Lecture 4:

The SPMD Programming Model

(a good example of abstraction vs. implementation)

Parallel Computer Architecture and Programming

CMU / 清华大学, Summer 2017

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:

(that happens to be very useful for assignment 1)

Programming with ISPC

ISPC

- Intel SPMD Program Compiler (ISPC)
- SPMD = “single program multiple data”

- http://ispc.github.com/

Recall: example program from last class

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$
for each element of an array of N floating-point numbers

```
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;
   }
}
```

sin(x) in ISPC

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

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);
```

SPMD programming abstraction:

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

All instances run ISPC code concurrently

Upon return, all instances have completed

ISPC code: `sinx.ispc`

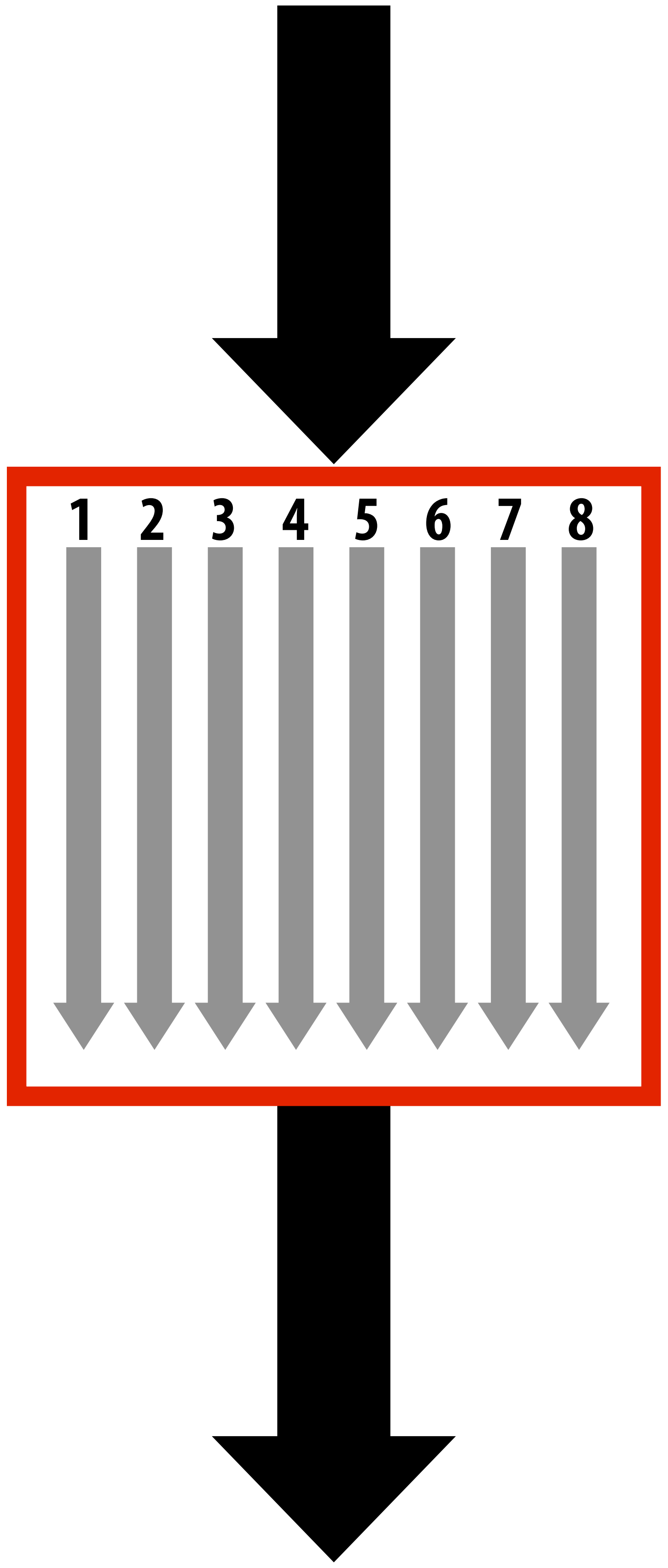
```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assume N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      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;
   }
}
```

sin(x) in ISPC

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

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);
```



Sequential execution (C code)

Call to `sinx()`
Begin executing `programCount` **instances of** `sinx()` **(ISPC code)**

`sinx()` **returns.**
Completion of ISPC program instances.
Resume sequential execution

Sequential execution (C code)

SPMD programming abstraction:

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

All instances run ISPC code concurrently

Upon return, all instances have completed

In this illustration `programCount` = 8

sin(x) in ISPC

“Interleaved” assignment of array elements to program instances

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 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;
   }
}
```

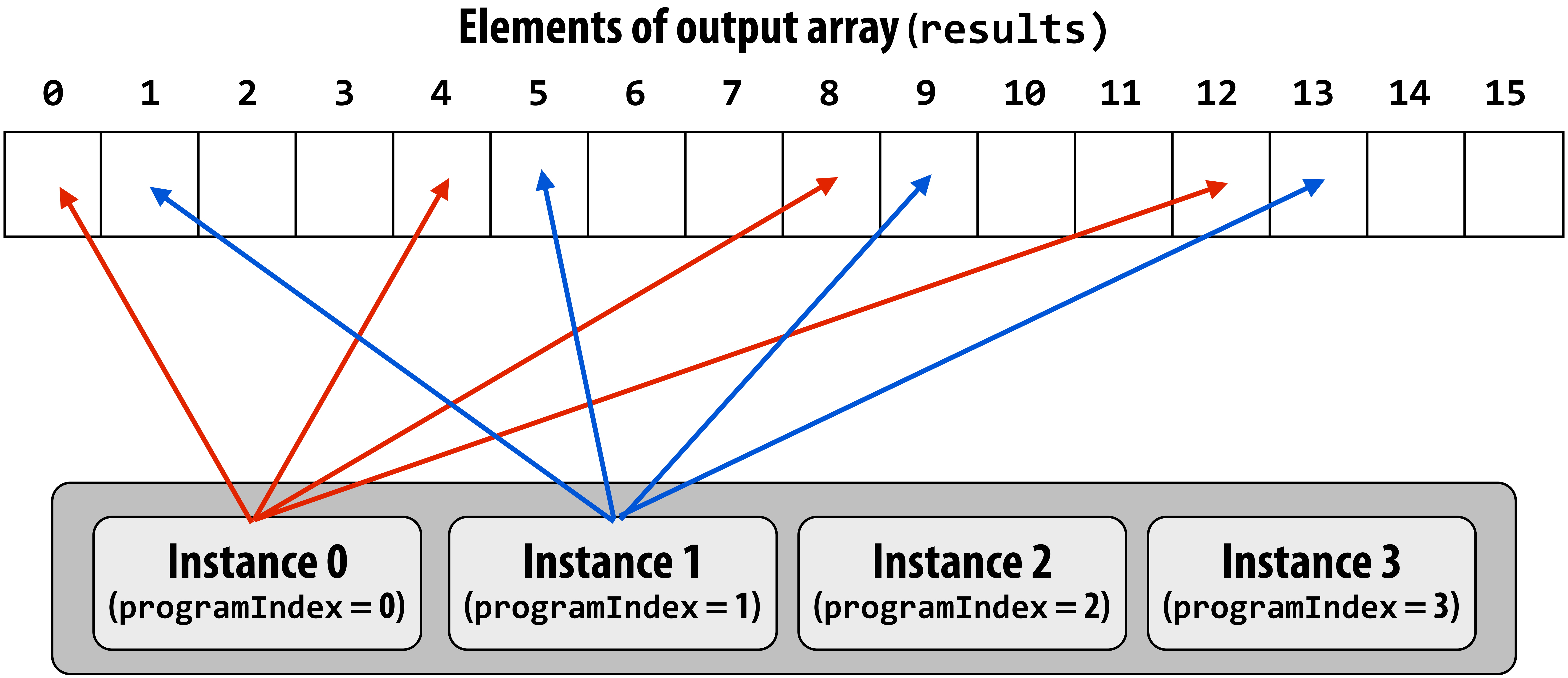
ISPC Keywords:

`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



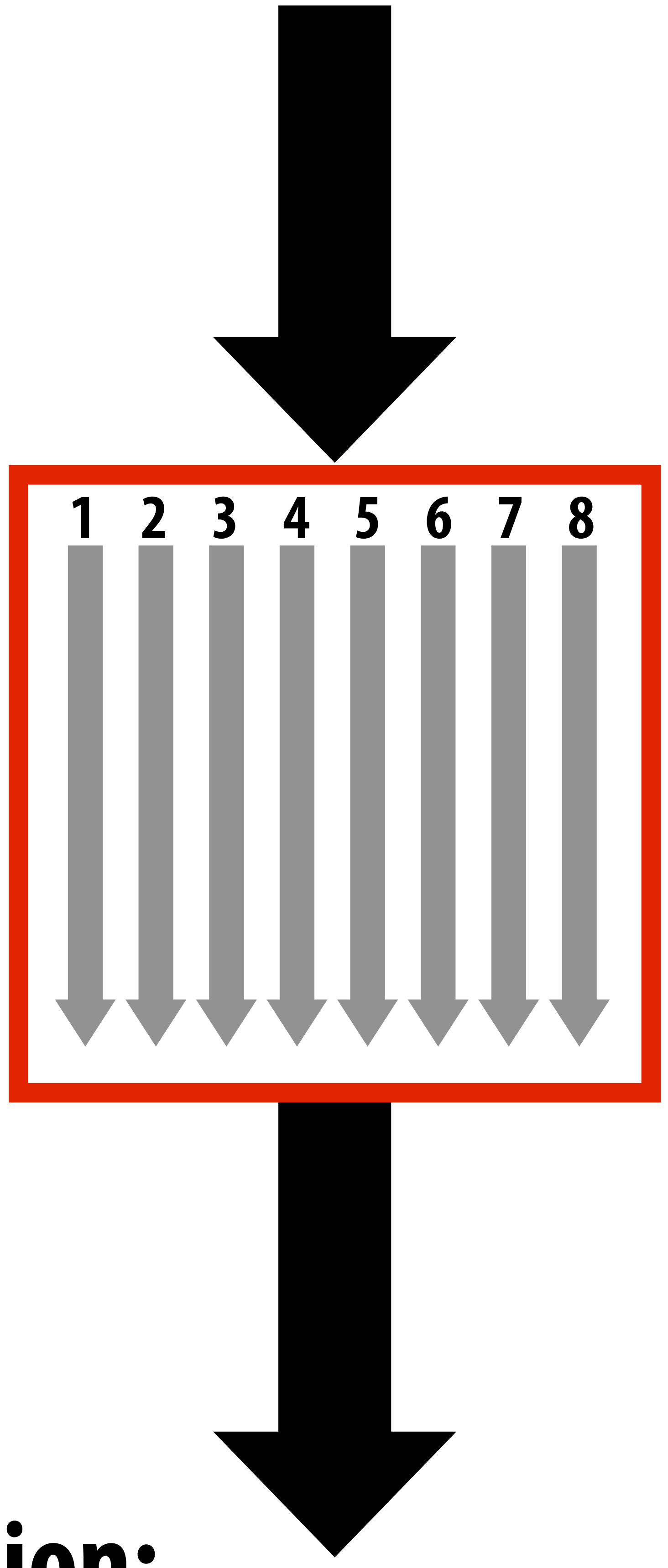
"Gang" of ISPC program instances

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

ISPC <u>implements</u> the gang abstraction using SIMD instructions

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);
```



Sequential execution (C code)

Call to `sinx()`
Begin executing `programCount` instances of `sinx()` (ISPC code)

`sinx()` returns.
Completion of ISPC program instances.
Resume sequential execution

Sequential execution
(C code)

SPMD programming abstraction:

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

All instances run ISPC code concurrently

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 binary (.o) with SIMD instructions

C++ code links against object file as usual

sin(x) in ISPC: version 2

"Blocked" assignment of elements to instances

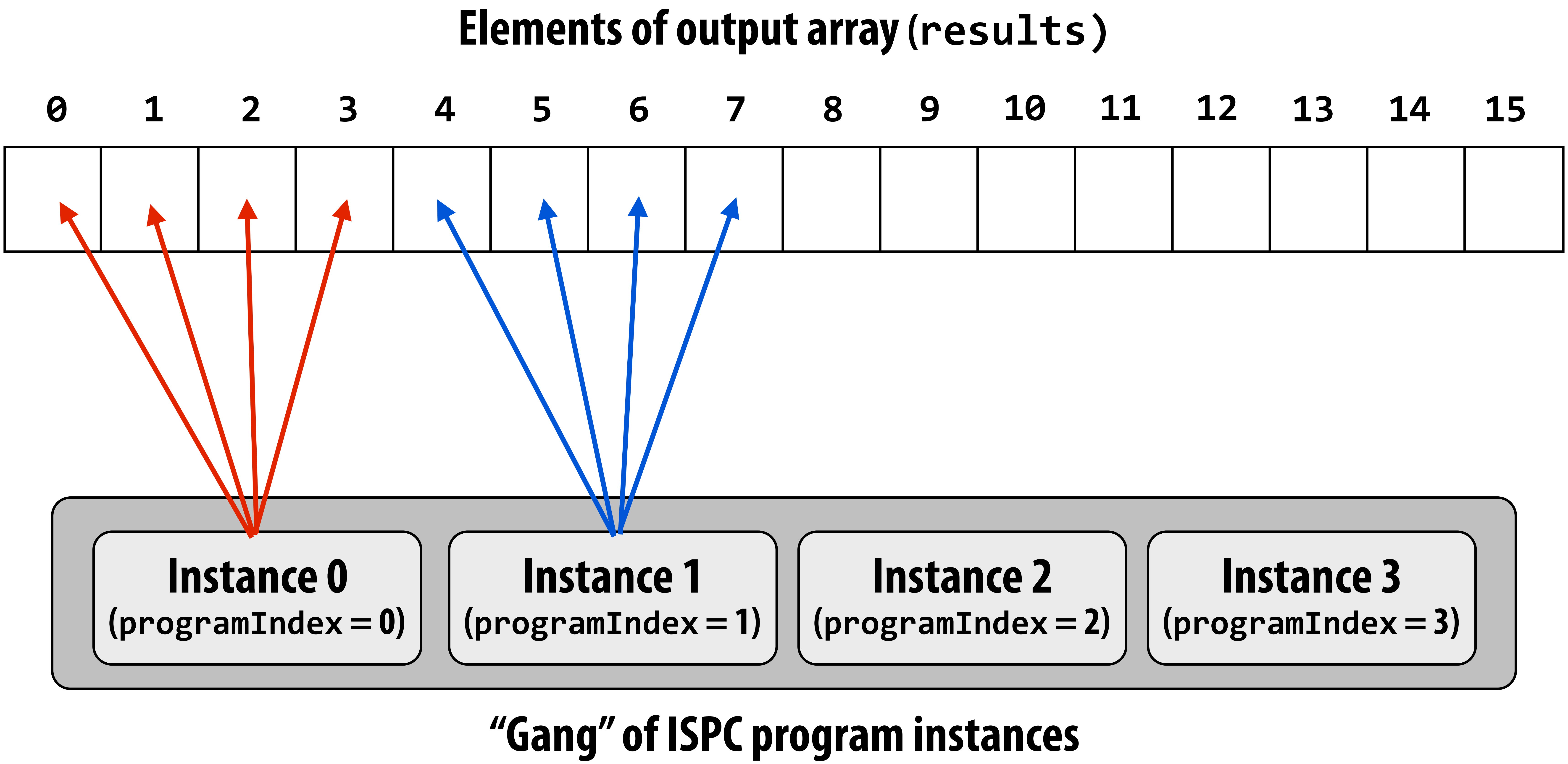
C++ code: `main.cpp`

```
#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 sinx(
   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`)

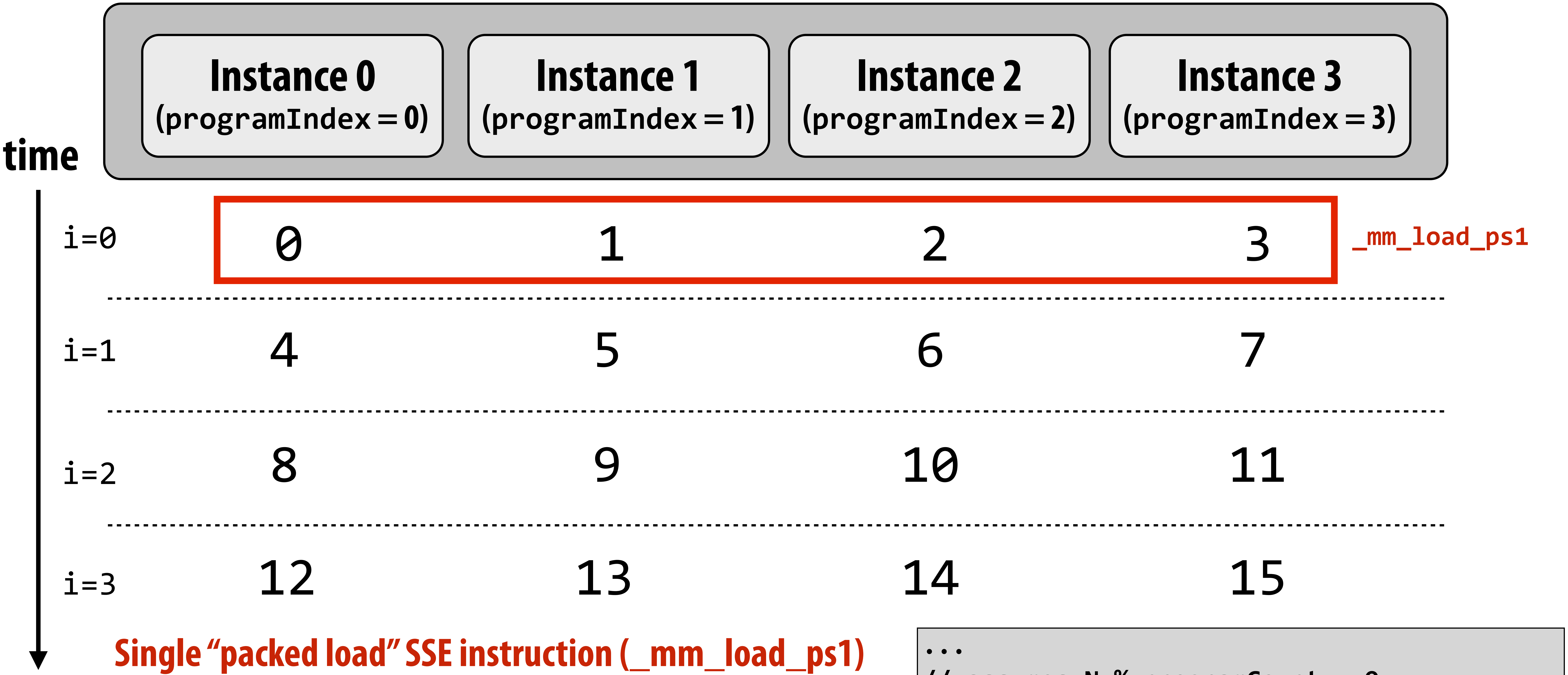
"Gang" of ISPC program instances

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

Schedule: interleaved assignment

“Gang” of ISPC program instances

Gang contains four instances: `programCount` = 4



	Instance 0 (`programIndex` = 0)	Instance 1 (`programIndex` = 1)	Instance 2 (`programIndex` = 2)	Instance 3 (`programIndex` = 3)
`i=0`	0	1	2	3
`i=1`	4	5	6	7
`i=2`	8	9	10	11
`i=3`	12	13	14	15

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

`float value = x[idx];`

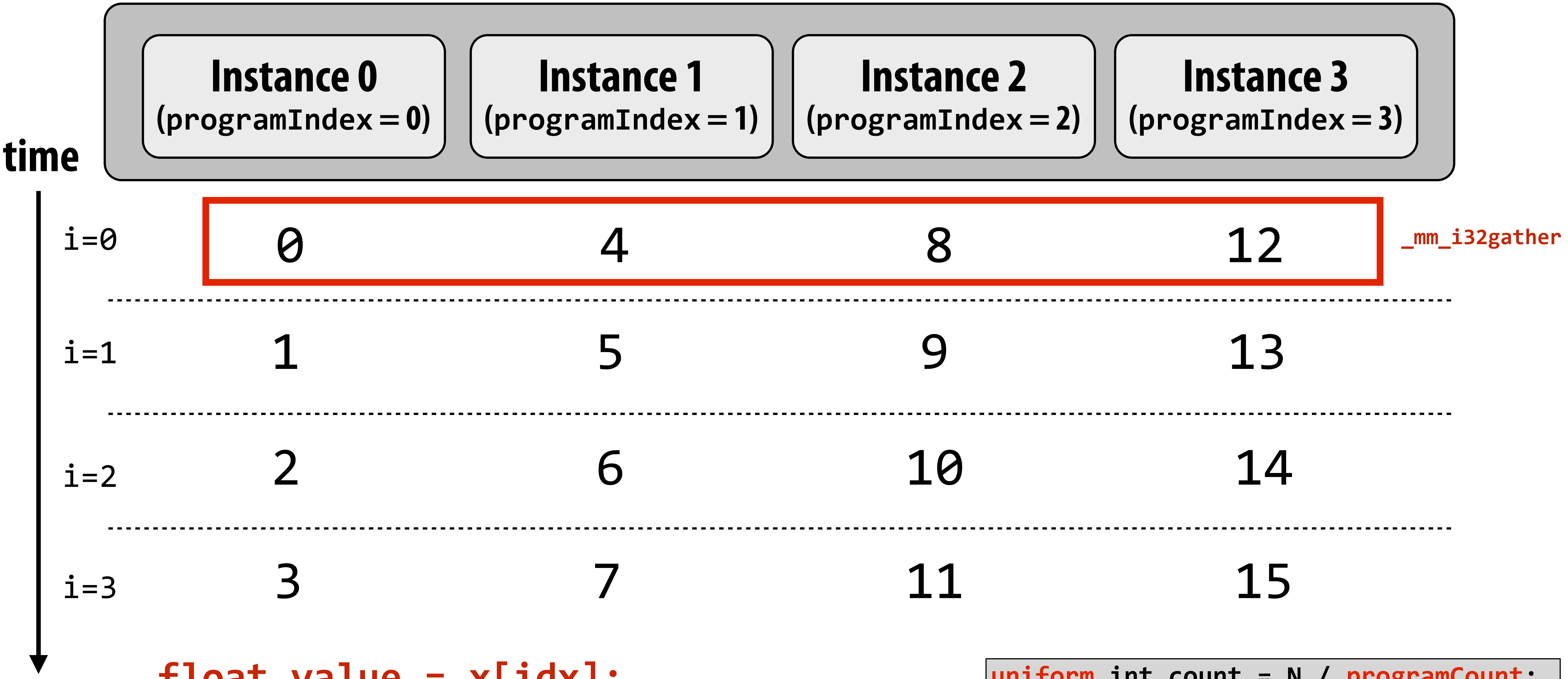
for all program instances, since the four values are contiguous in memory

```
...
// 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



time	Instance 0 (`programIndex` = 0)	Instance 1 (`programIndex` = 1)	Instance 2 (`programIndex` = 2)	Instance 3 (`programIndex` = 3)	
`i=0`	0	4	8	12	`_mm_i32gather`
`i=1`	1	5	9	13	
`i=2`	2	6	10	14	
`i=3`	3	7	11	15	

`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)

```
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];
...
```

Raising level of abstraction with foreach

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

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);
```

`foreach`: key ISPC language construct

- `foreach` declares parallel loop iterations
  - Programmer says: these are the iterations the instances in a gang cooperatively must perform
- ISPC implementation assigns iterations to program instances in gang
  - Current ISPC implementation will perform a static interleaved assignment (but the abstraction permits a different assignment)

ISPC code: `sinx.ispc`

```
export void 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;
   }
}
```

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 <u>abstraction</u>**
  - **Program is written in terms of this abstraction**

- **Single instruction, multiple data (SIMD) <u>implementation</u>**
  - **ISPC compiler emits vector instructions (SSE4 or AVX) 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.)**

- **Semantics of ISPC can be tricky**
  - **SPMD abstraction + uniform values**
    (allows implementation details to peak through abstraction a bit)

ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

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

   return sum;
}
```

```
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;
}
```

Correct ISPC solution

`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. *

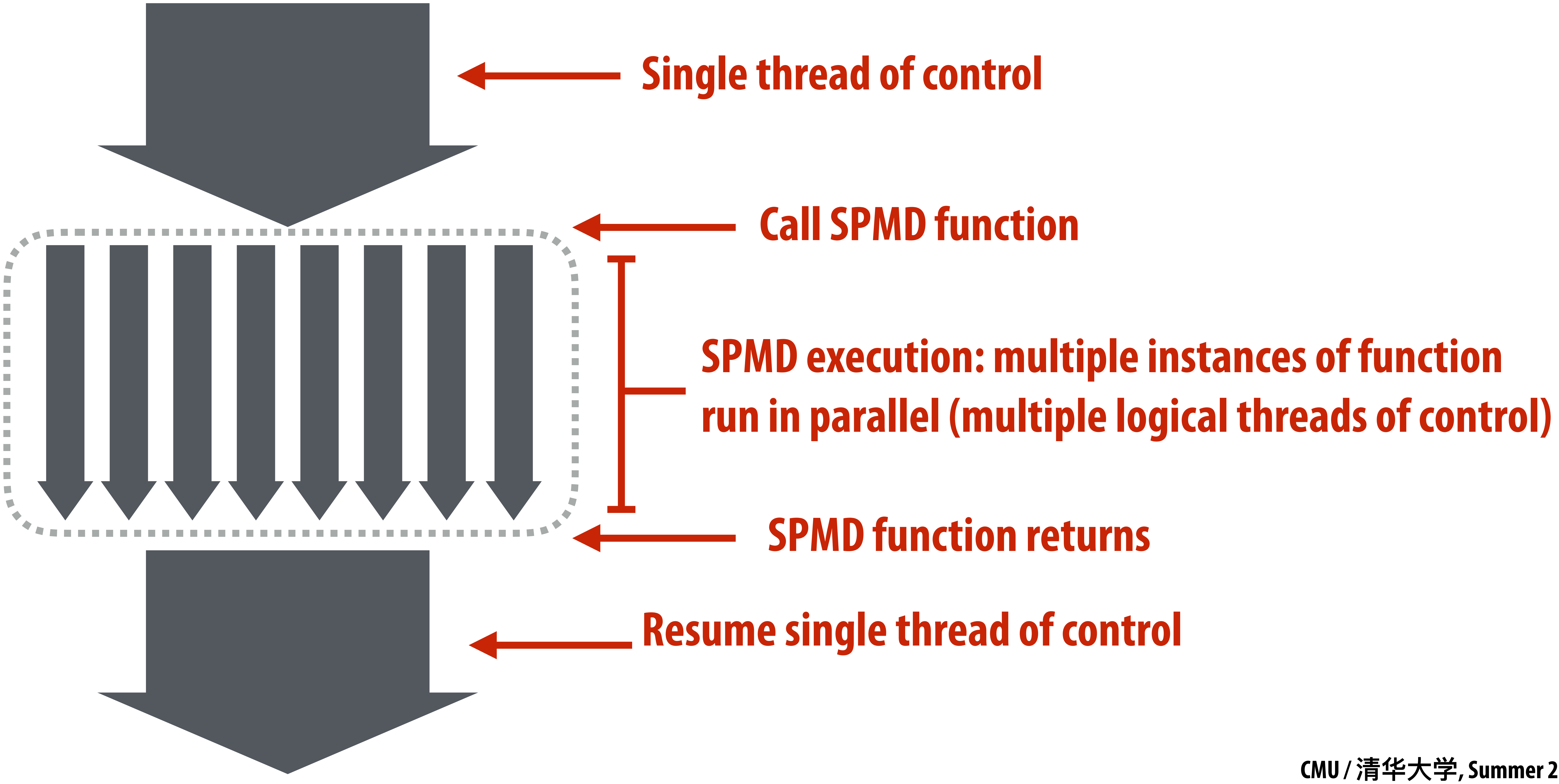
```
export uniform float sumall2(
   uniform int N,
   uniform float* x)
{
   uniform float sum;
   float partial = 0.0f;
   foreach (i = 0 ... N)
   {
      partial += x[i];
   }

   // from ISPC math library
   sum = reduce_add(partial);

   return sum;
}
```

```
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



Single thread of control

Call SPMD function

SPMD execution: multiple instances of function run in parallel (multiple logical threads of control)

SPMD function returns

Resume single thread of control

ISPC tasks

- The ISPC gang abstraction is implemented by SIMD instructions on one core.

- So... all the code I've shown you in the previous slides would have executed on only one core of a CPU

- ISPC contains another abstraction: a "task" that is used to achieve multi-core execution. I'll let you read about that when you do assignment 1.