Lecture 3:
Parallel Programming Abstractions
(and their corresponding HW/SW implementations)

Parallel Computing
Stanford CS149, Fall 2020
Today’s theme is a critical idea in this course.
And today’s theme is:

Abstraction vs. implementation

Conflating abstraction with implementation is a common cause for confusion in this course.
An example:
Programming with ISPC
ISPC

- Intel SPMD Program Compiler (ISPC)
- SPMD: single program multiple data

- http://ispc.github.com/

- A great read: “The Story of ISPC” (by Matt Pharr)
  - https://pharr.org/matt/blog/2018/04/30/ispc-all.html
Recall: example program from last class

Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...$

for each element of an array of N floating-point numbers

```c
void sinx(int N, int terms, float* x, float* result)
{
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;   // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```
Invoking \texttt{sinx()}

\textbf{C++ code: main.cpp}

```cpp
#include "sinx.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here
    sinx(N, terms, x, result);

    return 0;
}
```

\textbf{C++ code: sinx.cpp}

```cpp
void sinx(int N, int terms, float* x, float* result) {
    for (int i=0; i<N; i++)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6;  // 3!
        int sign = -1;
        for (int j=1; j<=terms; j++)
        {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```


**sinx() in ISPC**

**C++ code: main.cpp**

```cpp
#include "sinx_ispc.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here

    // execute ISPC code
    ispc_sinx(N, terms, x, result);
    return 0;
}
```

**ISPC code: sinx.ispc**

```cpp
export void ispc_sinx(  
    uniform int N,  
    uniform int terms,  
    uniform float* x,  
    uniform float* result)
{
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount) {  
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

**SPMD programming abstraction:**

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

Each instance has its own copy of local variables (blue variables in code, we’ll talk about “uniform” later)

Upon return, all instances have completed
Invoking sinx() in ISPC

C++ code: main.cpp

```c++
#include "sinx_ispc.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here

    // execute ISPC code
    ispc_sinx(N, terms, x, result);

    return 0;
}
```

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”
All instances run ISPC code concurrently
Each instance has its own copy of local variables
Upon return, all instances have completed

In this illustration programCount = 8
**sinx() in ISPC**

“Interleaved” assignment of array elements to program instances

**C++ code: main.cpp**
```
#include "sinx_ispc.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here

    // execute ISPC code
    ispc_sinx(N, terms, x, result);
    return 0;
}
```

**ISPC code: sinx.ispc**
```
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

**ISPC language keywords:**

- **programCount**: number of simultaneously executing instances in the gang (uniform value)
- **programIndex**: id of the current instance in the gang. (a non-uniform value: “varying”)
- **uniform**: A type modifier. All instances have the same value for this variable. Its use is purely an optimization. Not needed for correctness.
Interleaved assignment of program instances to loop iterations

“Gang” of ISPC program instances
In this illustration: gang contains four instances: \texttt{programCount} = 4
**ISPC implements the gang abstraction using SIMD instructions**

**C++ code:** main.cpp

```c++
#include "sinx_ispc.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here

    // execute ISPC code
    ispc_sinx(N, terms, x, result);
    return 0;
}
```

**SPMD programming abstraction:**

Call to ISPC function spawns “gang” of ISPC “program instances”

- All instances run ISPC code simultaneously
- Upon return, all instances have completed

**ISPC compiler generates SIMD implementation:**

- Number of instances in a gang is the SIMD width of the hardware (or a small multiple of SIMD width)
- ISPC compiler generates a C++ function binary (.o) whose body contains SIMD instructions
- C++ code links against generated object file as usual
**sinx() in ISPC: version 2**

“Blocked” assignment of elements to instances

### C++ code: main.cpp

```cpp
#include "sinx_ispc.h"

int main(int argc, void** argv) {
    int N = 1024;
    int terms = 5;
    float* x = new float[N];
    float* result = new float[N];

    // initialize x here

    // execute ISPC code
    ispc_sinx_v2(N, terms, x, result);
    return 0;
}
```

### ISPC code: sinx.ispc

```ispc
export void ispc_sinx_v2(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;
    for (uniform int i=0; i<count; i++)
    {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```
Blocked assignment of program instances to loop iterations

Elements of output array (results)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
</table>

“Gang” of ISPC program instances

In this illustration: gang contains four instances: `programCount = 4`
Schedule: interleaved assignment

“Gang” of ISPC program instances

Gang contains four instances: programCount = 4

<table>
<thead>
<tr>
<th>Instance 0 (programIndex = 0)</th>
<th>Instance 1 (programIndex = 1)</th>
<th>Instance 2 (programIndex = 2)</th>
<th>Instance 3 (programIndex = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Single “packed vector load” instruction (_mm_load_ps1) efficiently implements:

float value = x[idx];

for all program instances, since the four values are contiguous in memory

// assumes N % programCount = 0
for (uniform int i=0; i<N; i+=programCount) {
    int idx = i + programIndex;
    float value = x[idx];
...
Schedule: blocked assignment

“Gang” of ISPC program instances

Gang contains four instances: programCount = 4

<table>
<thead>
<tr>
<th>Instance 0 (programIndex = 0)</th>
<th>Instance 1 (programIndex = 1)</th>
<th>Instance 2 (programIndex = 2)</th>
<th>Instance 3 (programIndex = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

float value = x[idx];

Now touches four non-contiguous values in memory.

Need “gather” instruction to implement

(gather is a more complex, and more costly SIMD instruction: only available since 2013 as part of AVX2)
Raising level of abstraction with foreach

C++ code: main.cpp

```cpp
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: sinx.ispc

```ispc
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

foreach: key ISPC language construct

- **foreach** declares parallel loop iterations
  - Programmer says: these are the iterations the gang (not each instance) must perform

- ISPC implementation assigns iterations to program instances in the gang
  - Current ISPC implementation will perform a static interleaved assignment (but the abstraction permits a different assignment)
ISPC: abstraction vs. implementation

- Single program, multiple data (SPMD) programming model
  - Programmer “thinks”: running a gang is spawning programCount logical instruction streams (each with a different value of programIndex)
  - This is the programming abstraction
  - Program is written in terms of this abstraction

- Single instruction, multiple data (SIMD) implementation
  - ISPC compiler emits vector instructions (e.g., AVX2) that carry out the logic performed by a ISPC gang
  - ISPC compiler handles mapping of conditional control flow to vector instructions (by masking vector lanes, etc. like you do manually in assignment 1)

- Semantics of ISPC can be tricky
  - SPMD abstraction + uniform values (allows implementation details to peek through abstraction a bit)
ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

**Correct ISPC solution**

```ispc
export uniform float sumall2(
    uniform int N,
    uniform float* x)
{
    uniform float sum;
    foreach (i = 0 ... N)
    {
        partial += x[i];
    }
    // from ISPC math library
    sum = reduce_add(partial);
    return sum;
}
```

sum is of type uniform float (one copy of variable for all program instances)

x[i] is not a uniform expression (different value for each program instance)

Result: compile-time type error
ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

Each instance accumulates a private partial sum (no communication)

Partial sums are added together using the `reduce_add()` cross-instance communication primitive. The result is the same total sum for all program instances (`reduce_add()` returns a uniform float)

The ISPC code at right will execute in a manner similar to handwritten C + AVX intrinsics implementation below.*

```c
float sumall2(int N, float* x) {
    float tmp[8]; // assume 16-byte alignment
    __mm256 partial = _mm256_broadcast_ss(0.0f);

    for (int i=0; i<N; i+=8)
        partial = _mm256_add_ps(partial, _mm256_load_ps(&x[i]));

    _mm256_store_ps(tmp, partial);

    float sum = 0.f;
    for (int i=0; i<8; i++)
        sum += tmp[i];

    return sum;
}
```

*Self-test: If you understand why this implementation complies with the semantics of the ISPC gang abstraction, then you’ve got a good command of ISPC
SPMD programming model summary

- **SPMD** = “single program, multiple data”
- Define one function, run multiple instances of that function in parallel on different input arguments
ISPC tasks

- The ISPC gang abstraction is implemented by SIMD instructions that execute within on thread running on one x86 core of a CPU.

- So all the code I’ve shown you in the previous slides would have executed on only one of the four cores of the myth machines.

- ISPC contains another abstraction: a “task” that is used to achieve multi-core execution. I’ll let you read up about that.
Part 2 of today’s lecture

- Three parallel programming models
  - That differ in what communication abstractions they present to the programmer
  - Programming models are important because they (1) influence how programmers think when writing programs and (2) influence the design of parallel hardware platforms designed to execute them efficiently

- Corresponding machine architectures
  - Abstraction presented by the hardware to low-level software

- We’ll focus on differences in communication/synchronization
Three programming models (abstractions)

1. Shared address space
2. Message passing
3. Data parallel
Shared address space model
What is memory?

- On the first day of class, we described a program as a sequence of instructions
- Some of those instructions read and write from memory
- But what is memory?
  - To be precise, what I’m really asking is: what is the logical abstraction of memory as presented to a program
A program’s memory address space

- A computer’s memory is organized as an array of bytes
- Each byte is identified by its “address” in memory (its position in this array)
  (in this class we assume memory is byte-addressable)

"The byte stored at address 0x8 has the value 32."
"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program’s memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0x1F)
The implementation of the linear memory address space abstraction on a modern computer is complex.

The instruction “load the value stored at address X into register R0” might involve a complex sequence of operations by multiple data caches and access to DRAM.
Shared address space model (abstraction)

Threads communicate by reading/writing to shared variables

Thread 1:
```c
def spawn_thread(x):
    x = 0
    spawn_thread(1, &x)
    // write to address holding
    // contents of variable x
    x = 1;
```

Thread 2:
```c
void foo(int* x) {
    // read from addr storing
    // contents of variable x
    while (x == 0) {} // write to address holding
    print x;
}
```

(Pseudocode provided in a fake C-like language for brevity.)
A common metaphor

Image credit:
Shared address space model

Threads must synchronize their reads and writes to shared variables

Synchronization primitives are also shared variables: e.g., locks

Thread 1:

```c
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);

mylock.lock();
x++;
mylock.unlock();
```

Thread 2:

```c
void foo(int* x, lock* my_lock) {
    my_lock->lock();
x++;
    my_lock->unlock();
    print x;
}
```
Review: why do we need mutual exclusion?

- Each thread executes
  - Load the value of `diff` from location in memory into register `r1` *(this stores a copy of the value in memory in the register)*
  - Add the register `r2` to register `r1`
  - Store the value of register `r1` into `diff`

- One possible interleaving: (let starting value of `diff`=0, `r2`=1)

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T0 reads value 0</th>
<th>T1 reads value 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r1 ← diff</code></td>
<td><code>r1 ← diff</code></td>
<td>T0 sets value of its <code>r1</code> to 1</td>
<td>T1 sets value of its <code>r1</code> to 1</td>
</tr>
<tr>
<td><code>r1 ← r1 + r2</code></td>
<td><code>r1 ← r1 + r2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>diff ← r1</code></td>
<td><code>diff ← r1</code></td>
<td>T0 stores 1 to <code>diff</code></td>
<td>T1 stores 1 to <code>diff</code></td>
</tr>
</tbody>
</table>

- Need this set of three instructions must be “atomic”
Mechanisms for preserving atomicity

- Lock/unlock mutex around a critical section
  
  ```
  LOCK(mylock);
  // critical section
  UNLOCK(mylock);
  ```

- Some languages have first-class support for atomicity of code blocks
  
  ```
  atomic {
    // critical section
  }
  ```

- Intrinsics for hardware-supported atomic read-modify-write operations
  
  ```
  atomicAdd(x, 10);
  ```
Review: shared address space model

- Threads communicate by:
  - Reading/writing to shared variables in a shared address space
    - Inter-thread communication is implicit in memory loads/stores
  - Thread 1 stores to X
  - Later, thread 2 reads X (and observes update of value by thread 1)
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks

- This is a natural extension of sequential programming
  - In fact, all our discussions in class have assumed a shared address space so far!
HW implementation of a shared address space

Key idea: any processor can directly reference contents of any memory location

* Caches (not shown) are another implementation of a shared address space (more on this in a later lecture)
Shared address space HW architecture

Example: Intel Core i7 processor (Kaby Lake)

Intel Core i7 (quad core)
(intercconnect is a ring)
Intel’s ring interconnect

Introduced in Sandy Bridge microarchitecture

- Four rings
  - request
  - snoop
  - ack
  - data (32 bytes)

- Six interconnect nodes: four “slices” of L3 cache + system agent + graphics

- Each bank of L3 connected to ring bus twice

- Theoretical peak BW from cores to L3 at 3.4 GHz is approx. 435 GB/sec
  - When each core is accessing its local slice
SUN Niagara 2 (UltraSPARC T2): crossbar interconnect

Note area of crossbar (CCX):
about same area as one core on chip

Eight cores
72 cores, arranged as 6 x 6 mesh of tiles (2 cores/tile)

YX routing of messages:
- Message travels in Y direction
- “Turn”
- Message travels in X direction
Non-uniform memory access (NUMA)

The latency of accessing a memory location may be different from different processing cores in the system.

Example: latency to access address $x$ is higher from cores 5-8 than cores 1-4.

Example: modern dual-socket configuration.

Bandwidth from any one location may also be different to different CPU cores.
Summary: shared address space model

- **Communication abstraction**
  - Threads read/write variables in shared address space
  - Threads manipulate synchronization primitives: locks, atomic ops, etc.
  - Logical extension of uniprocessor programming *

- **Requires hardware support to implement efficiently**
  - Any processor can load and store from any address
  - Can be costly to scale to large numbers of processors
    (one of the reasons why high-core count processors are expensive)

* But NUMA implementation requires reasoning about locality for performance
Message passing model of communication
Message passing model (abstraction)

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier (“tag”)
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2
    - Why?

Illustration adopted from Culler, Singh, Gupta
A common metaphor: snail mail
Message passing (implementation)

- Hardware need not implement system-wide loads and stores to execute message passing programs (it need only communicate messages between nodes).
  - Can connect commodity systems together to form a large parallel machine (message passing is a programming model for clusters and supercomputers).
Keep in mind (again): programming model abstraction is distinct from its implementation

- Common to implement message passing abstractions on machines that implement a shared address space in hardware
  - “Sending message” = copying memory from message library buffers
  - “Receiving message” = copy data from message library buffers

- Can implement shared address space abstraction on machines that do not support it in HW (via less efficient SW implementations)
  - OS marks all pages with shared variables as invalid
  - OS page-fault handler issues appropriate network requests

- Keep clear in your mind: what is the programming model (abstractions used to specify program)? And what is the HW implementation?
The data-parallel model
Programming models provide a way to think about the organization of parallel programs (by imposing structure)

- **Shared address space**: very little structure to communication
  - All threads can read and write to all shared variables

- **Message passing**: communication is structured in the form of messages
  - All communication occurs in the form of messages (communication is explicit in source code—the sends and receives)

- **Data parallel structure**: more rigid structure to computation
  - Perform same function on elements of large collections
Data-parallel model *

- Organize computation as operations on sequences of elements
  - e.g., perform same function on all elements of a sequence

- Historically: same operation on each element of an array
  - Matched capabilities SIMD supercomputers of 80’s
  - Connection Machine (CM-1, CM-2): thousands of processors, one instruction decode unit
  - Early Cray supercomputers were vector processors
    - \texttt{add(A, B, n)} ← this was one instruction on vectors A, B of length n

- A well-known modern example: NumPy: \( C = A + B \)
  (A, B, and C are vectors of same length)

* We’ll have multiple lectures in the course about data-parallel programming and data-parallel thinking: this is just a taste
Key data type: sequences

- Ordered collection of elements
- For example, in a C++ like language: Sequence<T>
- e.g., Scala lists: List[T]
- In a functional language (like Haskell): seq T

- Program can only access elements of sequence through specific operations
**Map**

- Higher order function (function that takes a function as an argument) that operates on sequences
- Applies side-effect free unary function \( f : : a \rightarrow b \) to all elements of input sequence, to produce output sequence of the same length
- In a functional language (e.g., Haskell)
  - `map :: (a -> b) -> seq a -> seq b`
- In C++:
  ```cpp
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```
Parallelizing map

- Since \( f : : a \rightarrow b \) is a function (side-effect free), then applying \( f \) to all elements of the sequence can be done \textbf{in any order} without changing the output of the program.

- The implementation of map has flexibility to reorder/parallelize processing of elements of sequence however it sees fit.
Optimizing data movement in map

- Consider code that performs two back-to-back maps (like that to left)
- An optimizing compiler or runtime can reorganize code (bottom-left) to eliminate memory loads and stores (“map fusion”)
- Additional optimizations: highly optimized implementations of map can also perform optimizations like prefetching next element of input sequence (to hide memory latency)
- Think to yourself: why are these complex optimizations possible?

```cpp
const int N = 1024;
Sequence<float> input(N);
Sequence<float> tmp(N);
Sequence<float> output(N);

map(foo, input, tmp);
map(bar, tmp, output);

parallel_for(int i=0; i<N; i++)
{
    output[i] = bar(foo(input[i]));
}
```
Data parallelism in ISPC

// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x here
absolute_value(N, x, y);

// ISPC code:
export void absolute_value(
  uniform int N,
  uniform float* x,
  uniform float* y)
{
  foreach (i = 0 ... N)
  {
    if (x[i] < 0)
      y[i] = -x[i];
    else
      y[i] = x[i];
  }
}

foreach construct

Think of loop body as a function

Given this program, it is reasonable to think of the program as using foreach to “map the loop body onto each element” of the arrays X and Y.

But if we want to be more precise: a sequence is not a first-class ISPC concept. It is implicitly defined by how the program has implemented array indexing logic in the foreach loop.

(There is no operation in ISPC with the semantic: “map this code over all elements of this sequence”)
Data parallelism in ISPC

// ISPC code:
export void absolute_repeat(
    uniform int N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
        if (x[i] < 0)
            y[2*i] = -x[i];
        else
            y[2*i] = x[i];
        y[2*i+1] = y[2*i];
    }
}

Think of loop body as a function

The input/output sequences being mapped over are implicitly defined by array indexing logic

This is also a valid ISPC program!

It takes the absolute value of elements of x, then repeats it twice in the output array y

(Less obvious how to think of this code as mapping the loop body onto existing sequences.)
Data parallelism in ISPC

// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x
shift_negative(N, x, y);

思之为循环体为一个函数

输入/输出序列被映射的输入/输出序列由数组索引逻辑隐式定义

// ISPC code:
export void shift_negative(
    uniform int N,
    uniform float* x,
    uniform float* y)
{
    foreach (i = 0 ... N)
    {
        if (i >= 1 && x[i] < 0)
            y[i-1] = x[i];
        else
            y[i] = x[i];
    }
}

输出该程序的输出是未定义的!

可能多次循环体的迭代写入同一内存位置

数据并行模型（foreach）提供没有指定迭代之间发生的顺序的描述

但数据并行模型提供了没有提供细粒度的互斥/同步的原语。
它不是帮助程序员写具有该结构的程序
Gather/scatter: two key data-parallel sequence operations

Map absolute_value() onto stream produced by gather:

```cpp
const int N = 1024;
Sequence<float> input(N);
Sequence<int> indices;
Sequence<float> tmp_input(N);
Sequence<float> output(N);

stream_gather(input, indices, tmp_input);
absolute_value(tmp_input, output);
```

ISPC equivalent:

```cpp
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        float tmp = input[indices[i]];
        if (tmp < 0)
            output[i] = -tmp;
        else
            output[i] = tmp;
    }
}
```

Map absolute_value() onto stream, scatter results:

```cpp
const int N = 1024;
Sequence<float> input(N);
Sequence<int> indices;
Sequence<float> tmp_output(N);
Sequence<float> output(N);

absolute_value(input, tmp_output);
stream_scatter(tmp_output, indices, output);
```

ISPC equivalent:

```cpp
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        if (input[i] < 0)
            output[indices[i]] = -input[i];
        else
            output[indices[i]] = input[i];
    }
}
```
Gather instruction

gather(R1, R0, mem_base);  "Gather from buffer mem_base into R1 according to indices specified by R0."

Array in memory with (base address = mem_base)

Index vector: R0

Result vector: R1

Gather supported with AVX2 in 2013
But AVX2 does not support SIMD scatter (must implement as scalar loop)
Scatter instruction exists in AVX512

Hardware supported gather/scatter does exist on GPUs.
(still an expensive operation compared to load/store of contiguous vector)
Summary: data-parallel model

- Data-parallelism is about imposing rigid program structure to facilitate simple programming and advanced optimizations

- Basic structure: map a function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among distinct function invocations
    (allow invocations to be scheduled in any order, including in parallel)

- Other data parallel operators express more complex patterns on sequences: gather, scatter, reduce, scan, shift, etc.
  - This will be a topic of a later lecture

- You will think in terms of data-parallel primitives often in this class, but many modern performance-oriented data-parallel languages do not enforce this structure in the language
  - Many languages (like ISPC, CUDA, etc.) choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form
Summary
Summary

- Programming models provide a way to think about the organization of parallel programs.

- They provide abstractions that permit multiple valid implementations.

- *I want you to always be thinking about abstraction vs. implementation for the remainder of this course.*
Summary

Restrictions imposed by these abstractions are designed to:

1. Reflect realities of parallelization and communication costs to programmer (help a programmer write efficient programs)
   - Shared address space machines: hardware supports any processor accessing any address
   - Messaging passing machines: hardware may accelerate message send/receive/buffering
   - Desirable to keep “abstraction distance” low so programs have predictable performance, but want abstractions to be high enough for code flexibility/portability

2. Provide useful information to implementors of optimizing compilers/runtimes/hardware to help them efficiently implement programs using these abstractions
   - Consider optimizations possible when implementing ISPC foreach vs higher-order map
Modern practice: mixed programming models

- Use shared address space programming within a multi-core node of a cluster, use message passing between nodes
  - Very common in practice
  - Offer convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere

- Data-parallel-ish programming models often support shared-memory style synchronization primitives in functions
  - e.g., CUDA, OpenCL

- In a future lecture... CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate.
Questions to consider

- Programming models enforce different forms of structure on programs. What are the benefits of data-parallel structure?

- With respect to the goals of efficiency/performance... what do you think are problems of adopting a very high level of abstraction in a programming system?
  - What about potential benefits?

- Choose a popular parallel programming system (for example Hadoop, Spark, or Cilk) and try and describe its programming model (how are communication and execution expressed?)