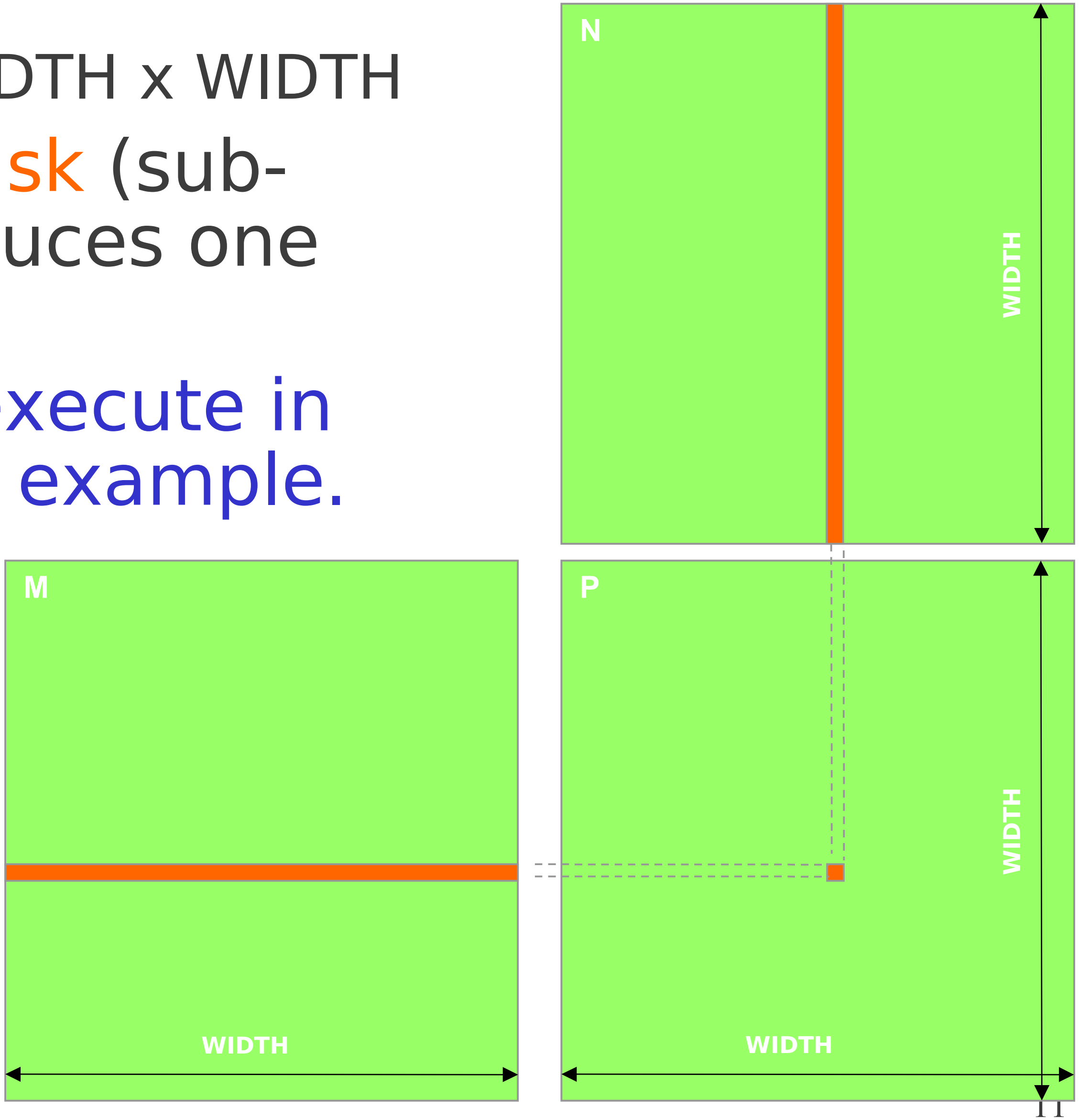
Task Decomposition

- Many large problems can be naturally decomposed into tasks – CUDA kernels are largely tasks
    - The number of tasks used should be adjustable to the execution resources available.
    - Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.
    - Tasks should maximize reuse of sequential program code to minimize effort.

"In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens."
- Mattson, Sanders, Massingill

Task Decomposition Example - Square Matrix Multiplication

- P = M * N of WIDTH x WIDTH
  - One natural task (sub-problem) produces one element of P
  - All tasks can execute in parallel in this example.

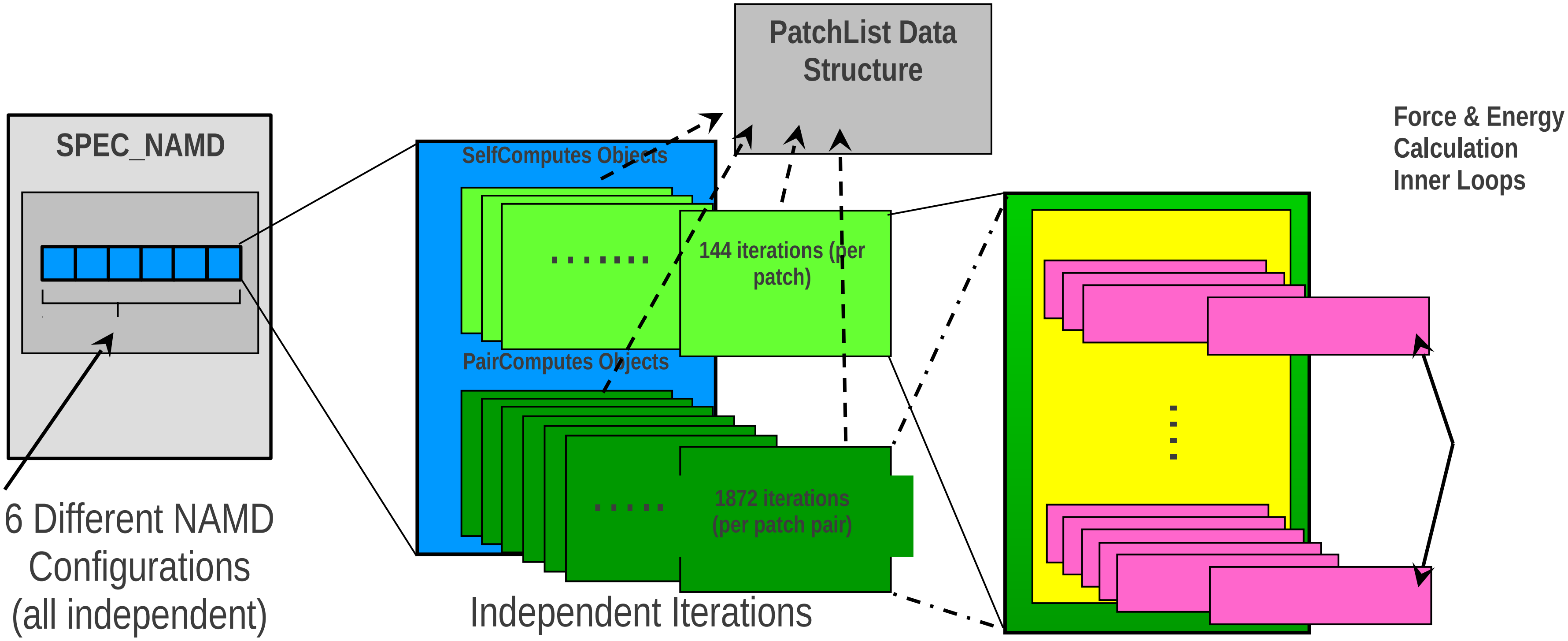


Task Decomposition Example – Molecular Dynamics

- Simulation of motions of a large molecular system
- For each atom, there are natural tasks to calculate
  - Vibrational forces
  - Rotational forces
  - Neighbors that must be considered in non-bonded forces
  - Non-bonded forces
  - Update position and velocity
  - Misc physical properties based on motions
- Some of these can go in parallel for an atom

It is common that there are multiple ways to decompose any given problem.

NAMD



6 Different NAMD Configurations (all independent)

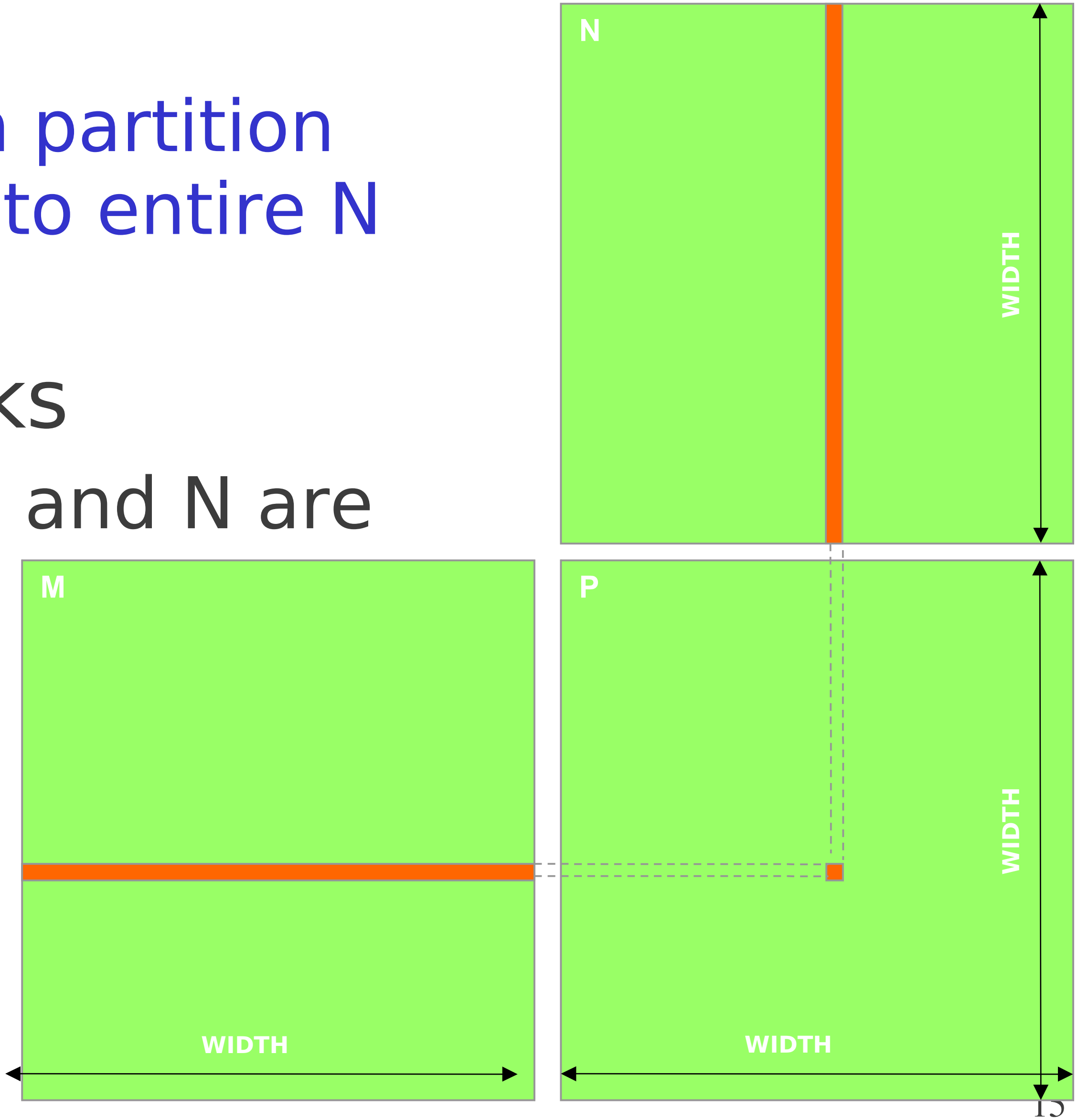
Independent Iterations

Data Decomposition

- The most compute intensive parts of many large problem manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner.
  - This is what CUDA is optimized for.
- The data decomposition should lead to
  - Efficient data usage by tasks within the partition
  - Few dependencies across the tasks that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics

Data Decomposition Example - Square Matrix Multiplication

- Row blocks
  - Computing each partition requires access to entire N array
- Square sub-blocks
  - Only bands of M and N are needed

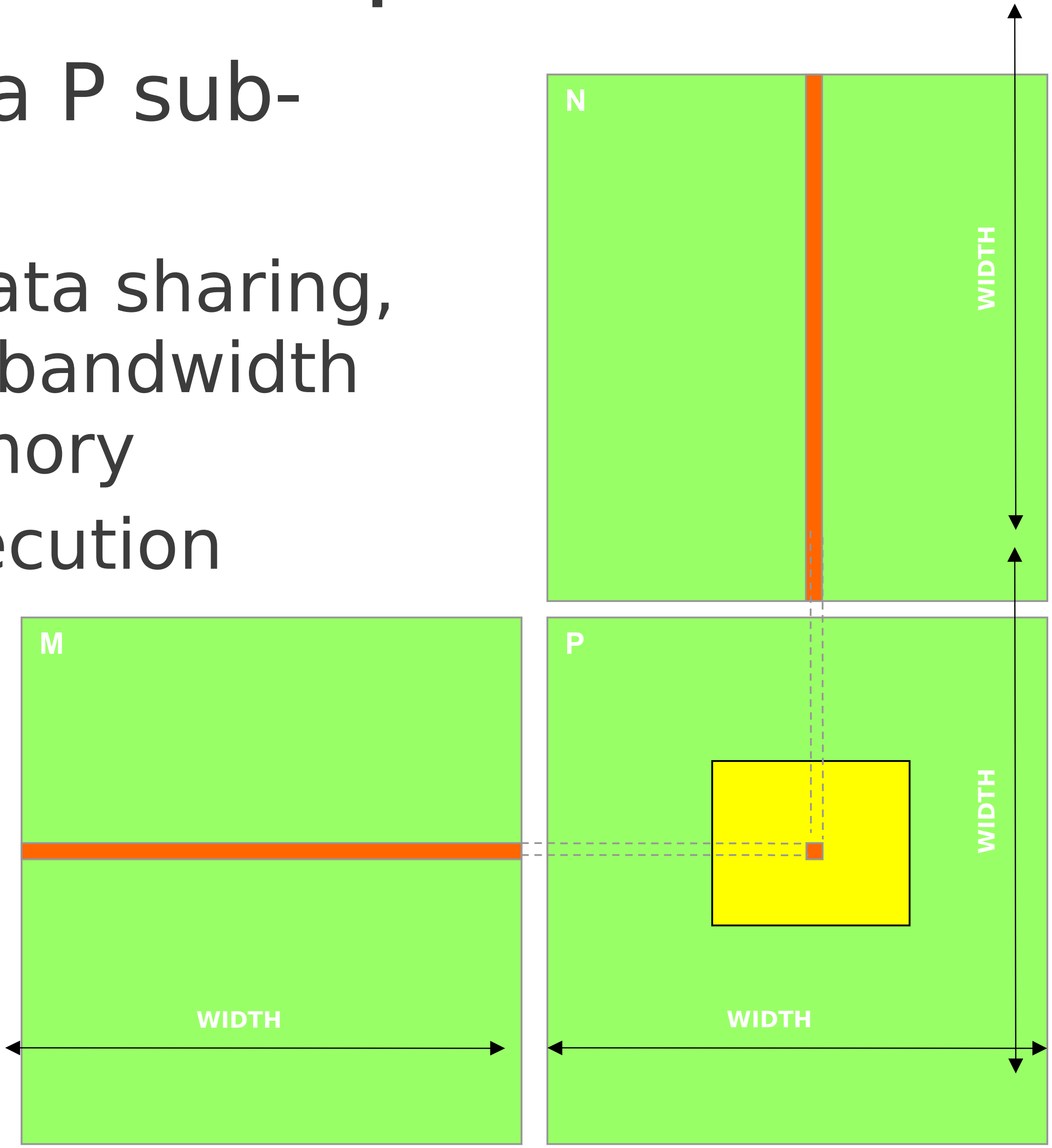


Tasks Grouping

- Sometimes natural tasks of a problem can be grouped together to improve efficiency
  - Reduced synchronization overhead – all tasks in the group can use a barrier to wait for a common dependence
  - All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shard Memory)
  - Grouping and merging dependent tasks into one task reduces need for synchronization
  - CUDA thread blocks are task grouping examples.

Task Grouping Example - Square Matrix Multiplication

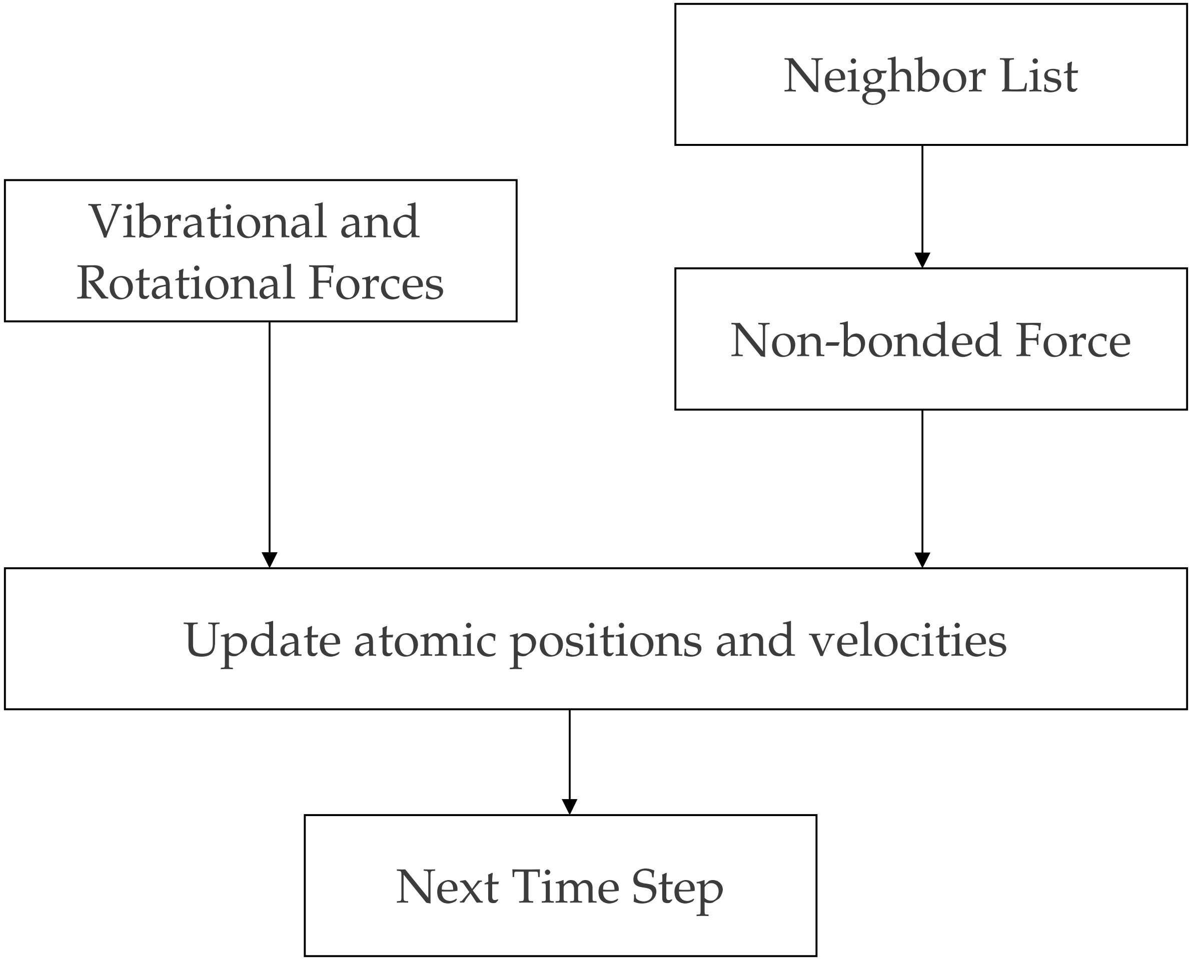
- Tasks calculating a P sub-block
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution



Task Ordering

- Identify the data and resource required by a group of tasks before they can execute them
  - Find the task group that creates it
  - Determine a temporal order that satisfy all data constraints

Task Ordering Example: Molecular Dynamics



Neighbor List

Vibrational and Rotational Forces

Non-bonded Force

Update atomic positions and velocities

Next Time Step

Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing can drastically reduce 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
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires synchronization

Data Sharing Example – Matrix Multiplication

- Each task group will finish usage of each sub-block of N and M before moving on
  - N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  - Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block
- Read-only shared data can be more efficiently accessed as Constant or Texture data

Data Sharing Example – Molecular Dynamics

- The atomic coordinates
  - Read-only access by the neighbor list, bonded force, and non-bonded force task groups
  - Read-write access for the position update task group
- The force array
  - Read-only access by position update group
  - Accumulate access by bonded and non-bonded task groups
- The neighbor list
  - Read-only access by non-bonded force task groups
  - Generated by the neighbor list task group

Key Parallel Programming Steps

- To find the concurrency in the problem
- **To structure the algorithm to translate concurrency into performance**
- To implement the algorithm in a suitable programming environment
- To execute and tune the performance of the code on a parallel system

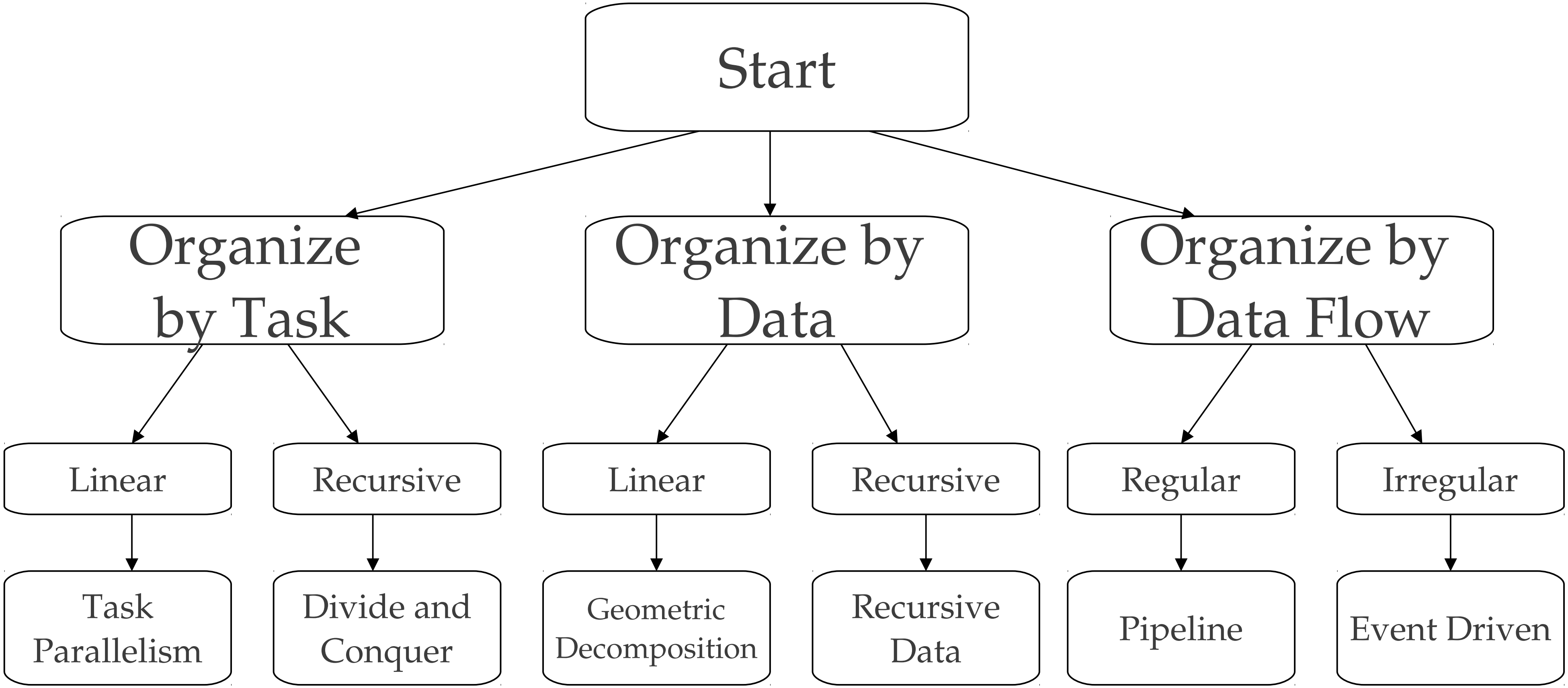
Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.

Algorithm

- A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  - Definiteness – the notion that each step is precisely stated
  - Effective computability – each step can be carried out by a computer
  - Finiteness – the procedure terminates

- Multiple algorithms can be used to solve the same problem
  - Some require fewer steps
  - Some exhibit more parallelism
  - Some have larger memory footprint than others

Choosing Algorithm Structure



Start

Organize by Task

Organize by Data

Organize by Data Flow

Linear

Recursive

Linear

Recursive

Regular

Irregular

Task Parallelism

Divide and Conquer

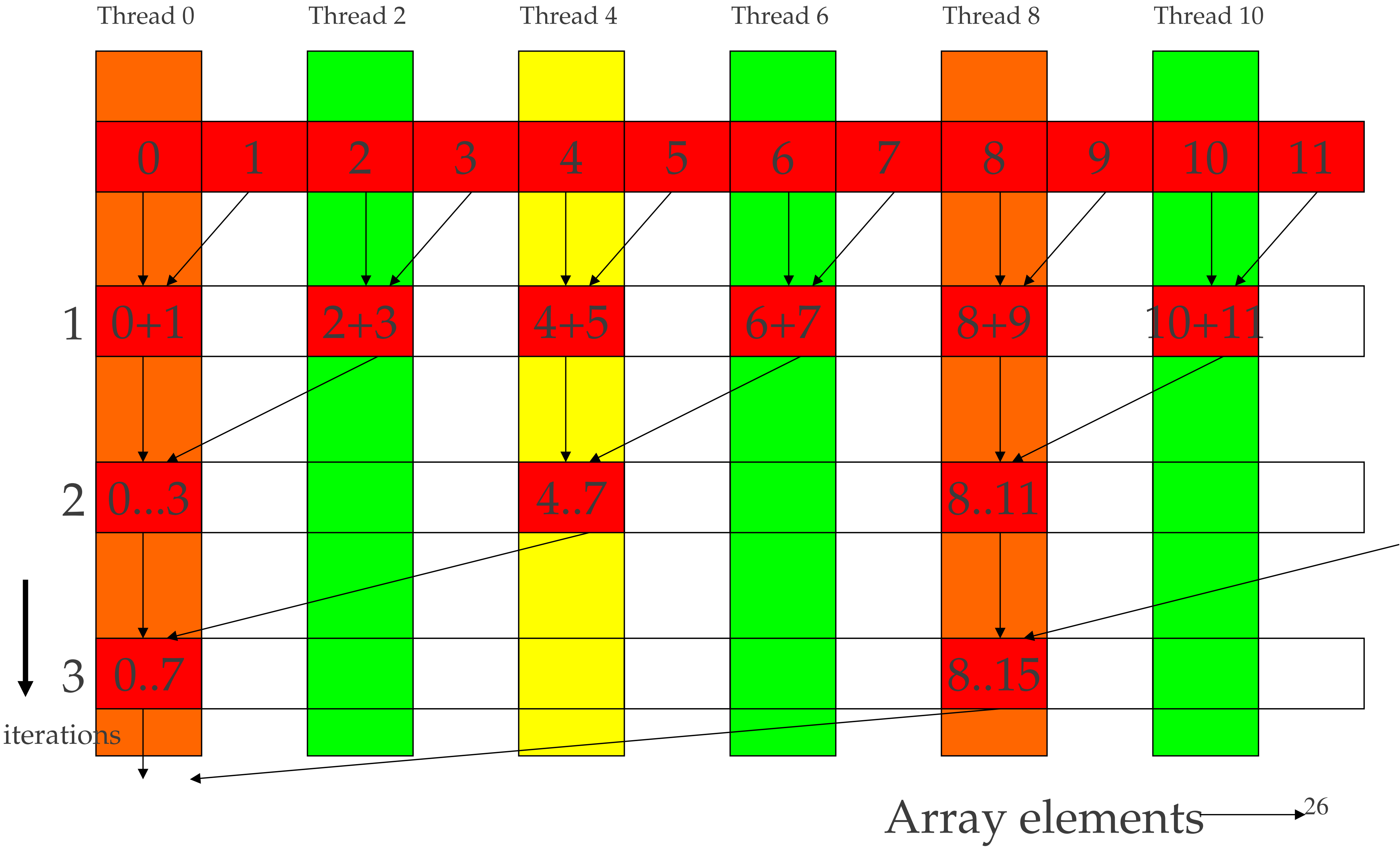
Geometric Decomposition

Recursive Data

Pipeline

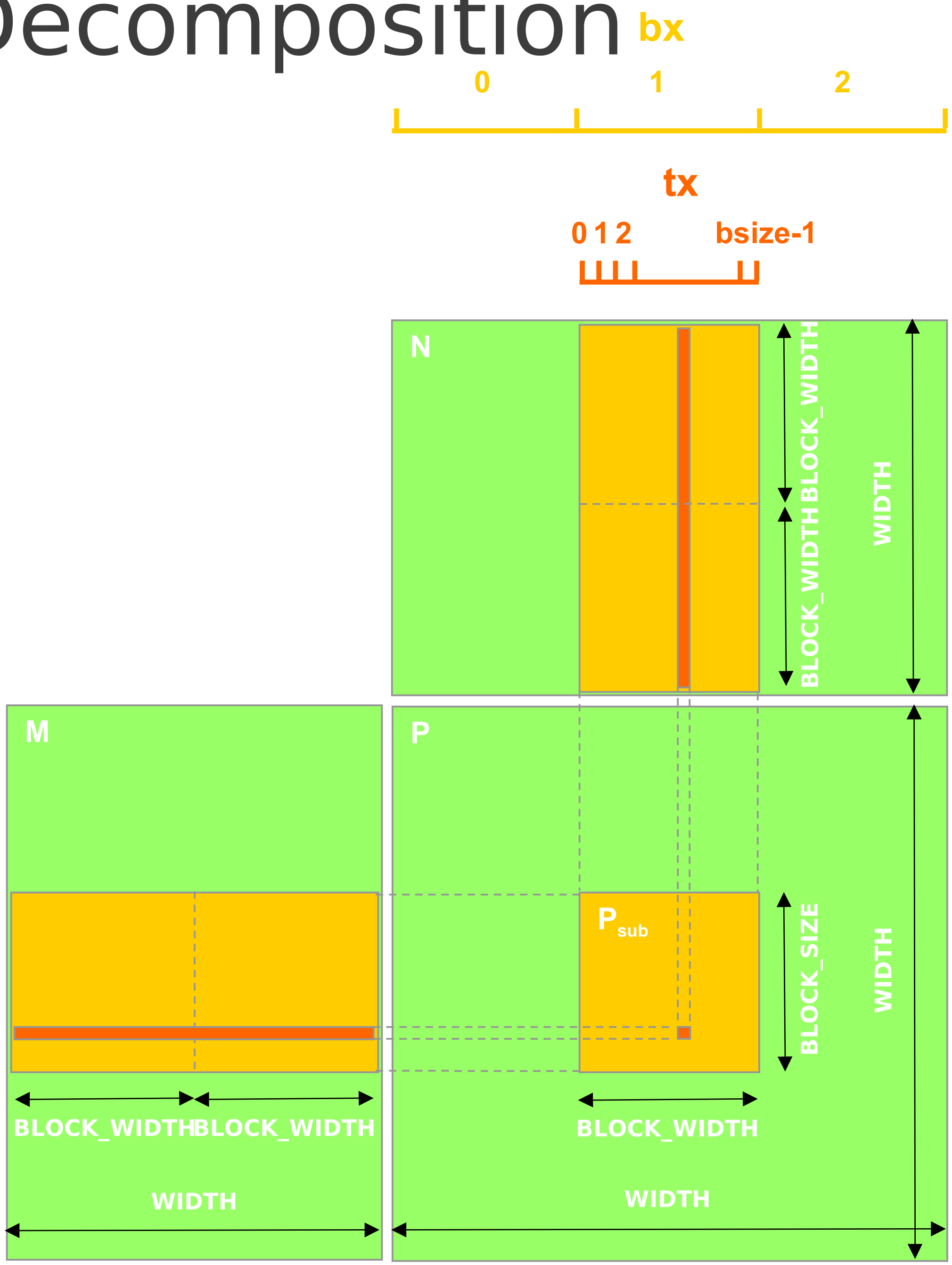
Event Driven

Mapping a Divide and Conquer Algorithm



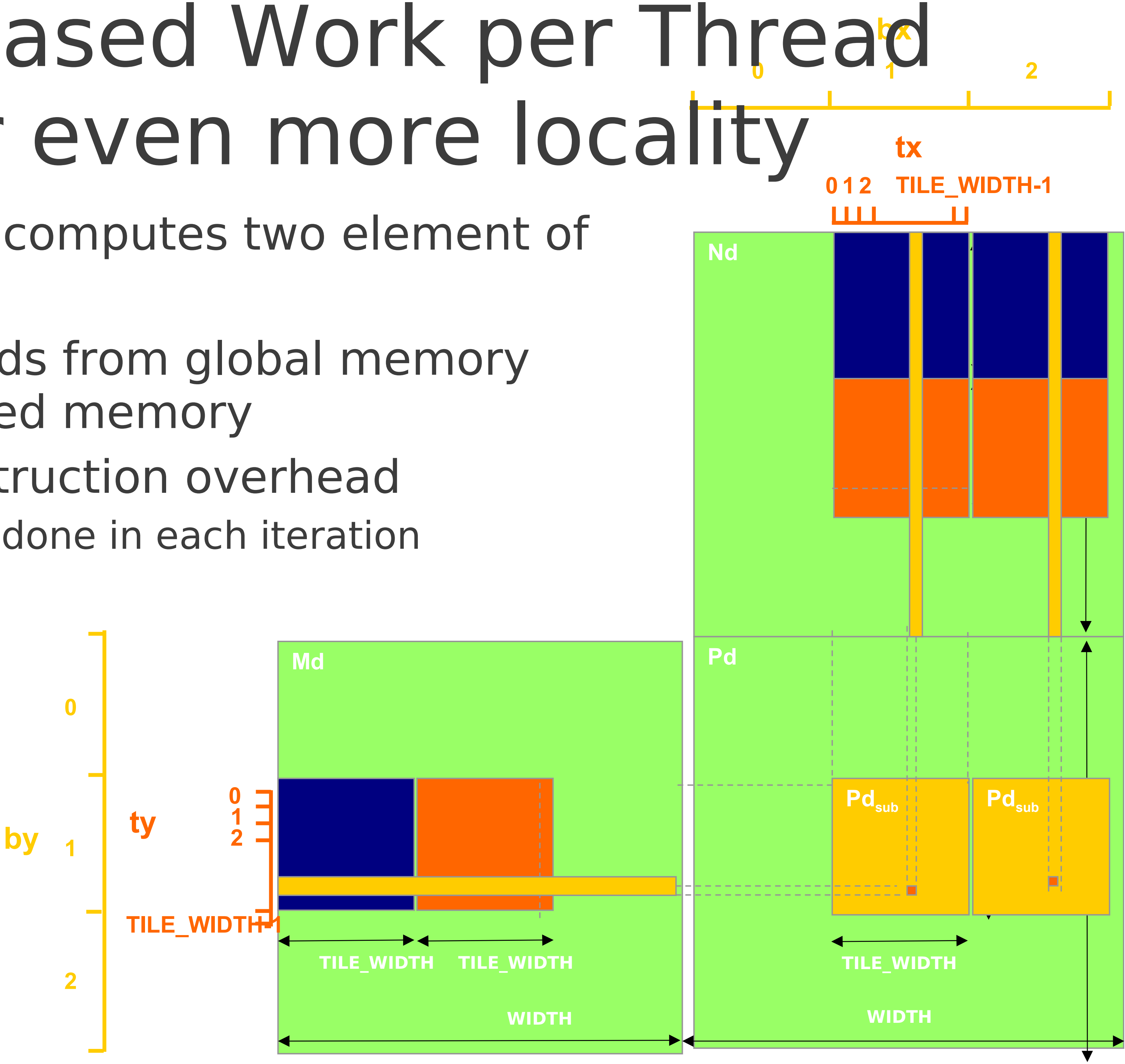
Tiled (Stenciled) Algorithms are Important for Geometric Decomposition

- A framework for memory data sharing and reuse by increasing data access locality.
  - Tiled access patterns allow small cache/scartchpad memories to hold on to data for re-use.
  - For matrix multiplication, a 16X16 thread block perform 2*256 = 512 float loads from device memory for 256 * (2*16) = 8,192 mul/add operations.
- A convenient framework for organizing threads (tasks)



Increased Work per Thread for even more locality

- Each thread computes two element of Pd_{sub}
- Reduced loads from global memory (Md) to shared memory
- Reduced instruction overhead
  - More work done in each iteration



Double Buffering
- a frequently used algorithm pattern

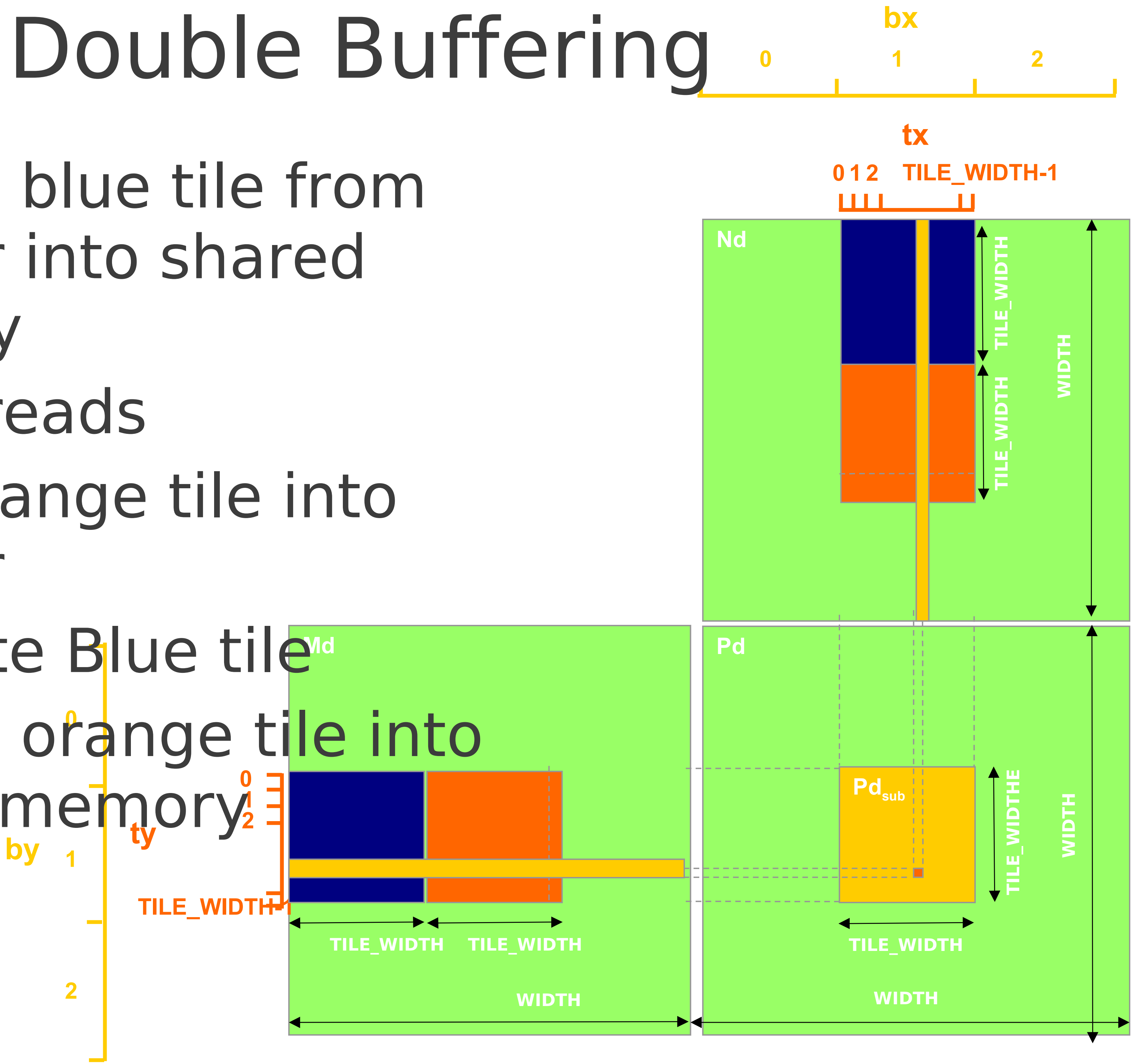
- One could double buffer the computation, getting better instruction mix within each thread
  - This is classic software pipelining in ILP compilers

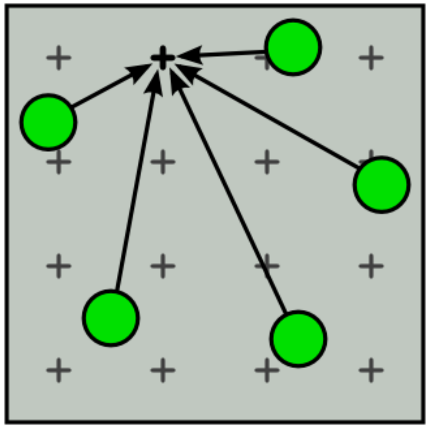
```
Loop {

  Load current tile to shared memory

  syncthreads()

  Compute current tile

  syncthreads()
}
```

```
Load next tile from global memory

Loop {
  Deposit current tile to shared memory

  syncthreads()

  Load next tile from global memory

  Compute current tile

  syncthreads()
}
```

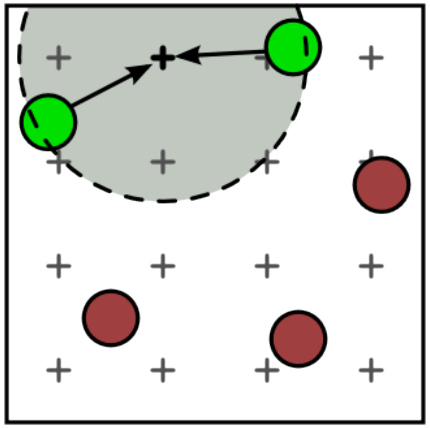
Double Buffering

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ….

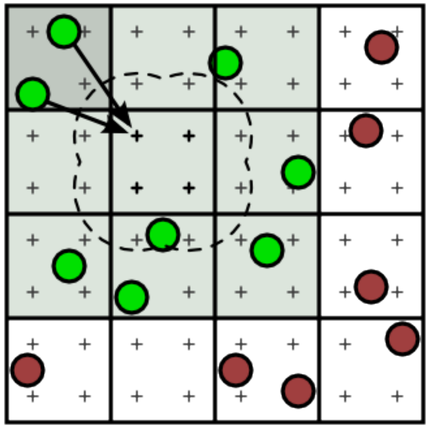




(a) Direct summation
At each grid point, sum the electrostatic potential from all charges



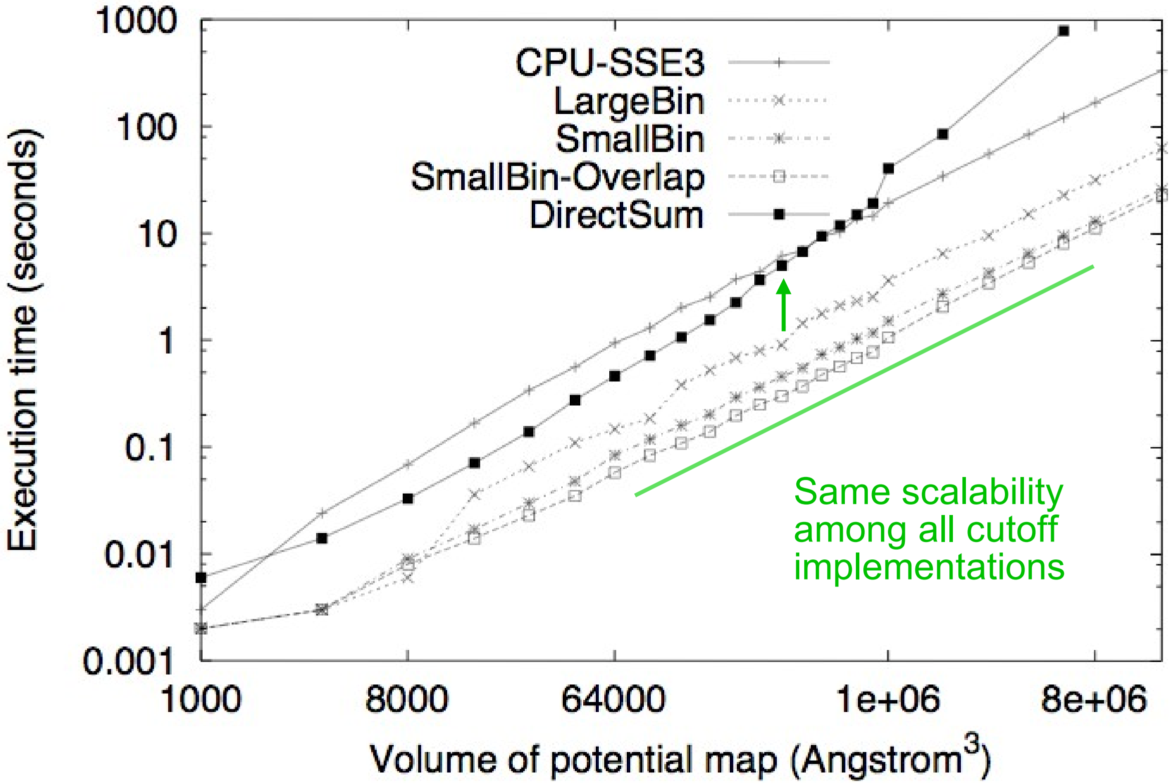
(b) Cutoff summation
Electrostatic potential from nearby charges summed; spatially sort charges first



(c) Cutoff summation using direct summation kernel
Spatially sort charges into bins; adapt direct summation to process a bin

Figure 10.2 Cutoff Summation algorithm

Cut-Off Summation Restores Data Scalability



Same scalability among all cutoff implementations

Scalability and Performance of different algorithms for calculating electrostatic potential map.