Massively Parallel Computing with Cuda

- Synchronization & Sorting -

Hendrik Lensch
Robert Strzodka

Today

- **Last lecture**
  - PrefixSum
- **Today**
  - Parallel Programming Patterns
  - Synchronization
  - Sorting

Parallel Programming Design

[following slides from David Kirk]

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

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

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.

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
- **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

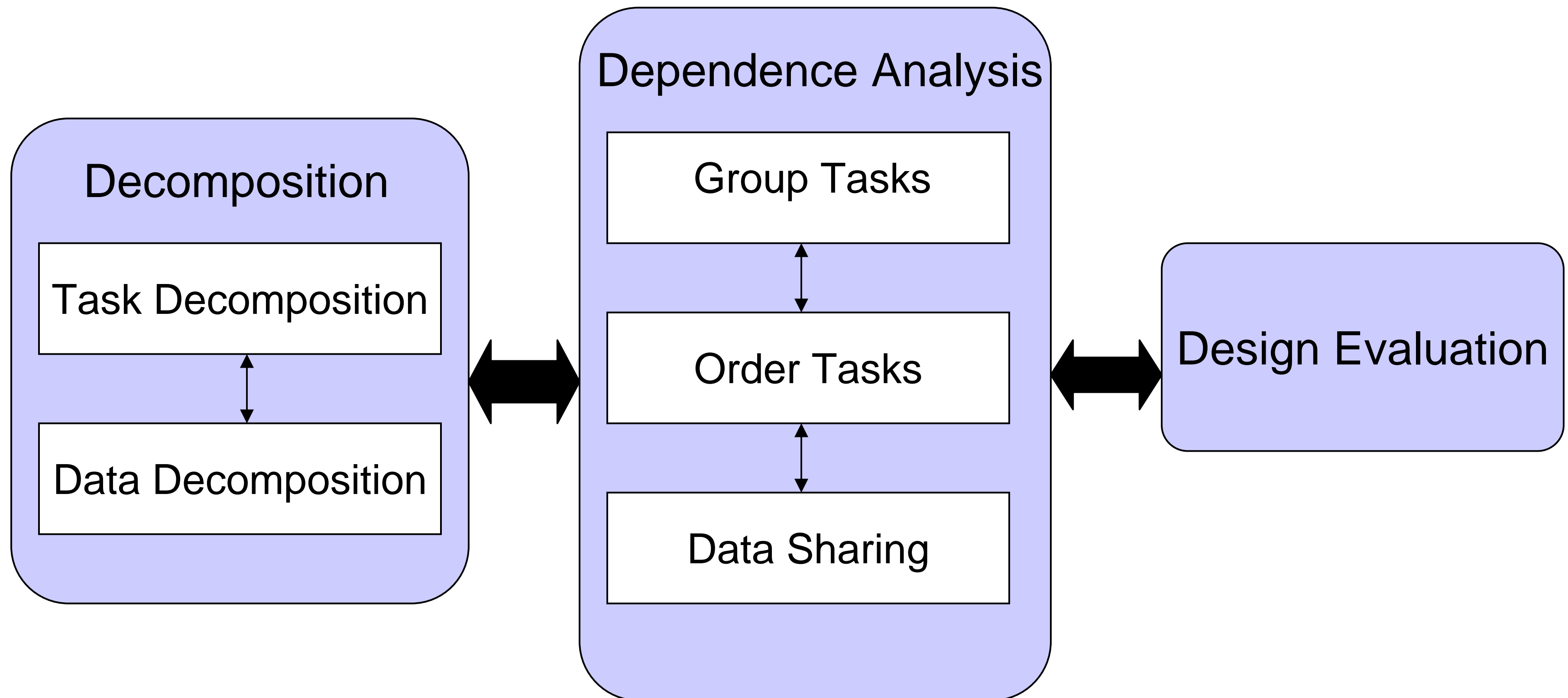
Shared Memory vs. Message Passing

- **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!**

Finding Concurrency in Problems

- **Identify a decomposition of the problem into sub-problems that can be solved simultaneously**
  - A task decomposition that identifies tasks that can execute concurrently
  - 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



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

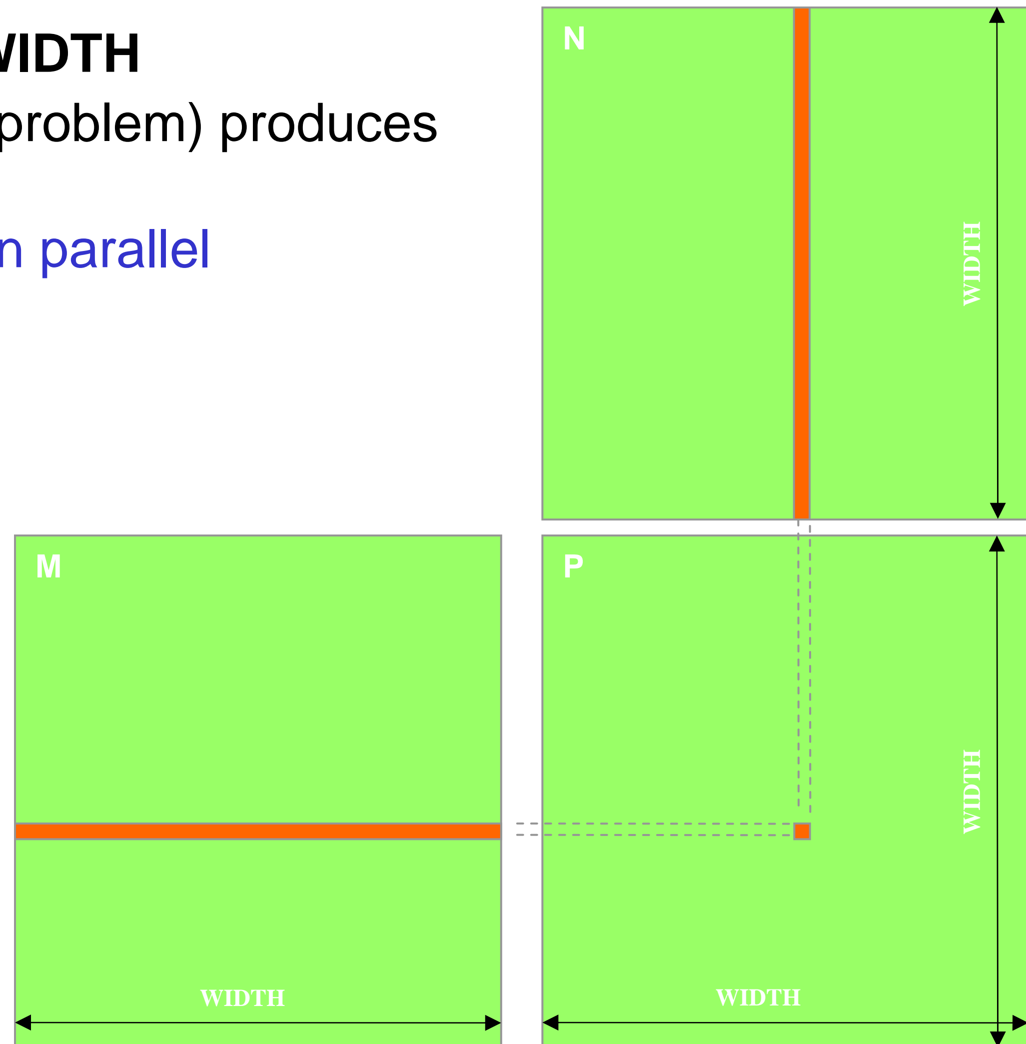
Task Decomposition

- **Many large problems have natural independent 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 • WIDTH**
  - One natural task (sub-problem) produces one element of P
  - All tasks can execute in parallel



Task Decomposition Example – Molecular Dynamics

- **Simulation of motions of a large molecular system**
- **A natural task is to perform calculations of individual atoms in parallel**
- **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.

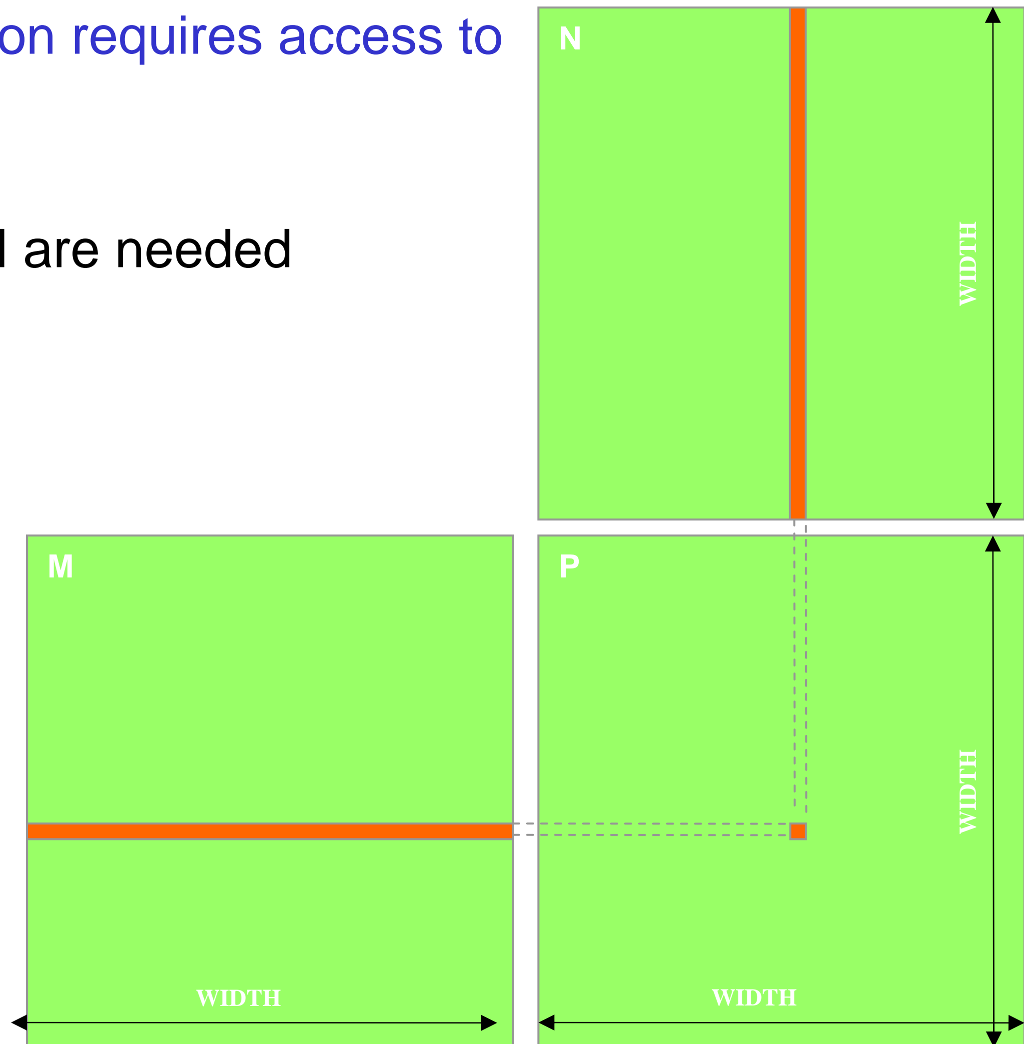
Hendrik Lensch and Robert Strzodka

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
    - minimum amount of global reads and writes
  - Few dependencies across the tasks that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics

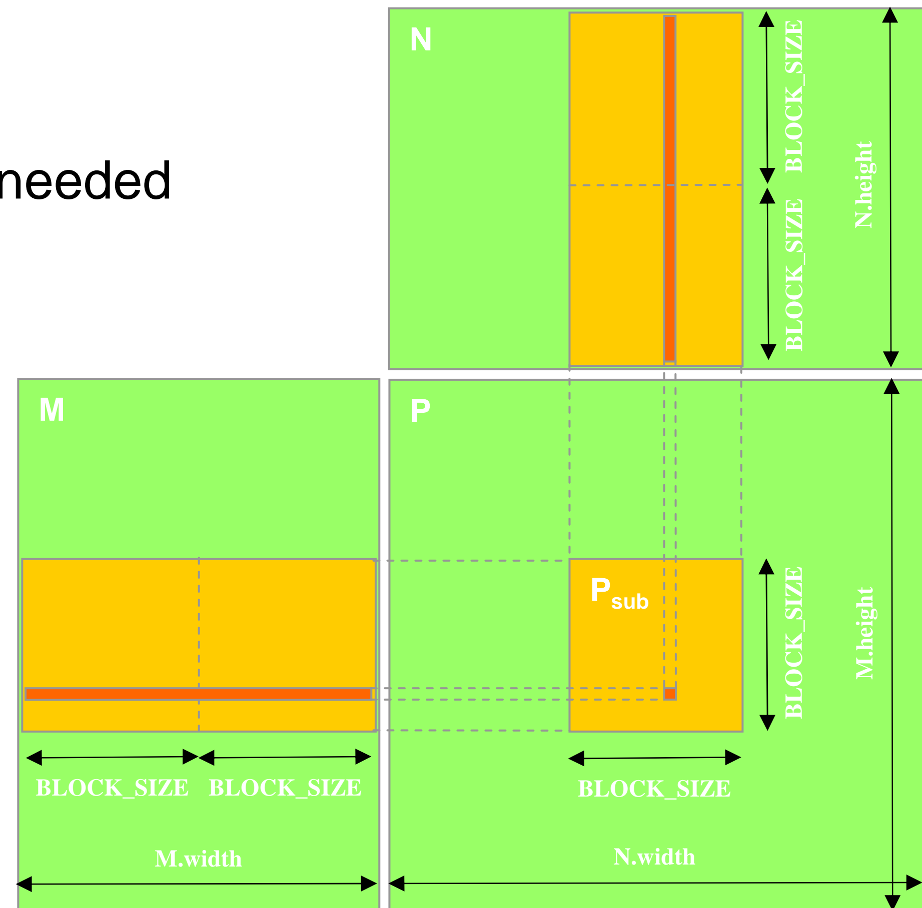
Data Decomposition Example - Square Matrix Multiplication

- **P Row blocks (adjacent rows)**
  - Computing each partition requires access to entire N array
- **Square sub-blocks**
  - Only bands of M and N are needed



Data Decomposition Example - Square Matrix Multiplication

- **P Row blocks (adjacent rows)**
  - Computing each partition requires access to entire N array
- **Square sub-blocks**
  - Only bands of M and N are needed

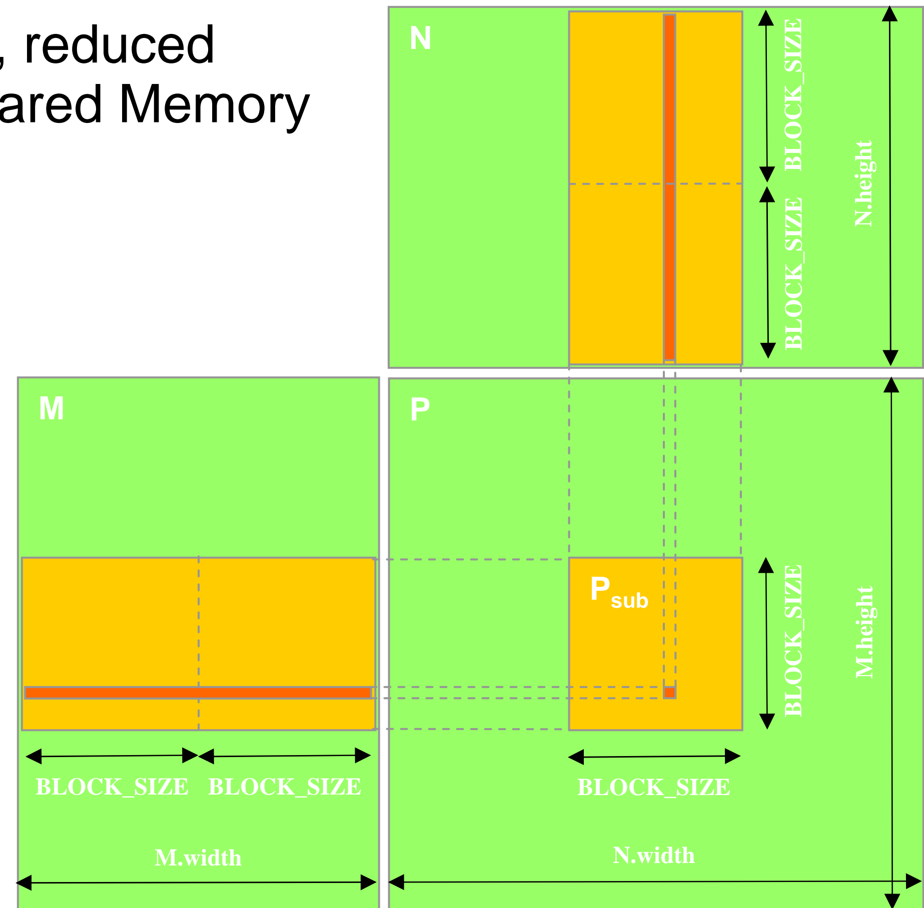


Tasks Grouping

- **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 (Shared Memory)
  – Grouping and merging dependent tasks into one task reduces need for synchronization

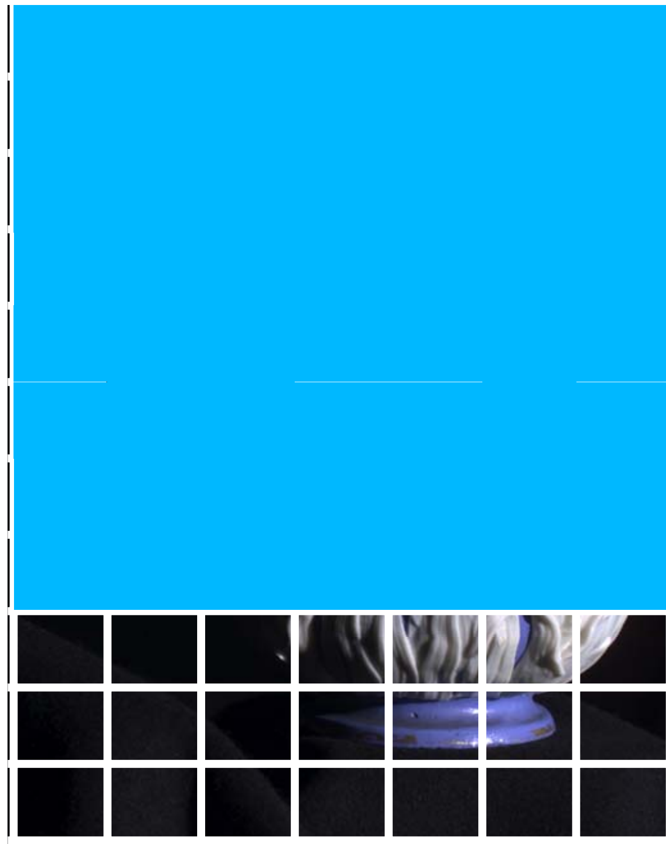
Task Grouping Example - Square Matrix Multiplication

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



Task Grouping Example - Cross Correlation

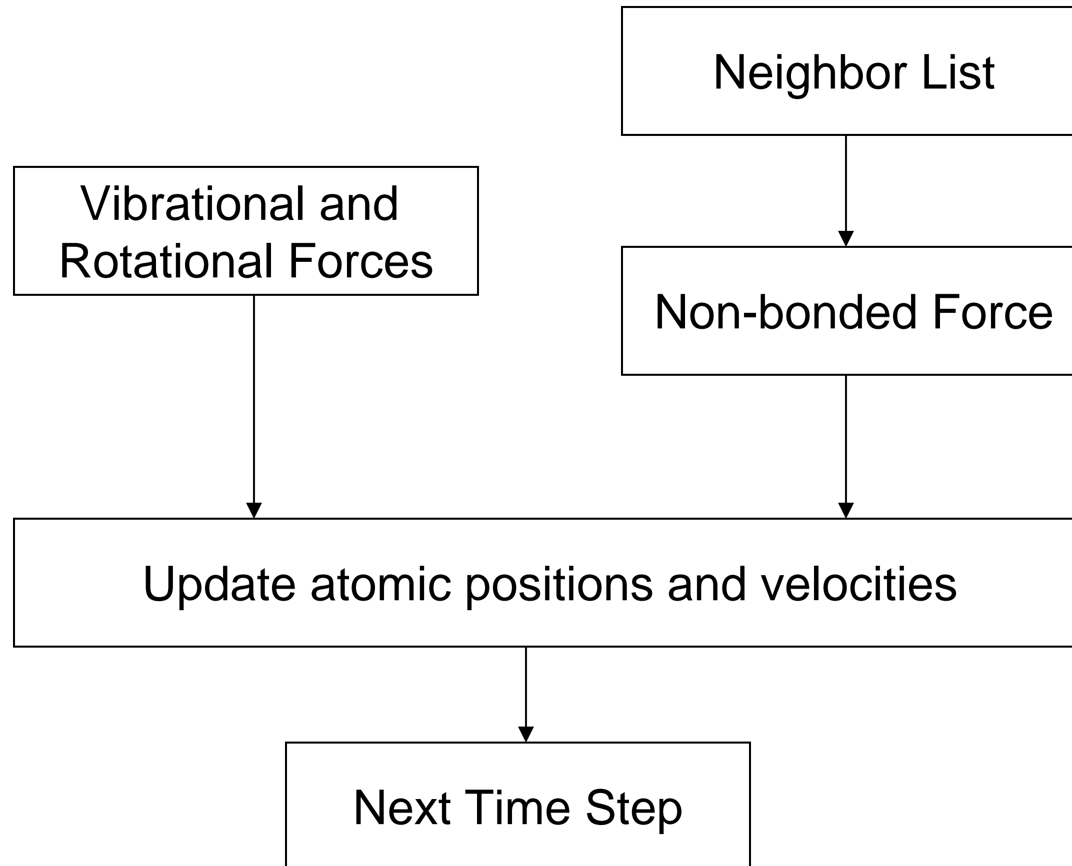
- **Tasks calculating an image sub-block**
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution



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



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

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?

Synchronization

How to avoid Race Conditions

- **Read-after-Write Hazards**
- **Write-after-Read Hazards**

- **Thanks to Mikael Kalms**

Simple Example - 2 Threads only

```
__global__ void testKernel(int* input, int* output)
{
  __shared__ int sharedMemory[2];
  for (int i = 0; i < 4; i++)
    {
      int globalMemoryOffset = i * 2;
      // Fetch value from device memory, store to shared memory
      sharedMemory[threadIdx.x] = input[globalMemoryOffset +
                                       threadIdx.x];
      // Compute product of the values fetched by threads 0 & 1
      int product = sharedMemory[0] * sharedMemory[1];
      // Store result out to device memory
      output[globalMemoryOffset + threadIdx.x] = product;
    }
}
```

- **input: a,b,c,d,e,f,g,h**
- **output (2 threads): a*b, a*b, c*d, c*d, e*f, e*f, g*h, g*h**

Lock-Step Execution (within Warp)

```
Thread 0                                Thread 1
--------                                --------
sharedMemory[0] = input[0]              sharedMemory[1] = input[1]
output[0] = sharedMemory[0]             output[1] = sharedMemory[0]
     * sharedMemory[1]                   * sharedMemory[1]

sharedMemory[0] = input[2]              sharedMemory[1] = input[3]
output[2] = sharedMemory[0]             output[3] = sharedMemory[0]
     * sharedMemory[1]                   * sharedMemory[1]

sharedMemory[0] = input[4]              sharedMemory[1] = input[5]
output[4] = sharedMemory[0]             output[5] = sharedMemory[0]
     * sharedMemory[1]                   * sharedMemory[1]

sharedMemory[0] = input[6]              sharedMemory[1] = input[7]
output[6] = sharedMemory[0]             output[7] = sharedMemory[0]
     * sharedMemory[1]                   * sharedMemory[1]

... thread 0 done                       ... thread 1 done
```

Read-after-Write Hazard

```
Thread 0                                  Thread 1
--------                                  --------
sharedMemory[0] = input[0]
output[0] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[2]
output[2] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[4]
output[4] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[6]
output[6] = sharedMemory[0]
     * sharedMemory[1]
... thread 0 done
                                          ... now thread 1 begins to
                                          its first operation!
                                          sharedMemory[1] = input[1]
                                          output[1] = sharedMemory[0]
                                                   * sharedMemory[1]
                                          sharedMemory[1] = input[3]
                                          output[3] = sharedMemory[0]
                                                   * sharedMemory[1] …
```

Without synchronization thread0 might start reading sharedMemory[1] before thread1 has written anything!

Add Synchronization

```
__global__ void testKernel(int* input, int* output)
{
  __shared__ int sharedMemory[2];
  for (int i = 0; i < 4; i++)
    {
      int globalMemoryOffset = i * 2;
      // Fetch value from device memory, store to shared memory
      sharedMemory[threadIdx.x] = input[globalMemoryOffset +
                                         threadIdx.x];
      // Wait until both threads have written a value each to
         shared memory
      __syncthreads();
      // Compute product of the values fetched by threads 0 & 1
      int product = sharedMemory[0] * sharedMemory[1];
      // Store result out to device memory
      output[globalMemoryOffset + threadIdx.x] = product;
    }
}
```

Write-after-Read Hazard

```
Thread 0                         Thread 1
--------                         --------
sharedMemory[0] = input[0]
                                 sharedMemory[1] = input[1]
-------------------- syncthreads() --------------------------

output[0] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[2]
                                 output[1] = sharedMemory[0]
                                           * sharedMemory[1]
                                 sharedMemory[1] = input[3]
-------------------- syncthreads() --------------------------

output[2] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[4]
                                 output[3] = sharedMemory[0]
                                           * sharedMemory[1]
                                 sharedMemory[1] = input[5]
-------------------- syncthreads() --------------------------
…
```

Write-after-Read Hazard

```
Thread 0                            Thread 1
--------                            --------
sharedMemory[0] = input[0]
                                    sharedMemory[1] = inp
-------------------- syncthreads() -------------

output[0] = sharedMemory[0]
     * sharedMemory[1]
sharedMemory[0] = input[2]
                                             = sharedMemory[0]
                                             * sharedMemory[1]
                                    aredMemory[1] = input[3]
----------------                  ads() --------------------------

output[2]             ory[0]
                   y[1]
sh         0] = input[4]
                                    output[3] = sharedMemory[0]
                                              * sharedMemory[1]
                                    sharedMemory[1] = input[5]
-------------------- syncthreads() --------------------------
…
```

thread0 might rewrite sharedMemory[0] before thread1 has accessed the old value!

Yet Another Synchronization

```
__global__ void testKernel(int* input, int* output)
{
  __shared__ int sharedMemory[2];
  for (int i = 0; i < 4; i++)
    {
      int globalMemoryOffset = i * 2;

      // Wait until both threads are finished using the shared
         memory data
      __syncthreads();

      // Fetch value from device memory, store to shared memory
      sharedMemory[threadIdx.x] = input[globalMemoryOffset +
                                        threadIdx.x];

      // Wait until both threads have written a value each to
         shared memory
      __syncthreads();

      // Compute product of the values fetched by threads 0 & 1
      int product = sharedMemory[0] * sharedMemory[1];

      // Store result out to device memory
      output[globalMemoryOffset + threadIdx.x] = product;
    }
}
```

Now everything is fine:

```
Thread 0                         Thread 1
--------                         --------
--------------------- syncthreads() ---------------------------
sharedMemory[0] = input[0]
                                 sharedMemory[1] = input[1]
--------------------- syncthreads() ---------------------------
output[0] = sharedMemory[0]
     * sharedMemory[1]
                                 output[1] = sharedMemory[0]
                                           * sharedMemory[1]
--------------------- syncthreads() ---------------------------
sharedMemory[0] = input[2]
                                 sharedMemory[1] = input[3]
--------------------- syncthreads() ---------------------------
output[2] = sharedMemory[0]
     * sharedMemory[1]
                                 output[3] = sharedMemory[0]
                                           * sharedMemory[1]
--------------------- syncthreads() ---------------------------
sharedMemory[0] = input[4]
                                 sharedMemory[1] = input[5]
```

Cross-Correlation Example

```
// This code is race condition free
for(lots of iterations)
  for(lots of iterations)
    {
     __syncthreads(); // protect from write-after-read hazard

     sharedMemory[threadIdx.x] = input[...];

     __syncthreads(); // protect from read-after-write hazard

     for (lots of iterations)
       for (lots of iterations)
       {
         result = computations;
       }

     output[...] = result;
    }
```

Summary

- **If threads are accessing data in shared (or device) memory which has been produced by other threads, you need to `__syncthreads()`.**
- **You need to look for two different kinds of hazards: read-after-write, and write-after-read.**
- **Generally speaking, you need `__syncthreads()` whenever the code switches back and forth between reading and writing to a shared memory location.**
- **If the code is in a loop, remember that it might switch between reading & writing between two iterations in the loop. Unrolling the loop makes it easier to spot any such pattern.**

Atomic Functions

Atomic Functions

- **Compute capability 1.1 – global memory only**
- **Compute capability 1.2 – global and shared memory**
- **Guaranteed to be executed without being interrupted by any other thread!**

- `int atomicAdd(int* addr, int val);`
- `int atomicSub(int* addr, int val);`
- `int atomicMin(int* addr, int val);`
- `int atomicMax(int* addr, int val);`
- `int atomicDec(int* addr, int val);`
- `int atomicExch(int* addr, int val);`

- **Read and modify!**
- **Returns old value of** `*addr`

Atomic Functions 2

- **Compare and Swap**
- `int atomicCAS(int* addr, int compare,int val);`
- **Evaluates**

```
*addr = (old == compare ? val : old)
```

- **and returns** `old`

Warp Vote Functions

- **`int __all(int predicate);`**
  - evaluates predicate for *all* threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for all of them
- **`int __any(int predicate);`**
  - evaluates predicate for *all* threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for any of them.

Histogram Idioms

[from David Kirk]

- **Making histograms is difficult without atomic operations (e.g. increment) to shared memory**
- **Several different techniques depending on how many histogram bins you need**
- **One example with 27-bit bins (top 5 bits store tid/warp):**

```
// one histogram array per 32-thread warp:
shared u32 histogram[NHIST][NBIN];

u16 bin = *pixPtr++;          // load a pixel from memory
histogram[myHist][bin] += 1; // potential conflicts across warp

// do this until update is not overwritten:
do {
  u32 myVal = histogram[myHist][bin] & 0x7FFFFFF; // read the current bin val
      myVal = ((tid & 0x1F) << 27) | (myVal + 1); // tag my updated val
      histogram[myHist][bin] = myVal;             // attempt to write the bin
} while (histogram[myHist][bin] != myVal);        // while update overwritten
```

- **If random bins updated, average #iterations <= 3**

My Histogram Version

- **Within a warp all threads are executed in lock-step**

```
int MyAtomicInc() {
  int res;
  do {
     res = a.x;
     a.x += 1;
     a.w = threadIdx.x
  } while (a.w != threadIdx.x);
  return res;
}
```

- **(no guarantee given)**

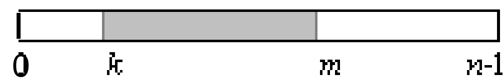
Parallel Sorting

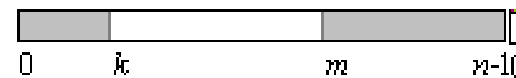
Bitonic Merge

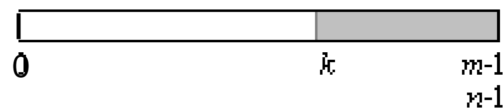
[see http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm]

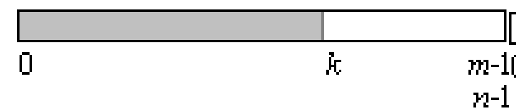
Bitonic Sequence

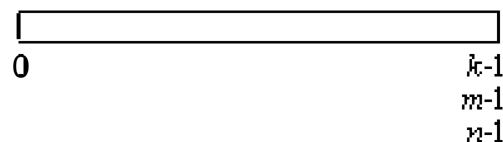
- **Binary sequence b_n-1 with at max two transitions at k and m.**

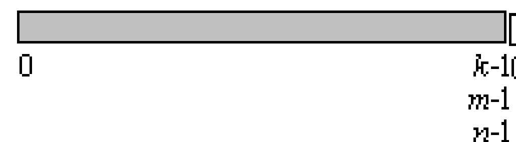












- **A sequence $A = a_0, a_1, \ldots, a_{n-1}$ is bitonic iff**

  1. There is an index i, $0 < i < n$, s.t.

  $a_0 .. a_i$ is increasing

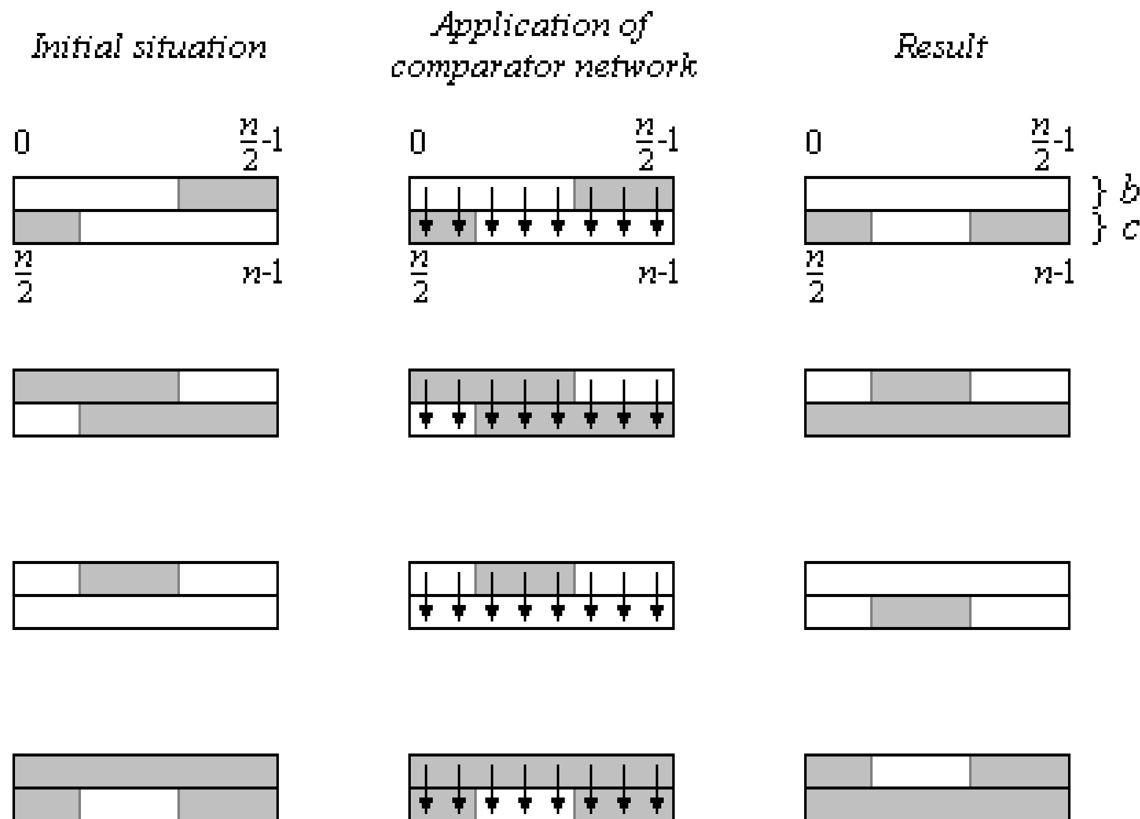
  and

  $a_i .. a_{n-1}$ is decreasing

or 2. There is a cyclic shift of A for which 1 holds.

Comparison Network

- **Given two bitonic sets b_n, c_n perform an element wise comparison storing the larger(smaller) value in c.**
- **Output is guaranteed to be bitonic again.**



Bitonic Split

- **A bitonic split divides a bitonic sequence in two:**

BitSplit(BS)=

$$S1 = (\min(bs_0, bs_{n/2}),\ \min(bs_1, bs_{n/2+1}),\ ..\ ,\ \min(bs_{n/2-1}, bs_{n-1}))$$

$$S2 = (\max(bs_0, bs_{n/2}),\ \max(bs_1, bs_{n/2+1}),\ ..\ ,\ \max(bs_{n/2-1}, bs_{n-1}))$$

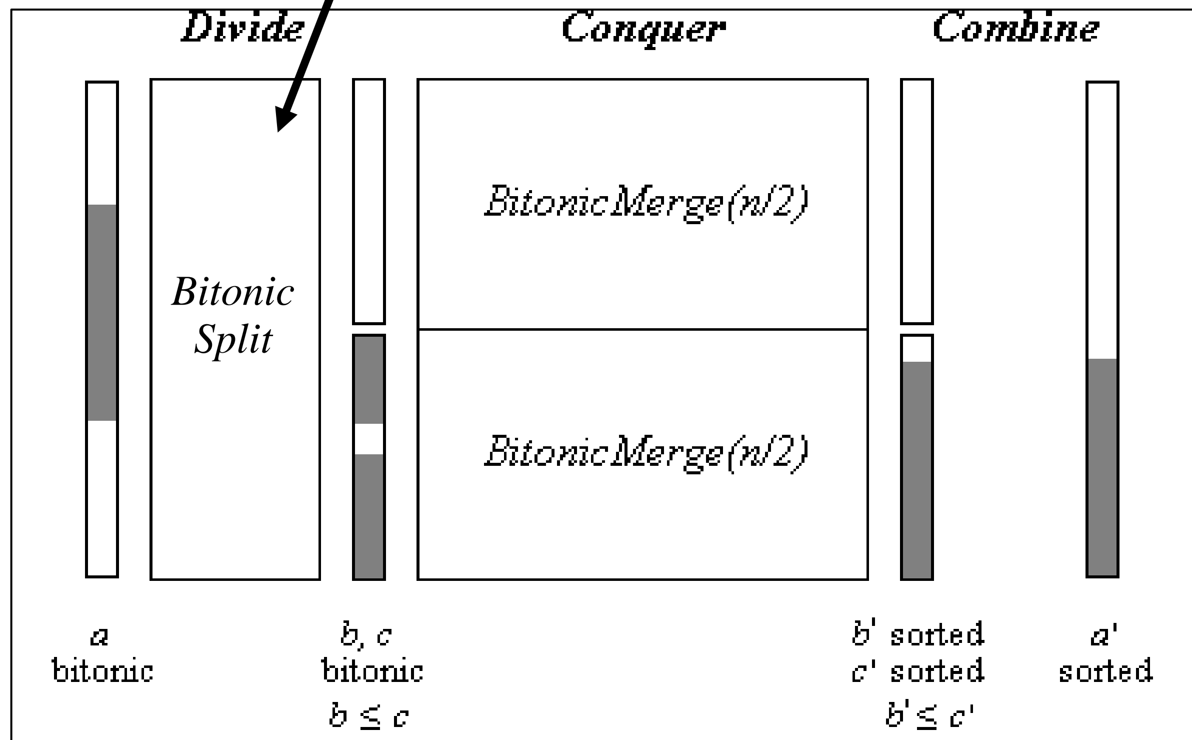
- **Theorem :**

  S1 and S2 are both bitonic

  $S1 < S2$

Bitonic Merger

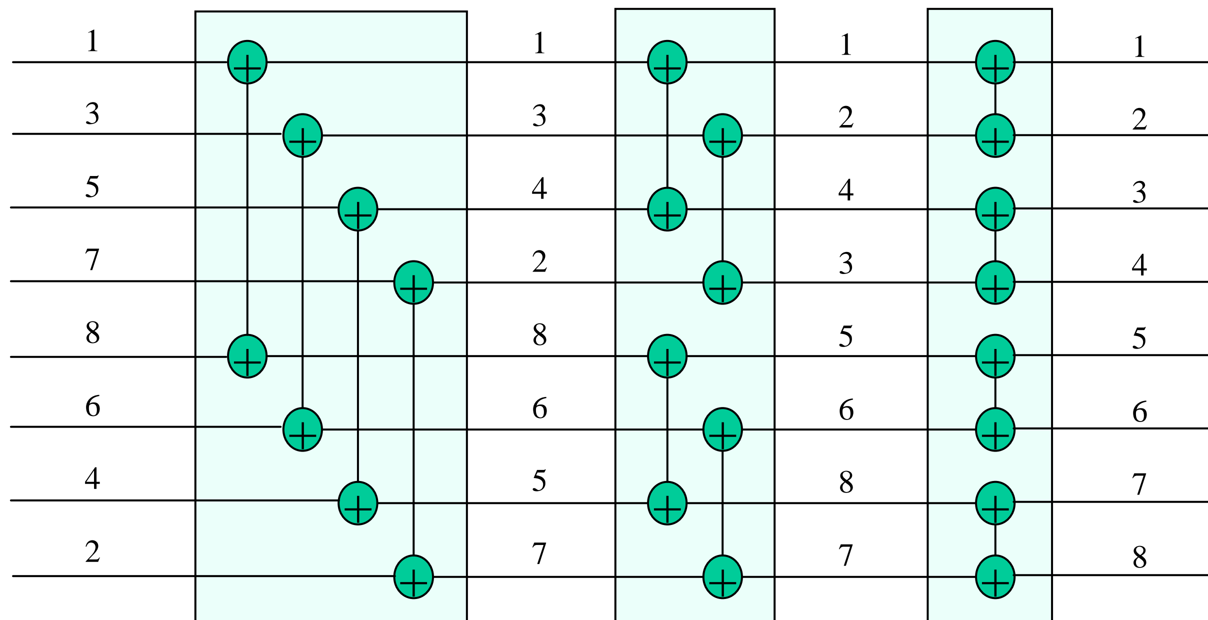
- **Given a bitonic vector, it will be sorted by applying a comparison network (divide) to its to halves.**
- **Then recurse.**



BitonicMerge(n)

Bitonic Merge

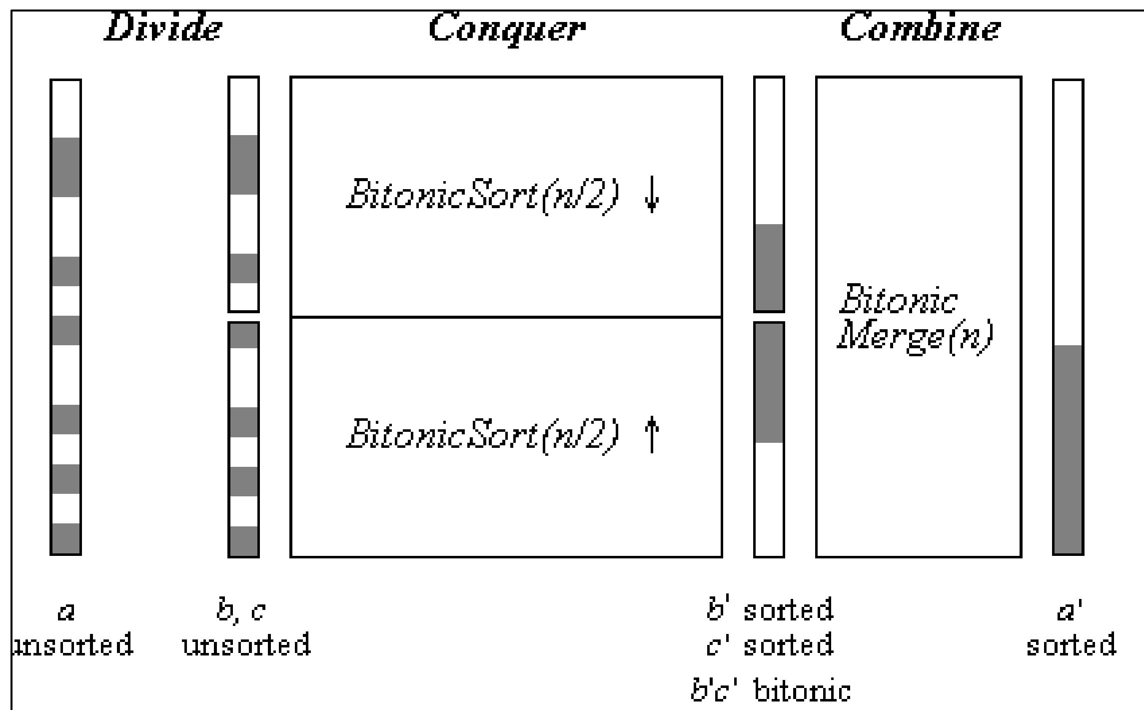
- **Given: a Bitonic Sequence BS of size n = 2^m**
- **Sort BS using m (parallel) Bitonic Splits stages**



Hendrik Lensch and Robert Strzodka

Bitonic Sort

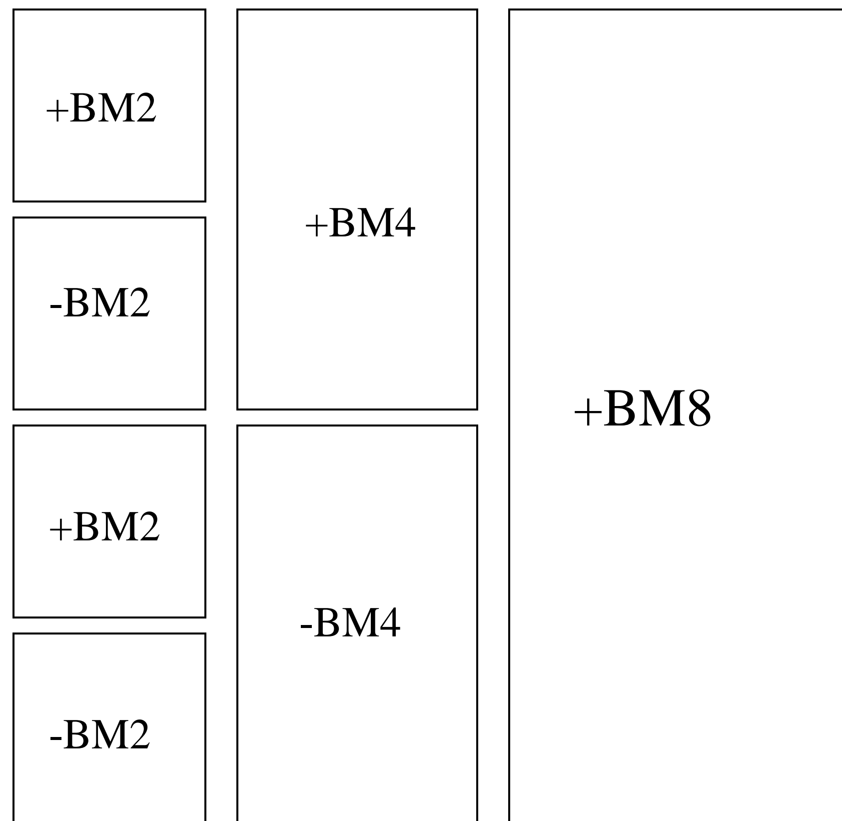
- **Split into two halves**
- **Sort ascending/descending**
- **Merge**



BitonicSort(n)

Bitonic Sort = log(n) Bitonic Merge Stages

- **Same number of comparison operations in each iteration**



Bitonic - Summary

- ***Complexity** $O(n \log(n)^2)$*
- **Fixed comparison networks**
  - they are all operating on the list in parallel
- **Simple to implement**

```
   // Copy input to shared mem.
   shared[tid] = values[tid];
 __syncthreads();

   // Parallel bitonic sort.
   for (int k = 2; k <= NUM; k *= 2)
   {
      // Bitonic merge:
      for (int j = k / 2; j>0; j /= 2)
         {
             int ixj = tid ^ j;  // XOR
             if (ixj > tid)
             {
                 if ((tid & k) == 0) // ascending - descending
                 {
                     if (shared[tid] > shared[ixj])
                     {
                         swap(shared[tid], shared[ixj]);
                     }
                 }
                 else
                 {
                     if (shared[tid] < shared[ixj])
                     {
                         swap(shared[tid], shared[ixj]);
                     }
                 }
             }
           __syncthreads();
    }    }
    // Write result.
    values[tid] = shared[tid];
}
```

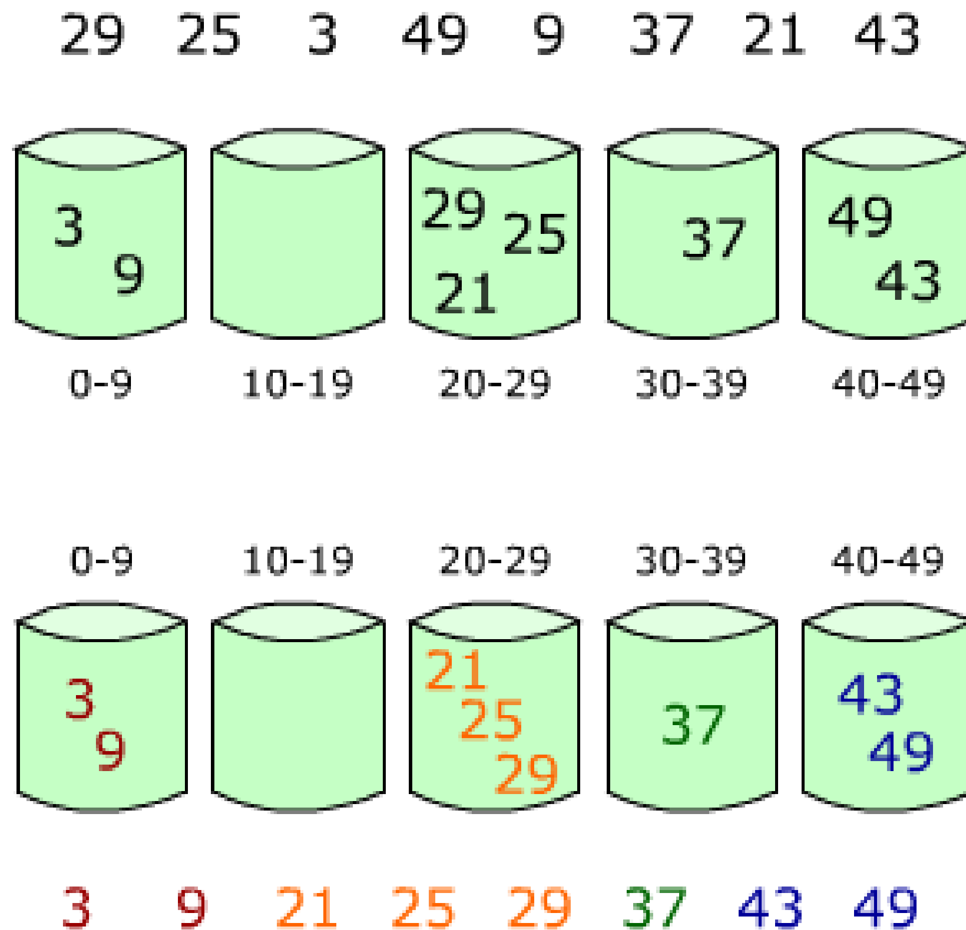
[from SDK]

Bucket Sort

Algorithm

1. **Set up an array of initially empty "buckets."**
2. **Scatter: Go over the original array, putting each object in its bucket.**
3. **Sort each non-empty bucket.**
4. **Gather: Visit the buckets in order and put all elements back into the original array.**

Example



[by Wikipedia]

Parallel Bucket Sort

- **Compute histogram**
  - e.g. using atomics (or following the example in the SDK)
  - `idx = atomicsInc().`
- **Compute Prefix Sum of histogram bins**
  - calculating the starting index of each bin –> `bin_start`
- **Write each element in parallel to (`bin_start+idx`)**
- **Now sort within bucket**
  - each bucket in parallel

Radix Sort

iterated bucket sort on individual digits
requires stable sorting

Review of Radix Sort

- **Sort from the least significant bit to the most significant bit using counting sort.**

- **Initial situation**
- **After sorting on 2nd digit**
- **After sorting on 1st digit**

89	28	81	69	14	31	29	18	39	17
81	31	14	17	28	18	89	69	29	39
14	17	18	28	29	31	39	69	81	89

Radix Sort - Review

- **Complexity: O(n)**
  - Fast
  - Simple to code
  - Requires stable sorting

Parallel Implementation

- **Main problem: keep sorting stable**
- **Compute radix histogram in each block**
- **Use PrefixSum to calculate offset for each bin and scatter**
- **Iterate**

```
for (uint shift = 0; shift < bits; shift += RADIX)
   {
       // Perform one round of radix sorting
       // Generate per radix group sums radix counts across a radix
          group
       RadixSum<<< … >>>(pData0, elements, shift);
       // Prefix sum in radix groups, and then between groups
          throughout a block
       RadixPrefixSum<<<…>>>();
       // Sum the block offsets and then shuffle data into bins
       RadixAddOffsetsAndShuffle<<<…>>>(pData0, pData1, elements,
                                        shift);
       // Exchange data pointers
       KeyValuePair* pTemp = pData0;
       pData0 = pData1;
       pData1 = pTemp;
   }
```

[from SDK]

Review of Parallel Radix Sort

- **Outer loop is independent of the problem size:**
- **Parallel complexity: O(1)**
  - given a sufficient number of processors

- **Problem:**
  - Current approaches write the entire array multiple times.

Hybrid Algorithm

Fast Parallel GPU-Sorting Using a Hybrid Algorithm

Erik Sintorn, Ulf Assarsson

Overview of the Algorithm

- **Main idea: Bucket Sort with load balancing**
  - **each bucket should receive approximately the same number of elements for internal sorting**
  - **need to determine bucket boundaries**
  - **sorting into buckets requires following a binary tree**

1. **Randomly shuffle elements O(*N*)**
2. **Compute initial pivot points O(*N*)**
3. **Bucket Sort O(*N*)**
4. **Vector Merge Sort O(*N*·log(*L*))**

Initial Pivot Points

- Calculate minimum and maximum
  - **glMinMax (~1% of total execution time)**
- Choose initial pivot points
  - **Linearly interpolate from min to max**
  - ***L*-1 pivot points | *L* ≥ 2*#Processors***
  - **With 32 SP (GeForce 8600), L=1024 was best**

Bucket Sort

1. **Compute histogram (optimized bucket counter)**
   - **Relies on linearly interpolated pivot points**
2. **Refine pivot points**
   - **Assume uniform distribution over range of each bucket**
3. **Count elements per bucket**
   - $\forall x \mid 1 \leq x \leq L : \text{size}(\text{sublist}[x]) \approx \frac{N}{L}$
4. **Reposition elements**
   - $\forall a \in \text{sublist}[l+1], b \in \text{sublist}[l] : a > b$

- **Drawback: Irregular bucket boundaries**

Summary

- **Parallel Design Patterns**
  - Decomposition
  - Dependence Analysis
  - Design Evaluation
- **Synchronization**
  - Beware of read-after-write and write-after-read hazards
  - Atomic Functions
- **Sorting**
  - Bitonic
  - Bucket, Radix, Hybird -- all rely on PrefixSum/Scan algorithm