Lecture 3:

Multi-Core Architecture, Part II (latency/bandwidth issues) + Parallel Programming Abstractions

Parallel Computing

Stanford CS149, Fall 2025

Reviewing last time…

- Three ideas in throughput computing hardware
    - Multi-core execution
    - SIMD execution
    - Hardware multi-threading (we actually didn't get to it… so let's do it now)

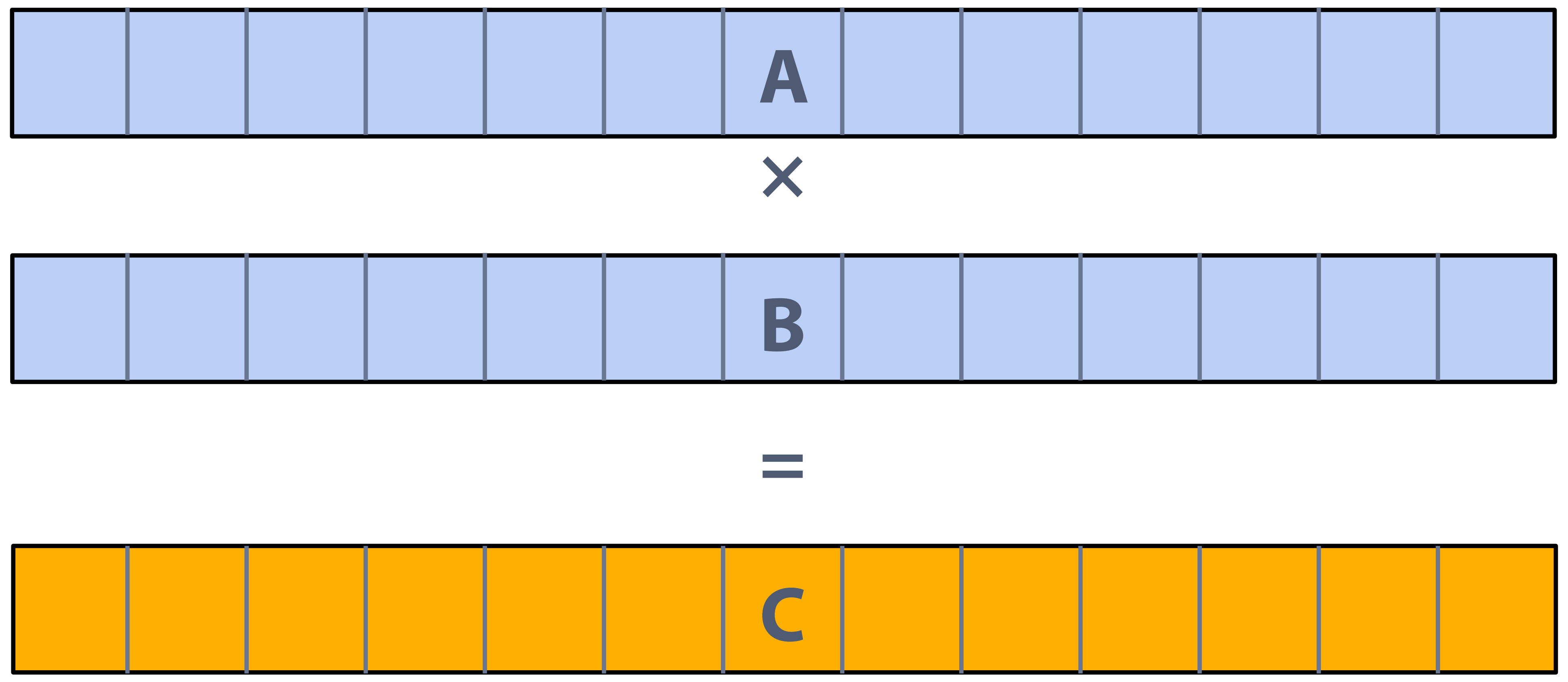
- [Will review by going over slides from the end of lecture 2]

Here's a 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]



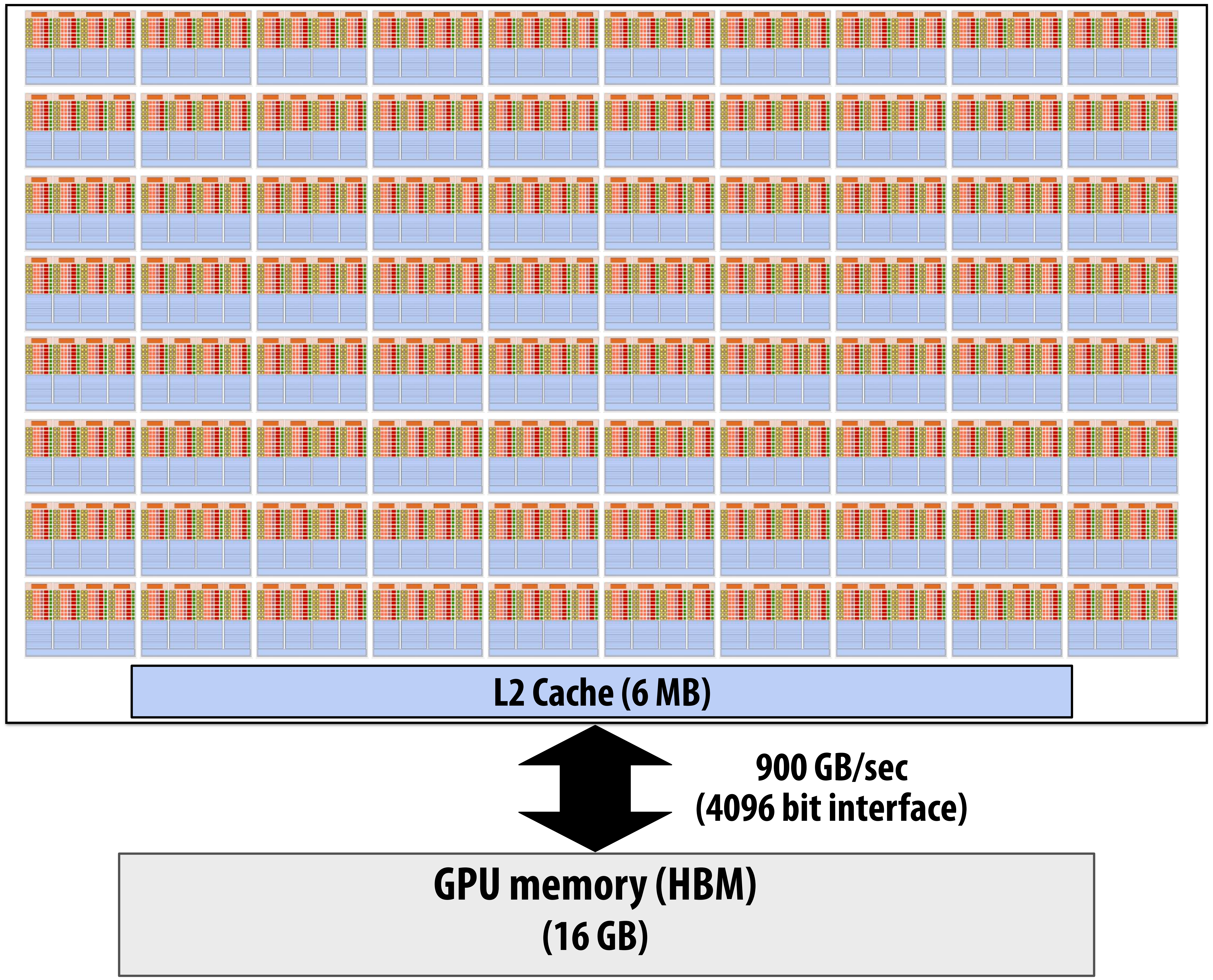
Is this a good application to run on a modern throughput-oriented parallel processor?

NVIDIA V100

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

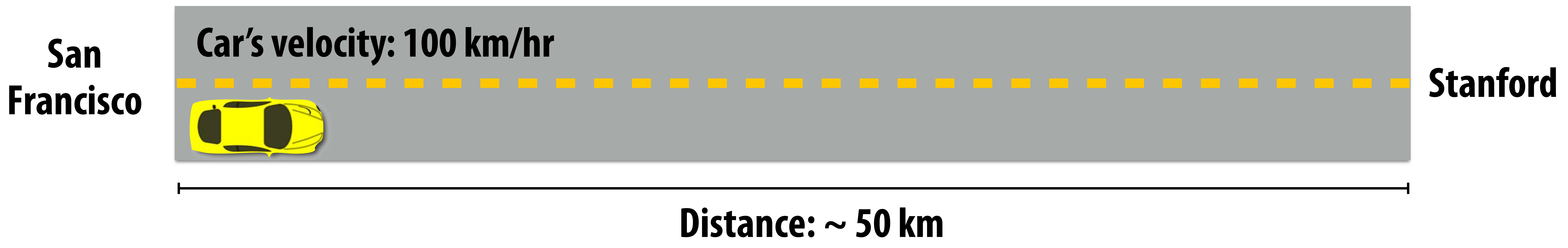
The school year is starting… gotta get back to Stanford

San Francisco fog vs. South Bay sun

Everyone wants to get to back to the South Bay!

Assume only one car in a lane of the highway at once.

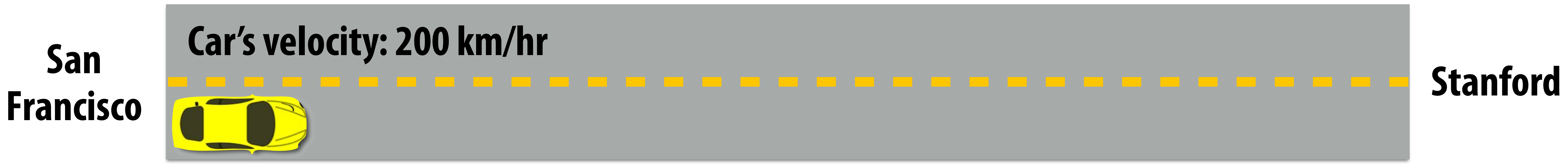
When car on highway reaches Stanford, the next car leaves San Francisco.



Latency of driving from San Francisco to Stanford: 0.5 hours

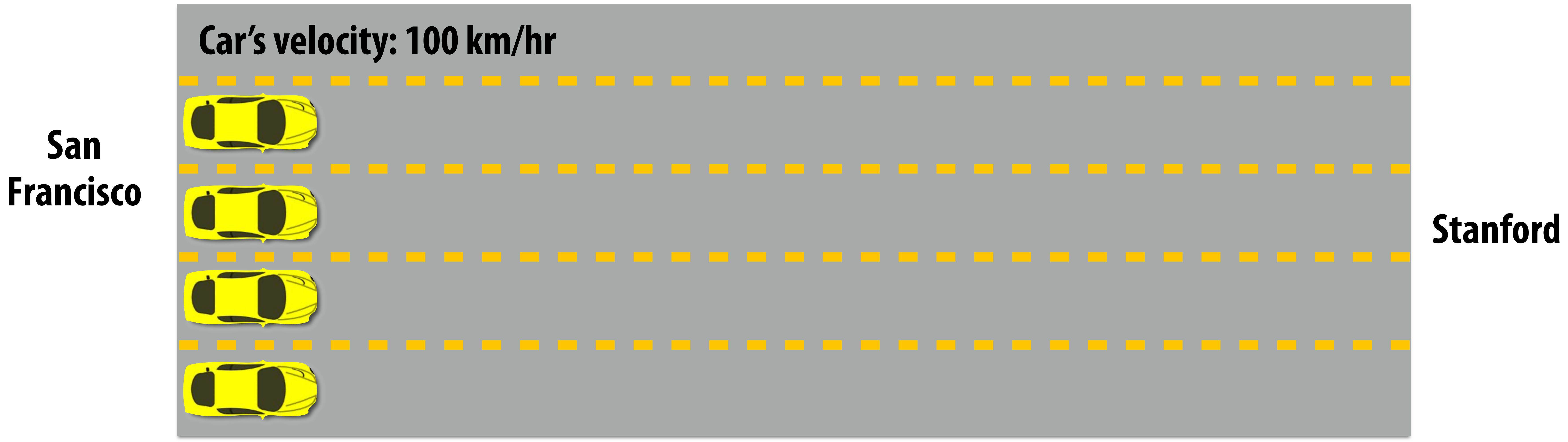
Throughput: 2 cars per hour

Improving throughput



Approach 1: drive faster!

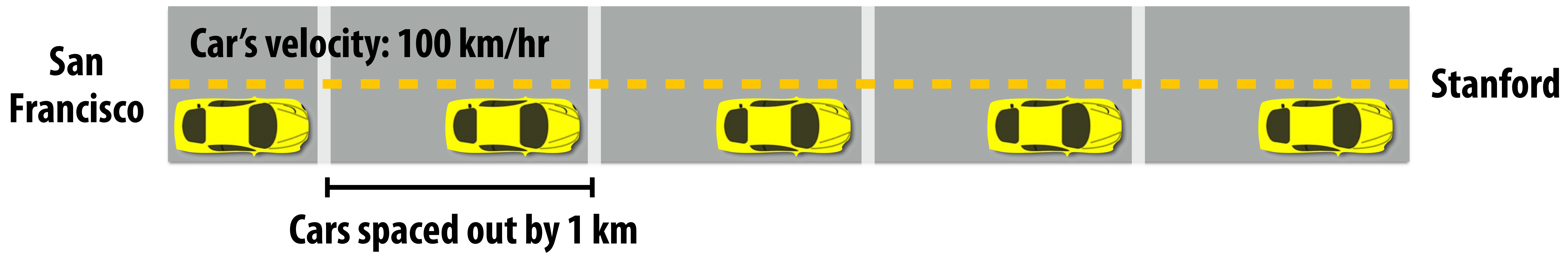
Throughput = 4 cars per hour



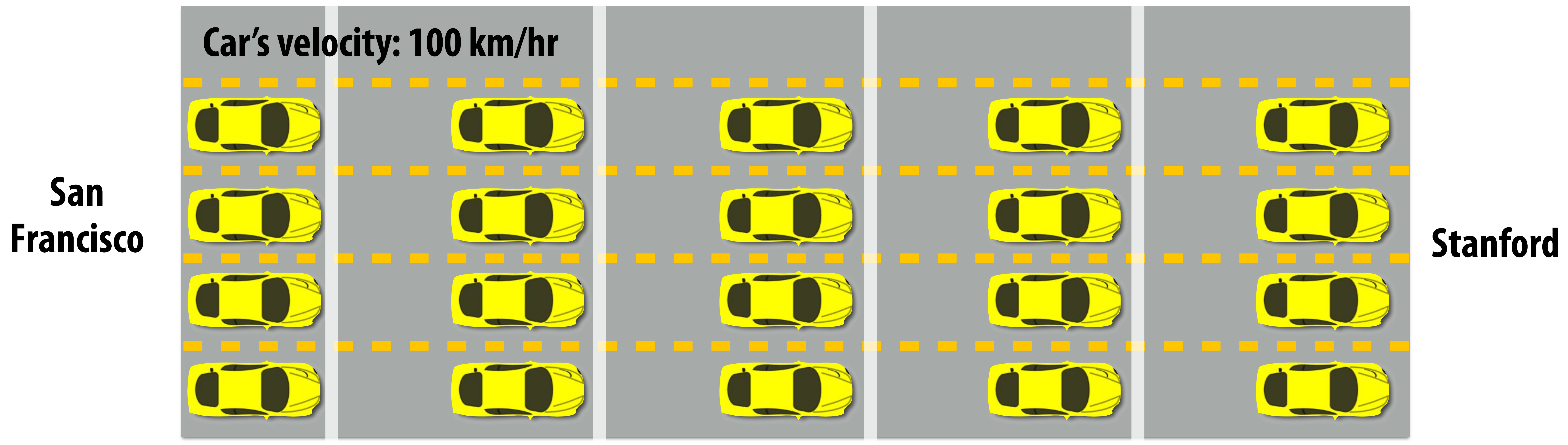
Approach 2: build more lanes!

Throughput = 8 cars per hour (2 cars per hour per lane)

Using the highway more efficiently



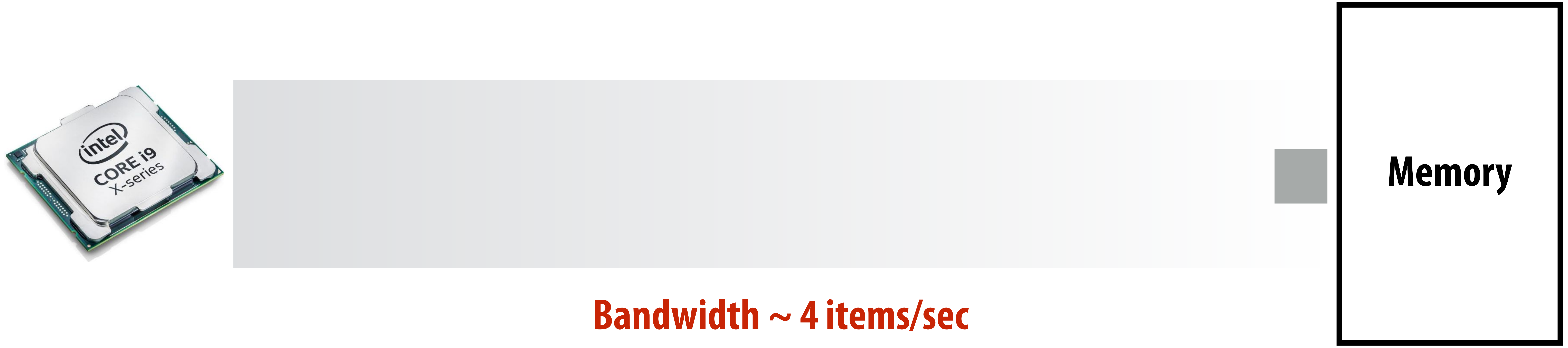
Throughput: 100 cars/hr (1 car every 1/100th of hour)



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

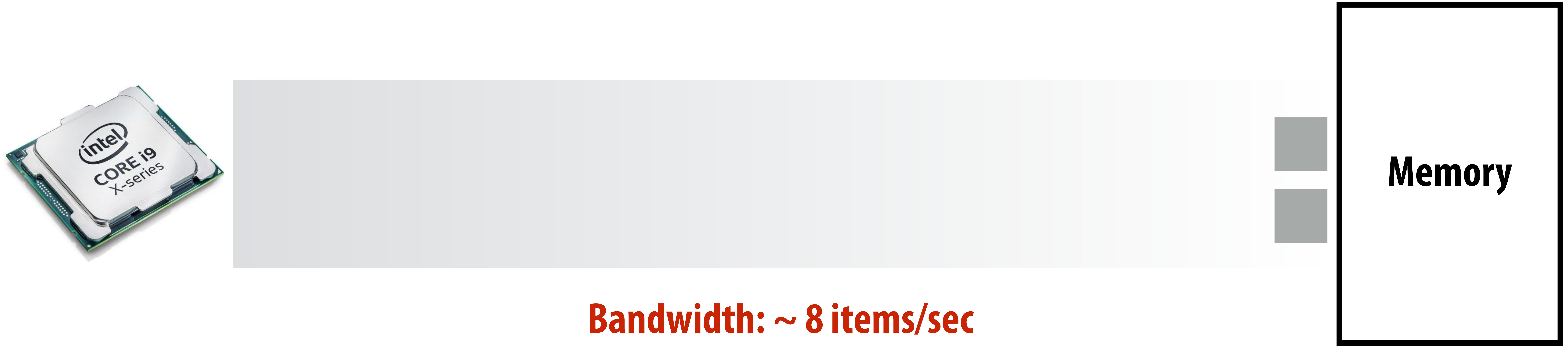


Bandwidth ~ 4 items/sec

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

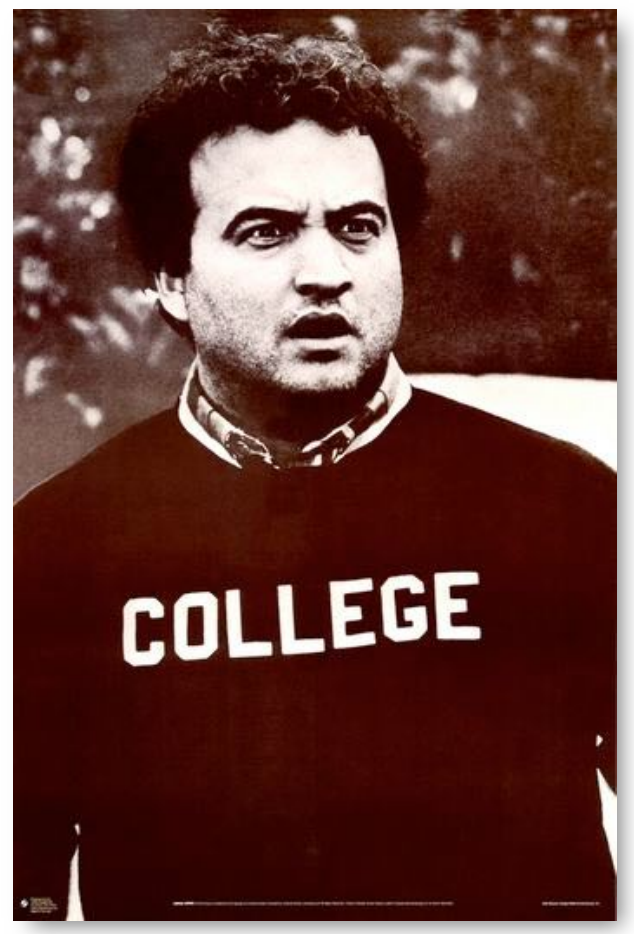
Example: doing your laundry

Operation: do your laundry

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

Washer
45 min

Dryer
60 min



College Student
15 min

Latency of completing 1 load of laundry = 2 hours

Increasing laundry throughput

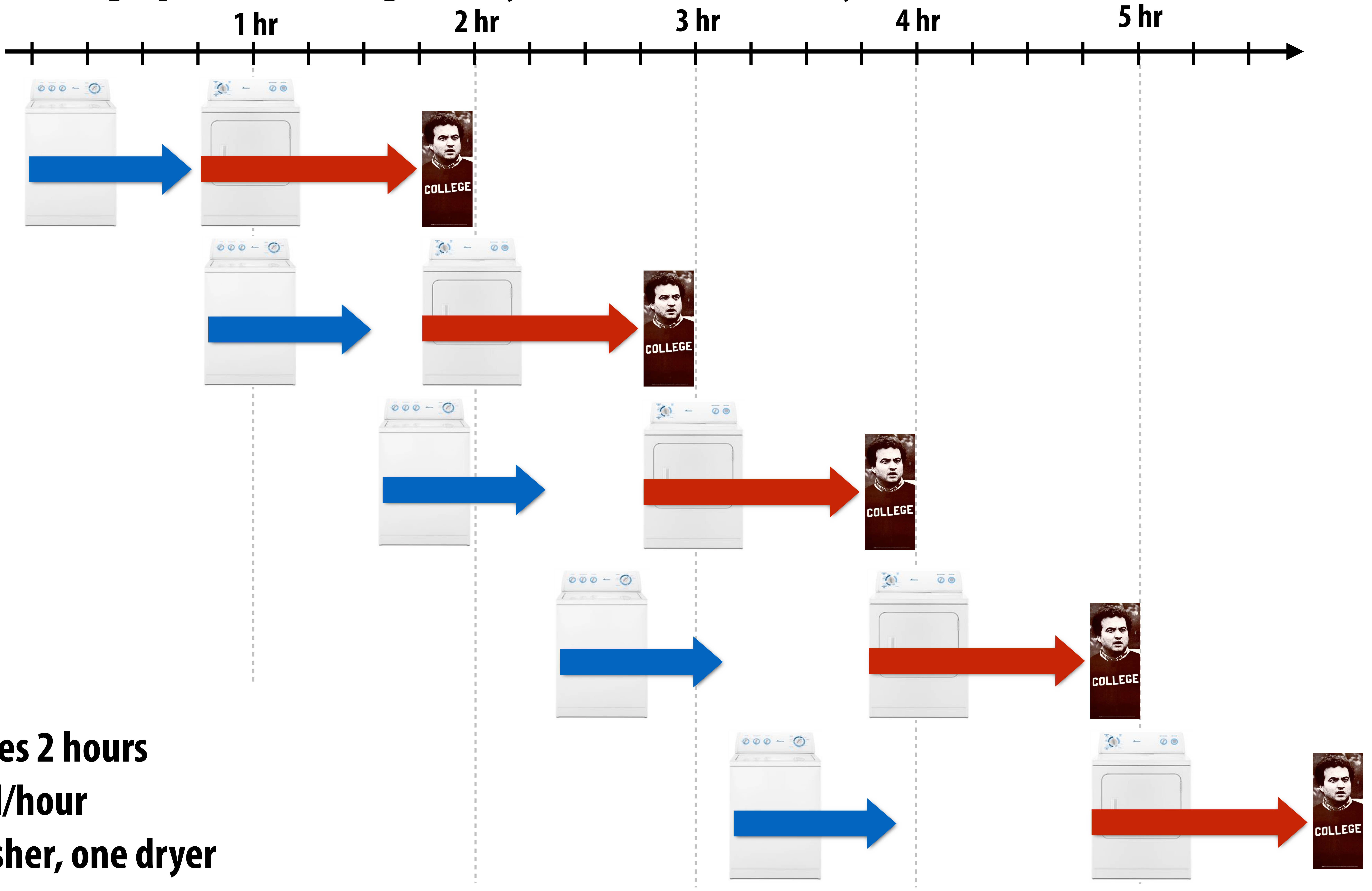
Goal: maximize throughput of many loads of laundry

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 laundry

Goal: maximize throughput of doing many loads of laundry



Latency: 1 load takes 2 hours

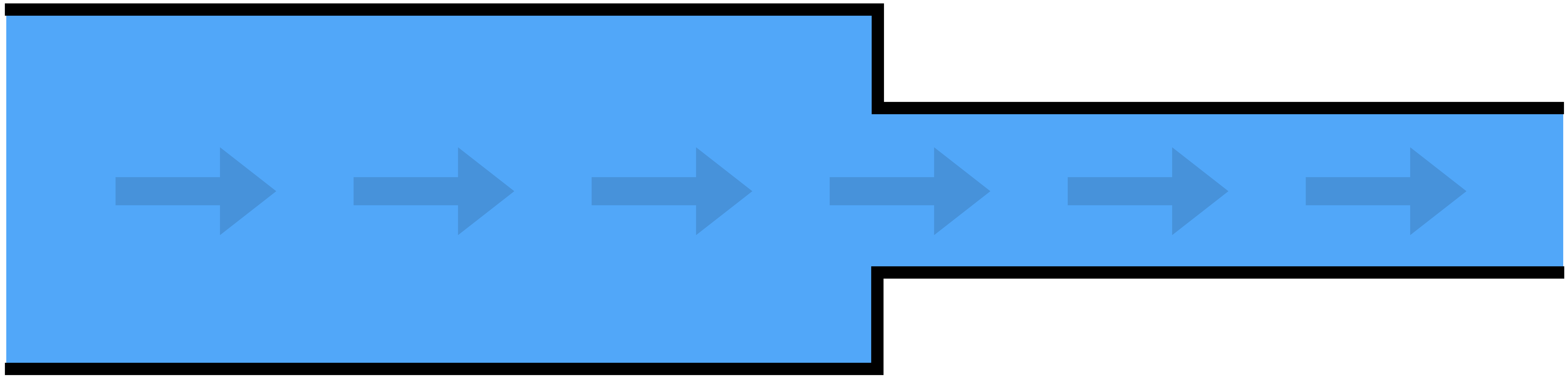
Throughput: 1 load/hour

Resources: one washer, one dryer

Another example: two connected pipes

Pipe 1: max flow 100 liters/sec

Pipe 2: max flow 50 liters/sec



If you connect the pipes, what is the maximum flow you can push through the system?

50 liters/sec

Applying this concept to a computer...

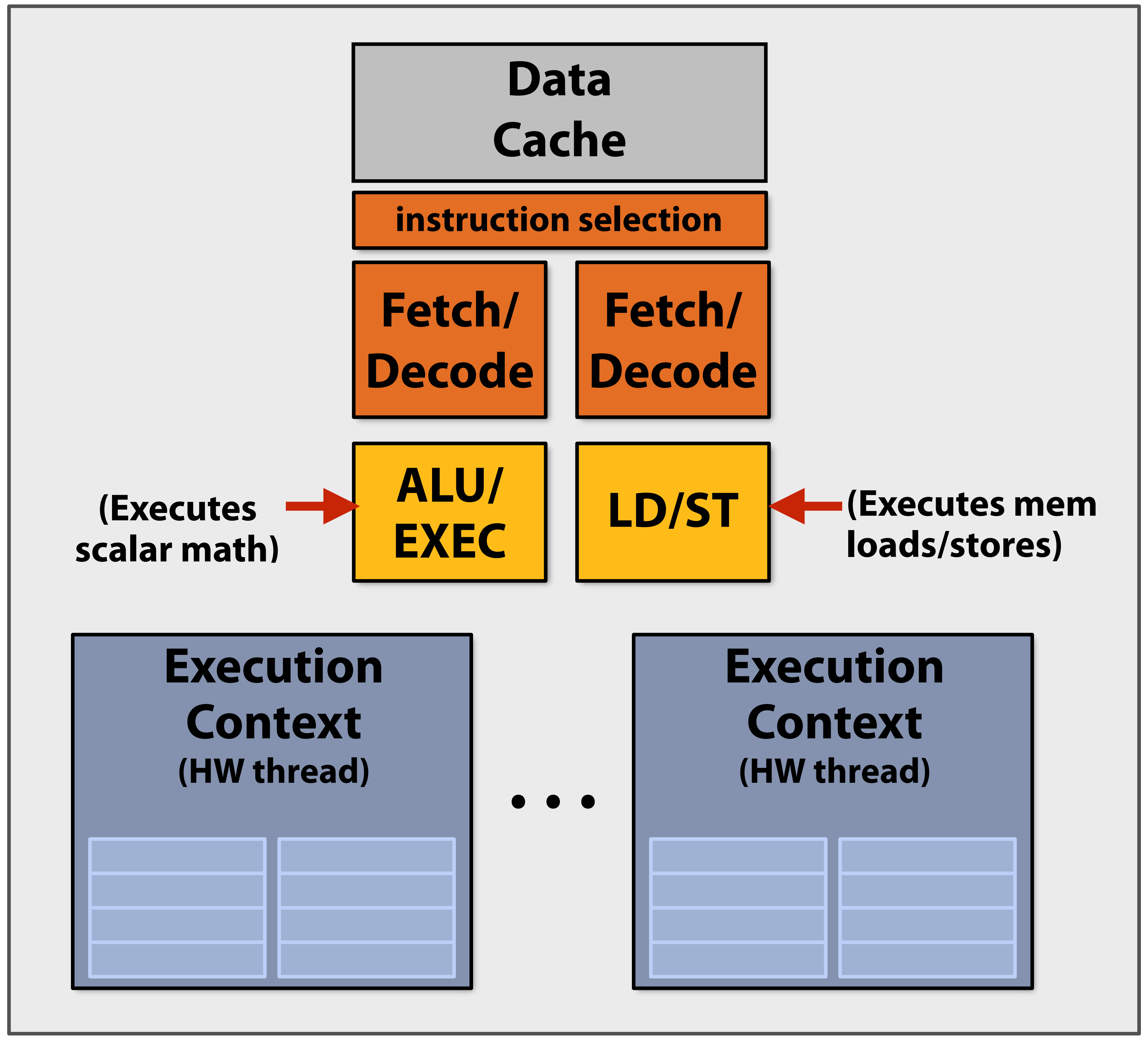
Consider a program that runs threads that repeat the following sequence of three dependent instructions

```
1. X = load 64 bytes
2. Y = add x + x
3. Z = add x + y
```

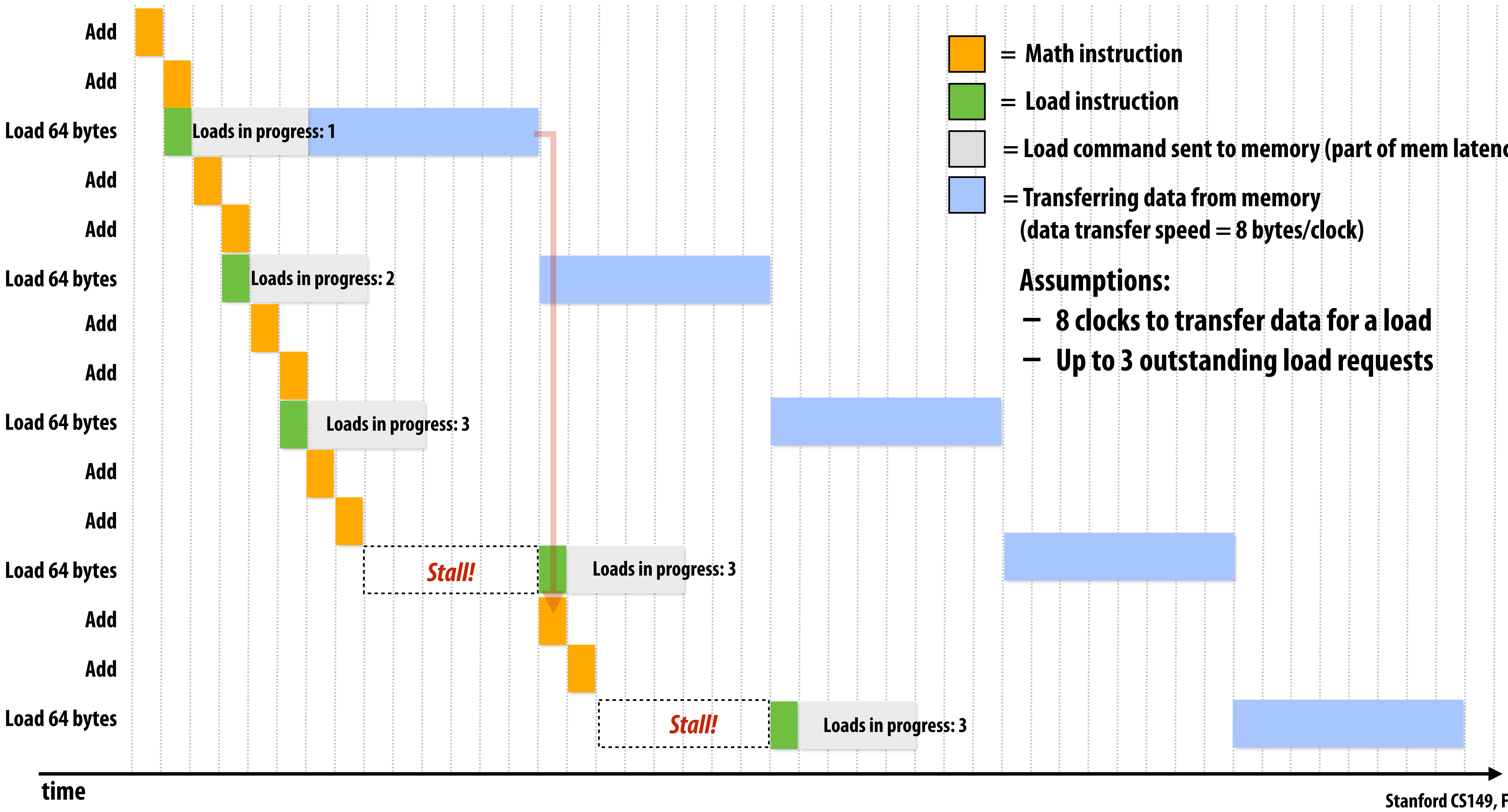
Let's say we're running this sequence on many threads of a multi-threaded* core that:

- Executes one math operation per clock
- Can issue load instructions in parallel with math
- Receives 8 bytes/clock from memory

(* Assume there are plenty of hardware threads to hide memory latency)



Processor that can do one add per clock (+ co-issues LDs)



🟧 = Math instruction

🟩 = Load instruction

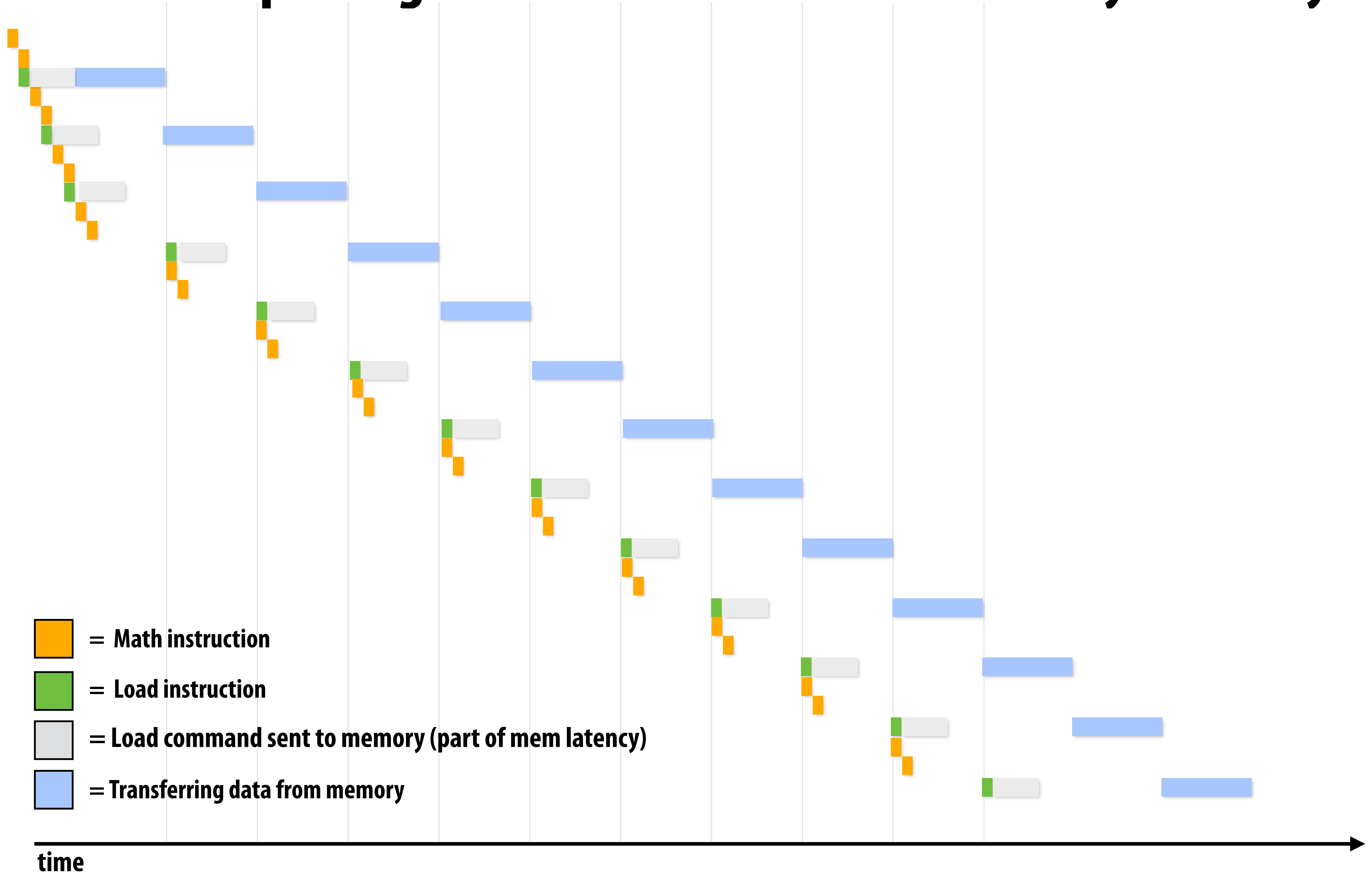
⬜ = Load command sent to memory (part of mem latency)

🟦 = Transferring data from memory (data transfer speed = 8 bytes/clock)

Assumptions:

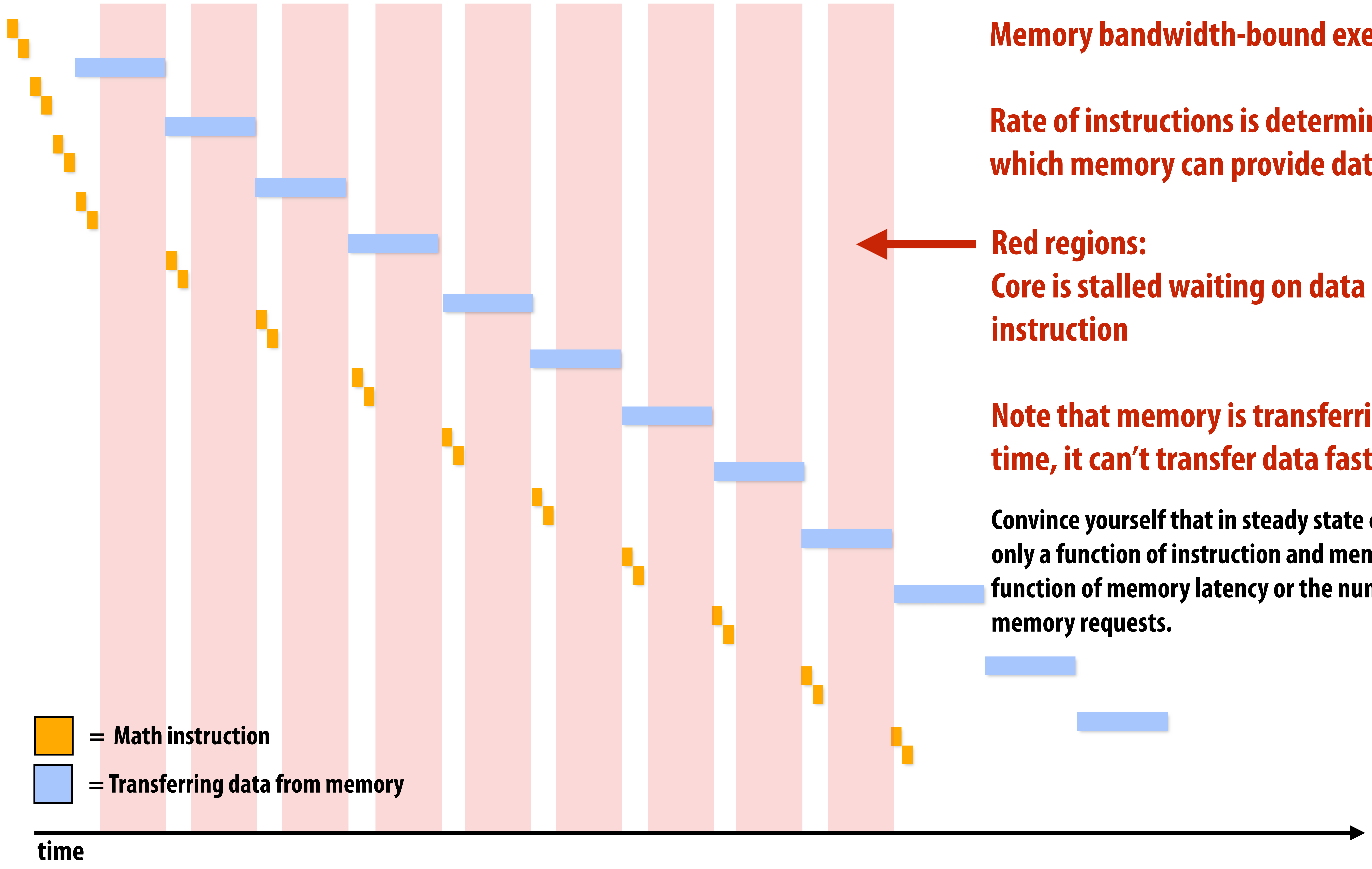
- 8 clocks to transfer data for a load
- Up to 3 outstanding load requests

Rate of completing math instructions is limited by memory bandwidth



= Math instruction
= Load instruction
= Load command sent to memory (part of mem latency)
= Transferring data from memory
time

Rate of completing math instructions is limited by memory bandwidth



Memory bandwidth-bound execution!

Rate of instructions is determined by the rate at which memory can provide data.

Red regions:
Core is stalled waiting on data for next instruction

Note that memory is transferring data 100% of time, it can't transfer data faster.

Convince yourself that in steady state core underutilization is only a function of instruction and memory throughput, not a function of memory latency or the number of outstanding memory requests.

High bandwidth memories

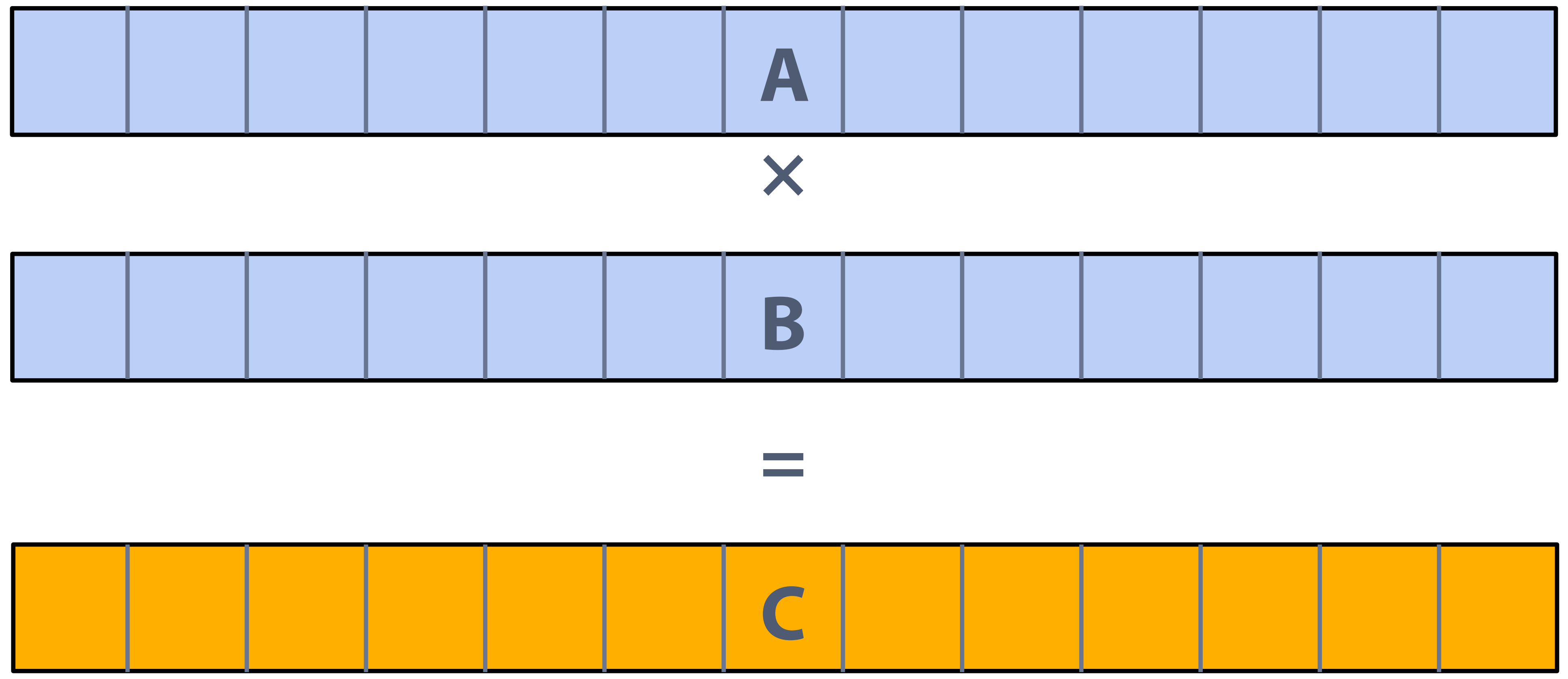
- Modern GPUs leverage high bandwidth memories located near processor
- Example:
  - V100 uses HBM2
  - 900 GB/s

Back to our 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 must faster than an 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.

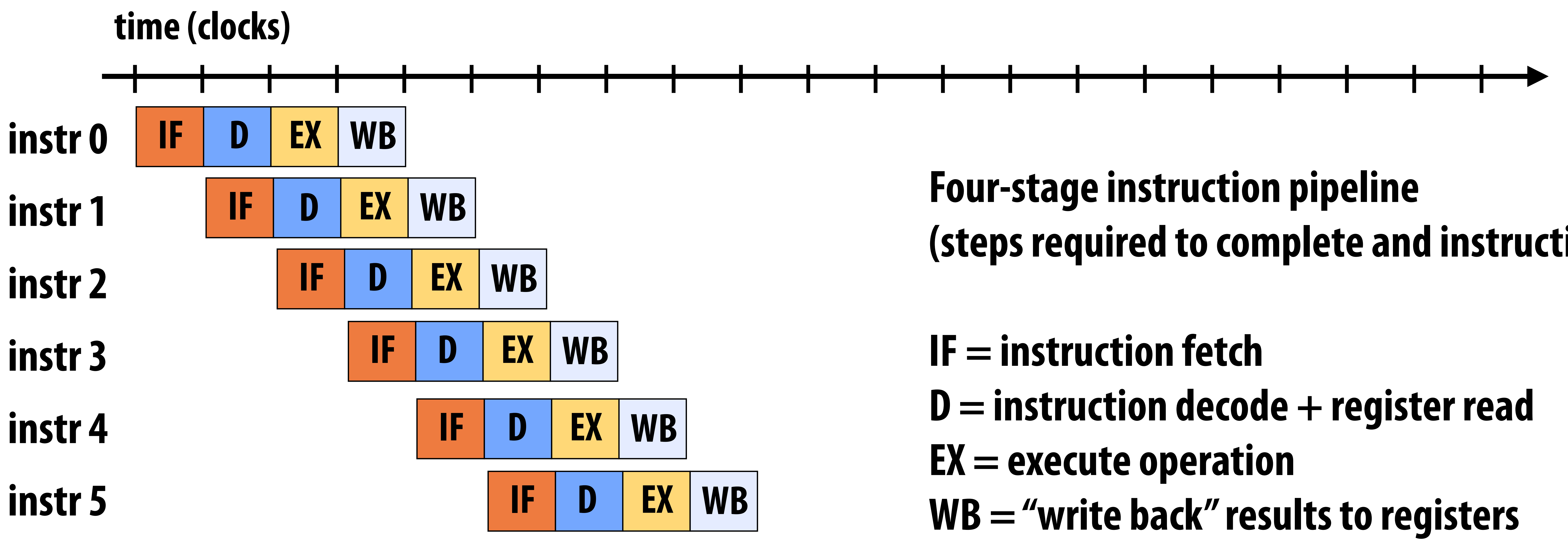
In modern computing, bandwidth is the critical resource

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 pipelining example: an instruction pipeline

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



Four-stage instruction pipeline
(steps required to complete and instruction):

IF = instruction fetch
D = instruction decode + register read
EX = execute operation
WB = "write back" results to registers

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.)

Actual instruction pipelines can be variable length (depending on the instruction) deep as ~20 stages in modern CPUs

Part 2:

The theme of the second half of today's lecture is:

Abstraction vs. implementation

Conflating semantics (meaning) of an abstraction with details of its implementation is a common cause for confusion in this course.

Abstraction vs. implementation

Semantics: what do the operations provided by a programming model mean?

Given a program, and given the semantics (meaning) of the operations used, what is the answer that the program will compute?

Implementation (aka scheduling): how will the answer be computed on a parallel machine?

In what (potentially parallel) order will be a program's operations be executed?
Which operations will be computed by each thread?
Each execution unit? Each lane of a vector instruction?

Your goal as a student:

Given a program and knowledge of how a parallel programming model is implemented, in your head can you "trace" through what each part of the parallel computer is doing during each step of program.

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
  - Go read it!

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;
   }
}
```

Invoking sinx()

C++ code: `main.cpp`

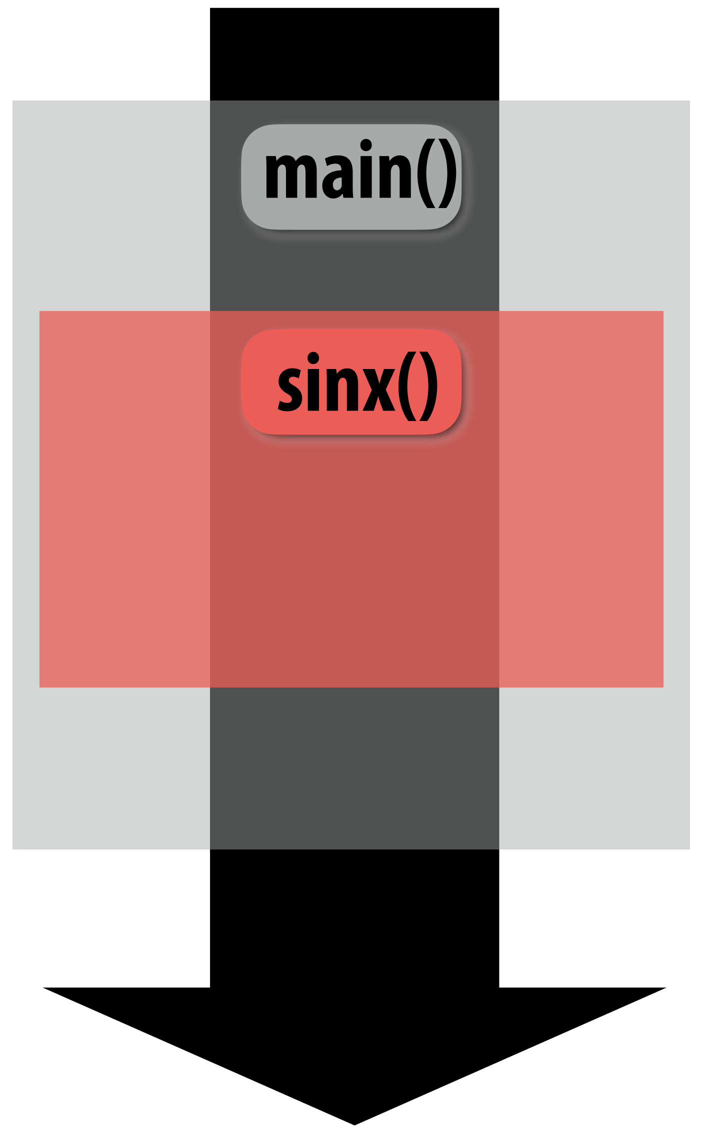
```
#include “sinx.h”

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;
}
```



Call to `sinx()`
Control transferred to sinx() func

Return from `sinx()`
Control transferred back to main()

C++ code: `sinx.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`

```
#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)
{
   // 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

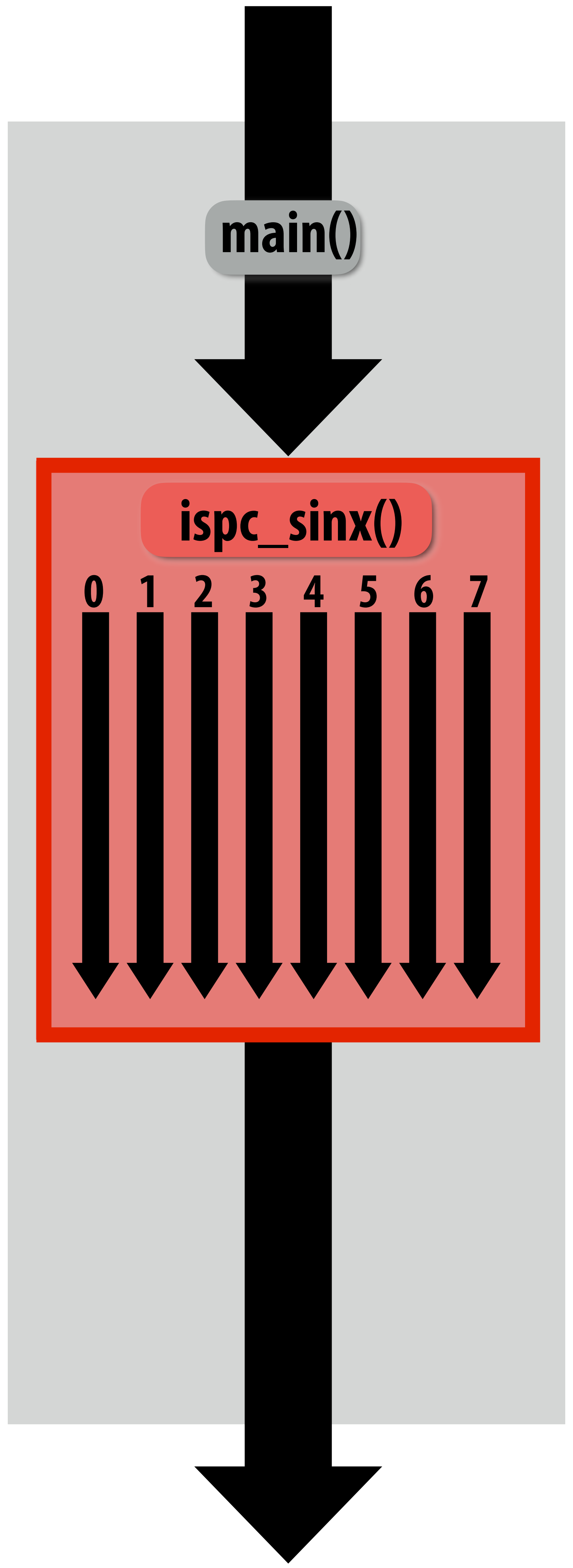
C++ code: `main.cpp`

```
#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;
}
```



Sequential execution (C code)

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

`ispc_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

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 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.

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;
   }
}
```

Interleaved assignment of program instances to loop iterations

Elements of output array (results)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
Instance 0 (programIndex = 0)
Instance 1 (programIndex = 1)
Instance 2 (programIndex = 2)
Instance 3 (programIndex = 3)
Instance 4 (programIndex = 4)
Instance 5 (programIndex = 5)
Instance 6 (programIndex = 6)
Instance 7 (programIndex=7)

"Gang" of ISPC program instances

In this illustration: gang contains eight instances: `programCount` = 8

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

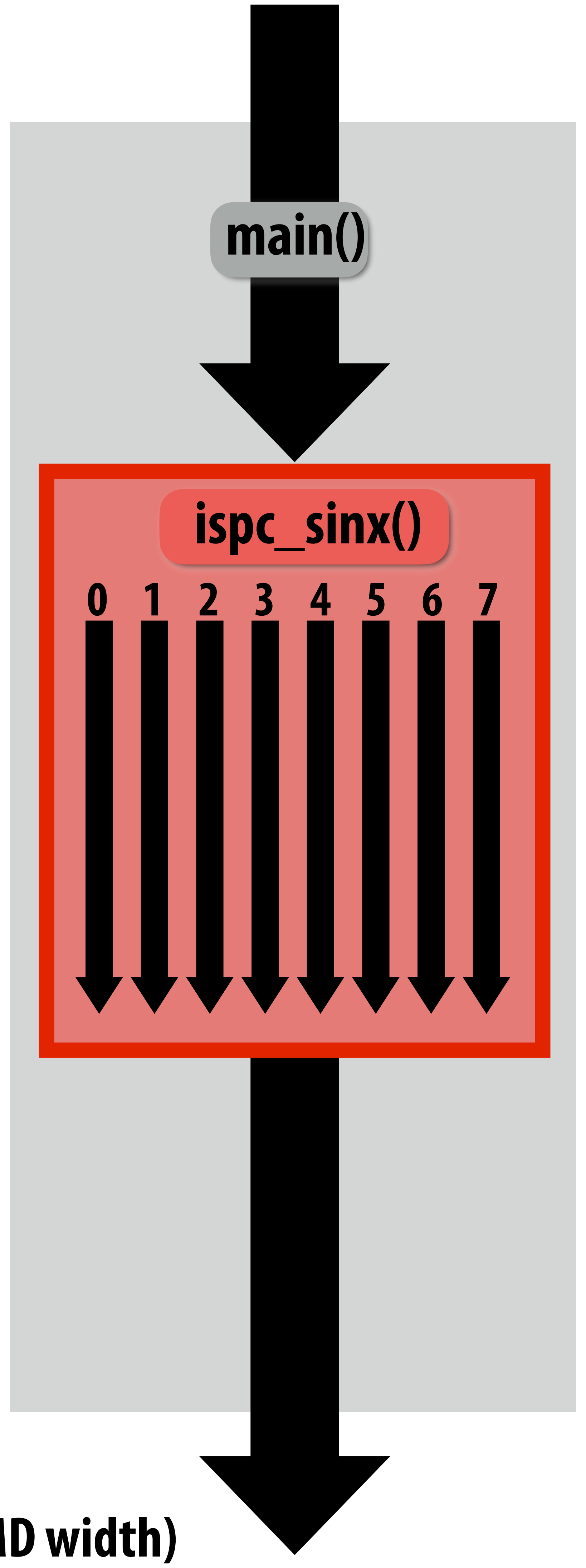
C++ code: `main.cpp`

```
#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;
}
```



Sequential execution (C code)

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

`ispc_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 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 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_v2(N, terms, x, result);
  return 0;
}
```

ISPC code: `sinx.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)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
Instance 0
(programIndex = 0)
Instance 1
(programIndex = 1)
Instance 2
(programIndex = 2)
Instance 3
(programIndex = 3)
Instance 4
(programIndex = 4)
Instance 5
(programIndex = 5)
Instance 6
(programIndex = 6)
Instance 7
(programIndex=7)

"Gang" of ISPC program instances

In this illustration: gang contains eight instances: `programCount` = 8

Schedule: interleaved assignment

“Gang” of ISPC program instances

Gang contains four instances: `programCount` = 8

time

	Instance 0 (programIndex = 0)	Instance 1 (programIndex = 1)	Instance 2 (programIndex = 2)	Instance 3 (programIndex = 3)	Instance 4 (programIndex = 4)	Instance 5 (programIndex = 5)	Instance 6 (programIndex = 6)	Instance 7 (programIndex = 7)
i=0	0	1	2	3	4	5	6	7
i=1	8	9	10	11	12	13	14	15
i=2	16	17	18	19	20	21	22	23
i=3	24	25	26	27	28	29	30	31

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

`float value = x[idx];`

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

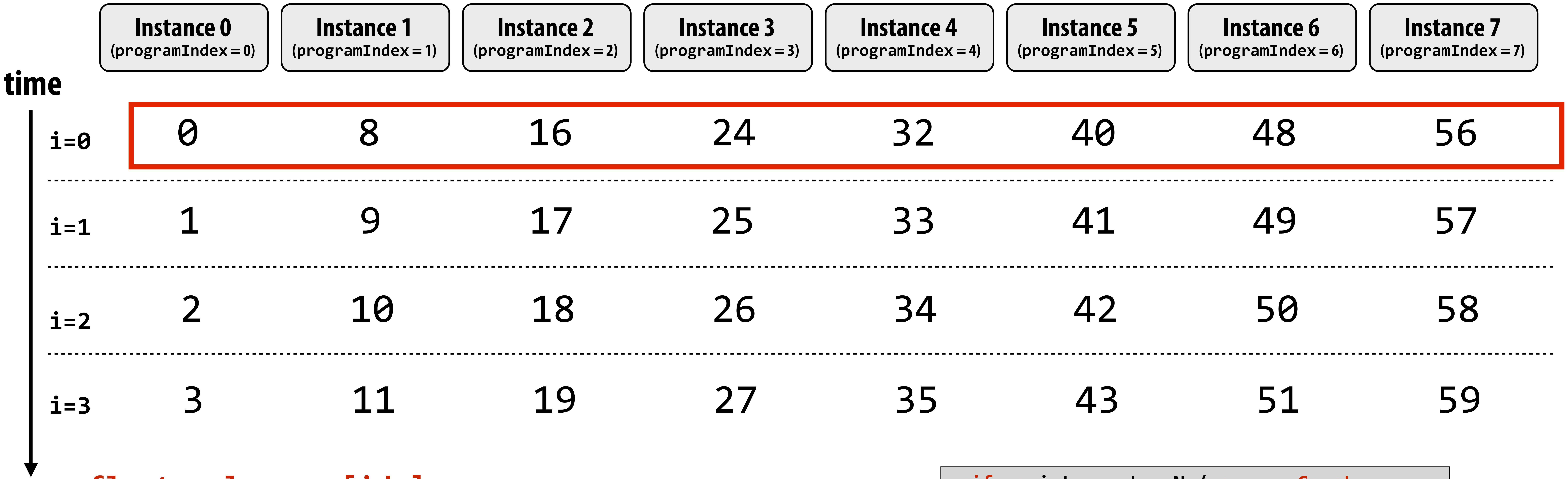
```
...
// assumes N % programCount = 0
for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
...
```

* see _mm256_load_ps() intrinsic function

Schedule: blocked assignment

"Gang" of ISPC program instances

Gang contains four instances: `programCount` = 8



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

`float value = x[idx];`

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

```
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];
...
```

* 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 takes responsibility for assigning iterations to program instances in the gang

How might foreach be implemented?

Code written using `foreach` abstraction:

```
foreach (i = 0 ... N)
{
   // do work for iteration i here...
}
```

Implementation 1: program instance 0 executes all iterations

```
if (programCount == 0) {
   for (int i=0; i<N; i++) {
     // do work for iteration i here…
   }
}
```

Implementation 2: interleave iterations onto program instances

```
// assume N % programCount = 0
for (uniform int loop_i=0; loop_i<N; loop_i+=programCount)
{
  int i = loop_i + programIndex;
  // do work for iteration i here...
}
```

Implementation 3: block iterations onto program instances

```
// assume N % programCount = 0
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int loop_i=0; loop_i<count; loop_i++)
{
    int i = start + loop_i;
    // do work for iteration i here...
}
```

Implementation 4: dynamic assignment of iterations to instances

```
uniform int nextIter;
if (programCount == 0)
  nextIter = 0;

int i = atomic_add_local(&nextIter, 1);
while (i < N) {

  // do work for iteration i here...

  i = atomic_add_local(&nextIter, 1);
}
```

Thinking about iterations, not parallel execution

In many simple cases, using `foreach` allows the programmer to express their program almost as if it was a sequential program

```
export void ispc_function(
   uniform int    N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
     float val = x[I];
     float result;

     // do work here to compute
     // result from val

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

What does this program do?

```
// 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

// call ISPC function
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];
   }
}
```

This ISPC program computes the absolute value of elements of x, then repeats it twice in the output array y

What does this program do?

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

// initialize N elements of x

// call ISPC function
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];
   }
}
```

The output of this program is undefined!

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

Computing the sum of all elements in an array (incorrectly)

What's the error in this program?

```
export uniform float sum_incorrect_1(
    uniform int N,
    uniform float* x)
{
    float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }

    return sum;
}
```

`sum` is of type `float`
(different variable for all program instances)

Cannot return many copies of a varianble to the calling C code, which expects one return value of type float
Result: compile-time type error

What's the error in this program?

```
export uniform float sum_incorrect_2(
    uniform int N,
    uniform float* x)
{
    uniform float sum = 0.0f;
    foreach (i = 0 ... N)
    {
        sum += x[i];
    }

    return sum;
}
```

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

`x[i]` has a different value for each program instance
So what gets copied into sum?
Result: compile-time type error

Computing the sum of all elements in an array (correctly)

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

   // reduce_add() is part of ISPC's cross
   // program instance standard library
   sum = reduce_add(partial);

   return sum;
}
```

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 left will execute in a manner similar to the C code with AVX intrinsics implemented below. *

```
float sum_summary_AVX(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 correctly implements the semantics of the ISPC gang abstraction, then you've got a good command of ISPC**

ISPC's cross program instance operations

Compute sum of a variable's value in all program instances in a gang:

```
uniform int64 reduce_add(int32 x);
```

Compute the min of all values in a gang:

```
uniform int32 reduce_min(int32 a);
```

Broadcast a value from one instance to all instances in a gang:

```
int32 broadcast(int32 value, uniform int index);
```

For all `i`, pass value from instance `i` to the instance `i+offset % programCount`:

```
int32 rotate(int32 value, uniform int offset);
```

ISPC: abstraction vs. implementation

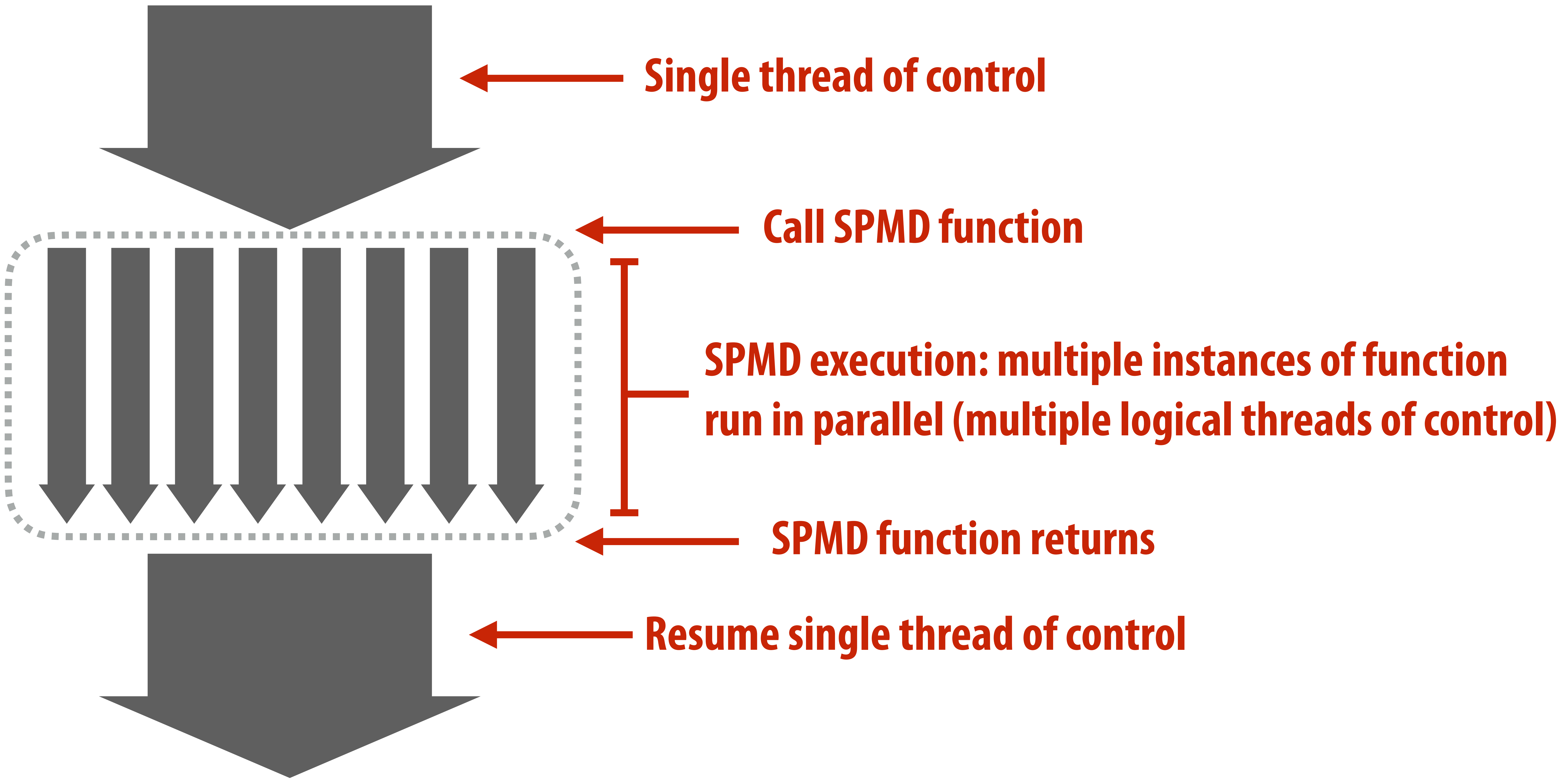
- **Single program, multiple data (SPMD) programming model**
  - Programmer "thinks": running a gang is spawning `programCount` logical instruction streams (each with a different value of `programIndex`)
  - This is the programming <u>abstraction</u>
  - Program is written in terms of this abstraction

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


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 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 as you do assignment 1.

Thinking about operating on data in parallel?

- In many simple cases, using ISPC foreach allows the programmer to express their program almost as if it was a sequential program
  - Almost want to explain code as: "independently, for each element in the input array… do this…"

- Exceptions:
  - Uniform variables
  - Cross-instance operations (in standard library, like reduceAdd)

- But ISPC is a low-level programming language: by exposing programIndex and programCount, it allows programmer to define what work each program instance does and what data each instance accesses
  - Can implement programs with undefined output
  - Can implement programs that are correct only for a specific programCount

```
export void ispc_function(
   uniform int    N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
     float val = x[i];
     float result;

     // do work here to compute
     // result from val

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

But can express very advanced cooperation

Here's a program that computes the product of all elements of an array in lg(8) = 3 steps

```
// compute the product of all eight elements in the
// input array. Assumes the gang size is 8.
export void vec8product(
   uniform float* x,
   uniform float* result)
{
   float val1  = x[programIndex];
   float val2 = shift(val1, 1);

   if (programIndex % 2 == 0)
     val1 = val1 * val2;

   val2 = shift(val1, 2);
   if (programIndex % 4 == 0)
      val1 = val1 * val2;
   }

   val2 = shift(val1, 4);
   if (programIndex % 8 == 0) {
      *result = val1 * val2
   }
}
```

But what if ISPC was not trying to be a low-level language?

- Example: change language so there is no access to programIndex, programCount
- Expect programmer to just use foreach

- Now there's very little need to think about program instances at all.
  - Everything outside a foreach must be uniform values and uniform logic. Why?

```
export void ispc_function(
   int    N,
   float* x,
   float* y)
{

   int twoN = 2 * N;

   foreach (i = 0 ... twoN)
   {
     float val = x[i];
     float result;

     // do work here to compute
     // result from val

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

Another alternative

- Don't even allow array indexing!
- Invoke computation once per element of a "collection" data structure
- Programmer writes no loops, performs no data indexing

```
float dowork(float x) {
  // do work here to compute
  // result from x
}

Collection x;  // data structure of N

// invoke doWork for all elements of x,
// placing results in collection y
Collection y = map(doWork, x, y);
```

- This model should be very family to NumPy, PyTorch, etc. programmers, right?

```
import numpy as np

def addOne(i):
    return i+1
mapAddOne = np.vectorize(addOne);

X = np.arange(15) # create numPy array [0, 1, 2, 3, ...]
Y = np.arange(15) # create numPy array [0, 1, 2, 3, ...]

Z = X + Y;              # Z = [0, 2, 4, 6, … ]
Zplus1 = mapAddOne(Z);  # Zplus1 = [1, 3, 5, 7, …]
```

- Much more on this to come

Summary

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

- They provide <u>abstractions</u> that permit multiple valid <u>implementations</u>.

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