Scientific and Technical Computing

Hardware and Code Optimization

Lars Koesterke

UT Austin, 9/15/22 & 9/22/22

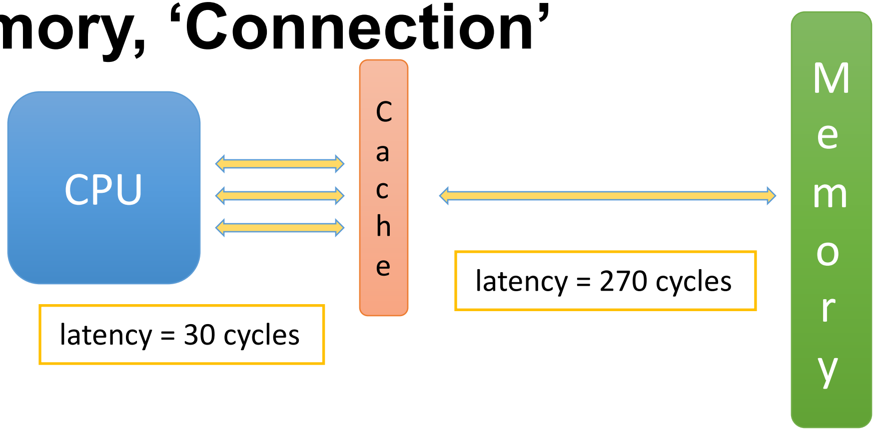
Our Computer: CPU, Cache, Memory, ‘Connection’

CPU
1. Pipelined operation
System designed to get 1 opc

Memory
1. Data streams
System designed to support 1 wpc (for one row)

Caches
1. Managed by run-time
2. Cache size (for stencil update)
System designed for ‘enough’ bandwidth to support 2 rows
Size: at least 3×n words

Our computer has been somewhat ‘hypothetical’ so far
We have designed the specs so that we get ‘optimal’ performance for a stencil update



But there is a problem with our cache

Our Computer: CPU, Cache, Memory, 'Connection'

CPU
1. Pipelined operation

System designed to get 1 opc

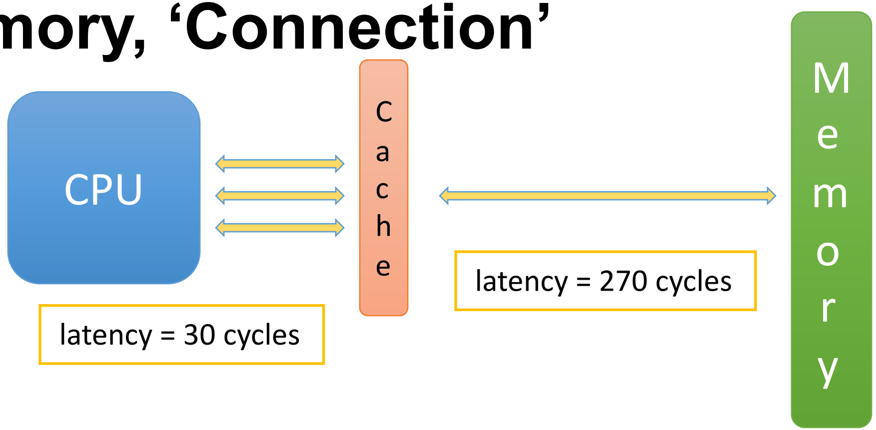
Memory
1. Data streams

System designed to support 1 wpc (for one row)

Caches
1. Managed by run-time
2. Cache size (for stencil update)

System designed for 'enough' bandwidth to support 2 rows
Size: at least 3×n words

Our computer has been somewhat 'hypothetical' so far
We have designed the specs so that we get 'optimal' performance for a stencil update



Requirement: Size of the cache = 3×n

n could be any number, any large number

Size of cache in hardware certainly not adjustable
Also differences between chip generations

TACC

Outline

CPU & Memory, latency, bandwidth, wpc, opc …

Data streaming, pipelining, caches (part 1)

Caches: software (short)

Caches (working principles)

There are at least 4 'working principles' that we will have to cover

My 'big' plan
Cover many hardware fundamentals as they guide code design

- loosely in decreasing order of importance

For each hardware feature:
Add details as necessary to describe a simplified, yet functional 'working model'

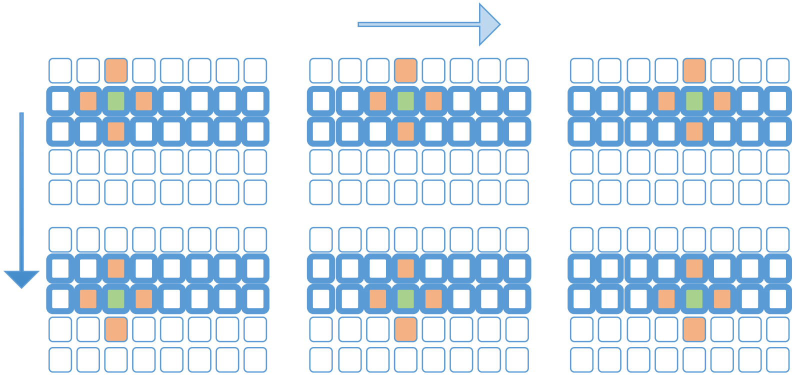
Discussion: Working around the size limitation

Example:

n=500, cache size=300 words

At what iteration '`i`' do we (approximately) start to replace data in the cache?

- '`i`' is inner loop, left to right
- '`j`' is outer loop, top to bottom



```
do j=1, n
  do i=1, n
    y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
  enddo
enddo
```

Discussion: Working around the size limitation

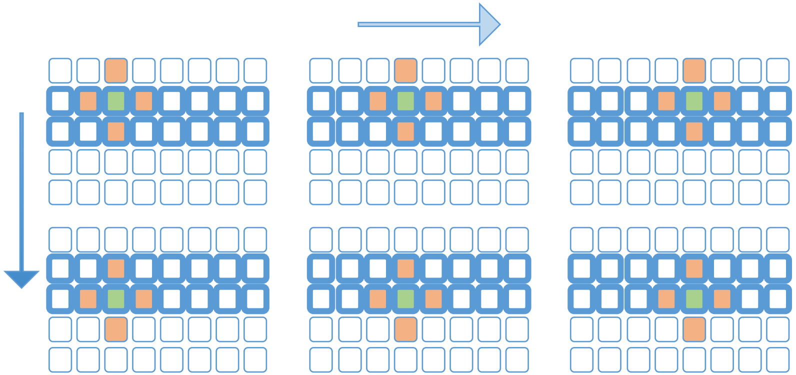
Example:

n=500, cache size=300 words

At what iteration '`i`' do we (approximately) start to replace data in the cache? `i~100`

What do we do when we reach '`i=100`'?

- Hint: going further to the right is a 'dead end'



```
do j=1, n
  do i=1, n
    y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
  enddo
enddo
```

Discussion: Working around the size limitation

Example:

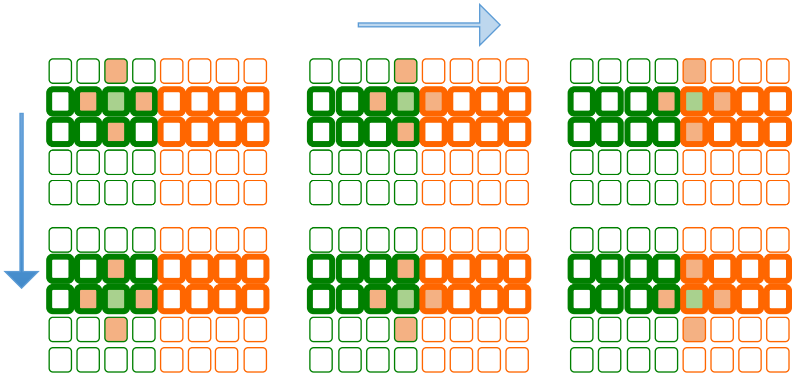
n=500, cache size=300 words

At what iteration 'i' do we (approximately) start to replace data in the cache? i~100

What do we do when we reach 'i=100'

- Hint: going further to the right is a 'dead end'
- Instead, we go one row down
- The green area first, then the orange area

How do we do this in code?



```
do j=1, n
  do i=1, n
    y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
  enddo
enddo
```

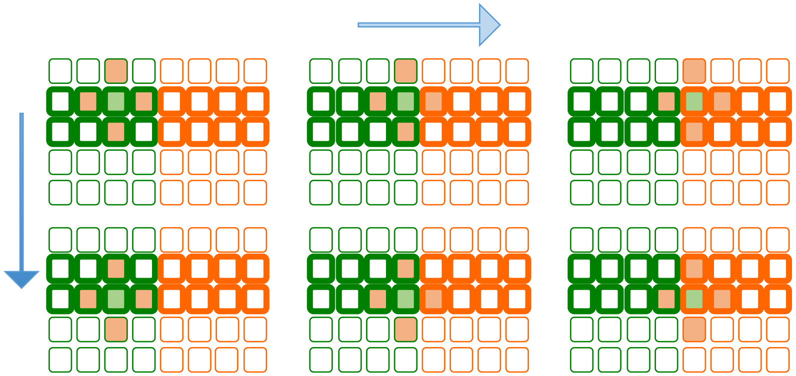
Discussion: Working around the size limitation

Example:

n=500, cache size=300 words

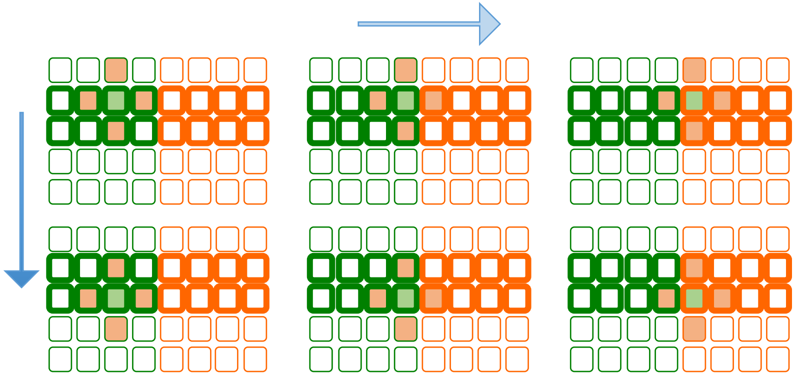
<u>So how do we do this in code?</u>

- What is the width of a stripe/block?
- How many stripes?
- How many loops in the code?



```
do j=1, n
  do i=1, n
    y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
  enddo
enddo
```

Discussion: Working around the size limitation



Example:

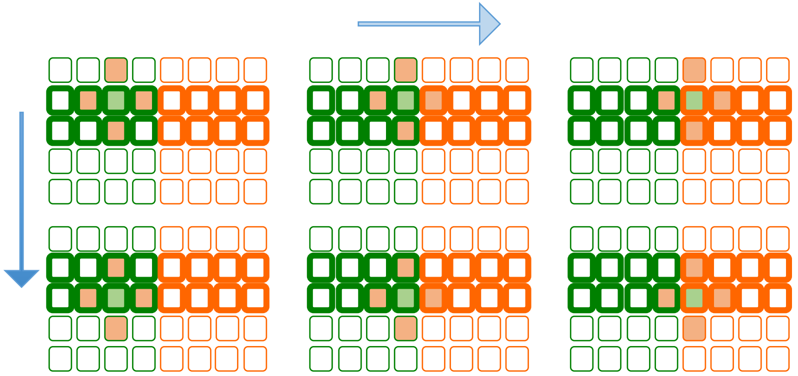
n=500, cache size=300 words

<u>So how do we do this in code?</u>

- What is the width of a stripe? 100
- How many stripes? 5
- How many loops in the code? 3 (up from 2)

```
n = 500; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Discussion: Working around the size limitation



Example:

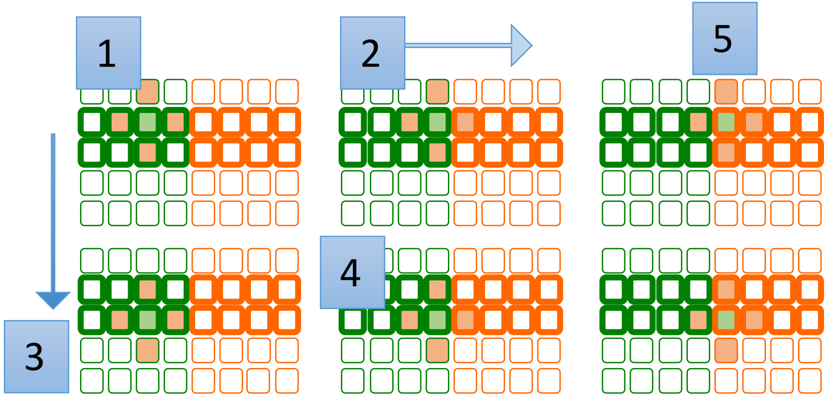
n=500, cache size=300 words

<u>So how do we do this in code?</u>

- What is the width of a stripe? 100
- How many stripes? 5
- How many loops in the code? 3 (up from 2)

```
n = 500; ns = 100
do iout=1, n/ns
  do j=1, n
    is = (iout-1) * ns + 1      (1, 101, 201, 301, 401)
    ie = iout * ns              (100, 200, 300, 400, 500)
    ie = is + ns - 1            (equivalent formulation)
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Discussion: Working around the size limitation



Example:

n=500, cache size=300 words

So how do we do this in code?

- What is the width of a stripe? 100
- How many stripes? 5
- How many loops in the code? 3 (up from 2)

```
n = 500; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 *
    enddo
  enddo
enddo
```

We go fast left to right (**x**-direction)
We go slowly in **y**-direction
Hence left to right is the inner loop. Loop index is '**i**'

The 'fast' loop, i.e. the inner loop 'exceeds' the size of the cache
We split up the inner loop in 2 loops. Indexes '**i**' and '**iout**'
The loop '**j**' that re-uses the data of the inner loop is in the middle

is and **ie** are set so that the pair of loops (**i** and **iout**) covers the whole computational domain, left to right, exactly once

Cache blocking

Re-use data before it is evicted

Breaking a loop into 2 (or more) parts

(There can be cache blocking for multiple loops)

Note:
In our example we have been overly optimistic
Width of the strip stretched to the max
Real application: other data is also stored in cache
(there are also other processes)

Let's fill in the blanks

```
n = 500; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Cache blocking

Re-use data before it is evicted

Breaking a loop into 2 (or more) parts

There can be cache blocking for multiple loops

Note:
In our example we have been overly optimistic
Width of the strip stretched to the max
Real application: other data is also stored in cache
(there are also other processes)

Be aware that for arbitrary pairs (n,ns) the code will be more complicated

Consider:
N=495; ns=100

```
n = 495; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Cache blocking

Note:
In our example we have been overly optimistic
Width of the strip stretched to the max
Real application: other data is also stored in cache
(there are also other processes)

In our example, why should the width of the strip (ns) be smaller than 50?

Re-use data before it is evicted

Breaking a loop into 2 (or more) parts

There can be cache blocking for multiple loops

1. Be aware that for arbitrary pairs (n,ns) the code will be more complicated

2. Cache size=300 ➔ ns=100
ns is too large (by 2×)
Why?

```
n = 500; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Cache blocking

Re-use data before it is evicted

Breaking a loop into 2 (or more) parts

There can be cache blocking for multiple loops

1. Be aware that for arbitrary pairs (n,ns) the code will be more complicated

2. Cache size=300 ➔ ns=100
ns is too large (by 2×)
Why?

Note:
In our example we have been overly optimistic
Width of the strip stretched to the max
Real application: other data is also stored in cache
(there are also other processes)

In our example, why should the width of the strip (ns) be smaller than 50?

Usually numerical tests (trials) are used to determine a suitable size for the cache blocking

Tests are repeated, if:
- Architecture changes (different machine)
- Implementation changes (more/less data in loop kernel)

```
n = 500; ns = 100
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Cache blocking

Re-use data before it is evicted

Breaking a loop into 2 (or more) parts

There can be cache blocking for multiple loops

2. Cache size=300 ➔ ns=100
ns is too large (by 2×)
Why?

Everything moving between memory and CPU is cached:

Not just '**x**' but also '**y**'

Note:
In our example we have been overly optimistic
Width of the strip stretched to the max
Real application: other data is also stored in cache
(there are also other processes)

In our example, why should the width of the strip (ns) be smaller than 50?

```
n = 500; ns = 50
do iout=1, …
  do j=1, n
    is = …
    ie = …
    do i=is, ie
      y(i,j) = 0.25 * (x(i-1,j) + x(i+1,j) + x(i,j-1) + x(i,j+1))
    enddo
  enddo
enddo
```

Limitations

Conflicting goals

- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In - First Out) or LRU

For illustration purposes
Cache is drawn as a 2d 'array'
You can imagine that the cache is organized as a 1d array

Match or no-match?

How do we find a match?

How do we find an element to evict?

4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Problems to tackle

1. Speed
2. FIFO
3. Storage efficiency

Basic principle
The cache is small, therefore it can hold data only for a (very) limited time (time in cycles)
How long (# of cycles) and how often data is re-used depends on the code (implementation of the algorithm)

Limitations

Keep in this in mind when thinking about how a cache works:
- The cache stores where the data came from, i.e. the original address
- The cache stores the actual value

These numbers are the addresses, not the actual data
Assume that in this example we encounter the addresses consecutive and in order.
Address ‘1’, and its content, was stored first, and is the oldest
Address ‘2’, and its content’, is the second oldest
Address ‘16’ was stored last

Conflicting goals
- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In - First Out) or LRU

Match or no-match?

How do we find a match?

How do we find an element to evict?

4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Problems to tackle
1. Speed
2. FIFO
3. Storage efficiency

Basic principle
The cache is small, therefore it can hold data only for a (very) limited time (time in cycles)
How long (# of cycles) and how often data is re-used depends on the code (implementation of the algorithm)

Limitations

Conflicting goals

- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In - First Out) or LRU

For illustration purposes
Cache is drawn as a 2d 'array'
You can imagine that the cache is organized as a (long) 1d array

Match or no-match?

How do we find a match?

How do we find the element to evict?

4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Problems to tackle

1. Speed
2. FIFO
3. Storage efficiency

Example:
Loading element #3: This element is stored in the cache. How do we find element #3
Loading element #17: This is not a match; #17 loaded from memory and will replace oldest element in cache

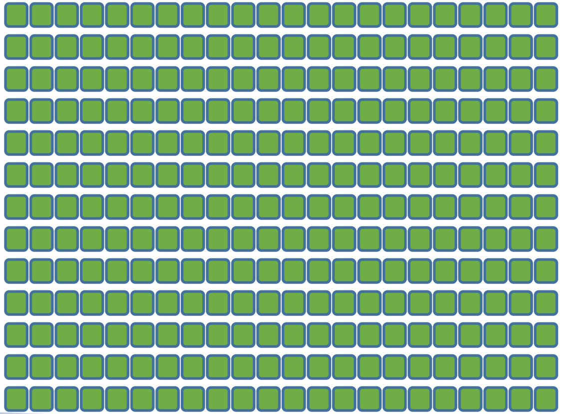
Limitations: The Cache can be quite large

Fully associative cache
- Data can be ‘anywhere’ in the cache
- Large number of checks to find elements in cache
- Bookkeeping of age for all elements

Conflicting goals
- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In- First Out) or LRU

Note:
For every cell in the cache the original address must be stored

Address = 1 word
Therefore, half the cache stores addresses
Half the cache stores data
… and then some ordering is needed



OK, for a cache holding 16 words we may be able to compare all addresses *and* also make FIFO work

Cache size may exceed 1Mw
More that a million entries
(1Mw = 4 or 8 Mb)

How can we devise a fast strategy?

Let’s tackle ‘fast access’ first,
and the add some FIFO
and then storage efficiency

Example: Let’s make our cache access fast

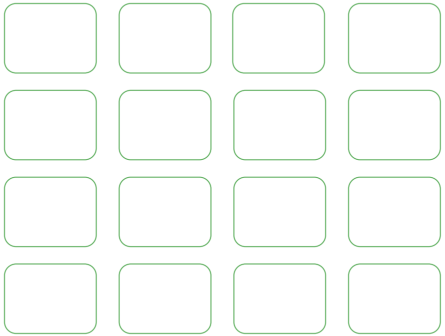
Let’s make up some address space notation
- Address (main memory) has 6 digits
- Each digit holds a value between 1 and 4
- Example 142314

We encounter addresses in this order

141423
141424
141431
443311
141432
141433
121111
…

How can we **map** our addresses into the cache address space?

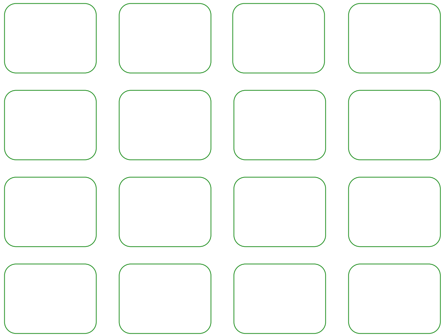
Our cache



Example: Let’s make our cache fast

Let’s make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4

Our cache



We encounter addresses in this order

What if I write it like this

```
1414 23
1414 24
1414 31
4433 11
1414 32
1414 33
1211 11
```

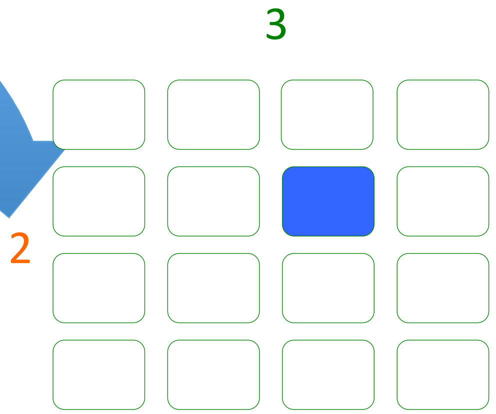
…

How can we **map** our addresses into the cache address space?

Example: Let’s make our cache fast

Let’s make up some address space notation
- Address has 6 digit
- Each digit ho value between 1 and 4

We enco r addresses in this order
What if I write it like this
1414 23
1414 24
1414 31
4433 11
1414 32
1414 33
1211 11
…
How can we **map** our addresses into the cache address space?



Algorithm
CPU reads address
CPU checks if data is stored in cache
(How many storage locations have to be checked?)
1. If yes, read it from there
2. If no, read data from memory and store it also in cache

How does this implementation accelerate data access?

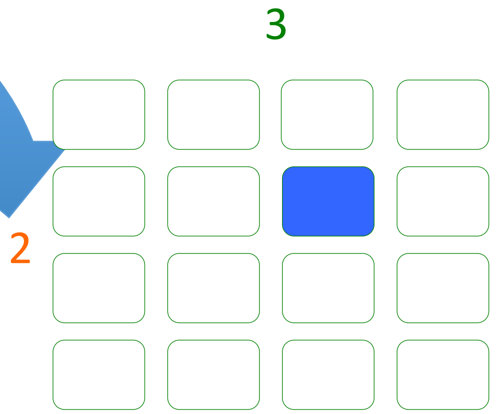
Cache with Direct Mapping

Let's make up some address space notation
- Address has 6 digit
- Each digit ho… value between 1 and 4

We enco… addresses in this order
What if I write it like this

```
1414 23
1414 24
1414 31
4433 11
1414 32
1414 33
1211 11
…
```

How can we **map** our addresses into the cache address space?



<u>Direct Mapping</u> (many to one)
Every element in main memory can be stored
In exactly one cache location

Problems
1. No FIFO --- Why?
2. Why do some access patterns render caching ineffective?

Cache with Direct Mapping: Problems

Conflicting goals
- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In- First Out) or LRU

Example: array with 16×16 elements
Assume row major ordering
Loop through the first column

How far are these elements apart in main memory addresses
`a(1,1), a(1,2), a(1,3)`?
or
`a[0,0], a[1,0], a[2,0]`   for column major ordering?

4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Cache with Direct Mapping: Problems

Conflicting goals

- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In- First Out) or LSR

Example: array with 16×16 elements
Assume row major ordering
Loop through the first column

How far are these elements apart in main memory addresses
`a(1,1), a(1,2), a(1,3)`? `a[0,0], a[1,0], a[2,0]?`
Answer: The distance is 16

Example addresses are:
4422 22
4423 22
4424 22

Where are these elements mapped in the cache?

4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Cache with Direct Mapping: Problems

Conflicting goals
- Purpose of cache: fast --- low latency/high bandwidth
- Organization of cache: FIFO (First In- First Out) or LSR

Example: array with 16×16 elements
Assume row major ordering
Loop through the first column

How far are these elements apart in main memory addresses
`a(1,1), a(1,2), a(1,3) or a[0,0], a[1,0], a[2,0]?`
Answer: The distance is 16

Example addresses are:
4422 22
4423 22
4424 22

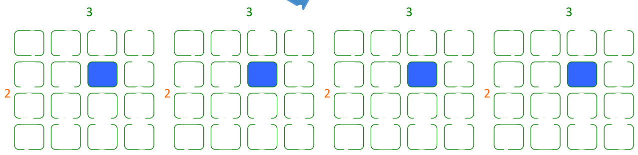
4	6	12	11
5	1	14	10
9	8	2	3
7	15	13	16

Where are these elements mapped in the cache? All to the same element
Therefore, our cache has a reduced effective size (In our case just one element)

Cache Associativity

Let's make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4

Example: Cache with 4-way associativity



We encounter addresses in this order
What if I [illegible]e it like this

1414 23
1414 24
1414 31
4433 11
1414 32
1414 33
1211 23
4433 23
…

An element of data may be cached in 1 of 4 locations
FIFO: replace the 'oldest' data element (out of the 4)

<u>Advantages of Caches with Associativity</u>
1. Ineffective mapping alleviated (somewhat)
2. Compare addresses only for few (4) locations
3. Eviction policy FIFO or LRU schedule
4. Keep 'age' for few (4) entries

(In principle, 'random' eviction could work, too)

TACC

Cache: So far

What we have addressed so far

- Why caches?
- Cache blocking
- Address mapping
- Associativity
  - fully associative, set-associative, direct mapping

Remaining problem

For each 'cell' in the cache we store the

- data (that's what we are interested in)
- where the data came from (i.e. the address in main memory)

50% is useful (to us)

50% is book keeping

We can do better! But how?

Requirements

Know what is stored (full address)

Know how 'old' the data is (within associativity)

Finding data

Compare address with address in cache

Two possibilities

1. Match: load data from cache
2. No match: Load form memory, evict oldest data, store new data in place

Does this help us?

$$a(i) = b(i) + c(i)$$

```
loop with index i
  a(i) = b(i) + c(i)
end loop
```

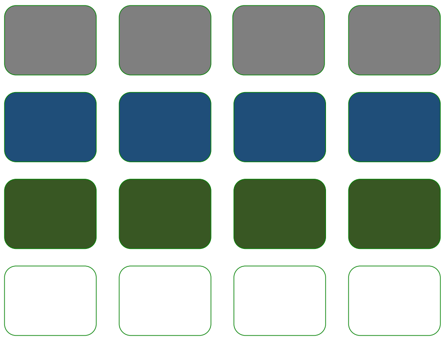
Can data streams help us reducing the overhead of book keeping?

Improved Cache Storage Efficiency

Let's make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4

Streams of data: a, b, c

```
1414 11
1414 12
1414 13
1414 14
3722 31
3722 32
3722 33
3722 34
2344 21
2344 22
2344 23
2344 24
```



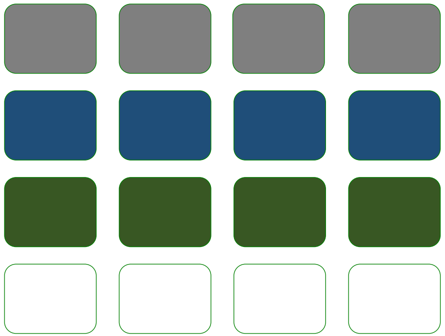
Consecutive (in memory) elements of a, b, and c are loaded
Elements of a, b, and c are cached in consecutive locations

Improved Cache Storage Efficiency

Let's make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4

Streams of data: a, b, c

1414 11
1414 12
1414 13
1414 14
3722 31
3722 32
3722 33
3722 34
2344 21
2344 22
2344 23
2344 24

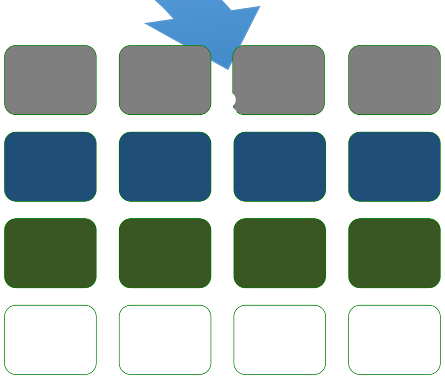


Consecutive (in memory) elements of a, b, and c are loaded
Elements of a, b, and c are cached in consecutive locations

Let's take advantage and keep only track of consecutive tuples of 8/16 words (double/single precision), i.e. 512 bits

Cache Lines

Let's make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4



Streams of data: a, b, c

```
1414 11
1414 12
1414 13
1414 14
3722 31
3722 32
3722 33
3722 34
2344 21
2344 22
2344 23
2344 24
```

Complete cache lines are moved between memory, cache, and CPU
1 cache line: 512 bits (x86 architecture), or 8 words (dp), or 16 words (sp)

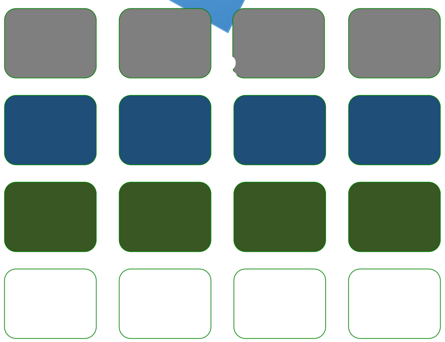
Book keeping is done per cache line (reduction by 7/8 or 15/16)

Assumption: It is likely that the code uses all elements rather than only 1
`a(i) = b(i) + c(i)`: 3 streams each with consecutive data access

Is this always efficient? What would be a setup (addresses) were this works with limited efficiency?

Cache Lines

Let’s make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4



Streams of data: a, b, c

```
1414 11
1414 12
1414 13
1414 14
3722 31
3722 32
3722 33
3722 34
2344 21
2344 22
2344 23
2344 24
```

Complete cache lines are moved between memory, cache, and CPU
1 cache line: 512 bits, or 8 words (dp), or 16 words (sp)

Book keeping: per cache line (reduction by 7/8 or 15/16)

Assumption: It is likely that the code uses all elements rather than only 1
`a(i) = b(i) + c(i)`: 3 streams each with consecutive data access

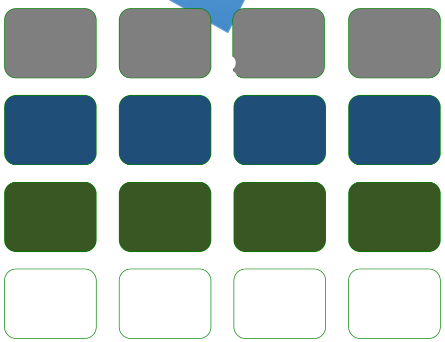
Is this always efficient? What would be a setup (addresses) were this works with limited efficiency?

What will happen here?

```
Loop with index i
a(2×i) = b(2×i) + c(2×i)
```

Cache Lines

Let’s make up some address space notation
- Address has 6 digits
- Each digit holds a value between 1 and 4



Streams of data: a, b, c

```
1414 11
1414 12
1414 13
1414 14
3722 31
3722 32
3722 33
3722 34
2344 21
2344 22
2344 23
2344 24
```

Complete cache lines are moved between memory, cache, and CPU
1 cache line: 512 bits, or 8 words (dp), or 16 words (sp)

Book keeping: per cache line (reduction by 7/8 or 15/16)

Assumption: It is likely that the code uses all elements rather than only 1
`a(i) = b(i) + c(i)`: 3 streams each with consecutive data access

Is this always efficient? What would be a setup (addresses) were this works with limited efficiency?

<u>What will happen here?</u>

`Loop with index i`
`a(2×i) = b(2×i) + c(2×i)`

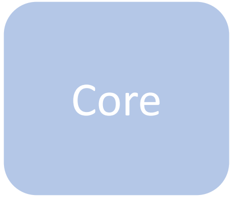
- Effective bandwidth reduced by 2×
- Effective cache size reduced by 2x

But wait …

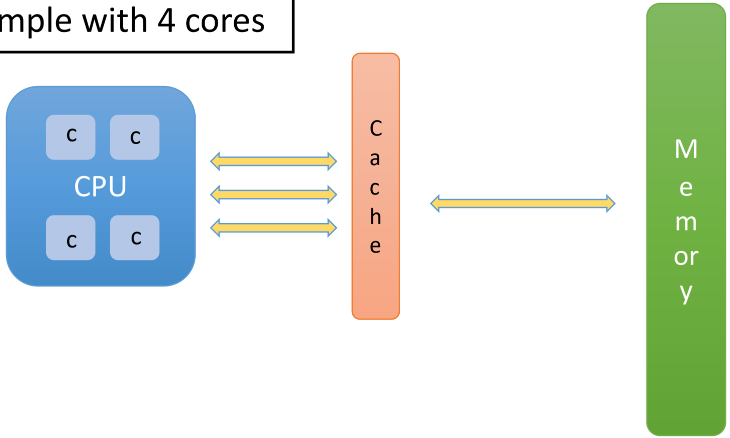
There is more!

about caches

Our Toy Computer with multi-core CPUs



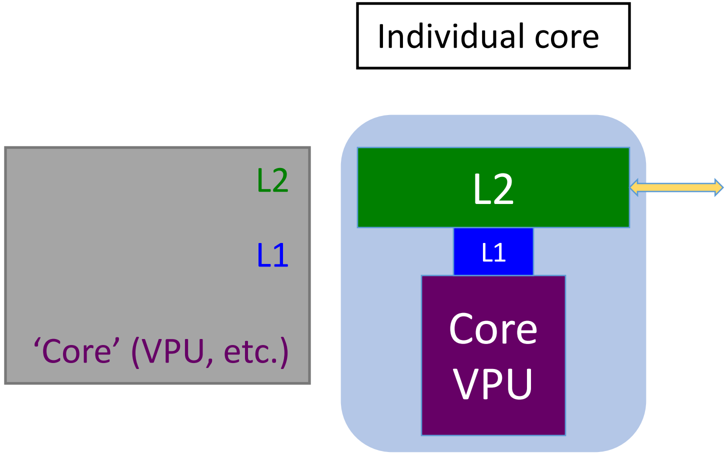
Example with 4 cores

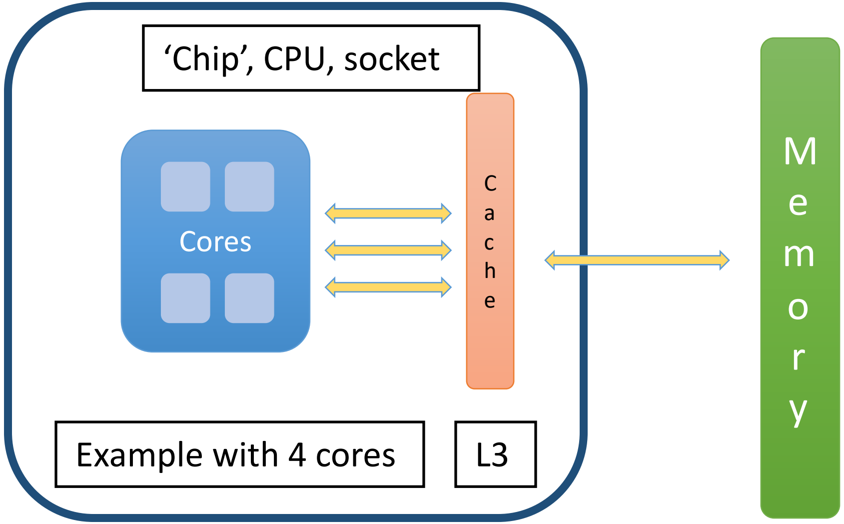


<u>Stampede2</u>
A socket/CPU has 24 cores
The total number of caches per socket is 49

How many caches are there
- Per core?
- Per CPU?

Our Toy Computer with multi-core CPUs





Stampede2

A socket/CPU has 24 cores
The total number of caches per CPU is 49

How many caches are there
- Per core? 2 (L1 and L2)
- Per CPU? 1 (L3)

Feel sorry for the 'bookkeeper' at least once!

Sizes

L1: 32 KB	very close, very fast
L2: 1MB	close and fast
L3: 33MB	Farther, but still on-chip

Stampede2 nodes have 2 sockets
Total number of caches is 98

More bookkeeping

Bookkeeping: MOESI Protocol

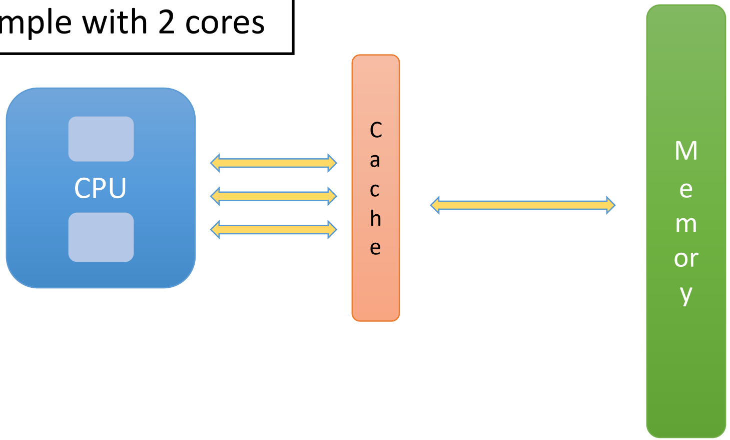
For now, only consider **reading data**
… `= b(i) + c(i)`

We start the loop, reading c(1) on core 0

1. Cache line for c(1:8) is in main memory
2. Cache line moves to L3
3. Cache line moves to L2
4. Cache line moves to L1
5. 'somehow' we perform the operation

So where do we read c(2) from?

Example with 2 cores



MOESI Protocol

For now, only consider **reading data**
… `= b(i) + c(i)`

We start the loop, reading c(1) on core 0

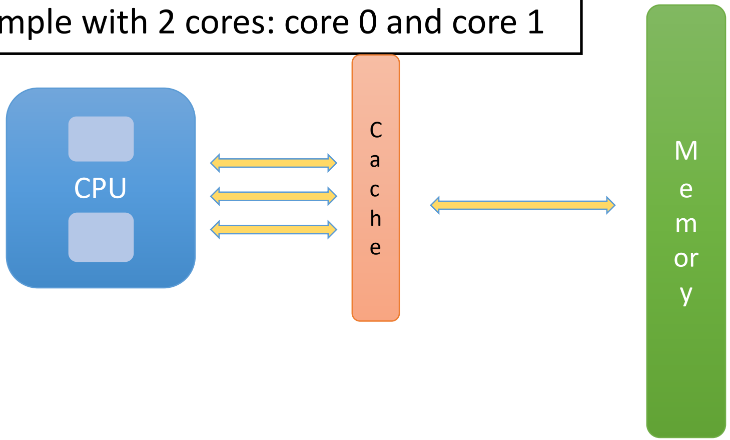
1. Cache line for c(1:8) is in main memory
2. Cache line moves to L3
3. Cache line moves to L2
4. Cache line moves to L1
5. ‘somehow’ we perform the operation

So where do we read c(2) from?

We continue the loop, reading c(2)

1. There are now 4 copies of the cache line
2. We can read the data from any copy, likely reading from L1
3. ‘somehow’ we perform the operation

Example with 2 cores: core 0 and core 1



MOESI Protocol

For now, only consider **reading data**

… = b(i) + <u>c(i)</u>

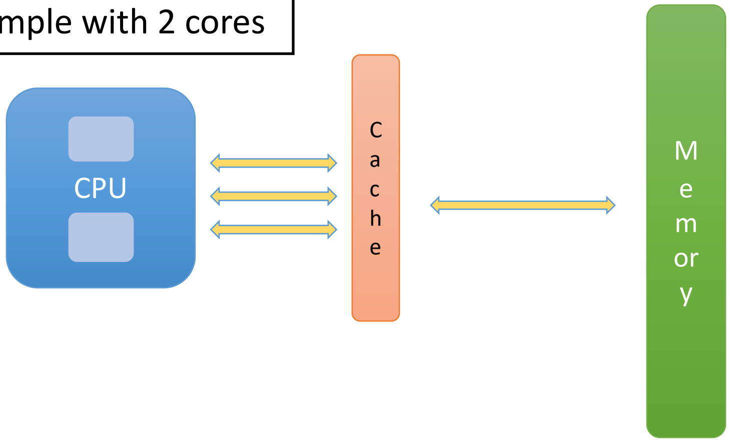
We start the loop, reading c(1) on <u>core 0</u>

1. Cache line for c(1:8) is in main memory
2. Cache line moves to L3
3. Cache line moves to L2
4. Cache line moves to L1
5. ‘somehow’ we perform the operation

We continue the loop, reading c(2)

1. There are now 4 copies of the cache line
2. We can read the data from any copy, likely reading from L1
3. ‘somehow’ we perform the operation

Example with 2 cores



Process hops to <u>core 1</u> and
we continue the loop, reading c(3)

1. There are now 4 copies of the cache line
2. 2 copies are far away, L1 and L2 of <u>core 0</u>
3. 1 copy is close: L3
4. 1 copy is far away (memory)
5. ‘somehow’ we perform the operation

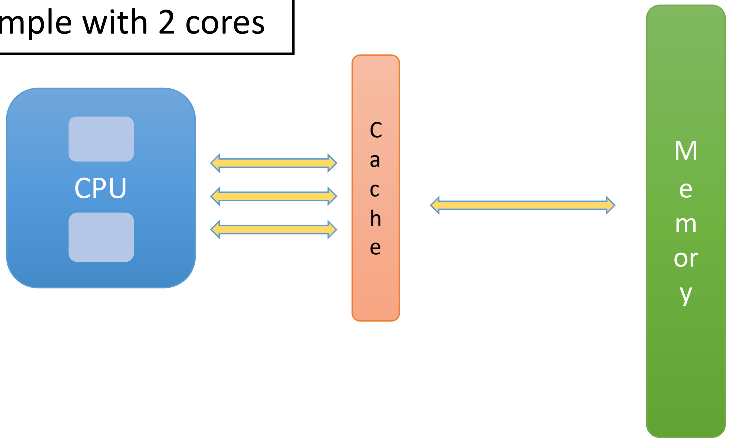
We can read from any copy (depending on availability and convenience)

MOESI Protocol

For now, only consider **writing data**
`a(i) = …`

We have to bring in cache line first (more later)
1. Cache line for a(1:8) is in main memory
2. Cache line moves to L3
3. Cache line moves to L2
4. Cache line moves to L1
5. 'somehow' we perform the operation
6. a(1) is modified
7. Modified cache line moves back to L1

Example with 2 cores



Process hops to core 1 and
we continue the loop, modifying a(2)
1. Again, there are now 4 copies of the cache line
2. Which one do we use?

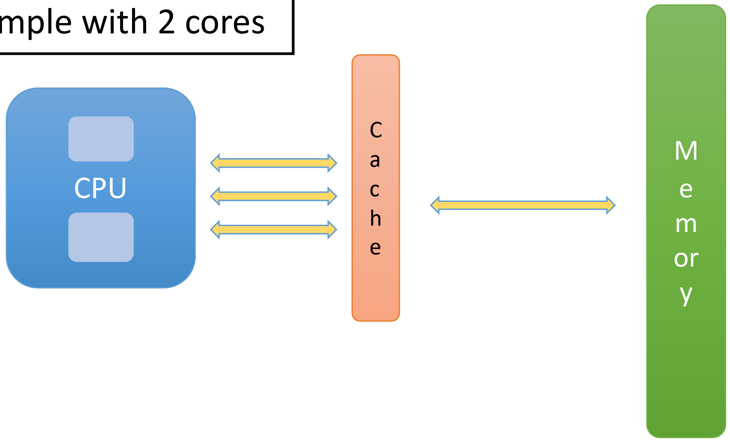
MOESI Protocol

For now, only consider **writing data**
`a(i) = …`

We have to bring in cache line first (more later)

1. Cache line for a(1:8) is in main memory
2. Cache line moves to L3
3. Cache line moves to L2
4. Cache line moves to L1
5. ‘somehow’ we perform the operation
6. a(1) is modified
7. Cache line moves back to L1

Example with 2 cores



Process hops to core 1 and
we continue the loop, modifying a(2)

1. Again, there are now 4 copies of the cache line
2. Which one do we use?
3. 3 of the 4 copies are invalid
4. Only the copy in L1 of core 0 is ‘good to go’

The MO(E)SI protocol keeps track of the validity of cache lines

MO(E)SI Protocol

A cache line can be in one of these states

- Modified (M): Only one cache has a valid copy (which has been modified). All other copies are invalid
- Owned (O): Multiple caches hold a copy, which may or may not be valid. Only one cache ‘owns’ the cache line
- Exclusive (E): Only one cache holds an exclusive copy
- Shared (S): Multiple caches hold a copy of the cache line
- Invalid (I): A copy of the cache line that is invalid

Writing
Before a process can modify data, it must own the cache line. Then the modification renders all other copies invalid

Reading
To read from a cache line, the status of the cache line has to be ‘shared’, or ‘owned’, or ‘modified’, but not ‘invalid’!

Complexity: very large
49 caches per socket (per CPU)
2 sockets: lot’s of traffic between sockets (cache snooping)

Cache coherency
MOESI protocol ensures cache coherency
Programmer not responsible for managing cache lines
Cornerstone of **any** shared-memory platform
Backbone of any thread-based parallel computing

Performance/programming implications from MO(E)SI

Parallel computing is not part of this class (see PCSE in the Spring)

Nevertheless, there is one implication that I'd like to discuss.

Imagine your code executes on 2 cores

- Two 'thingies' are executing. These 'thingies' are called threads (reference to OpenMP)
- A loop is split in two halves. Each thread is executing ½ of the loop ➔ 2x speed-up (theoretically)

Which solution will work better? (having cache-coherency in mind)

Scenario 1: Loop split into 2

Thread 0

```
do i=1, n/2
  a(i) = b(i) + c(i)
enddo
```

Thread 1

```
do i=n/2+1, n
  a(i) = b(i) + c(i)
enddo
```

Scenario 2: Loops for odd and even elements

Thread 0 (odd)

```
do i=1, n, 2
  a(i) = b(i) + c(i)
enddo
```

Thread 1 (even)

```
do i=2, n, 2
  a(i) = b(i) + c(i)
enddo
```

General syntax: Do <start>, <end>, <increment> (default increment is +1)

```
Odd  elements: do i=1, n, 2   (1, 3, 5, 7, …)
Even elements: do i=2, n, 2   (2, 4, 6, 8, …)
```

TACC

False Sharing of Cache Lines

Imagine your code executes on 2 cores

- Two ‘thingies’ are executing. These ‘thingies’ are called threads
- A loop is split in two halves. Each thread is executing ½ ➔ 2× speed-up (theoretically)

Which solution will work better? (having cache-coherency in mind)

Scenario 1: Loop split into 2

Thread 0

```
do i=1, n/2
  a(i) = b(i) + c(i)
enddo
```

Thread 1

```
do i=n/2+1, n
  a(i) = b(i) + c(i)
enddo
```

Scenario 2: Loops for odd and even elements

Thread 0

```
do i=1, n, 2
  a(i) = b(i) + c(i)
enddo
```

Thread 1

```
do i=2, n, 2
  a(i) = b(i) + c(i)
enddo
```

Scenario 2: Two threads are writing to the same cache line (at potentially the same time)
The cache lines are (potentially) bouncing around multiple times, negating any speed-up

This happens after one thread has modified a cache line (close to one core)

- Other copies are invalidated
- Before the other thread can write, the cache line has to be transferred
- Now both threads have a cache line available close by
- But once a thread is modifying its cache line, the other copy becomes ‘invalid’

False Sharing of Cache Lines

Imagine your code executes on 2 cores

- Two ‘thingies’ are executing. These ‘thingies’ are called threads
- A loop is split in two halves. Each thread is executing ½ ➔ 2× speed-up (theoretically)

Which solution will work better? (having cache-coherency in mind)

Scenario 1: Loop split into 2

Thread 0

```
do i=1, n/2
  a(i) = b(i) + c(i)
enddo
```

Thread 1

```
do i=n/2+1, n
  a(i) = b(i) + c(i)
enddo
```

Scenario 2: Loops for odd and even elements

Thread 0

```
do i=1, n, 2
  a(i) = b(i) + c(i)
enddo
```

Thread 1

```
do i=2, n, 2
  a(i) = b(i) + c(i)
enddo
```

Scenario 2: Two threads are writing to the same cache line (at potentially the same time)
The cache lines are (potentially) bouncing around multiple times, negating any speed-up

This happens after one thread has modified a cache line (close to one core)

- Other copies are invalidated
- Before the other thread can write, the cache line has to be transferred
- Now both threads have a cache line available close by
- But once a thread is modifying its cache line, the other copy becomes ‘invalid’

OK, why would anybody write code like this?

Here is a better example with a reduction

False Sharing of Cache Lines: Reduction Example

Task: Compute the sum of a vector using 2 threads

```
real, dimension(0:1) :: s   ! Partial sums, 2 elements

Thread 0                   Thread 1

s(0) = 0.                  s(1) = 0.
do i=1, n/2                do i=n/2+1, n
  s(0) = s(0) + a(i)         s(1) = s(1) + a(i)
enddo                      enddo

sum = s(0) + s(1)   ! Global sum from partial sums
```

Reduction:

- Decreasing/reducing complexity
- Calculating a result with a lower 'dimensionality' from higher 'dimensionality'

Examples

- sum of a vector (1-d ➔ scalar)
- Sum of an array ('*any*'-d ➔ scalar)

Problem: both threads update repeatedly data stored in the same cache line

So what would be a remedy?

Two threads are writing to the same cache line (at potentially the same time)
The cache lines are (potentially) bouncing around multiple times, wiping out any speed-up

This happens after one thread has modified a cache line (close to one core)

- Other copies are invalidated
- Before the other thread can write, the cache line has to be transferred
- Now both threads have a cache line available close by
- But once a thread is modifying its cache line, the other copy becomes 'invalid'

False Sharing of Cache Lines: Reduction Example

Task: Compute the sum of a vector using 2 threads

```
real, dimension(0:16) :: s   ! Partial sums, 17 elements

Thread 0                     Thread 1

s(0) = 0.                    s(16) = 0.
do i=1, n/2                  do i=n/2+1, n
  s(0) = s(0) + a(i)           s(16) = s(16) + a(i)
enddo                        enddo

sum = s(0) + s(16)   ! Global sum from partial sums
```

Move your partial sums to addresses that are guaranteed not to be on the same cache line

Note that OpenMP provides other mechanisms to avoid ‘false sharing’
However, sometimes it is useful to intervene manually

Example: Stencil update
If you use multiple threads you may put the domain boundaries on top of a cache line boundary

Recap

Topics that we have addressed so far

Data streams
Pipelining
Caches

- Why?
- Cache blocking (software)
- Address mapping
- Eviction policy (FIFO, etc.)
- Associativity
  - fully associative, set-associative, direct mapping
- Storage efficiency (for addresses)
  - Cache lines
- Cache coherency (MOSI protocol)
- Shared-memory architecture
- False sharing

All these topics come with these questions:

- What is it?
- Why do we need it?
- How is it implemented?
- Is there a specific trick?
- How does this increase concurrency?

One more hardware feature to cover (unrelated to caches)

And then we will start looking into ‘writing fast code’

Overarching idea

Single transaction is limited in speed

Increasing **concurrency** is our goal

Memory transactions and (floating point) operations