Utrecht University

B3CC: Concurrency

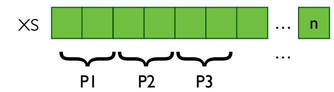
13: Data Parallelism (2)

Ivo Gabe de Wolff

Recap

- Data parallelism: well understood & supported approach to massive parallelism

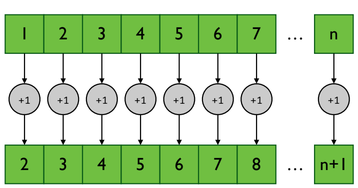
```
parallel_for (i = 1..N) {
    // ... do something to xs[i]
}
```



  - Single point of concurrency
  - Easy to implement: well supported (Fortran, MPI, OpenMP…), scales to large number of processors, etc.
  - Good cost model (work & span): conceptually very simple!
  - BUT! the "something" has to be sequential

Recap

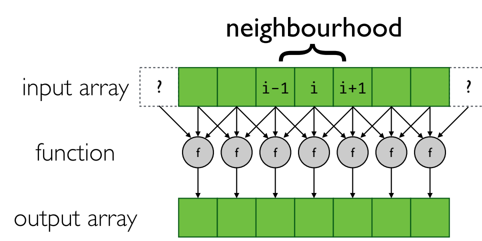
- The `map` operation applies the *same* function to each element of a set
  - This is a parallelisation of a loop with a *fixed* number of iterations
  - There must not be any dependencies between loop iterations: the function uses only the input element value and/or index




```
for (i = 0; i < len; ++i)
{
  x = xs[i];
  y = f(x);
  ys[i] = y;
}
```

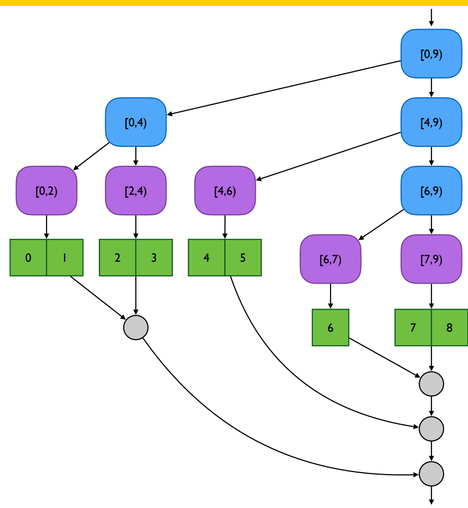

Recap

- A `map` with access to the neighbourhood around each element
  - The set of neighbours is fixed, and relative to the element
  - Ubiquitous in scientific, engineering, and image processing algorithms



Data parallelism on CPUs

- Distribute work via
  - Static schedule (like count & list mode of IBAN)
  - fork-join
  - divide-and-conquer (like search mode of IBAN)
  - …



Data parallelism on GPUs

- A GPU program consists of the kernel that runs on the GPU
  - Kernel functions are executed n times in parallel by n different threads
  - Each thread executes the same sequential program
  - Each thread can distinguish itself from all others *only* by it's thread identifier
    - Any information a thread needs should be directly derivable from this ID

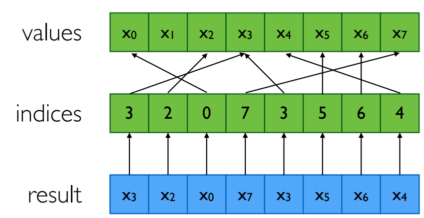
```
__global__ void kernel( float* xs, float* ys, int n, … )
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      // do something
   }
}
```


More parallel patterns

- We have seen:
  - Map
  - Stencil
- We will discuss today and next time:
  - Gather or backwards permute: random reads
  - Scatter or permutation: random writes
  - Fold or reduction: combined value of all items
  - Scan prefix sum: at each index, combined value of all prior elements

Gather

- The *gather* pattern performs independent random *reads* in parallel
  - Also known as a *backwards permutation*
  - Collects all the data from a source array at the given locations

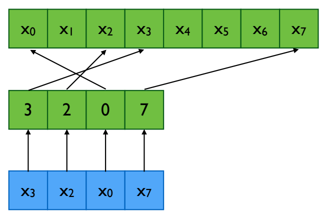


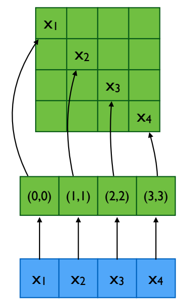

```
for (i = 0; i < len; ++i)
{
   idx = indices[i];
   val = values[idx];
   result[i] = val;
}
```

https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:29

Gather

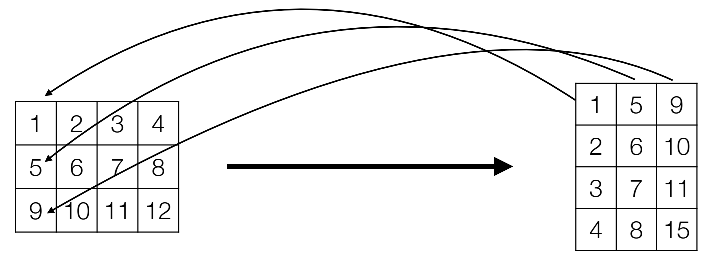
- The *gather* pattern performs independent random *reads* in parallel
  - Requires a function from output index to input index
  - Not all input values have to be read
  - Some values may be read twice
  - Input and output may have different dimensions





Example: matrix transpose

- Transpose rows and columns of a matrix



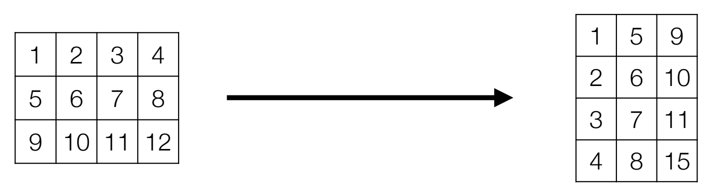
Example: matrix transpose

- Transpose the rows and columns of a matrix

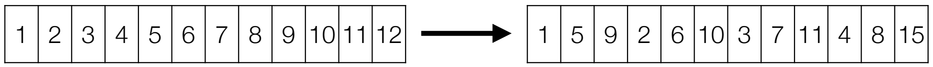
```
transpose :: Elt a ⇒ Acc (Matrix a) → Acc (Matrix a)
transpose xs =
  let I2 rows cols = shape xs
   in backpermute (I2 cols rows) (\(I2 y x) → I2 x y) xs

__global__ void transpose( float* xs, float* ys, int rows, int cols)
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if ( idx < n ) {
      int row = idx / rows;
      int col = idx % cols;
      …
   }
}
```


Example: matrix transpose



- In memory, this is stored as:



Example: matrix transpose

- To write one row of the output, we read one column of the input

1	2	3	4
5	6	7	8
9	10	11	12

→

1	5	9
2	6	10
3	7	11
4	8	15

1	2	3	4	5	6	7	8	9	10	11	12

→

1	5	9	2	6	10	3	7	11	4	8	15

Example: matrix transpose

- The memory access pattern for transpose is not ideal
  - On the CPU work in tiles to improve cache behaviour
  - On the GPU use shared memory explicitly to do coalesced reads & writes

Example: matrix vector multiply

- The *dense* matrix-vector multiply
  - Perform a dot-product of each row of the matrix against the vector
  - Can be parallelised in different ways

```
for (r = 0; r < rows; ++r) {
  result[r] = 0;
  for (c = 0; c < cols; ++c) {
    // dot product of this row with the vector
    result[r] += matrix[r][c] * vector[c];
  }
}
```


Example: sparse-matrix vector multiply

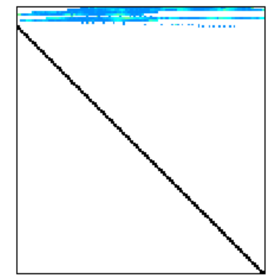
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 11 \\ 0 \\ 1 \\ 9 \end{pmatrix}$$

Example: sparse-matrix vector multiply

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ \boxed{11} \\ 0 \\ 1 \\ 9 \end{pmatrix}$$

Example: sparse-matrix vector multiply

- Multiply a *sparse* matrix by a dense vector
  - Example: Hardesty3 dataset
    - Matrix size is 8.2M x 7.6M
    - Only 40M non-zero entries (0.000065%)
  - Want to store only the non-zero entries, as only these will contribute to the result
  - Together with the row/column index of each element (various encodings possible)



https://sparse.tamu.edu/Hardesty/Hardesty3

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)
  - Stores only the non-zero elements together with their column index
  - Also need the number of non-zero elements in each row
  - …corresponds to:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

index-value pairs `[ (0, 1.0), (1, 1.0), (1, 7.0), (2, 3.0), (3, 2.0) , (1, 1.0), (3, 3.0), (4, 4.0) ]`

segment descriptor `[ 2, 3, 0, 1, 2 ]`

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)
  - Stores only the non-zero elements together with their column index
  - Also need the number of non-zero elements in each row
  - …corresponds to:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

index-value pairs `[ (0, 1.0), (1, 1.0), (1, 7.0), (2, 3.0), (3, 2.0) , (1, 1.0), (3, 3.0), (4, 4.0) ]`

segment descriptor `[ 2, 3, 0, 1, 2 ]`

Example: sparse-matrix vector multiply

- Store matrix in *compressed sparse row* format (CSR)

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 7 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 \end{pmatrix}$$

indices `[ 0, 1, 1, 2, 3, 1, 3, 4 ]`

values `[ 1.0, 1.0, 7.0, 3.0, 2.0, 1.0, 3.0, 3.0 ]`

segment descriptor `[ 2, 3, 0, 1, 2 ]`

vector `[ 3, 1, 0, 2, 1 ]`

- The sparse-matrix dense-vector multiply is then:
  1. *gather* the values from the input vector at the column indices
  2. pair-wise multiply (1) with the matrix values (*zipWith*)
  3. segmented *reduction* of (2) with the matrix segment descriptor
     - … *more on reductions and segmented operations next time!*

https://github.com/tmcdonell/accelerate-examples/tree/master/examples/smvm

Example: sparse-matrix vector multiply

- This can be viewed as a kind of *nested* data-parallel computation: parallel computations which spawn further parallel work
  - More difficult to parallelise (for both hardware and software)
  - Segmented operators allow us to convert nested parallel computations into *flat* parallel computations

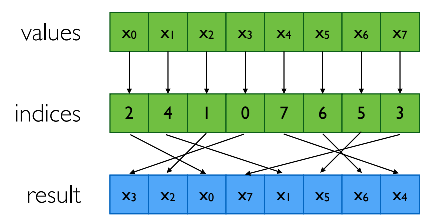
values	3	1	7	0	4	1	6	3
segment descriptor	2	3	0	1	2			
segmented fold	4	11	0	1	9			

Gather

- Gather or backwards permutation transforms indices in the *output* array to indices in the *input* array
  - But; arbitrary memory access patterns are slow (especially on the GPU)
  - Simple pattern; many common cases which can be made more efficient
- Next is scatter, forward permutation, which transforms indices in the *input* array to indices in the *output* array

Scatter

- The *scatter* pattern performs independent random *writes* in parallel
  - Also known as forward permutation
  - Puts data from the source array into the specified locations




```
for (i = 0; i < len; ++i)
{
   val = values[i];
   idx = indices[i];
   result[idx] = val;
}
```

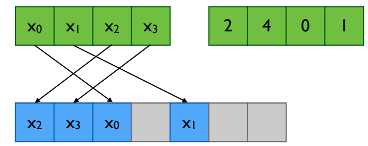
https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:28

Scatter

- The *scatter* pattern performs independent random *writes* in parallel
  - Analogously to gather, we can consider scatter as an index mapping f transforming indices in the *input* (source) array to indices in the *output* (destination) array
  - More complex than gather, especially if
    - f is not surjective: the range of f might not cover the entire codomain
    - f is not injective: distinct indices in the domain may map to the same index in the codomain
    - f is partial: elements in the domain may be ignored

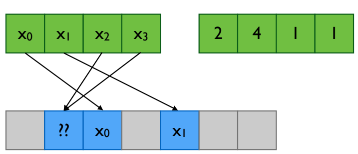
Scatter

- The index permutation might not cover every element in the output
  - We need to first initialise the output array



Collisions

- Multiple values may map to the same output index
  - Possible strategies to handle *collisions:*
    - Disallow
    - Non-deterministically, one write succeeds
    - Merge values with a given associative and commutative operation



Collisions: atomic instructions

Possible strategies to handle *collisions:*

1. Non-deterministically, one write succeeds
   - Requires atomic writes
   - Writes of single words are typically atomic, but that depends on architecture
2. Merge values with a given associative and commutative operation
   - Use an atomic read-modify-write instruction (e.g. atomic_fetch_add), if it exists for this operation
   - Use an atomic compare-and-swap loop, if a value is a single word
     - Maximal size of a word for compare-and-swap depends on the architecture
3. Use (per element) locks otherwise

Collisions: locks

- A general merge function might need to implement some locking strategy
  - If no atomic instruction exists; or multiple words are updated
  - Recall: this classic spin lock executed on the GPU can deadlock:

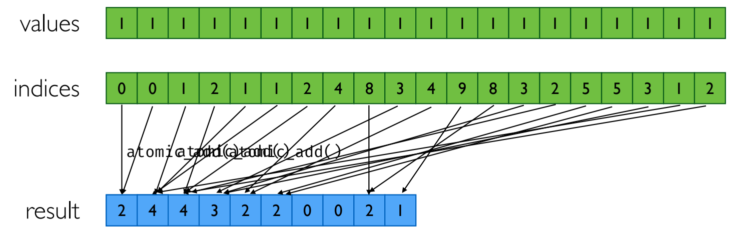
```
do {
  old = atomic_exchange(&lock[i], 1);
} while (old == 1);

/* critical section */

atomic_exchange(&lock[i], 0);
```


Example: histogram

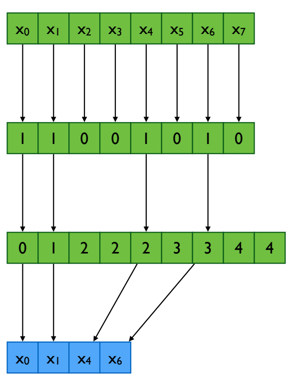
- Computing a histogram requires merging writes to the same location
- Sample data: `[0,0,1,2,1,1,2,4,8,3,4,9,8,3,2,5,5,3,1,2]`



https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#v:permute

Example: filter (compact)

- Return only those elements of the array which pass a predicate
  1. *map* the predicate function over the values to determine which to keep
  2. exclusive *scan* the boolean flags to determine the output locations and number of elements to keep
  3. *permute* the values into the position given by (2) if (1) is true



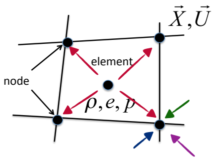
https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#v:filter

Scatter

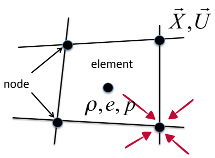
- Scatter is more expensive than gather for a number of reasons
  - Not only to handle collisions!
  - Due to the behaviour of caches, there is inter-core communication when threads access the same cache *line*, even if there is no actual collision
  - If the target locations are known in advance, scatter can be converted into a gather operation (this may require extra processing)

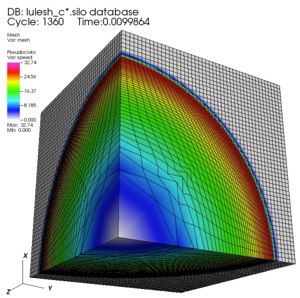
Scatter

- Reframing an algorithm can be key to converting scatter to gather
  - As always, there are different tradeoffs in computation vs. communication
  - Per element: scatter



  - Per node: gather





Summary

- Performance is often more limited by data movement than computation
  - Transferring data across memory layers is costly
  - Data organisation and layout can help to improve locality & minimise access times
  - Design the application around the data movement
- Similar consistency issues arise as when dealing with computation parallelism
- Might involve the creation of additional intermediate data structures
- Some applications are all about data movement: searching, sorting…