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

Parallel Computing
Stanford CS149, Fall 2021
Finishing up from last time + Review
REVIEW

HOW IT ALL FITS TOGETHER:

superscalar execution,
SIMD execution,
multi-core execution,
and hardware multi-threading
void sinx(int N, int terms, float* x, float* y)
{
    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;
        }

        y[i] = value;
    }
}

```
lod r0, addr[r1]
mul r1, r0, r0
add r2, r0, r0
mul r3, r1, r2
...  
...  
...  
...  
...  
...  
st addr[r2], r0
```
Running code on a simple processor

Instruction stream

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
...
st  addr[r2], r0
```

Single core processor, single-threaded core. Can run one scalar instruction per clock
Superscalar core

Instruction stream

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...   ...
...   ...
...   ...
st    addr[r2], r0
```

Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent scalar instructions per clock from one instruction stream (one hardware thread)
**SIMD execution capability**

*Instruction stream (now with vector instructions)*

- `vector_ld v0, vector_addr[r1]`
- `vector_mul v1, v0, v0`
- `vector_add v2, v0, v0`
- `vector_mul v3, v1, v2`
- ...
- ...
- ...
- ...
- `vector_st addr[r2], v0`

Single core processor, single-threaded core. Can run one 8-wide SIMD vector instruction from one instruction stream.
Single core processor, single-threaded core.  
Two-way superscalar core: can run up to two independent instructions per clock from one instruction stream, provided one is scalar and the other is vector.
Multi-threaded core

Instruction stream 0

```
ld  r0, addr[r1]
mul r1, r0, r0
add r2, r0, r0
mul r3, r1, r2
... ...
... ...
... ...
st addr[r2], r0
```

Instruction stream 1

```
ld  r0, addr[r1]
sub r1, r0, r0
add r2, r1, r0
mul r5, r1, r0
... ...
... ...
... ...
st addr[r2], r0
```

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

Single core processor, multi-threaded core (2 threads). Can run one scalar instruction per clock from one of the instruction streams (hardware threads)
Multi-threaded, superscalar core

In this example: two instructions from the same thread.
(Superscalar execution, interleaved multi-threading)

Instruction stream 0
- vector ld v0, addr[r1]
- vector mul v1, v0, v0
- vector add v2, v1, v1
- mul r2, r1, r1
- ...
- ...
- ...
- ...
- vector st addr[r2], v0

Instruction stream 1
- vector ld v0, addr[r1]
- sub r1, r0, r0
- vector add v2, v0, v0
- mul r5, r1, r0
- ...
- ...
- ...
- ...
- rect addr[r2], v0

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

Single core processor, multi-threaded core (2 threads).
Two-way superscalar core: in this example, my core is capable of running up to two independent instructions per clock, provided one is scalar and the other is vector
Multi-threaded, superscalar core

In this example: two instruction from different threads. (simultaneous multi-threading)

Instruction stream 0

Instruction stream 1

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Simultaneous execution of two hardware threads.

Single core processor, multi-threaded core (2 threads).
Two-way superscalar core: in this example, my core is capable of running up to two independent instructions per clock, provided one is scalar and the other is vector
Multi-threaded, superscalar core

Execution of hardware threads may or may not be interleaved in time (instructions from different threads may be running simultaneously)
Multi-core, with multi-threaded, superscalar cores

Dual-core processor, multi-threaded cores (4 threads/core).
Two-way superscalar cores: each core can run up to two independent instructions per clock from any of its threads, provided one is scalar and the other is vector.
Example: Intel Skylake core

Two-way multi-threaded cores (2 threads).
Each core can run multiple independent scalar instructions and multiple 8-wide vector instructions (up to 2 vector mul or 3 vector add)

Not shown on this diagram: units for LD/ST operations
Many modern GPUs execute hardware threads that run instruction streams with only scalar instructions.

GPU cores detect when different hardware threads are executing the same instruction, and implement simultaneous execution of up to SIMD-width threads using SIMD ALUs.

Here ALU 6 would be “masked off” since thread 6 is not executing the same instruction as the other hardware threads.
GPUs: extreme throughput-oriented processors

This is one NVIDIA V100 streaming multi-processor (SM) unit

64 “warp” execution contexts per SM

Wide SIMD: 16-wide SIMD ALUs (carry out 32-wide SIMD execute over 2 clocks)

64 x 32 = up to 2048 data items processed concurrently per “SM” core

64 KB registers per sub-core

256 KB registers in total per SM

Registers divided among (up to) 64 “warps” per SM

“Shared” memory + L1 cache storage (128 KB)

- SIMD fp32 functional unit, control shared across 16 units (16 x MUL-ADD per clock *)
- SIMD int functional unit, control shared across 16 units (16 x MUL-ADD per clock *)
- SIMD fp64 functional unit, control shared across 8 units (8 x MUL/ADD per clock **) (16 x MUL/ADD per clock *)
- Tensor core unit
- Load/store unit

* one 32-wide SIMD operation every 2 clocks
** one 32-wide SIMD operation every 4 clocks

Stanford CS149, Fall 2021
NVIDIA V100

There are 80 SM cores on the V100:

That’s 163,840 pieces of data being processed concurrently to get maximal latency hiding!
The story so far...

To utilize modern parallel processors efficiently, an application must:

1. Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)

2. Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)

3. Expose more parallel work than processor ALUs to enable interleaving of work to hide memory stalls
Thought experiment

Task: element-wise multiplication of two vectors A and B
Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute A[i] × B[i]
- Store result into C[i]

Is this a good application to run on a modern throughput-oriented parallel processor?
There are 80 SM cores on the V100:

80 SM x 64 fp32 ALUs per SM = 5120 ALUs

Think about supplying all those ALUs with data each clock. 🙀
Understanding latency and bandwidth
The school year is starting... gotta get back to Stanford
San Francisco fog vs. South Bay sun

When it looks like this in SF

It looks like this at Stanford
Why the south bay? Great social distancing opportunities

- Quick plug:
  - Kayvon’s guide to local bay area hikes
Everyone wants to get to back to the South Bay!

Assume only one car in a lane of the highway at once.
When car on highway reaches Stanford, the next car leaves San Francisco.

Distance: ~ 50 km

Latency of driving from San Francisco to Stanford: 0.5 hours
Throughput: 2 cars per hour
Improving throughput

Approach 1: drive faster!
Throughput = 4 cars per hour

Approach 2: build more lanes!
Throughput = 8 cars per hour (2 cars per hour per lane)
Using the highway more efficiently

Cars spaced out by 1 km

Throughput: 100 cars/hr (1 car every 1/100th of hour)

Throughput: 400 cars/hr (4 cars every 1/100th of hour)
Terminology

- Memory bandwidth
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

Bandwidth ~ 4 items/sec

Latency of transferring any one item: ~2 sec
Terminology

- Memory bandwidth
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

Bandwidth: ~ 8 items/sec
Latency of transferring any one item: ~2 sec
Example: doing your laundry

Operation: do your laundry

1. Wash clothes
2. Dry clothes
3. Fold clothes

Latency of completing 1 load of laundry = 2 hours
Increasing laundry throughput
Goal: maximize throughput of many loads of laundry

One approach: duplicate execution resources:
use two washers, two dryers, and call a friend

Latency of completing 2 loads of laundry = 2 hours
Throughput increases by 2x: 1 load/hour
Number of resources increased by 2x: two washers, two dryers
Pipelining

Goal: maximize throughput of doing many loads of laundry

Latency: 1 load takes 2 hours
Throughput: 1 load/hour
Resources: one washer, one dryer
Consider a processor that can do one add per clock (+ can co-issue LD)

Load 64 bytes

Add

Add

Load 64 bytes

Add

Add

Load 64 bytes

Add

Add

Add

Assumptions (8 clocks to transfer data)
Up to 3 outstanding load requests.
Rate of math instructions limited by available bandwidth

Bandwidth-bound execution!

Convince yourself that the instruction throughput is not impacted by memory latency, number of outstanding memory requests, etc.

Only the memory bandwidth!!!

(Note how the memory system is occupied 100% of the time)

= Math instruction
= Load instruction
= Occupancy of memory bus
(size of cache line / memory bus bandwidth)

Stanford CS149, Fall 2021
High bandwidth memories

- Modern GPUs leverage high bandwidth memories located near processor
- Example:
  - V100 uses HBM2
  - 900 GB/s
Thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

- Load input A[i]
- Load input B[i]
- Compute A[i] × B[i]
- Store result into C[i]

Three memory operations (12 bytes) for every MUL

NVIDIA V100 GPU can do 5120 fp32 MULs per clock (@ 1.6 GHz)

Need ~98 TB/sec of bandwidth to keep functional units busy

<1% GPU efficiency... but still 12x faster than eight-core CPU!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus: ~3% efficiency on this computation)
This computation is bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

Overcoming bandwidth limits is often the most important challenge facing software developers targeting modern throughput-optimized systems.
In modern computing, bandwidth is the critical resource

Performant parallel programs will:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread (temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- Favor performing additional arithmetic to storing/reloading values (the math is “free”)

- Main point: programs must access memory infrequently to utilize modern processors efficiently
Another example: an instruction pipeline

Many students have asked how a processor can complete a multiply in a clock. When we say a core does one operation per clock, we are referring to INSTRUCTION THROUGHPUT, NOT LATENCY.

Latency: 1 instruction takes 4 cycles
Throughput: 1 instruction per cycle
(Yes, care must be taken to ensure program correctness when back-to-back instructions are dependent.)

Intel Core i7 pipeline is variable length (it depends on the instruction) ~20 stages
And now today’s topic...
Pay attention!

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:  
\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots 
\]
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 sinx()**

**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;
}
```

**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

```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;
}
```

ISPC code: sinx.ispc

```c
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 \texttt{sinx()} in ISPC

\textbf{C++ code:} \texttt{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;
}
```

\textbf{SPMD programming abstraction:}
\begin{itemize}
    \item Call to ISPC function spawns “gang” of ISPC “program instances”
    \item All instances run ISPC code concurrently
    \item Each instance has its own copy of local variables
    \item Upon return, all instances have completed
\end{itemize}

In this illustration \texttt{programCount} = 8
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

```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;
    }
}
```

"Interleaved" assignment of array elements to program instances

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 eight instances: programCount = 8
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

Sequential execution (C code)
- Call to `ispc_sinx()`
- Execution of `ispc_sinx()` returns

Completion of ISPC program instances
- Resume sequential execution

Sequential execution (C code)
sinx() in ISPC: version 2

“Blocked” assignment of array elements to program 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

In this illustration: gang contains eight instances: programCount = 8
Schedule: interleaved assignment

"Gang" of ISPC program instances

Gang contains four instances: \( \text{programCount} = 8 \)

<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>
<th>Instance 4 (programIndex = 4)</th>
<th>Instance 5 (programIndex = 5)</th>
<th>Instance 6 (programIndex = 6)</th>
<th>Instance 7 (programIndex = 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>i=1</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>i=2</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>i=3</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
</tbody>
</table>

A single "packed vector load" instruction (vmovaps *) efficiently implements:

```c
float value = x[idx];
```

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

```c
... // assumes N % programCount = 0
for (uniform int i=0; i<N; i+=programCount) {
    int idx = i + programIndex;
    float value = x[idx];
}
...```

* see `_mm256_load_ps()` intrinsic function
Schedule: blocked assignment

"Gang" of ISPC program instances

Gang contains four instances: programCount = 8

<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>
<th>Instance 4 (programIndex = 4)</th>
<th>Instance 5 (programIndex = 5)</th>
<th>Instance 6 (programIndex = 6)</th>
<th>Instance 7 (programIndex = 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>i=0</td>
<td>i=1</td>
<td>i=2</td>
<td>i=3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 8 16 24 32 40 48 56</td>
<td>1 9 17 25 33 41 49 57</td>
<td>2 10 18 26 34 42 50 58</td>
<td>3 11 19 27 35 43 51 59</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

float value = x[idx];

For all program instances now touches eight non-contiguous values in memory. Need “gather” instruction \texttt{(vgatherdps *)} to implement (gather is a more complex, and more costly SIMD instruction…)

* see \_\_mm256\_i32gather\_ps() intrinsic function
Raising level of abstraction with foreach

**C++ code:** main.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
```
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 entire 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, ARM NEON) 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)
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
Review: 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 locations in a shared address space (shared variables)

Thread 1:
int x = 0;
spawn_thread(foo, &x);

// write to address holding
// contents of variable x
x = 1;

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

(Pseudocode provided in a fake C-like language for brevity.)
A common metaphor:
A shared address space is like a bulletin board

(Everyone can read/write)
Coordinating access to shared variables with synchronization

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 variable \( x \) from a location in memory into register \( r1 \)
    (this stores a copy of the value in memory in the register)
  - Add the contents of register \( r2 \) to register \( r1 \)
  - Store the value of register \( r1 \) into the address storing the program variable \( x \)

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 \leftarrow x )</td>
<td>( r1 \leftarrow x ) T1 reads value 0</td>
</tr>
<tr>
<td>( r1 \leftarrow r1 + r2 )</td>
<td>( r1 \leftarrow r1 + r2 ) T2 reads value 0</td>
</tr>
<tr>
<td>( X \leftarrow r1 )</td>
<td>( X \leftarrow r1 )</td>
</tr>
</tbody>
</table>

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

- Lock/unlock mutex around a critical section
  ```
  mylock.lock();
  // critical section
  mylock.unlock();
  ```

- 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
  - 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!
Hardware 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 hardware architecture

Any processor can directly reference any memory location

Example: Intel Core i7 processor (Kaby Lake)
**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 ~ 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 core processor
- 72 cores, arranged as 6x6 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. Bandwidth from any one location may also be different to different CPU cores.*

In practice, you’ll find NUMA behavior on a single-socket system as well (recall: different cache slices are a different distance from each core).
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 implementations requires reasoning about locality for performance optimization
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)
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

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

Something you’ve seen early in the lecture…

* 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>
- Scala lists: List[T]
- In a functional language (like Haskell): seq T

- Program can only access elements of sequence through sequence operators:
  - map, reduce, scan, shift, etc.
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 **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.
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];
    }
}
Data parallelism in ISPC

// main C++ code:
const int N = 1024;
float* x = new float[N/2];
float* y = new float[N];

// initialize N/2 elements of x here
absolute_repeat(N/2, x, y);

// 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);

// 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];
    }
}

Think of loop body as a function

The output of this program is undefined!

Possible for multiple iterations of the loop body to write to same memory location

Data-parallel model (foreach) provides no specification of order in which iterations occur
Compute the sum of all array elements in parallel

```ispc
export uniform float sumall1(
    uniform int N,
    uniform float* x)
{
    uniform float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }
    return sum;
}
```

The code above will result in a compile-time type error because `sum` is of type `uniform float` (one copy of variable for all program instances), while `x[i]` is not a uniform expression (different value for each program instance).

```ispc
Correct ISPC solution
export uniform float sumall2(
    uniform int N,
    uniform float* x)
{
    uniform float sum;
    float partial = 0.0f;
    foreach (i = 0 ... N)
    {
        partial += x[i];
    }
    // from ISPC math library
    sum = reduce_add(partial);
    return sum;
}
```
ISPC discussion: sum “reduction”

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
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.