Scientific and Technical Computing

Hardware and Code Optimization

Lars Koesterke

UT Austin, 10/1/19 & …

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

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

<u>CPU</u>

1. Pipelined operation

System designed to get 1 opc

<u>Memory</u>

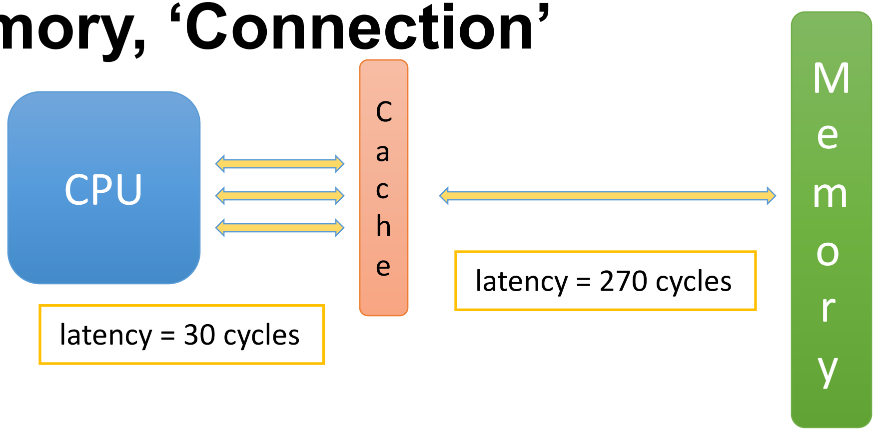
1. Data streams

System designed to support 1 wpc (for one row)

<u>Caches</u>

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

Concurrency!

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

CPU
1. Pipelined operation

System designed to get 1 opc

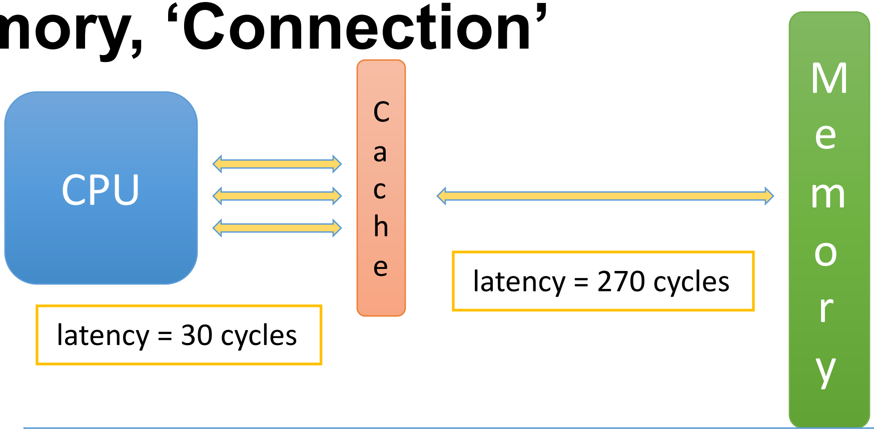
Memory
1. Data streams

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

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 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'

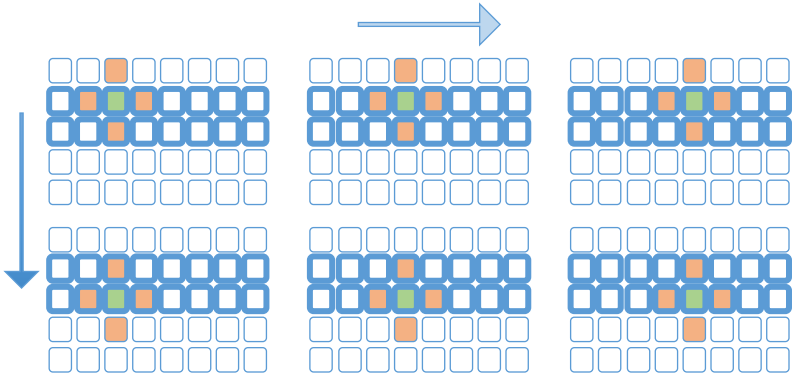
Discussion: Working around the size limitation

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



```
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

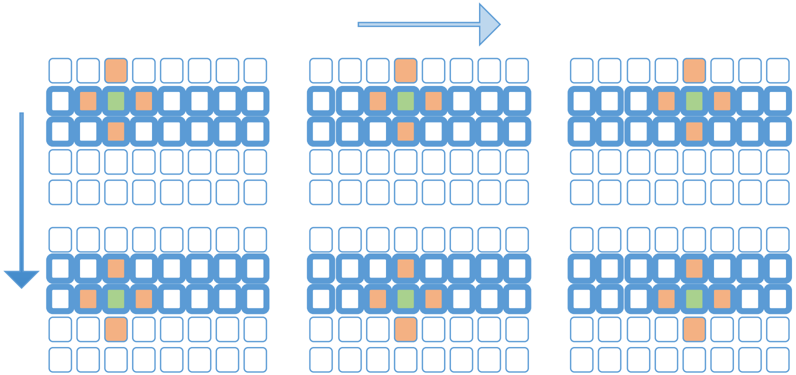
Example:

n=500, cache size=300 words

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

So 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:

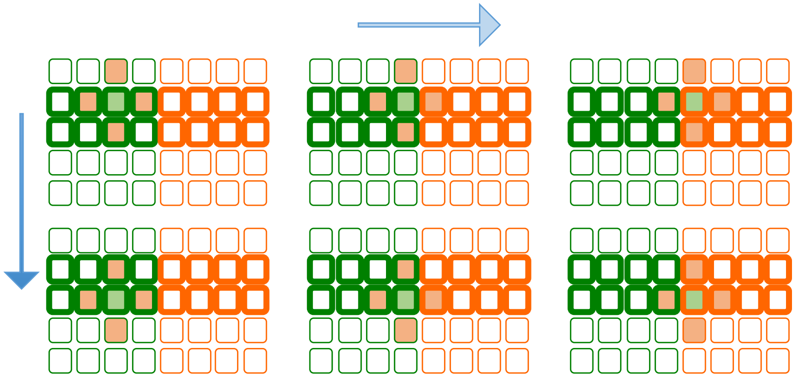
n=500, cache size=300 words

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

So what do we do when we reach 'i=100'

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

So 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
```

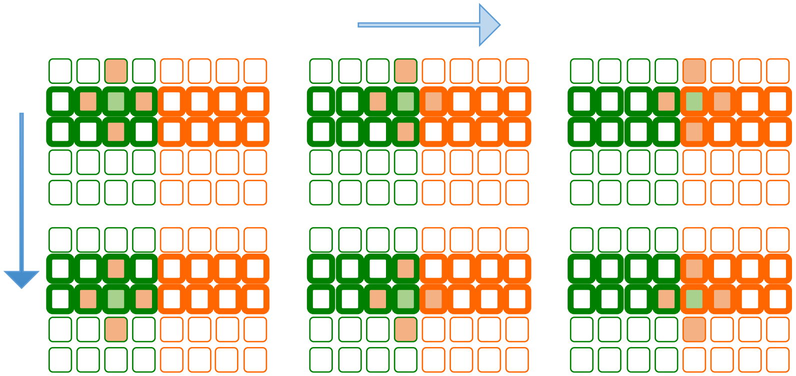
Discussion: Working around the size limitation

Example:

n=500, cache size=300 words

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

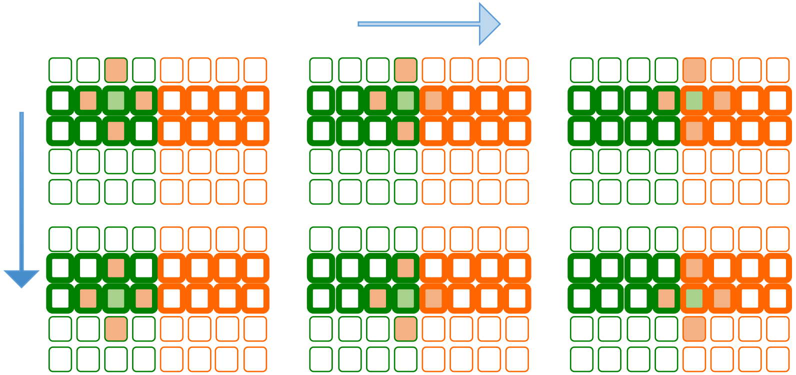
- What is the width of a strip?
- How many strips?
- How many loops in the code?



```
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
```

TACC

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 strip? 100
- How many strips? 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
```

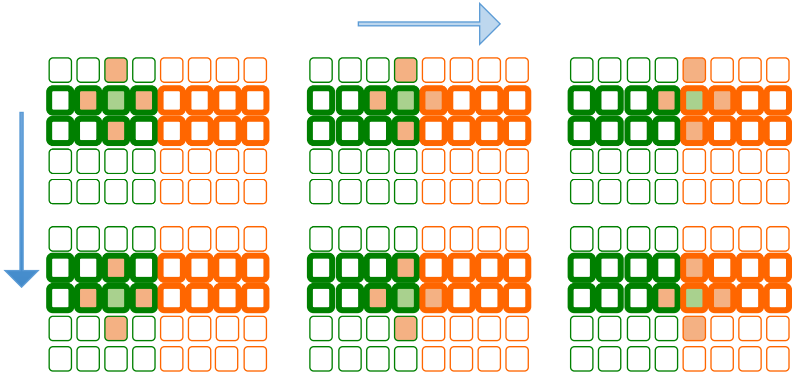
Discussion: Working around the size limitation

Example:

n=500, cache size=300 words

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

- What is the width of a strip? 100
- How many strips? 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 left to right first (**x**-direction)
We go in **y**-direction second
Hence left to right is the inner loop. Loop index is '**i**'

The 'fast' loop, i.e. the inner loop 'exceeds' to 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 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
```

TACC

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
```

TACC

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 way(!) 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 way(!) 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
```

TACC

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 were the data came from, i.e. the original address
- The cache stores the actual value

These are the addresses, not the actual data
Assume that in this example we encounter the addresses
Consequtive and in order.
So address ‘1’, and its content, was stored first,
i.e. 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

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

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

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

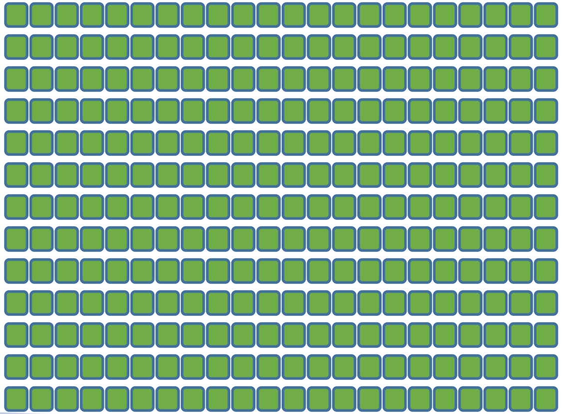
Limitations: The Cache can be quite large

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

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

Conflicting goals

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



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)

So how can we devise a fast strategy?

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

TACC