

CS 677: Parallel Programming for Many-core Processors

Lecture 9

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

Outline

- Computational Thinking
 - Chapter 13 in K&H
- Asynchronous Memory Transfer
- CUDA Streams

Kirk & Hwu Chapter 13: Computational Thinking

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
 - 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*.

Key Parallel Programming Steps

- 1) To find the concurrency in the problem
- 2) To structure the algorithm so that concurrency can be exploited
- 3) To implement the algorithm in a suitable programming environment
- 4) 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.

Amdahl's Law

- “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”
- Example
 - 95% of original execution time can be sped up by 100x on GPU
 - Speed up for entire application:

$$\frac{1}{(5\% + \frac{95\%}{100})} = \frac{1}{5\% + 0.95\%} = \frac{1}{5.95\%} = 17x$$

Challenges of Parallel Programming

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

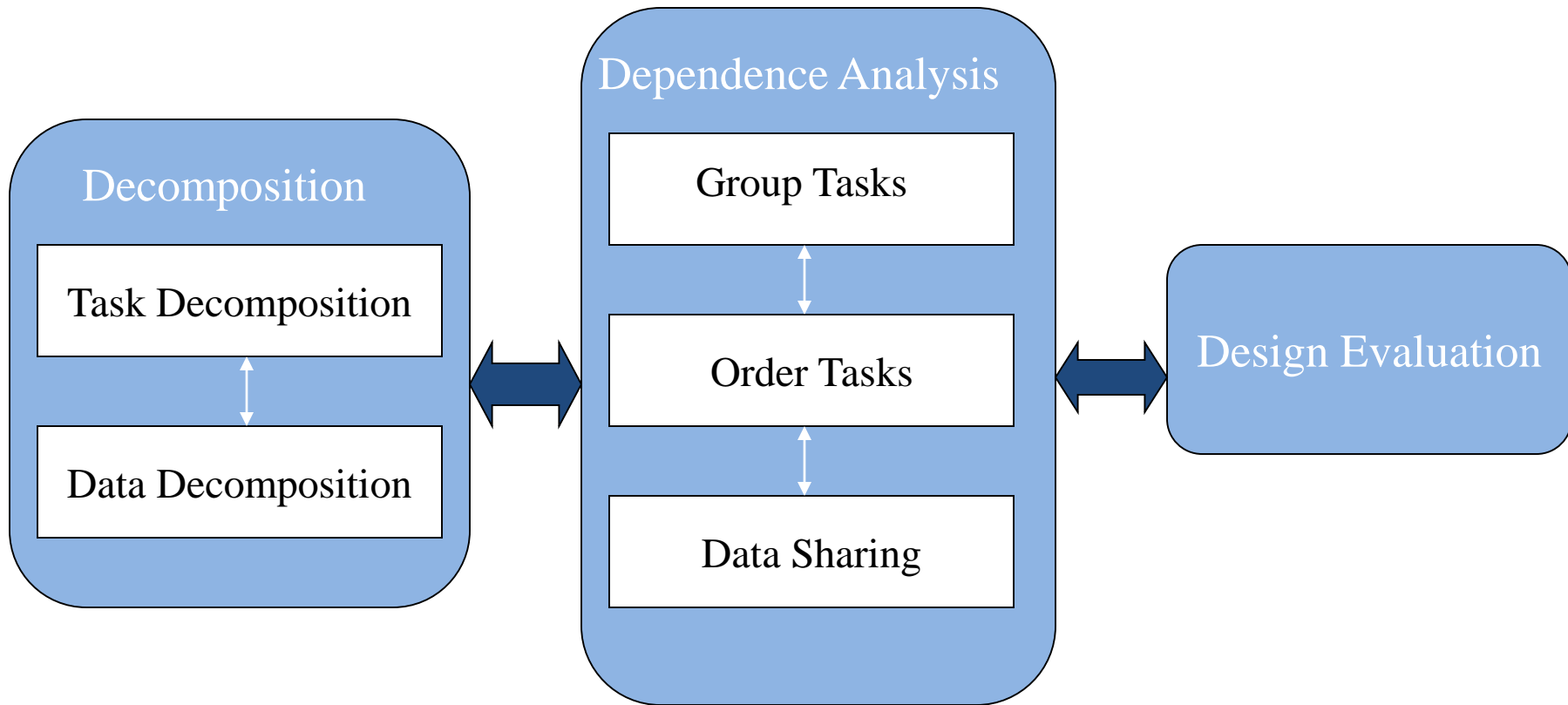
Shared Memory vs. Message Passing

- We have focused 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
 - This is different than global address space (single pointer space)
- The programming considerations of message passing model is quite different!
 - See MPI (Message Passing Interface)

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 the quality of the choices made in all the steps

Finding Concurrency - The Process



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

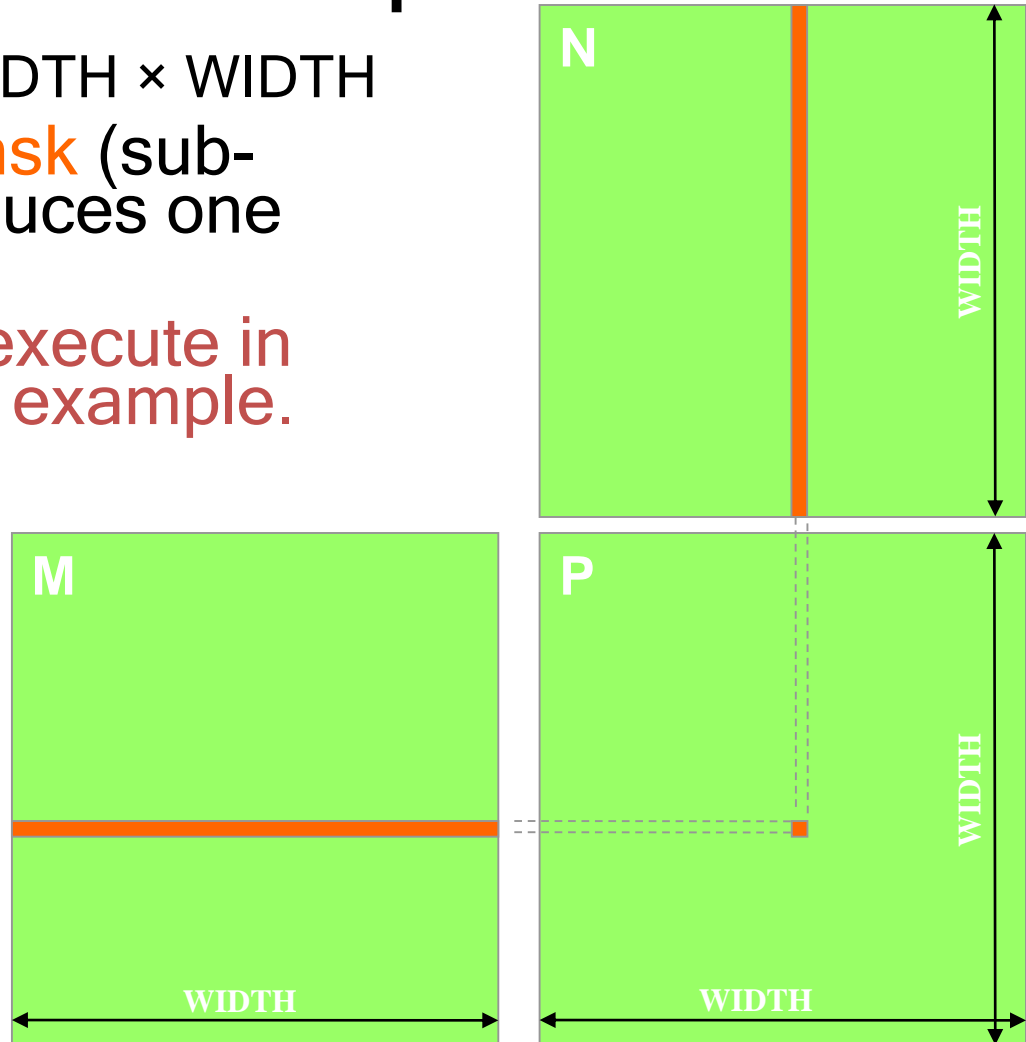
Task Decomposition

- 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

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

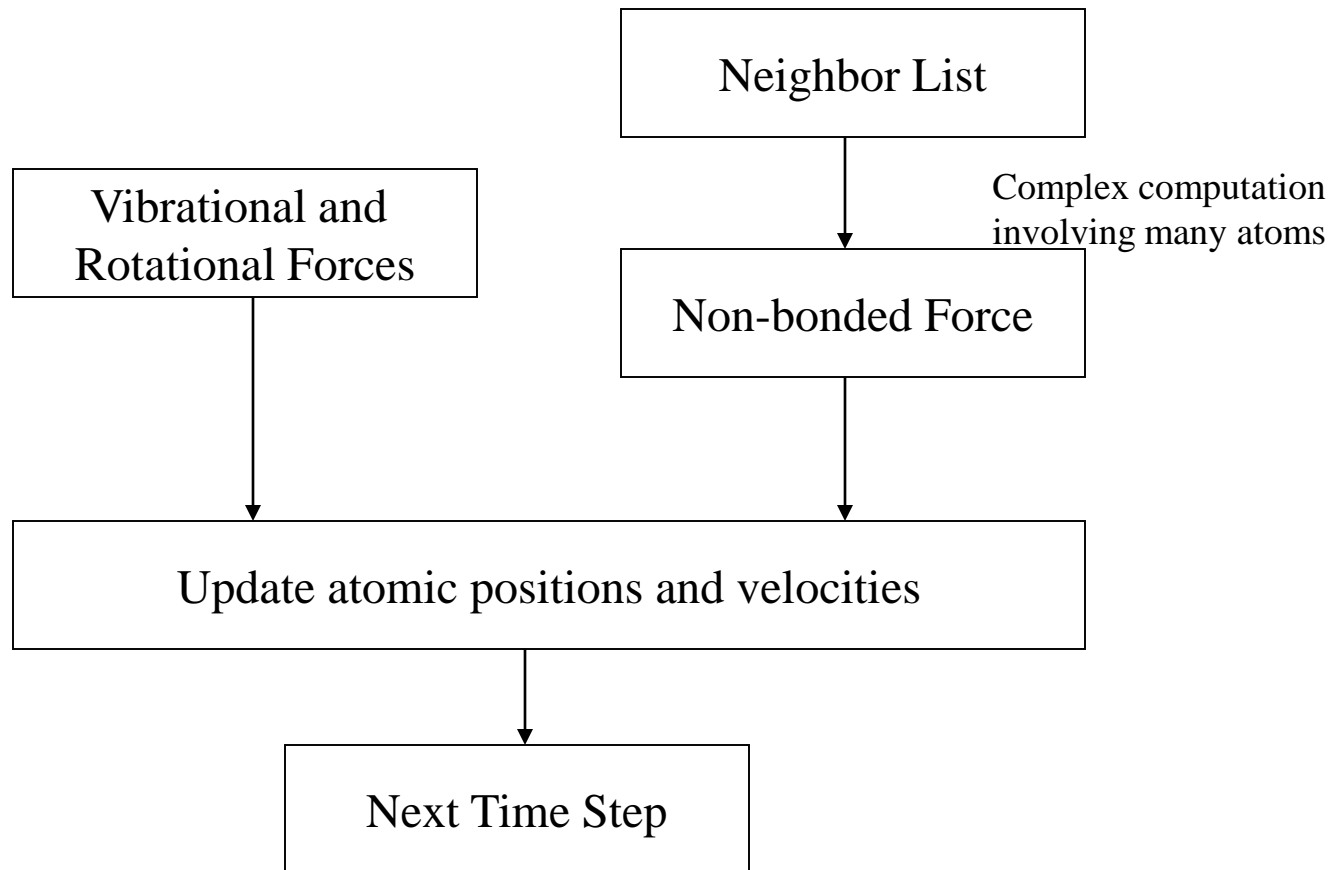


Task Decomposition Example - Molecular Dynamics

- 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

Often there are multiple ways to decompose any given problem.

Task Ordering Example: Molecular Dynamics



Data Decomposition

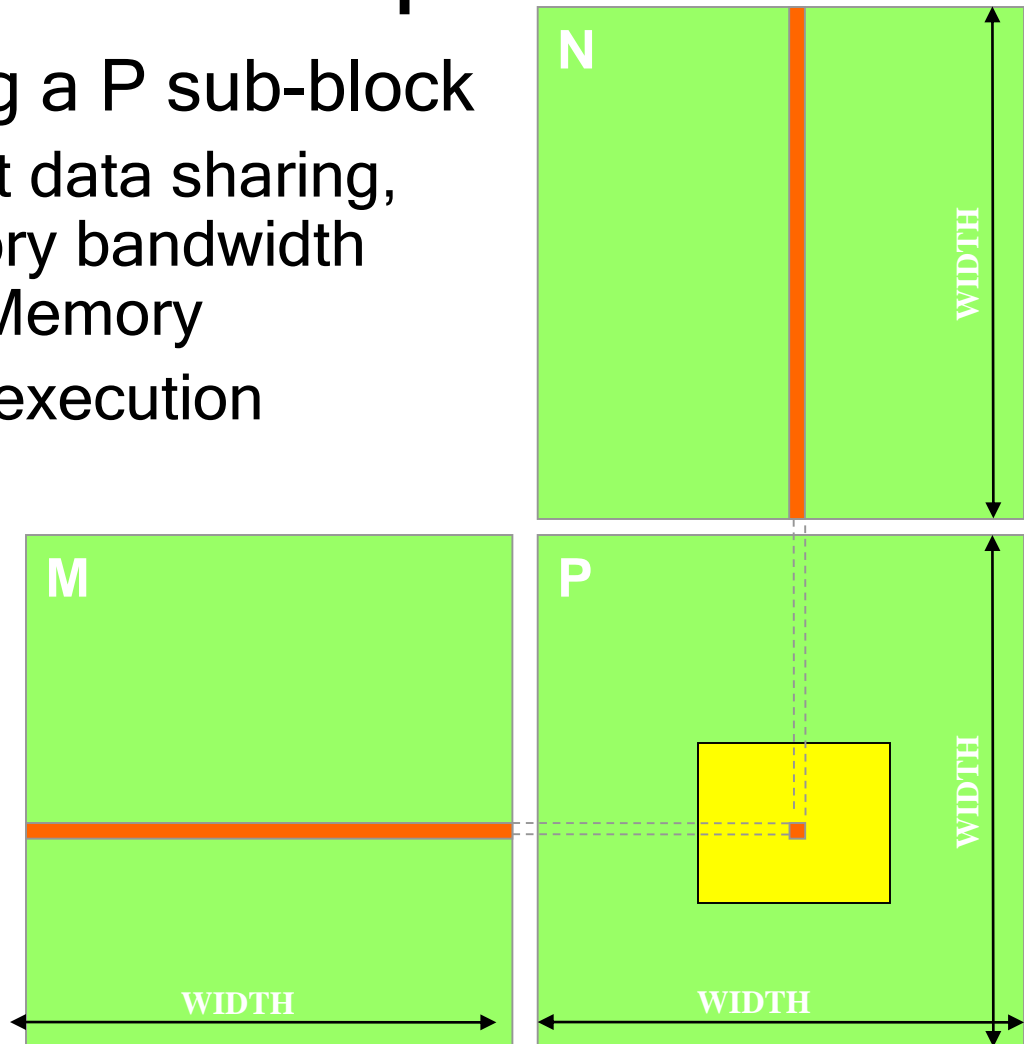
- The most compute intensive parts of many large problem manipulate a large data structure
 - Similar operations are being applied to different parts of the data structure, in a mostly independent manner
 - This is what CUDA is optimized for
- The data decomposition should lead to
 - Efficient **data 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

Task 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 (Shared 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 synced in execution



Task Ordering

- Identify the data and resources required by a group of tasks before they can be executed
 - Find the task group that creates them
 - Determine a temporal order that satisfies all data constraints

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

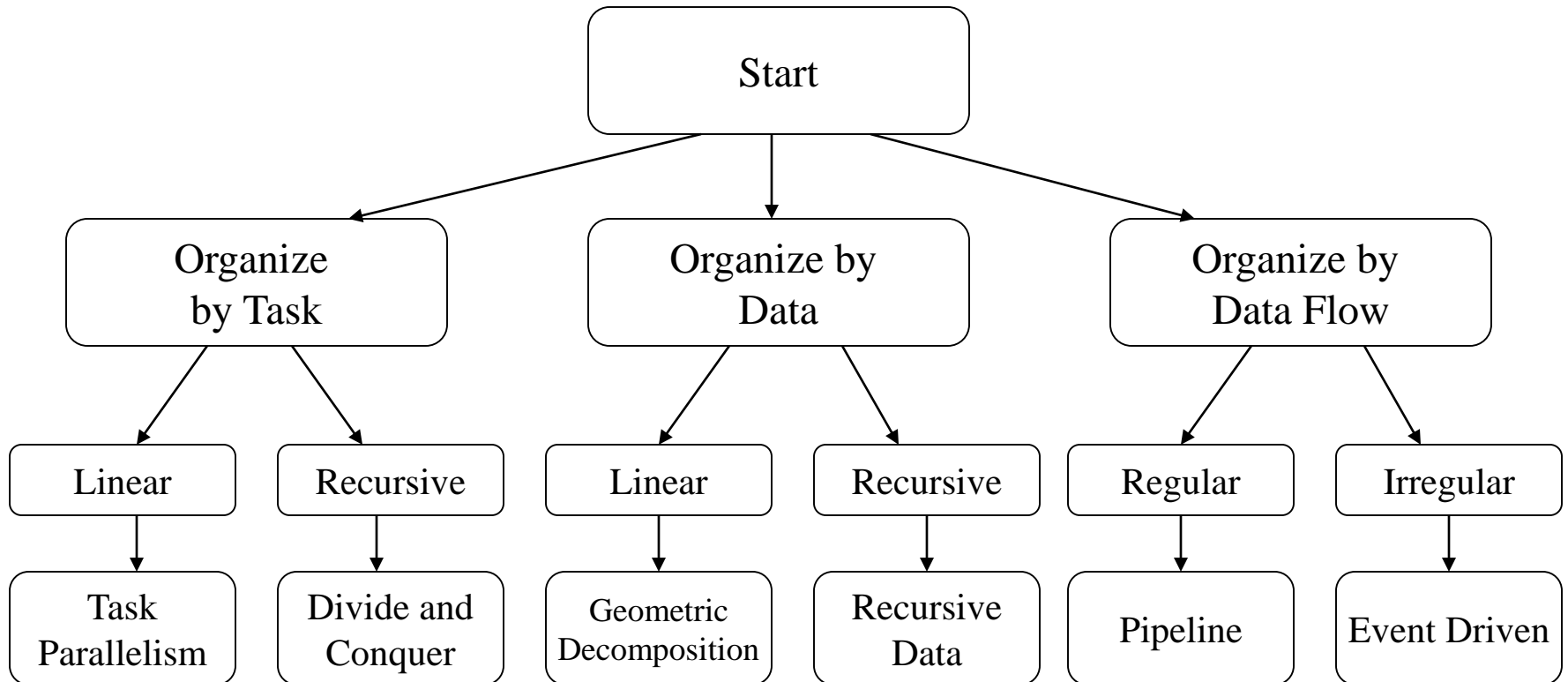
Key Parallel Programming Steps

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

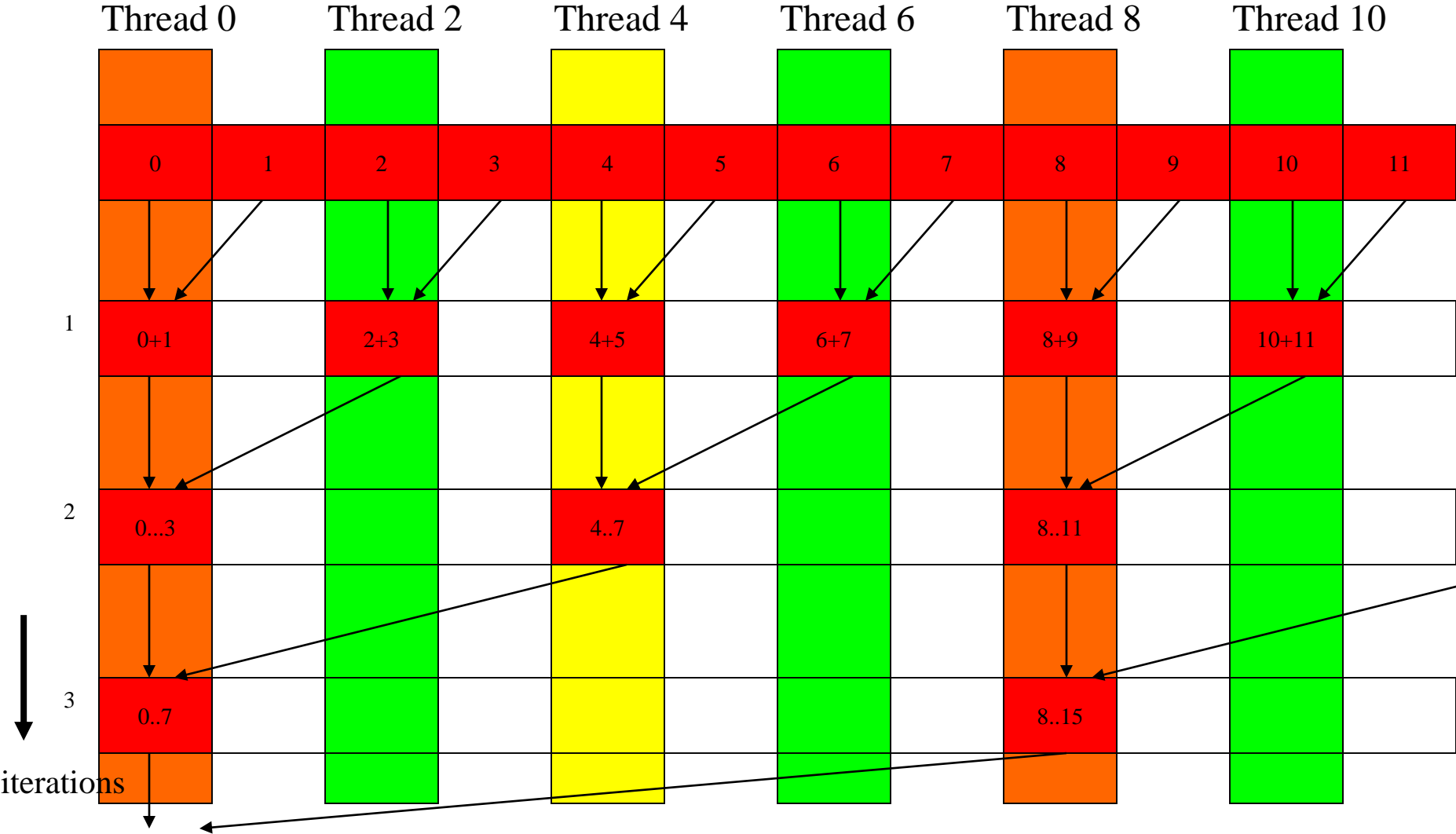
Algorithm

- 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

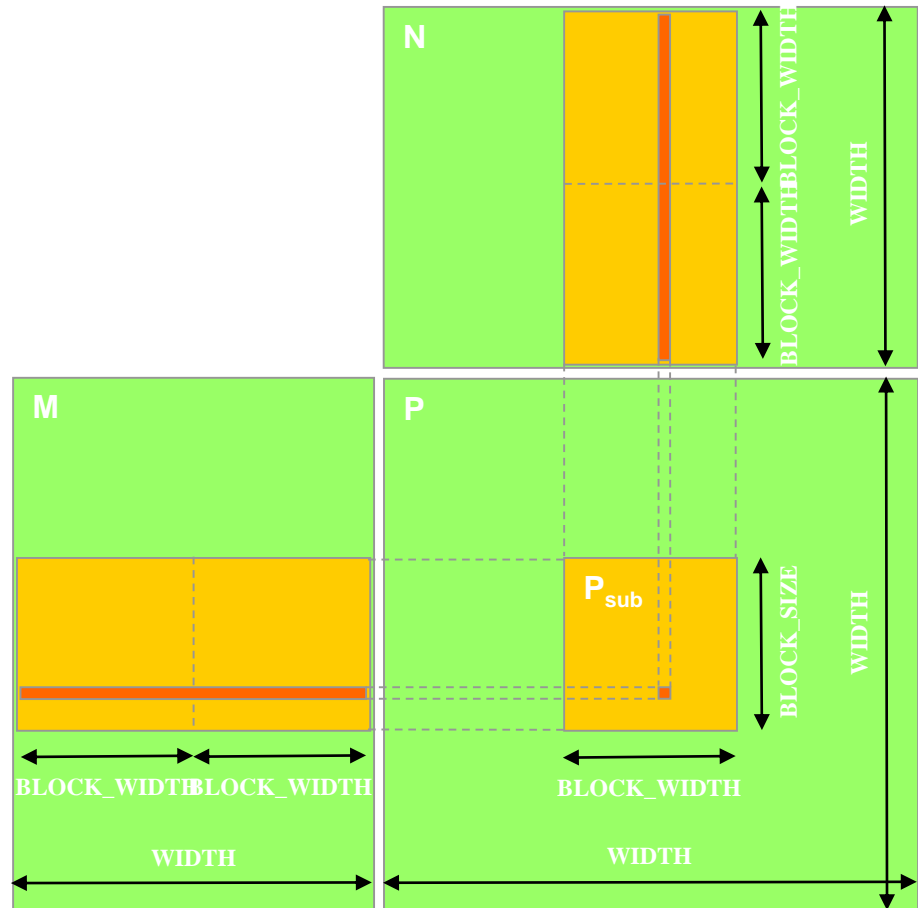


Mapping a Divide and Conquer Algorithm



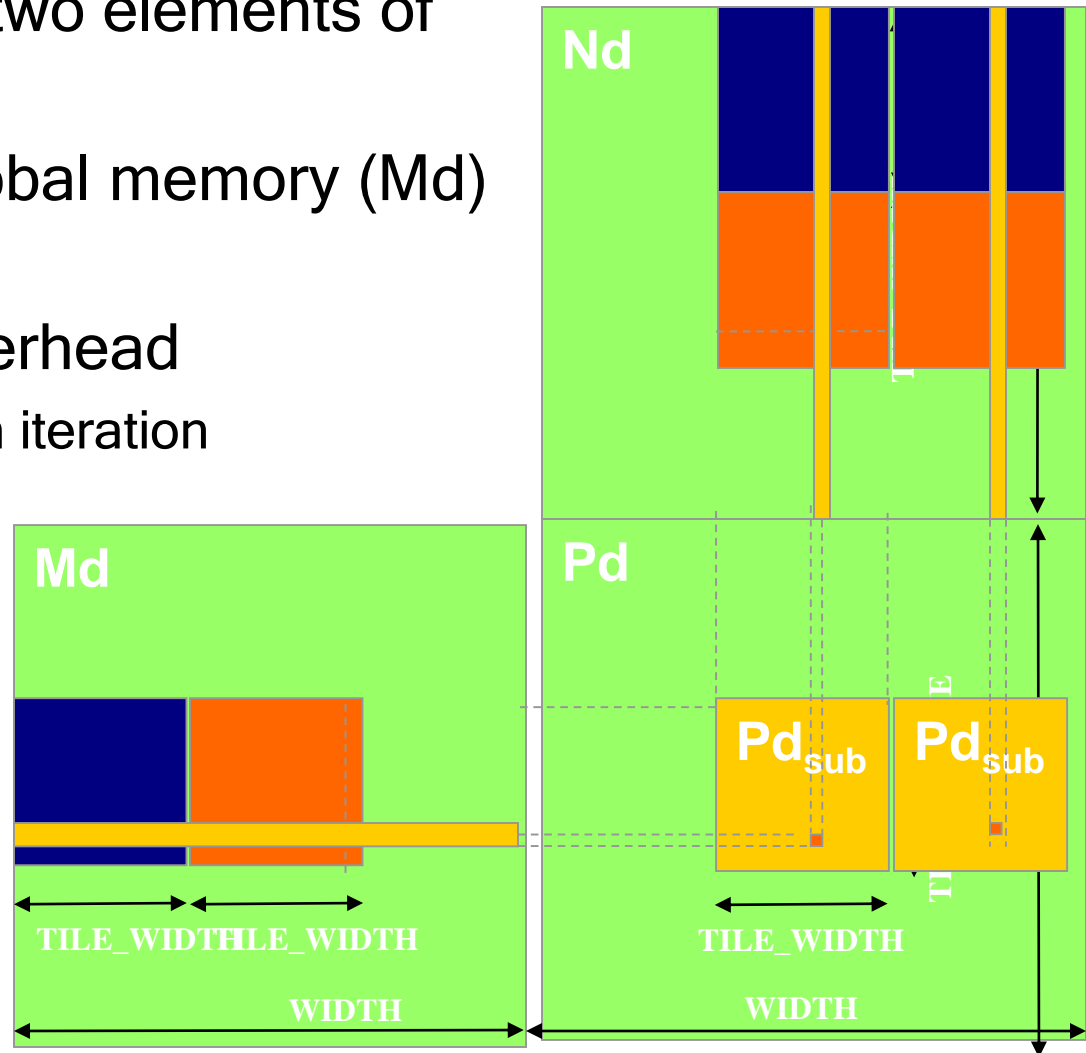
Tiled (Stenciled) Algorithms are Important for Geometric Decomposition

- 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 elements 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

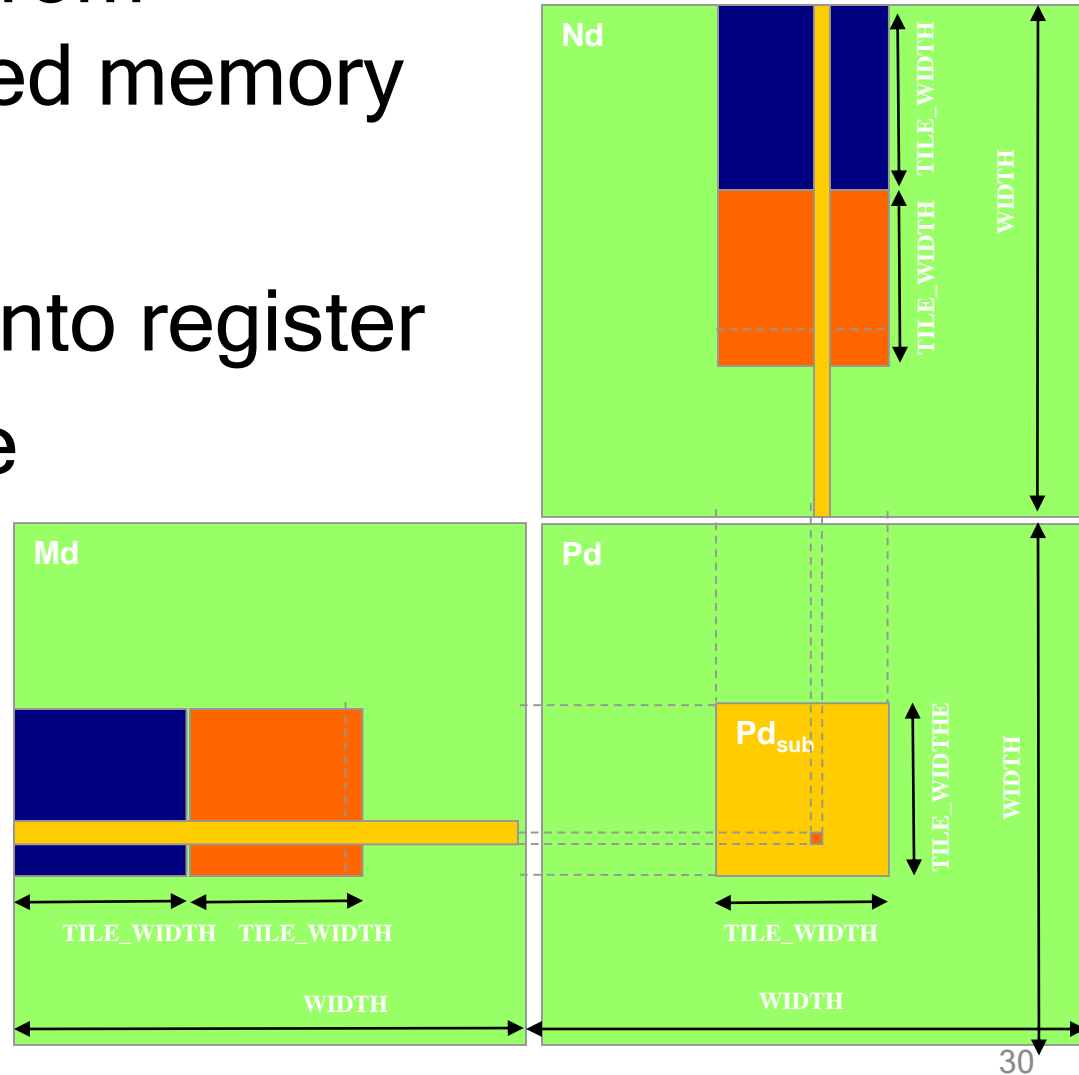
- One could double buffer the computation, getting better instruction mix within each thread
 - This is classic software pipelining in ILP compilers

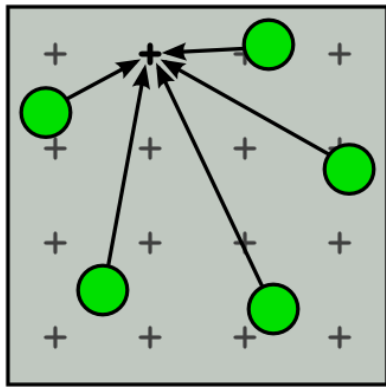
```
Loop {  
  
    Load current tile to shared memory  
  
    syncthreads()  
  
    Compute current tile  
  
    syncthreads()  
}
```

```
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

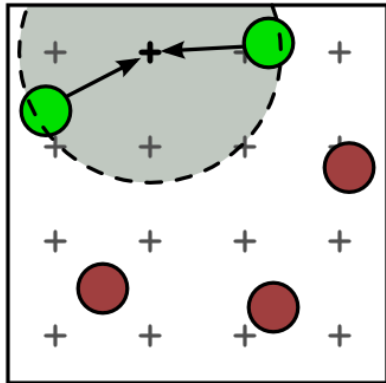
- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
-





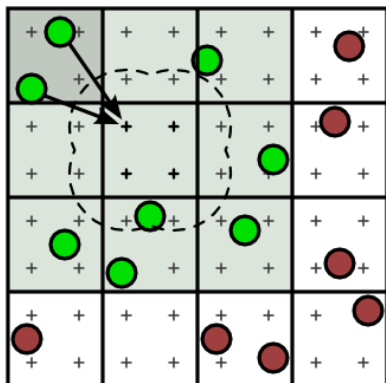
(a) Direct summation

At each grid point, sum the electrostatic potential from all charges



(b) Cutoff summation

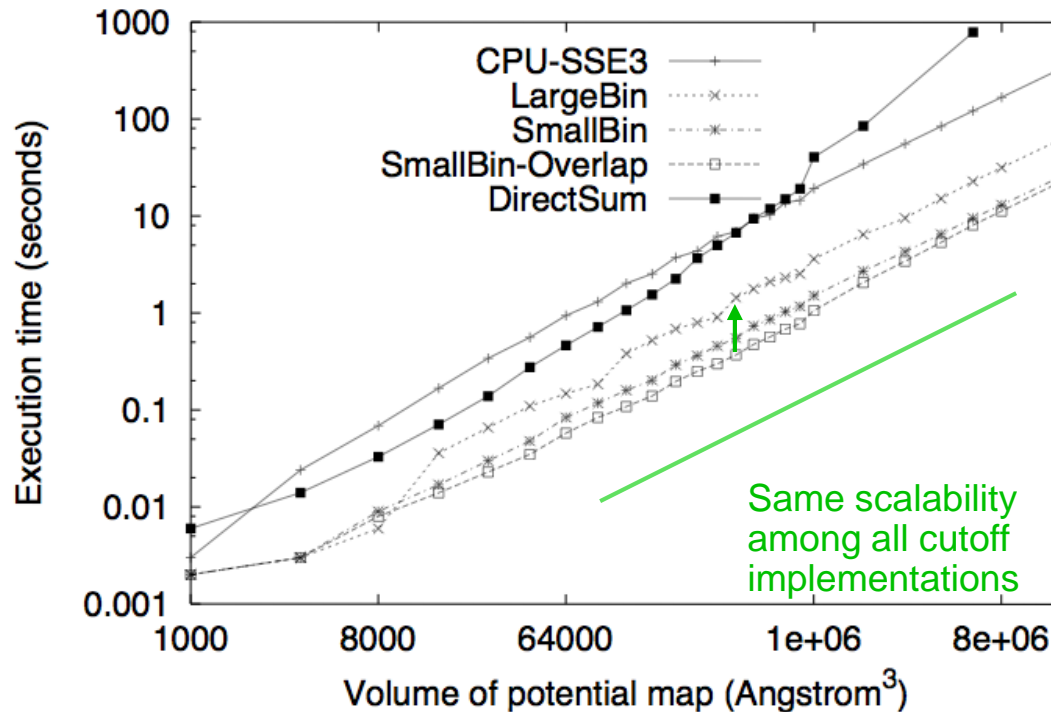
Electrostatic potential from nearby charges summed; spatially sort charges first



(c) Cutoff summation using direct summation kernel

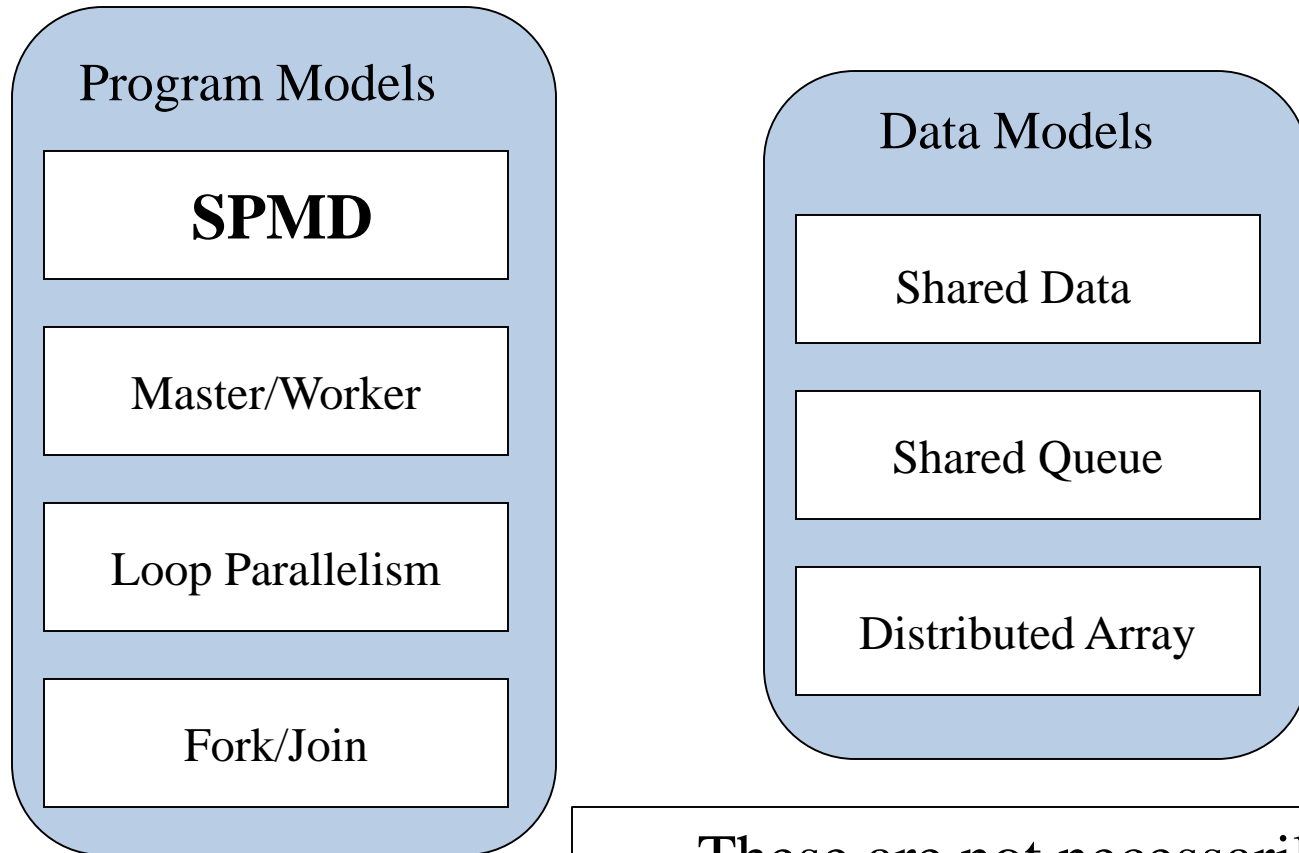
Spatially sort charges into bins; adapt direct summation to process a bin

Cut-Off Summation Restores Data Scalability



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

Parallel Programming Coding Styles - Program and Data Models



Program Models

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

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker
 - A Master thread sets up a 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)
- Fork/Join
 - Most general, generic way of creation of threads

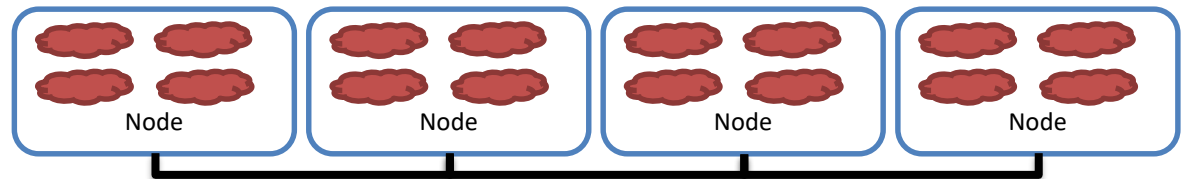
OpenMP

- API that supports shared memory multiprocessing in C, C++ and Fortran
- A master thread forks a specified number of slave threads and the system divides a task among them

```
int main(int argc, char **argv) {  
    int a[100000];  
  
    #pragma omp parallel for  
    int i;  
    for (i = 0; i < N; i++)  
        a[i] = 2 * i;  
  
    return 0;  
}
```

Message Passing Interface (MPI)

- Standardized, portable and language-independent message passing system
- Supports point-to-point and collective communication



```
int array[100];  
int root, total_p, *receive_array;
```

```
MPI_Comm_size(comm, &total_p);  
receive_array=malloc(total_p*100*sizeof(*receive_array));  
MPI_Gather(array, 100, MPI_INT, receive_array, 100,  
          MPI_INT, root, comm);
```

Algorithm Structures vs. Program Models

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

Program Models vs. Architectures

	OpenMP	MPI	CUDA/ OpenCL
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
 - A lot of 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 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, 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
 - Share/replicate 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
 - Use thread ID or conditions based on thread ID to branch to 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 remaining iterations

```
//num_steps = 1000000;  
...  
i_start = my_id * (num_steps/num_threads);  
i_end = i_start + (num_steps/num_threads);  
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 more iterations than others
- The number of extra iterations is up to the total number of threads - 1
 - This is not a big problem when the number of threads is small
 - When there are thousands of threads, this can create serious load imbalance problems
- Also, the extra if statement is a typical source of “branch divergence” in CUDA programs

SPMD Code Version 2

- Assign one more iteration 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) {  
    ....  
}
```

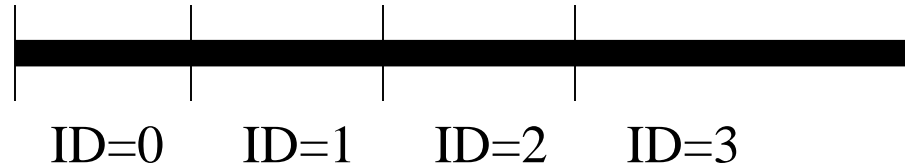
Less load imbalance

Loop branch divergence in the last Warp

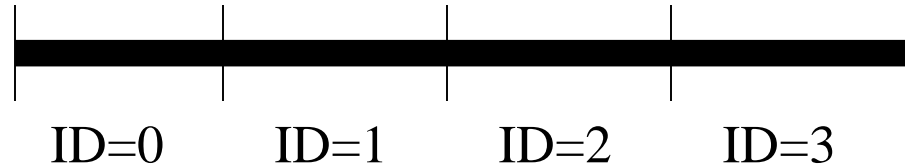
Data padding further eliminates divergence

Comparing the Three Versions

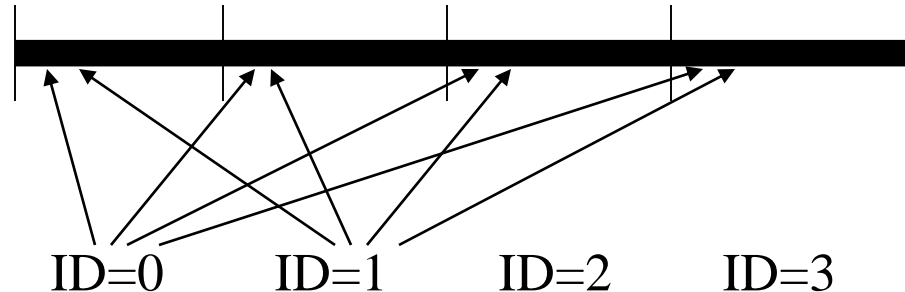
Version 1



Version 2



Version 3



Padded version 3 may be best
for some data access patterns

Pinned Memory

Pinned Memory

- *Page-locked* or *pinned* memory transfers attain the highest bandwidth between host and device
 - Ensures that host buffer does not get moved to virtual memory
- Allocated using the `cudaMallocHost()`
- Pinned memory should not be overused
 - Excessive use can reduce overall system performance
 - How much is too much is difficult to tell in advance

Asynchronous Transfers and Overlapping Transfers with Computation

- Data transfers between host and device using `cudaMemcpy()` are blocking transfers
 - Control is returned to the host thread only after the data transfer is complete.
- The `cudaMemcpyAsync()` function is a nonblocking variant of `cudaMemcpy()`
 - Unlike `cudaMemcpy()` the asynchronous transfer version requires pinned host memory

Asynchronous Transfers and Overlapping Transfers with Computation

```
cudaMemcpyAsync(a_d, a_h, size,  
    cudaMemcpyHostToDevice, stream);  
kernel<<<grid, block>>>(a_d);  
cpuFunction();
```

- Memory transfer and device execution are performed in parallel with host execution
- Last argument of `cudaMemcpyAsync()` specifies stream
 - 0 is the default - only nonzero streams are asynchronous (more details soon)
 - Kernel does not begin execution until memory transfer is complete

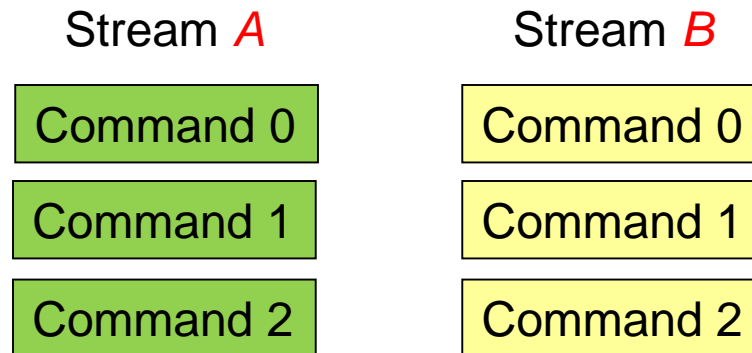
CUDA Streams

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Steve Rennich
NVIDIA

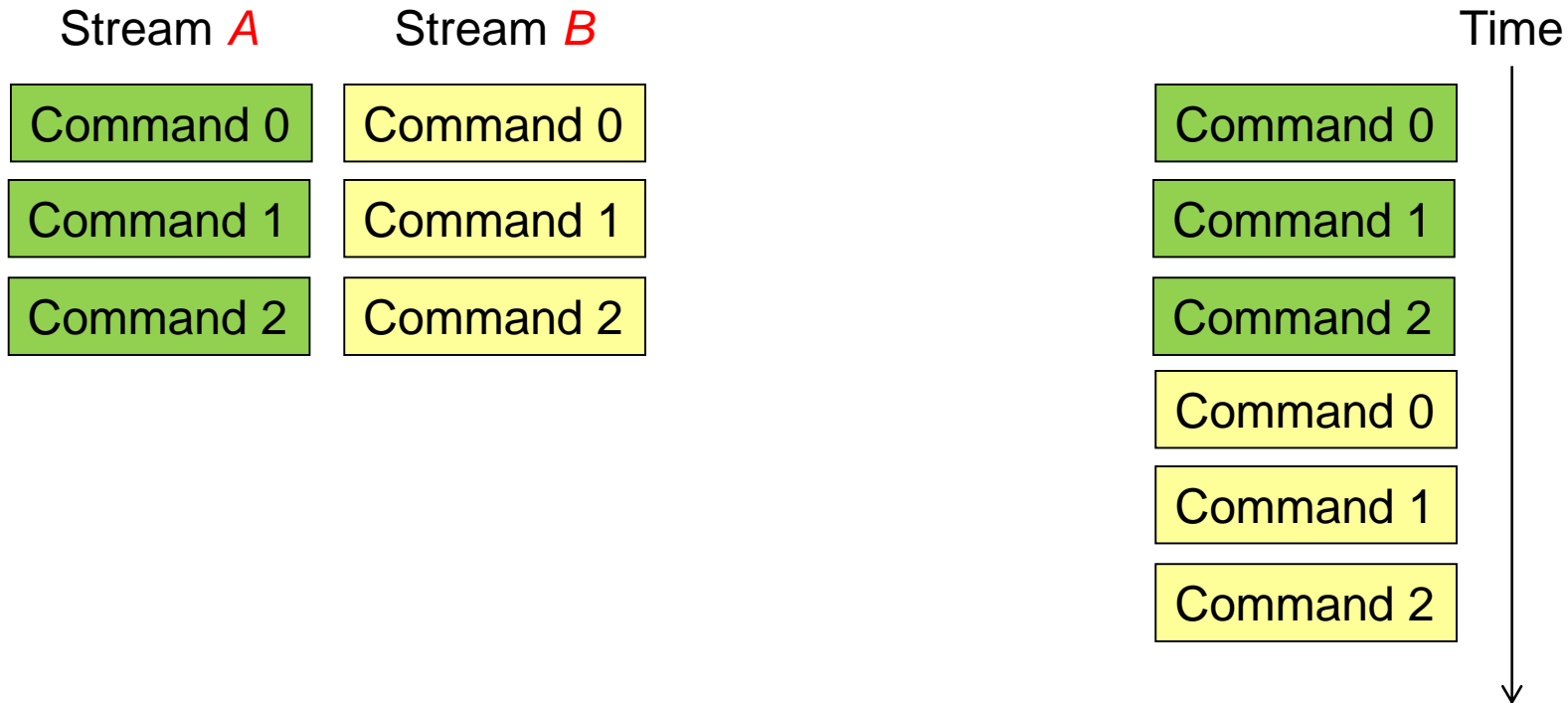
Streams

- **Stream**: Sequence of commands that execute in order
- Streams may execute their commands out-of-order or concurrently with respect to other streams



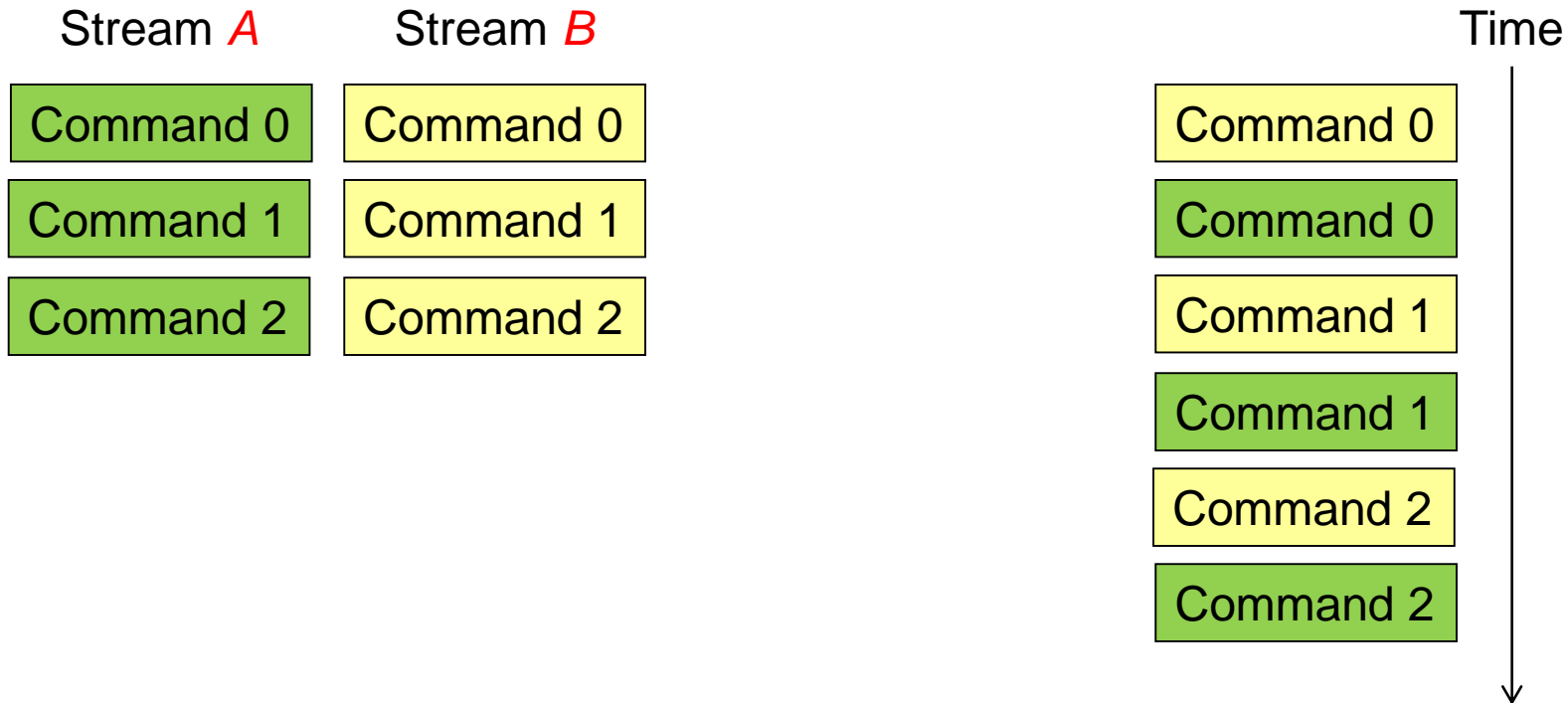
Streams

- Is this a possible order?



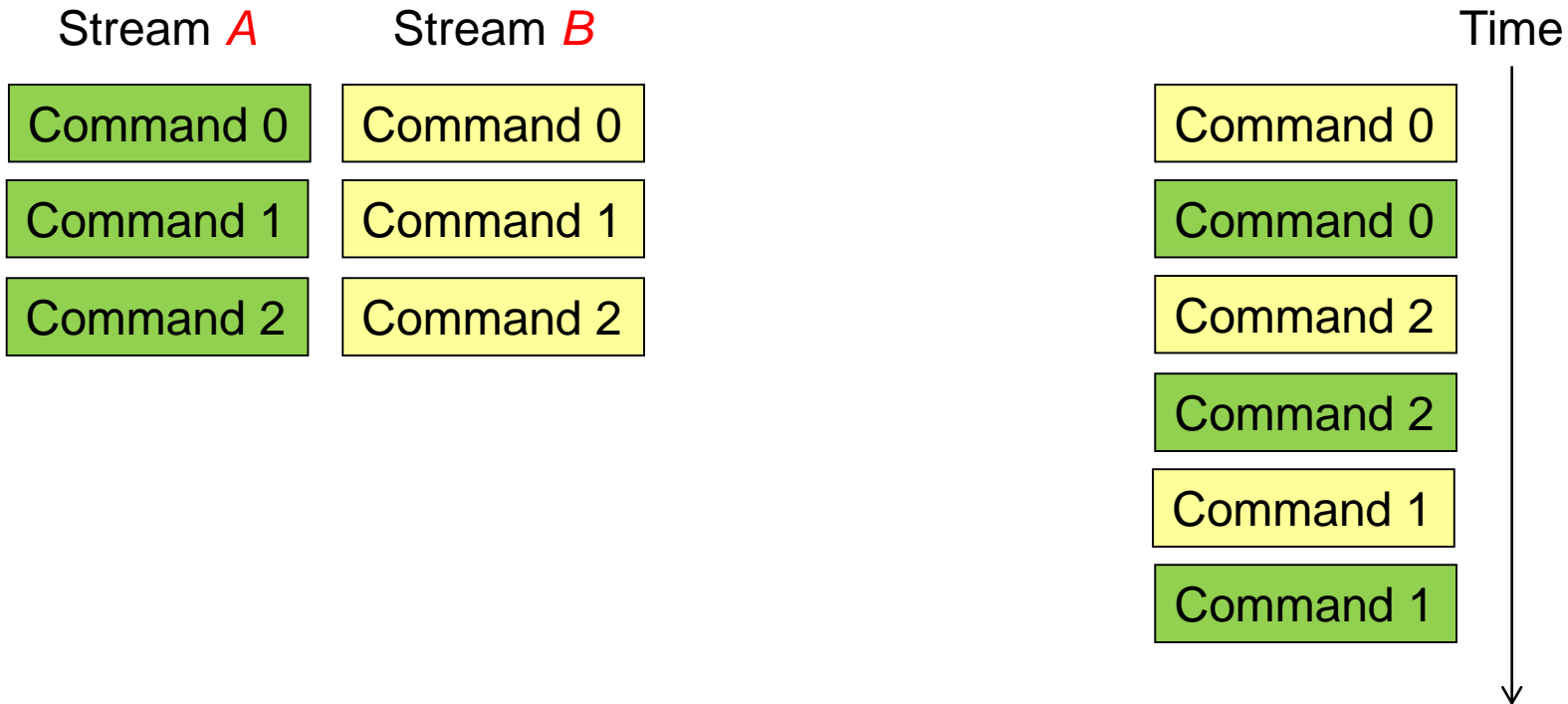
Streams

- Is this a possible order?



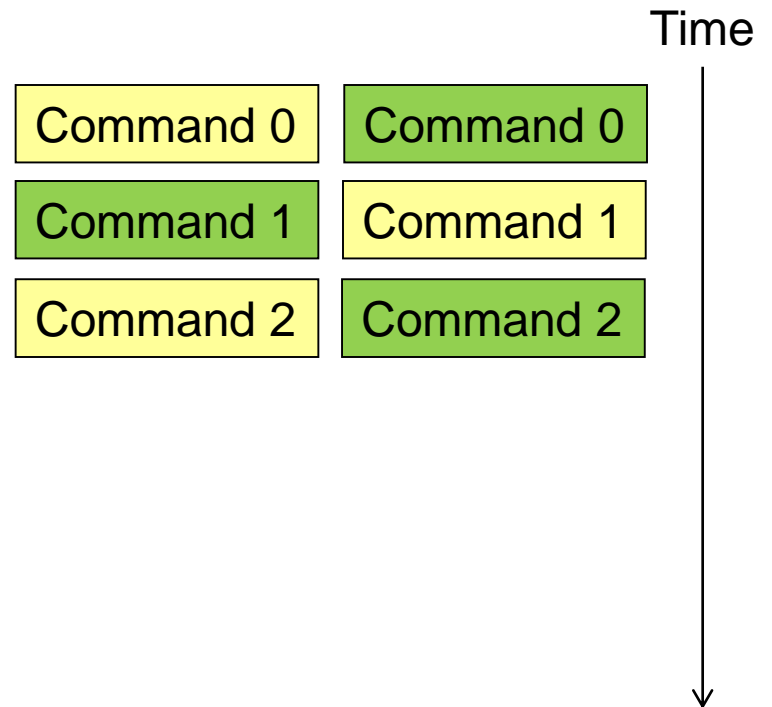
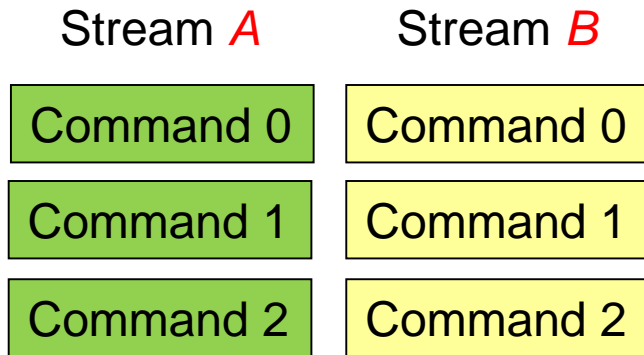
Streams

- Is this a possible order?




Streams

- Is this a possible order?



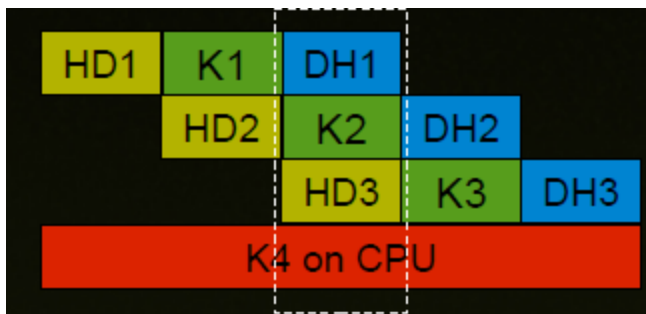
Streams

- In CUDA, what commands go in a stream?
 - Kernel launches
 - Host  device memory transfers

Streams on Fermi

- Architecture supports:
 - Up to 16 CUDA kernels on GPU
 - 2 cudaMemcpyAsync (must be in different directions)
 - Computation on the CPU
- Note: major improvements were made with Kepler architecture
 - Coming up later today

Amount of Concurrency



Default Stream (Stream '0')

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
 - As if `cudaDeviceSynchronize()` inserted before and after every CUDA operation
- Exceptions - asynchronous w.r.t. host
 - Kernel launches in the default stream
 - `cudaMemcpy*Async`
 - `cudaMemset*Async`
 - `cudaMemcpy` within the same device
 - H2D `cudaMemcpy` of 64kB or less

Requirements for Concurrency

- CUDA operations must be in different, non-0, streams
- `cudaMemcpyAsync` with host from 'pinned' memory
 - Page-locked memory
 - Allocated using `cudaMallocHost()` or `cudaHostAlloc()` (*)
- Sufficient resources must be available
 - `cudaMemcpyAsync`s in different directions
 - Device resources (SMEM, registers, blocks, etc.)

* Both commands are roughly equivalent, but not in all versions of CUDA

Streams

- Code Example
 1. Create two streams
 2. Each stream:
 1. Copy page-locked memory to device
 2. Launch kernel
 3. Copy memory back to host
 3. Destroy streams

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];  
for (int i = 0; i < 2; ++i)  
{  
    cudaStreamCreate(&stream[i]);  
}  
  
float *hostPtr;  
cudaMallocHost(&hostPtr, 2 * size);
```

Create two streams

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];  
for (int i = 0; i < 2; ++i)  
{  
    cudaStreamCreate (&stream[i]);  
}
```

```
float *hostPtr;
```

```
cudaMallocHost (&hostPtr, 2 * size);
```

Allocate two buffers in page-locked memory

Stream Example (Step 2 of 3)

```
for (int i = 0; i < 2; ++i)
{
    cudaMemcpyAsync (/* ... */,
                    cudaMemcpyHostToDevice, stream[i]);
    kernel<<<100, 512, 0, stream[i]>>>
    (/* ... */);
    cudaMemcpyAsync (/* ... */,
                    cudaMemcpyDeviceToHost, stream[i]);
}
```

Commands are assigned to, and executed by streams

Stream Example (Step 3 of 3)

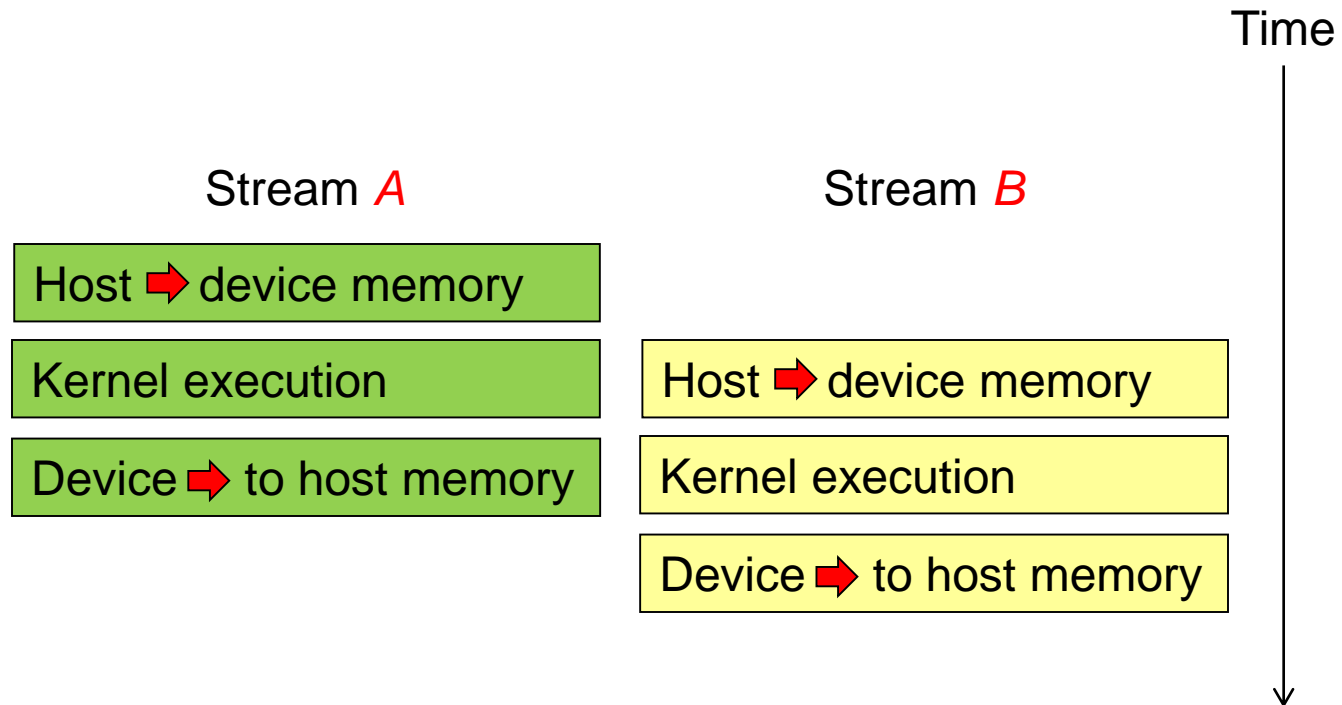
```
for (int i = 0; i < 2; ++i)
{
    // Blocks until commands complete
    cudaStreamDestroy(stream[i]);
}
```

Streams

- Assume compute capability 1.1 and above:
 - Overlap of data transfer and kernel execution
 - Concurrent kernel execution
 - Concurrent data transfer
- How can the streams overlap?

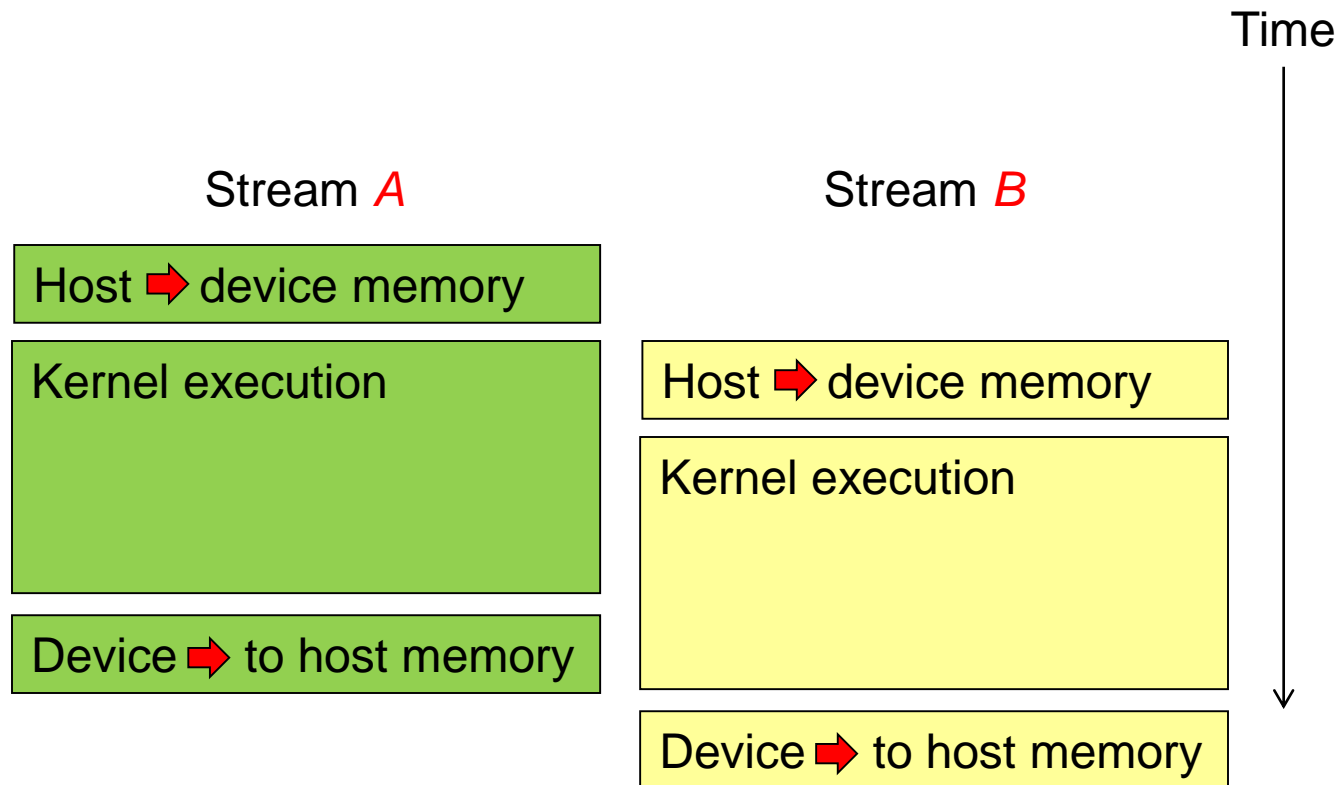
Streams

- Can we have more overlap than this?



Streams

- Can we have this?



Streams

- *Implicit Synchronization*
 - An operation that requires a dependency check to see if a kernel finished executing:
 - *Blocks* all kernel launches *from any stream* until the checked kernel is finished
- `cudaStreamQuery()` can be used to test if a stream has completed all operations

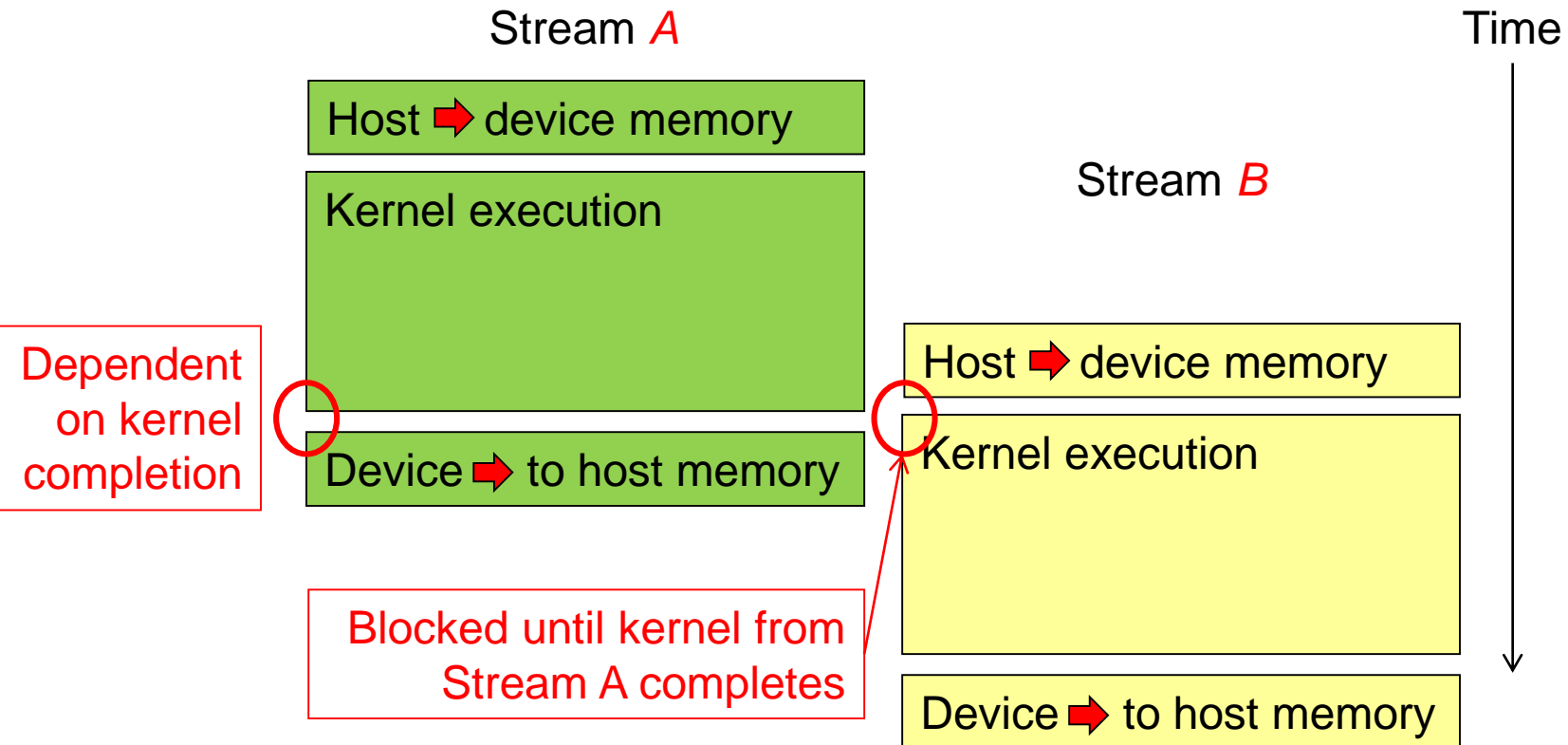
See 3.2.6.5.3 in the NVIDIA CUDA C Programming Guide for all limitations (version 3.2)

Implicit Synchronization

- These operations implicitly synchronize all other CUDA operations
 - Page-locked memory allocation
 - `cudaMallocHost`
 - `cudaHostAlloc`
 - Device memory allocation
 - `cudaMalloc`
 - Non-Async version of memory operations
 - `cudaMemcpy*` (no Async suffix)
 - `cudaMemset*` (no Async suffix)
 - Change to L1/shared memory configuration
 - `cudaDeviceSetCacheConfig`

Streams

- Can we have this?



Streams

- Performance Advice
 - Issue all independent commands before dependent ones
 - Delay synchronization (implicit or explicit) as long as possible

Streams

- Rewrite this to allow concurrent kernel execution

```
for (int i = 0; i < 2; ++i)
{
    cudaMemcpyAsync (/* ... */, stream[i]);
    kernel<<< /*... */ stream[i]>>> ();
    cudaMemcpyAsync (/* ... */, stream[i]);
}
```

Streams

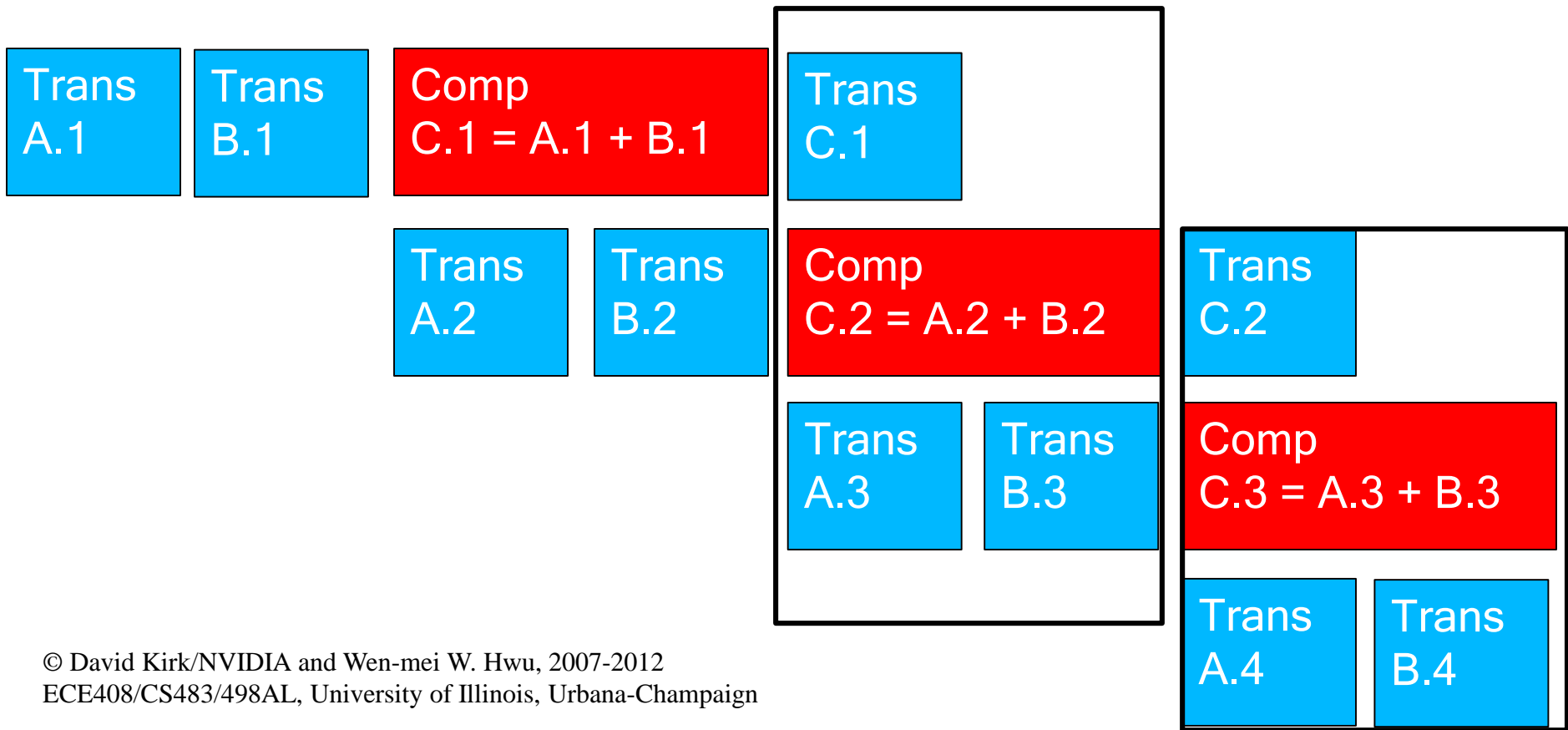
```
for (int i = 0; i < 2; ++i) // to device
    cudaMemcpyAsync(/* ... */, stream[i]);
```

```
for (int i = 0; i < 2; ++i)
    kernel<<< /*... */ stream[i]>>>();
```

```
for (int i = 0; i < 2; ++i) // to host
    cudaMemcpyAsync(/* ... */, stream[i]);
```

Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



Explicit Synchronization

- `cudaDeviceSynchronize()`
 - Blocks until commands in all streams finish
 - `cudaThreadSynchronize()` has been deprecated
- `cudaStreamSynchronize(streamid)`
 - Blocks until commands in a specific stream finish

Synchronization using Events

- Create specific 'Events', within streams, to use for synchronization
- `cudaEventRecord (event, streamid)`
- `cudaEventSynchronize (event)`
- `cudaStreamWaitEvent (stream, event)`
- `cudaEventQuery (event)`

Explicit Synchronization Example

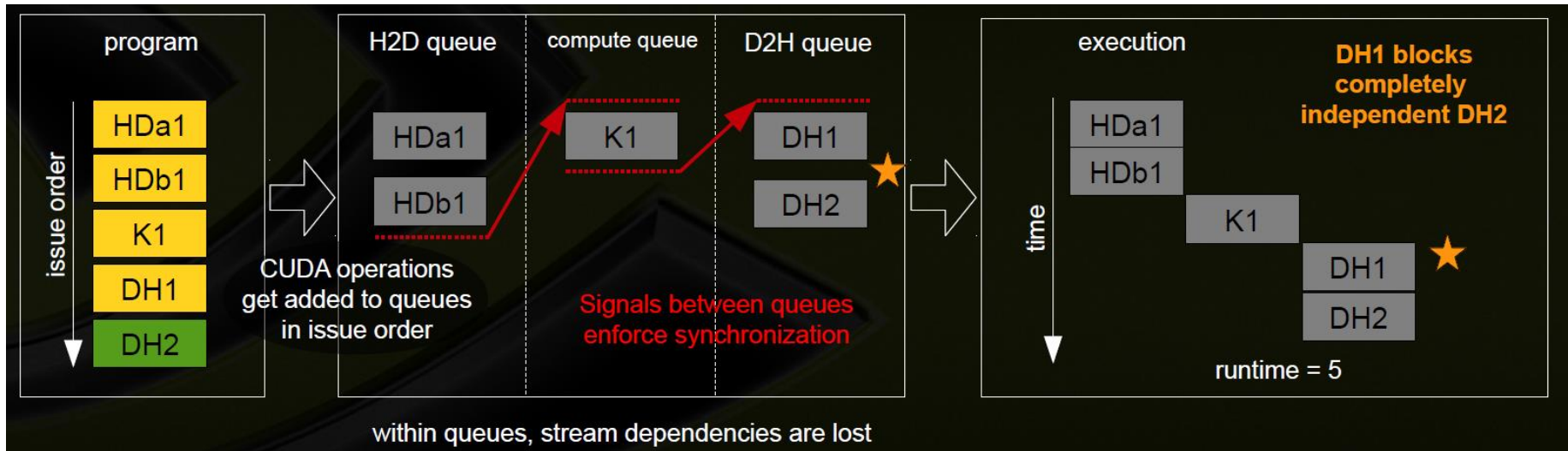
```
{  
  cudaEvent_t event;  
  cudaEventCreate (&event); // create event  
  
  // 1) H2D copy of new input  
  cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );  
  cudaEventRecord (event, stream1);           // record event  
  
  // 2) D2H copy of previous result  
  cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );  
  cudaStreamWaitEvent ( stream2, event );     // wait for event in stream1  
  
  kernel <<< , , , stream2 >>> ( d_in, d_out ); // 3) must wait for 1 and 2  
  asynchronousCPUmethod ( ... );  
}
```

Stream Scheduling

- Fermi hardware has 3 queues
 - 1 Compute Engine queue
 - 2 Copy Engine queues - one for H2D and one for D2H
- CUDA operations are dispatched to HW in the sequence they were issued
 - Placed in the relevant queue
 - Stream dependencies between engine queues are maintained, but lost within an engine queue
- A CUDA operation is dispatched from the engine queue if:
 - Preceding calls in the same stream have completed,
 - Preceding calls in the same queue have been dispatched, and
 - Resources are available
- CUDA kernels may be executed concurrently if they are in different streams
 - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- Note that a blocked operation blocks all other operations in the queue, even in other streams

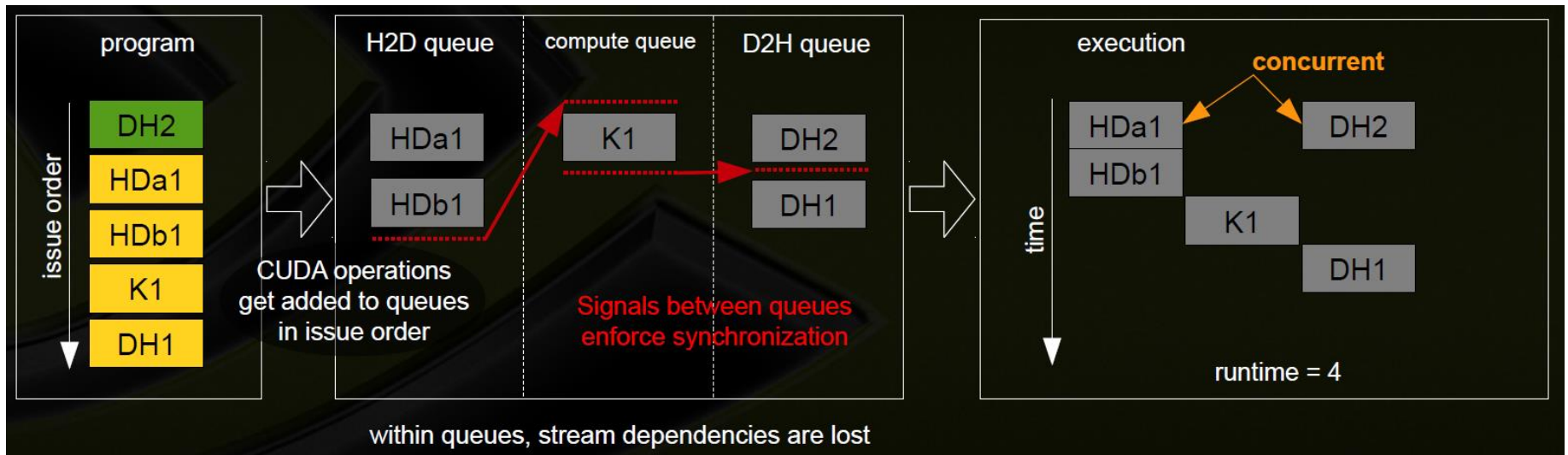
Example - Blocked Queue

- Two streams, stream 1 is issued first
 - Stream 1 : HDa1, HDb1, K1, DH1 (issued first)
 - Stream 2 : DH2 (completely independent of stream 1)



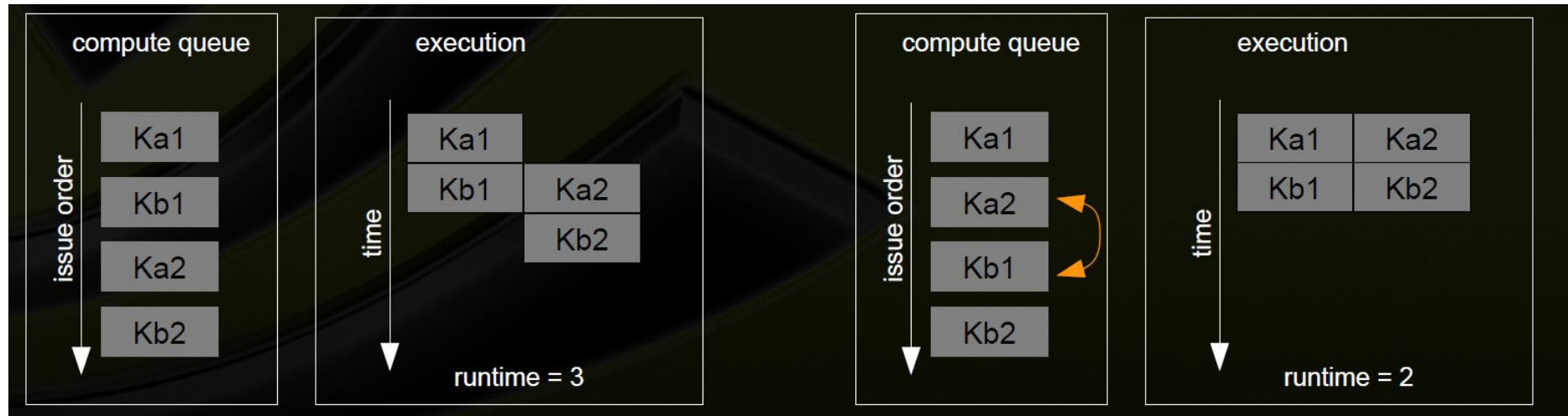
Example - Blocked Queue

- Two streams, stream 1 is issued first
 - Stream 1 : HDa1, HDb1, K1, DH1
 - Stream 2 : DH2 (issued first)



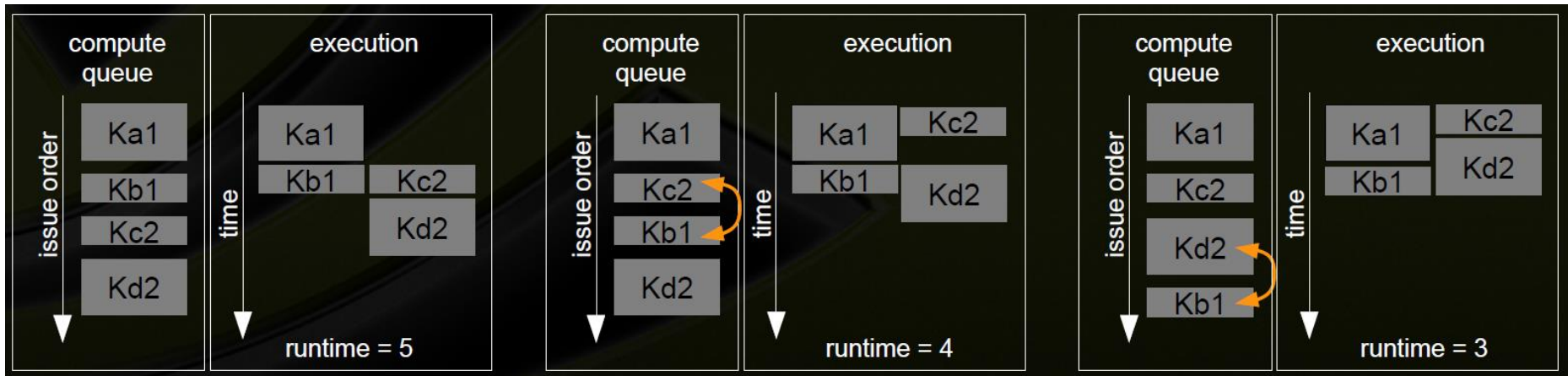
Example - Blocked Kernel

- Two streams - just issuing CUDA kernels
 - Stream 1 : Ka1, Kb1
 - Stream 2 : Ka2, Kb2
 - Kernels are similar size, fill $\frac{1}{2}$ of the SM resources



Example - Optimal Concurrency can Depend on Kernel Execution Time

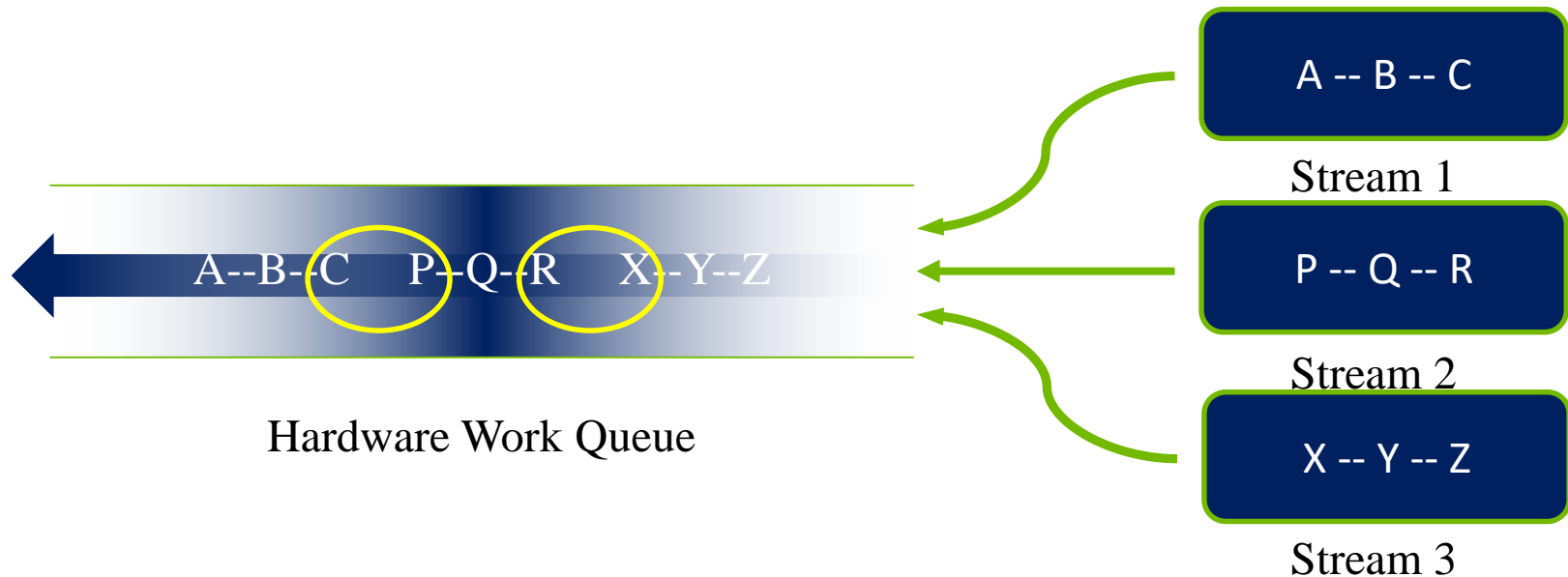
- Two streams - just issuing CUDA kernels - but kernels are different 'sizes'
 - Stream 1 : Ka1 {2}, Kb1 {1}
 - Stream 2 : Kc2 {1}, Kd2 {2}
 - Kernels fill $\frac{1}{2}$ of the SM resources



Hyper Queue

- Provide multiple real queues for each engine
- Allow much more concurrency by allowing some streams to make progress for an engine while others are blocked

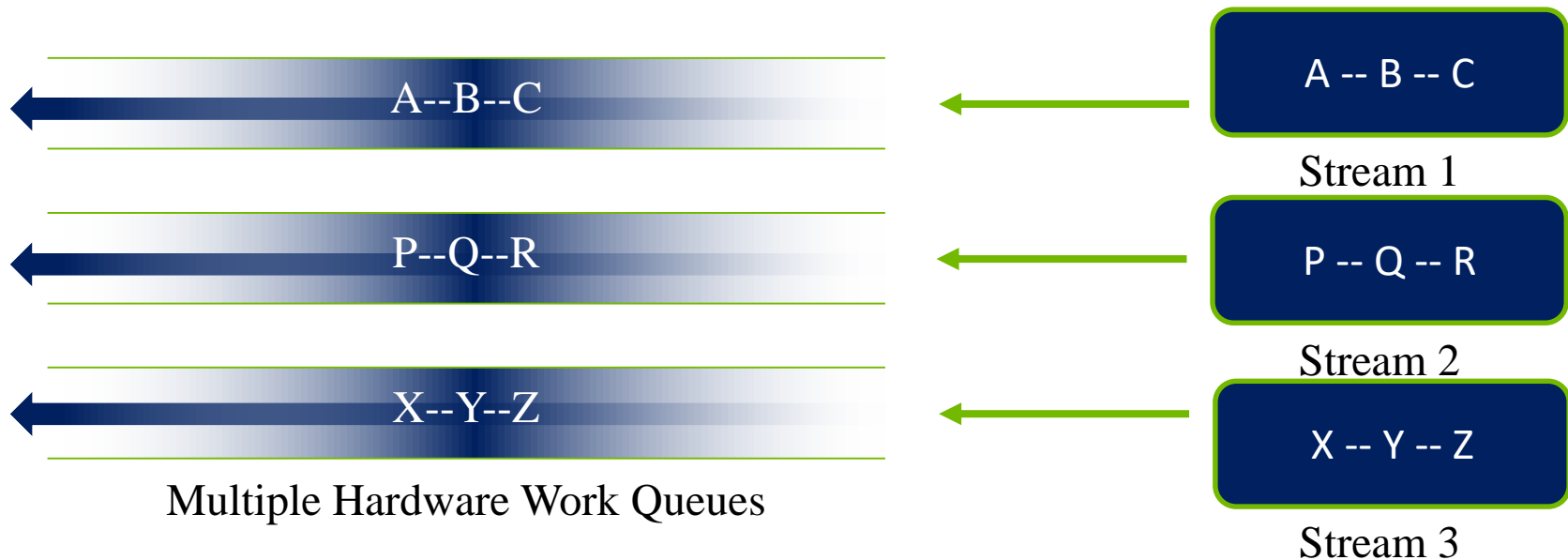
Fermi (and older) Concurrency



Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges

Improved Concurrency



Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

Pascal also supports 32-way concurrency

- One work queue per stream
- Dynamic scheduling