What is a Stream Programming Language?

- Describes kernels and streams
- Makes communication explicit
  - No ‘random’ memory references within kernels
- Easy to program
  - Sometimes at odds with explicit communication

What are the Issues?

Part I - Kernels

- How is a kernel described?
  - Implicit or explicit
  - Retained state or functional
  - Access across input streams
  - Access to multidimensional structures
  - Access to irregular structures (unstructured grids)
  - Access to ‘global’ data

Implicit vs Explicit

// loop over stream elements
for(i=0;i<MAX-FIRLEN;i++)
  s = 0
  for(j=0;j<FIRLEN;j++)
    s += a[i+FIRLEN-1-j]*h[j]
  b[i-FIRLEN+1] = s

kernel fir(floats a[i:0,FIRLEN-1], float h[FIRLEN], out floats b) {
  float s = 0
  for(j=0;j<FIRLEN;j++)
    s += a[FIRLEN-1-j]*h[j]
  b = s
}

fir(a, h, b)

Retained State vs Functional

// output stream is running sum of // Each element of b is only a function
// input stream // of the corresponding element of a
// kernel scan(istream a, ostream b) {
// kernel fn(floats a, out floats b) {
// each element of b is only a function
// scan requires "reduction" variables
// scan(floats a, out floats b, reduce float s) {
// scan with reduction variable
// return float b) {
// kernel fir(floats a[i:0,FIRLEN-1], float h[FIRLEN], out floats b) {
// return float b) {
// fir(a, h, b)
Access Across Input Streams

// sum pairs of input stream
// in Brook
kernel sumpair(floats a[1:-1,0], out floats b) {
    b = a + a[-1]
}
// note, new version of Brook requires
// stencil for a[-1,0]
// in KernelC – requires comm
kernel sumpair(istream a, ostream b) {
    loopstream (a) {
        a >> x;
        y = commucperm (…);
        // ugliness to deal with edge case
        z = x+y;
        b << z;
    }
}

// StreamIt – uses peek
Class Foo extends Filter {
    …
    void work() {
        x = input.peek(1) + input.pop() ;
        output.push(x) ;
    }
}

Access To Global Data

// in Brook
kernel lookup(istream table, istream a, ostream b) {
    i = 0;
    loopstream (table) {
        table >> tbl[i++];
    }
    loopstream (a) {
        a >> x;
        y = tbl[x];
        b << y;
    }
}
// in Brook
kernel lookup(ints a, int table[TSIZE], out ints b) {
    b = table[a];
}
// but aren’t we making random memory
// references here?

What are the Issues?
Part II - Streams

• How are streams connecting kernels described
  – How is a stream declared?
  – How is one stream derived from another?
  – How are common communication patterns implemented?
  – Are streams derived by copying or by reference?

Stream Declarations and Derivations

// StreamC
kernel1(a, b, c);
kernall2(b, c, d);
kernall3(c, d);
kernall4(d, e, f);

• StreamIt only allows the following constructors
  - Pipeline - one kernel follows another and consumes its output
  - Split-join - input stream is split and divided across kernels then joined
  - Feedback Loop - output 'split' passed through a kernel, and then 'joined' with input.

Communication Patterns

// StreamC
kernel1(a, b, c);
kernall2(b, c, d);
kernall3(c, d);
kernall4(d, e, f);

Brook

• What is the purpose of Brook?
  - Machine independent
  - No clusterisms
  - Suitable for parallel implementation
  - No serializations
  - No retained state
  - Reduction variables – can be converted to a ‘tree’
  - Support multidimensional arrays
  - Support irregular data structures (e.g., graphs)
  - Template declaration in argument list – details remain to be determined
Simple Example

typedef stream float floats;
floats x, y, z;
streamSetLength(x, 1024);
streamSetLength(y, 1024);
streamSetLength(z, 1024);
kernel double(floats a, out floats b){
b = 2*a;
}

void main() {
  // stuff to initialize x
double(x, y);
double(y, z);
}

2-D Array Access

typedef stream float floats;
floats x[1024];
streamShape(x, 2, 32, 32);
kernel neighborAvg(floats a[-1:1], out floats b){
  int i, j;
  float x = 0;
  b = 0.25*(a[-1,0]+a[1,0]+a[0,-1]+a[0,1]);
}

2-D Array Access
(new version of Brook)

typedef stream float floats;
typedef stream float floats2[3][3];
floats x;
floats2 y;
streamShape(x, 2, 32, 32);
streamStencil(y, x, STREAM_STENCIL_CLAMP, 2, -1, 1, -1, 1);
kernel void neighborAvg(floats2 a, out floats b){
  b = 0.25*(a[0][1]+a[2][1]+a[1][0]+a[1][2]);
}

Reduction

typedef stream float floats;
floats x, y;
streamSetLength(x, 1024); streamSetLength(y, 1024);
kernel void dotProduct(floats a, floats b, reduce float p){
p += a * b;
}

Irregular Structures

How would you code this in a stream language?

struct node {
float value;
float old_value;
int nr_neighbors;
struct node *neighbors;
}

For each node, *node
node->old_value = node->value;

For each node, *node
node->value = 0;

For each neighbor, *neighbor
node->value += neighbor->old_value;

Irregular Structures
One Possibility

struct node {
float value;
float old_value;
int nr_neighbors;
int start_neighbor;
}

typedef stream node nodes;
typedef stream int ints;
nodes nds[NR_NODES];
ints indices[NR_NEIGHBORS];
Nodes neighbors[NR_NEIGHBORS];
kernel neighborIndices(nodes nds, out ints indices) {
  int j;
  for(j = 0; j < nds.nr_neighbors; j++)
    push(nds.start_neighbor + j); // multiple outm args?
}

streamIndex(neighbors, nodes, indices); // want just the old_value field

kernel sumNeighbors(nodes nds, neighbors nds, out nodes new nds) {
  // need to consume the streams at different rates
}
Irregular Structures
A Cleaner Approach

```
struct node {
    float value;
    float old_value;
    int nr_neighbors;
    int start_neighbor;
};
typedef stream node nodes;
typedef stream int ints;

nodes nds[NR_NODES];
ints indices[NR_NEIGHBORS];

kernel sumNeighbors(nodes nds[indices[nds.start_neighbor..nds.start_neighbor+MAX_NEIGHBORS]],
{ int j;
    float sum = 0;
    for(j = 0 ; j< nds.nr_neighbors; j++)
        sum += nds[indices[nds.start_neighbor+j]].old_value;
    nds.value = sum;
})
```

Stream Languages
Summary

- Make communication explicit
  - By describing streams and kernels
- Narrow line between
  - Too difficult to express programs with non-trivial communication
  - Too easy to write inefficient programs
    - With unnecessary and unexposed communication
- Communication is declared
  - As input, output, and reduction streams
  - Restricting direction (no input/output) simplifies compilation
- Handling increasingly complex structures
  - Linear streams only – no access to other elements/data
  - Linear streams with access to neighbors (peek)
  - Arbitrary number of dimensions with access to “stencil”
  - Arbitrary structure with access to “template”

Stream Languages
Summary (cont)

- Kernel issues
  - Functional kernels make it easier for the compiler to exploit parallelism
    - Persistant state made explicit by “reduction variables”
    - Need an “imm” input type to allow different consumption rates of input streams
    - Sometimes want an “outer product” composition of input streams
  - Explicit kernels expose communication
  - Kernels should allow arbitrary accesses if declared
    - Nothing disallowed but no “hidden” global references
- Stream issues
  - Allow arbitrary connection of kernels
  - Often use “Indexing kernels”
  - Reference or copy semantics