Algorithmen für die Echtzeitgrafik

Algorithmen für die Echtzeitgrafik

Daniel Scherzer
scherzer@cg.tuwien.ac.at

LBI Virtual Archeology

Parallel Computing

Basics

Fundamentals of Parallel Computing

- Parallel computing requires
  - Problem decomposable into sub-problems
    - Savely solvable at the same time
  - Programmer structures code and data accordingly
- The goals of parallel computing are
  - To solve problems in less time
  - To solve bigger problems
- The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Key Parallel Programming Steps

1. Find concurrency in problem
2. Structure algorithm to exploit concurrency
3. Implement algorithm in suitable programming environment
4. Execute and tune the performance of the code on a parallel system
5. Probably start again from the top

- Dependent problems!

Challenges of Parallel Programming

- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
- 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

Challenges of Parallel Programming

- 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

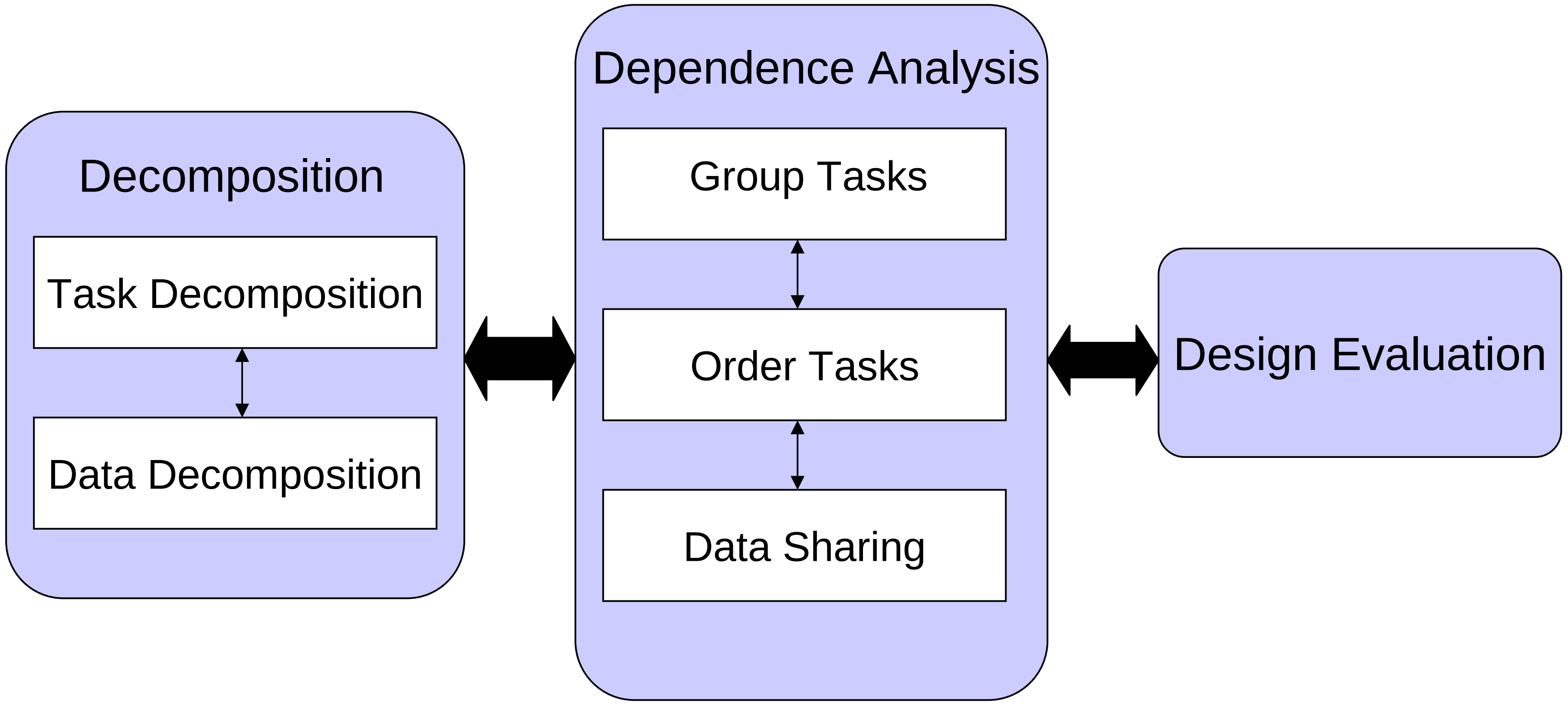
Shared Memory vs. Message Passing

- Focus on shared memory parallel programming
  - Basis of CUDA/OpenCL
  - Expectation: future massively parallel microprocessors support shared memory at the chip level
- The programming considerations of message passing model is quite different!

Finding Concurrency in Problems

- Identify decomposition of problem into sub-problems that can be solved simultaneously
  - Task decomposition: identifies tasks that can execute concurrently
  - Data decomposition: identifies data local to each task
  - 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 the quality of the choices made in all the steps

Finding Concurrency – The Process



This is typically an iterative process.

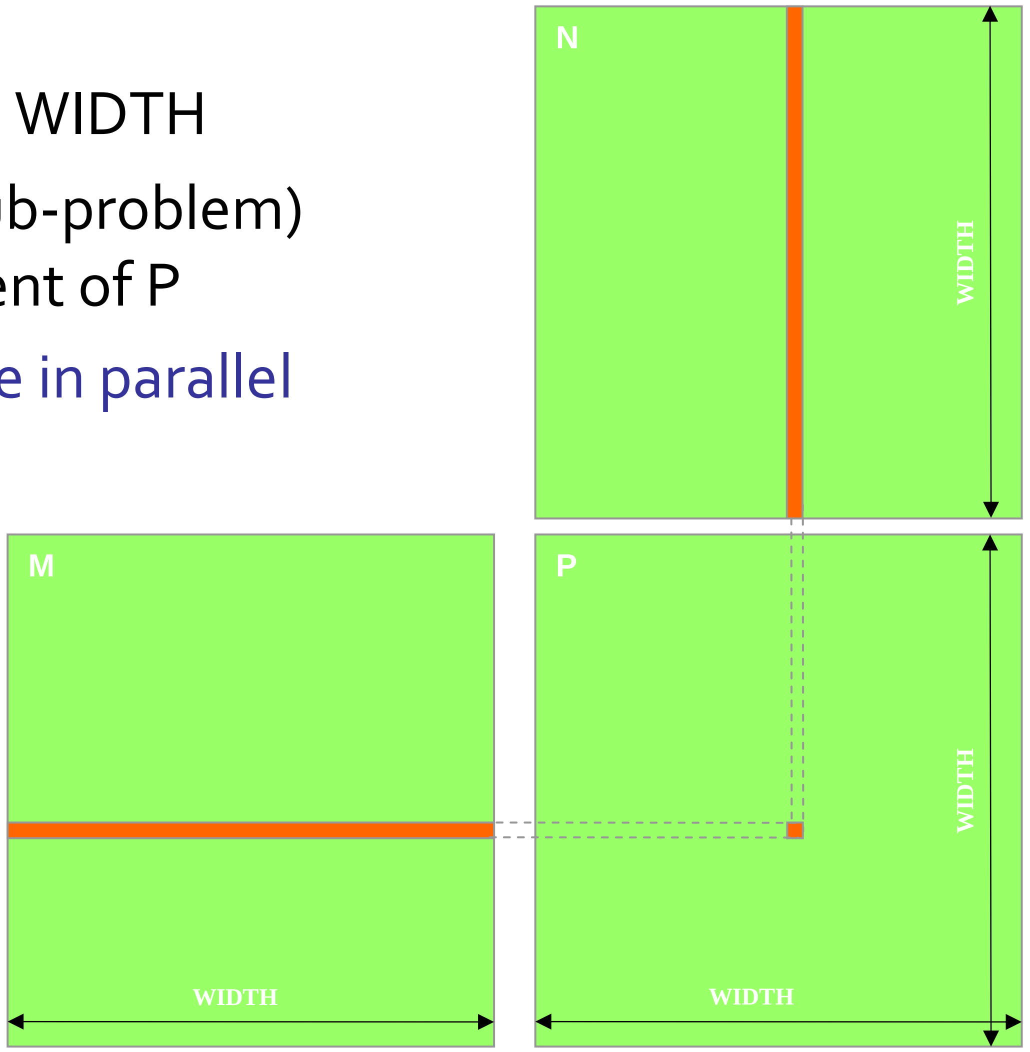
Task Decomposition

- Large problems often have independent tasks
  - # adjustable to execution resources available
  - Big enough to compensate for overhead of managing parallel execution
  - 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 Decomp. Example - Matrix Mul

- P = M * N of WIDTH ● WIDTH
  - One natural task (sub-problem) produces one element of P
  - All tasks can execute in parallel
  - Up to M*N tasks



Task Decomp. Example – Particle System

- Simulation of motions of a large system
- For each particle, there are natural tasks to calculate
  - Update position and velocity
  - Neighbors that must be considered
  - Misc physical properties based on motions
- Some of these can go in parallel for a particle

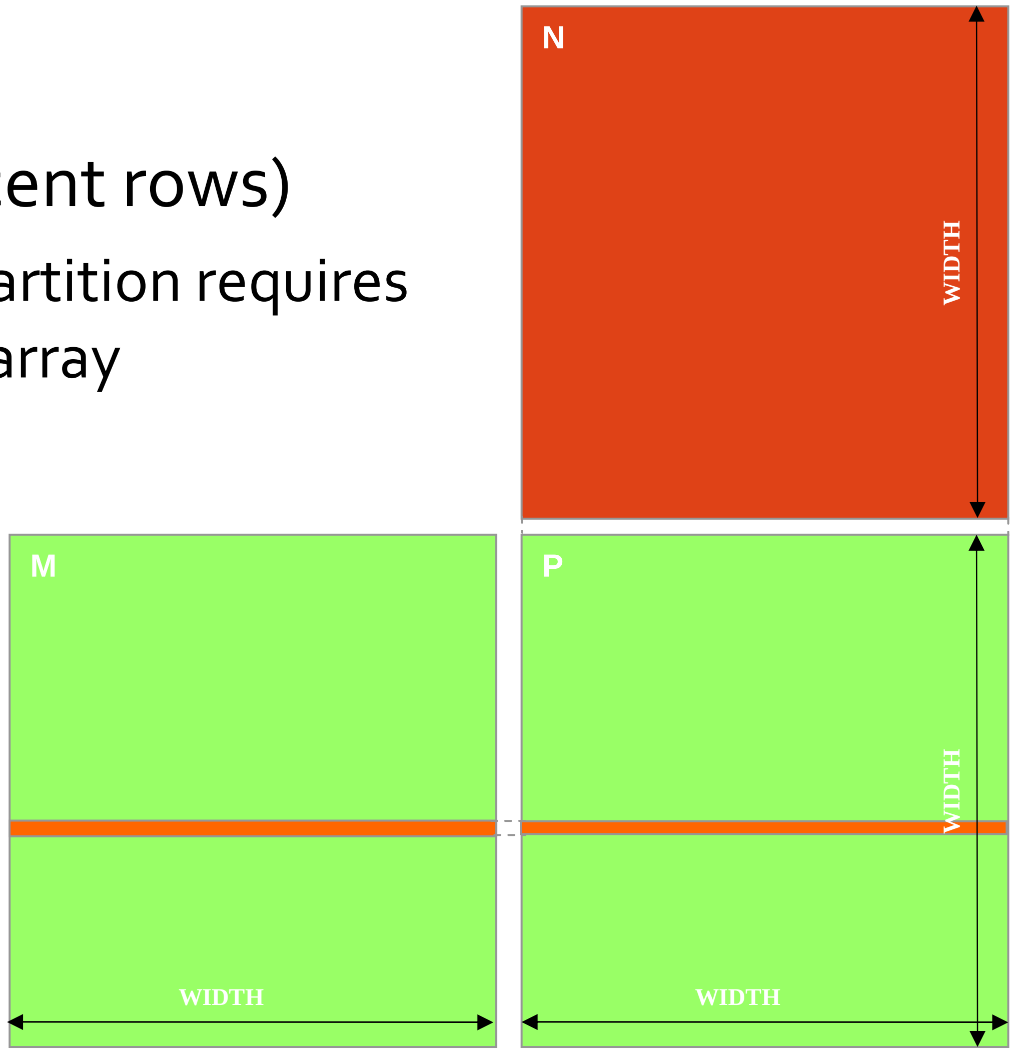
It is common that there are multiple ways to decompose any given problem.

Data Decomposition

- Most compute intensive parts of many large problem is manipulation of large data structure
  - Similar operations applied to different parts of data structure - mostly independent
  - CUDA is optimized for this.
- The data decomposition should lead to
  - Efficient data usage by tasks within partition
  - Few dependencies across tasks that work on different partitions
  - Adjustable partitions according to the hardware characteristics

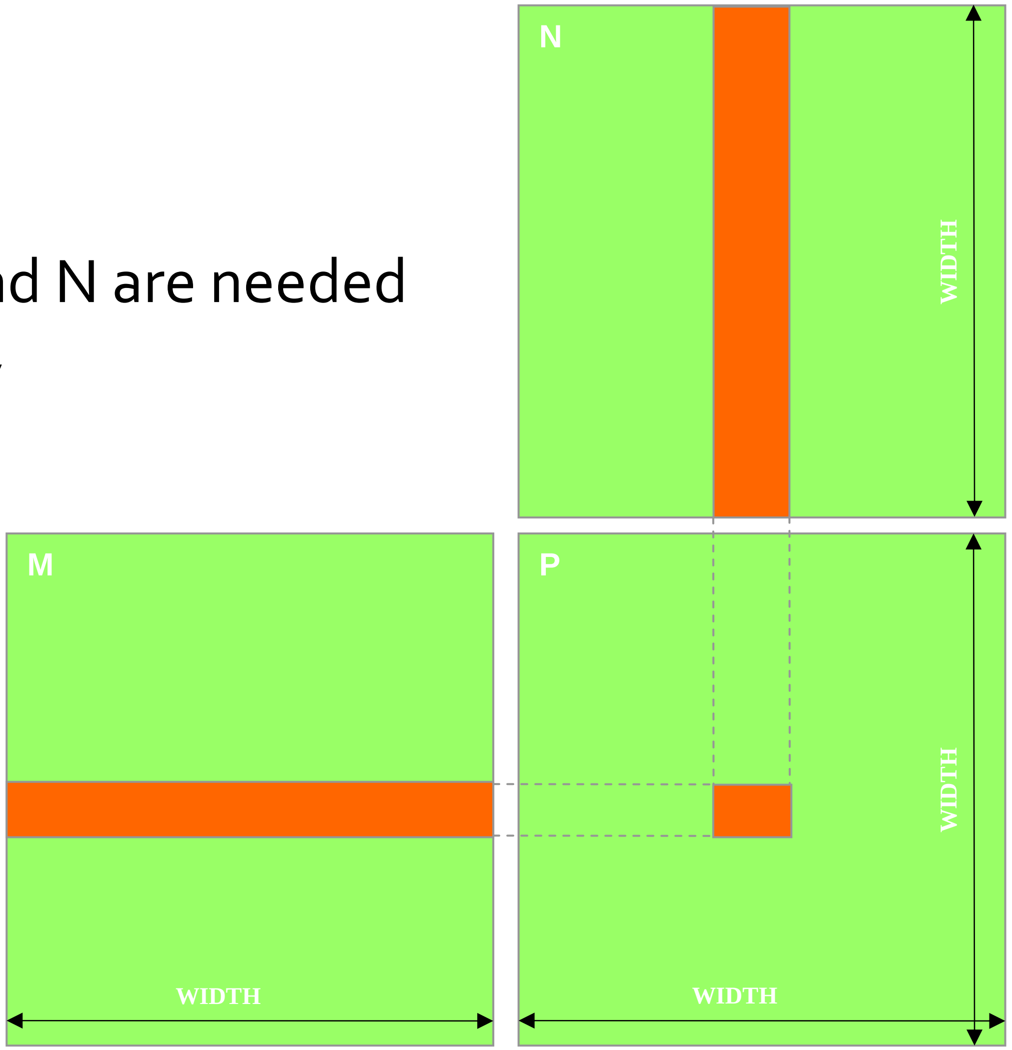
Data Decomp. Example - Matrix Mul.

- P Row blocks (adjacent rows)
  - Computing each partition requires access to entire N array
  - Bad data locality



Data Decomp. Example - Matrix Mul.

- Square sub-blocks
  - Only bands of M and N are needed
  - Better data locality
  - Less cache usage

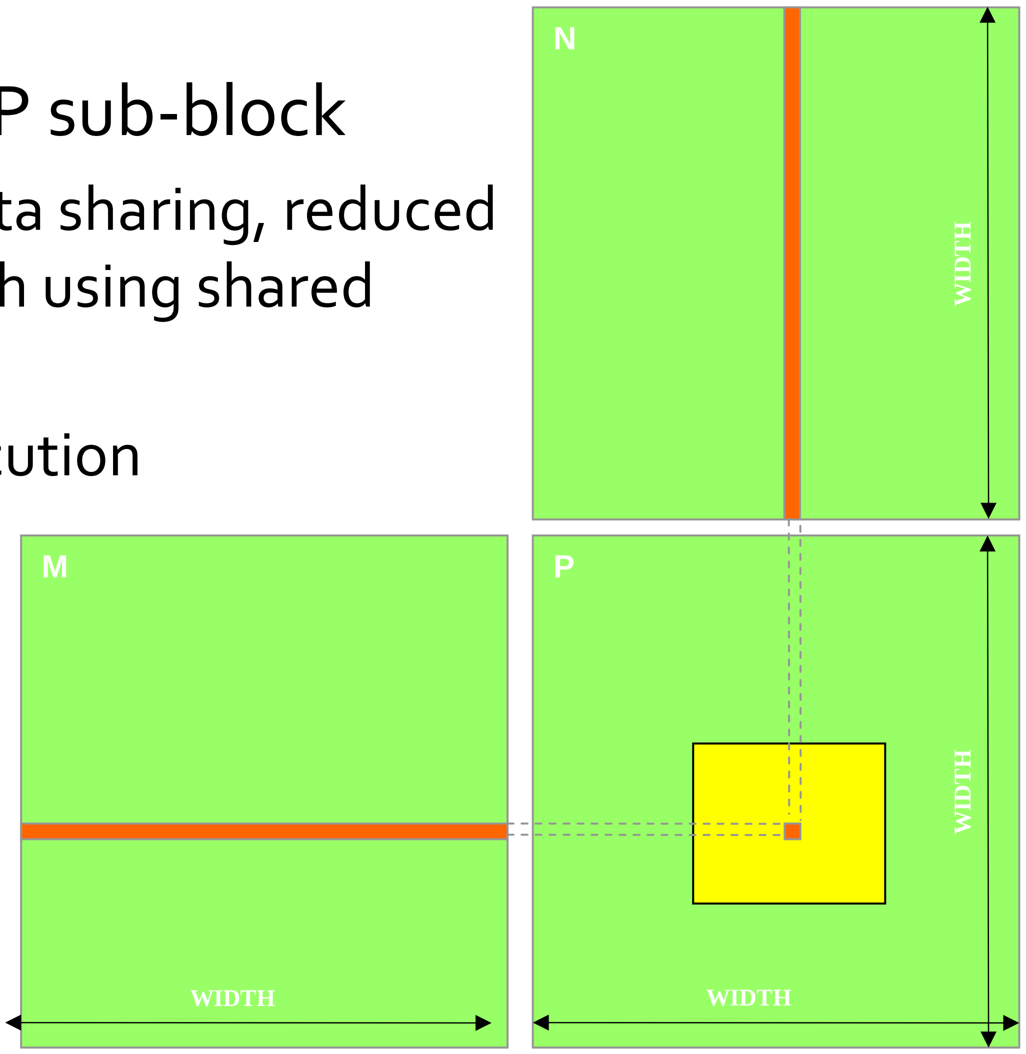


Tasks Grouping

- Sometimes natural tasks of a problem can be grouped together to improve efficiency
  - Reduced synchronization overhead
    - Use barrier to wait for common dependence
  - Efficiently share data
    - Data on chip (shared memory)
  - Reduces need for synchronization
    - Because of reduced dependencies
  - CUDA thread blocks are task grouping examples.

Task Grouping Example - Matrix Mul

- 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 creating task group
  - Determine temporal order that satisfy all data constraints

Data Sharing

- Double-edged sword
  - If excessive can reduce advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth
  - Synchronizing the execution of task groups
  - Coordinating their usage of memory data
  - Efficient use of on-chip, shared storage
- Read-only much faster than read-write (sync)

Data Sharing Example – Matrix Mul

- Task group finishes usage of each sub-block before moving on
  - N and M sub-blocks in shared memory used by all threads of P sub-block (coordinate use)
  - Amount of on-chip shared memory limits number of threads/sub-blocks (efficient use)
- CUDA: Read-only shared data can be more efficiently accessed as Constant or Texture data

Design Evaluation

- Key questions to ask
  - How many threads can be supported?
  - How many threads are needed?
  - How are the data structures shared?
  - Is there enough work in each thread between synchronizations to make parallel execution worthwhile?

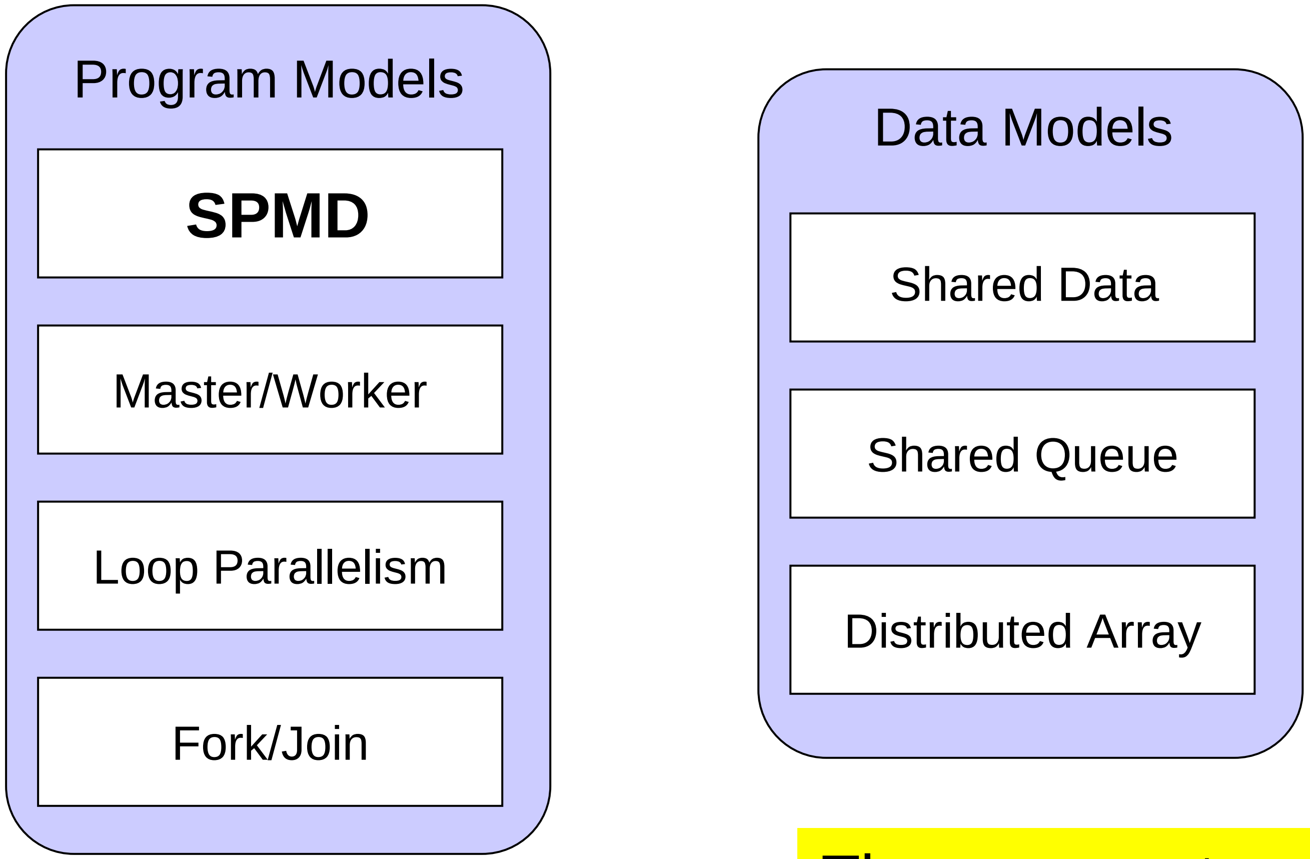
Design Eval Example – Matrix Mul On G92

- Min. 32 FOPS/thread between sync
- Min. 192 threads/block to fully utilize hardware
- Sub-blocks of each thread group must fit into 16KB of shared memory
- Result: 16*16 sub-block best
- Minimal matrix size
  - 1K*1K to make parallel execution worthwhile

Coding Styles - Objective

- Understand SPMD (CUDA coding style)
  - Relation to other parallel programming coding styles
  - Effect on applications
  - How to implement with this coding style
    - Apps with different levels of efficiency and complexity

Coding and Data Styles



These are not necessarily mutually exclusive.

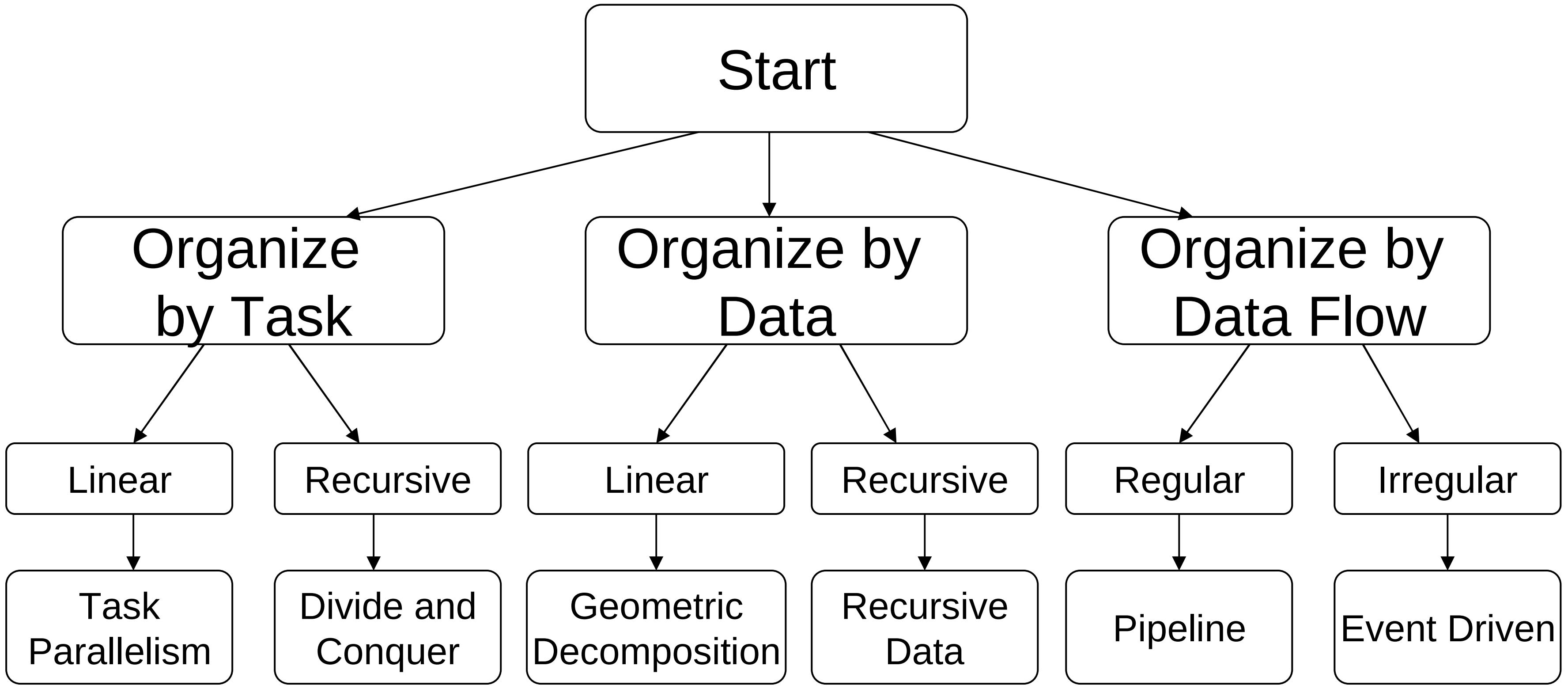
Program Models

- SPMD (Single Program, Multiple Data)
  - All parallel computational elements (PE) 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
  - CUDA grid model
  - SIMD is a special case
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

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

Algorithm Structure



Algorithm Structures vs. Coding Styles

	Task Parallel	Divide/ Conquer	Geometric Decomp.	Recursive Data	Pipeline	Event-based
SPMD	☺☺☺☺	☺☺☺	☺☺☺☺	☺☺	☺☺☺	☺☺
Loop Parallel	☺☺☺☺	☺☺	☺☺☺			
Master/ Worker	☺☺☺☺	☺☺	☺	☺	☺	☺
Fork/ Join	☺☺	☺☺☺☺	☺☺		☺☺☺☺	☺☺☺☺

Coding Styles vs. Programming Models

	OpenMP	MPI	CUDA
SPMD	☺ ☺ ☺	☺ ☺ ☺ ☺	☺ ☺ ☺ ☺ ☺
Loop Parallel	☺ ☺ ☺ ☺	☺	
Master/ Slave	☺ ☺	☺ ☺ ☺	
Fork/Join	☺ ☺ ☺		

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 parallel programs.

Typical Program Phases

1. Initialize
   - Establish localized data structure and communication channels
2. 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.

Typical Program Phases

3. 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
4. Run the core computation
   - More details in next slide…
5. Finalize
   - Reconcile global data structure
   - Prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of different threads
  - Use thread ID in loop index calculations to split loop iterations among threads
  - Use thread ID or conditions based on thread ID to branch to their specific actions

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

A Simple Example

- Assume
  - The computation being parallelized has 1,000,000 iterations.

- Sequential code:

```
Num_steps = 1000000;

for (i=0; i< num_steps, i++) {
    …
}
```

SPMD Code Version 1

- Assign a chunk of iterations to each thread
  - The last thread also finishes up the remainder iterations

```
num_steps  = 1000000;
i_start = my_id * (num_steps/num_threads);
i_end = i_start + (num_steps/num_threads);
//check if last thread
if my_id – (num_threads-1) i_end = num_steps;

for (i = i_start; i < i_end; i++) {
….
}
Reconciliation of results across threads if
necessary.
```

Problems with Version 1

- The last thread executes extra iterations
- Last thread can have nearly twice as many iterations
- Can create serious load imbalance problems.
- Also, the extra if statement is a typical source of “branch divergence” in CUDA programs.
  - This is typically handled by instruction predication.

SPMD Code Version 2

- Assign one more iterations to some of the threads

```
int rem = num_steps % num_threads;
i_start = my_id * (num_steps/num_threads);
i_end = i_start + (num_steps/num_threads);

if (rem != 0) {
  if (my_id < rem) {
            i_start += my_id;
            i_end += (my_id +1);
  }
  else {
            i_start += rem;
            i_end += rem;
  }
}
```

Less load imbalance

More branch divergence.

SPMD Code Version 3

- Use cyclic distribution of iteration

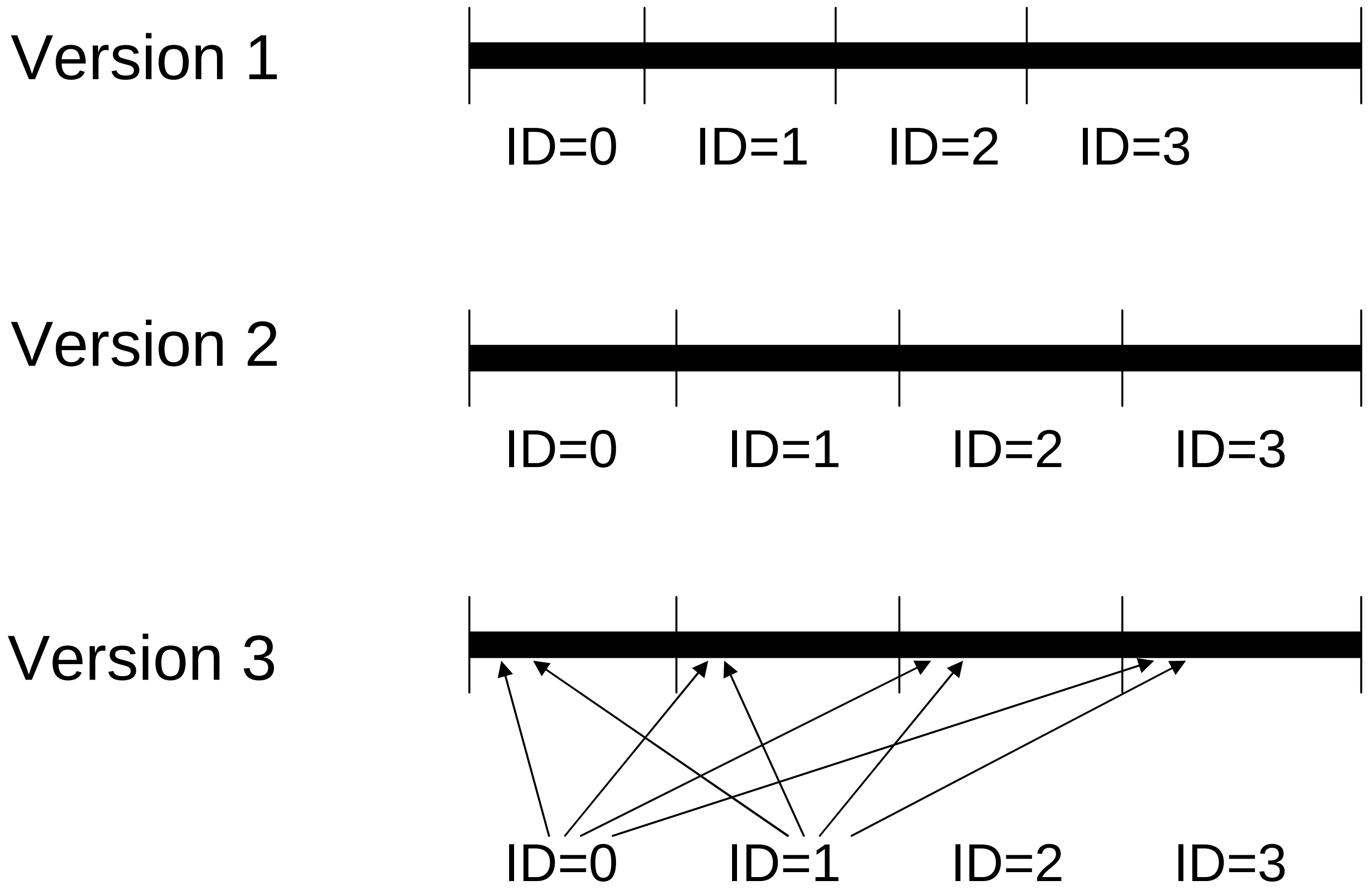
```
num_steps  = 1000000;

for (i = my_id; i < num_steps; i+= num_threads) {
….
}
```

Less load imbalance

No branch divergence.

Comparing the Three Versions



Data Styles

- Shared Data
    - All threads share a major data structure
    - This is what CUDA supports
- Shared Queue
    - All threads see a “thread safe” queue that maintains ordering of data communication
- Distributed Array
    - Decomposed and distributed among threads
    - Limited support in CUDA Shared Memory