Scientific and Technical Computing

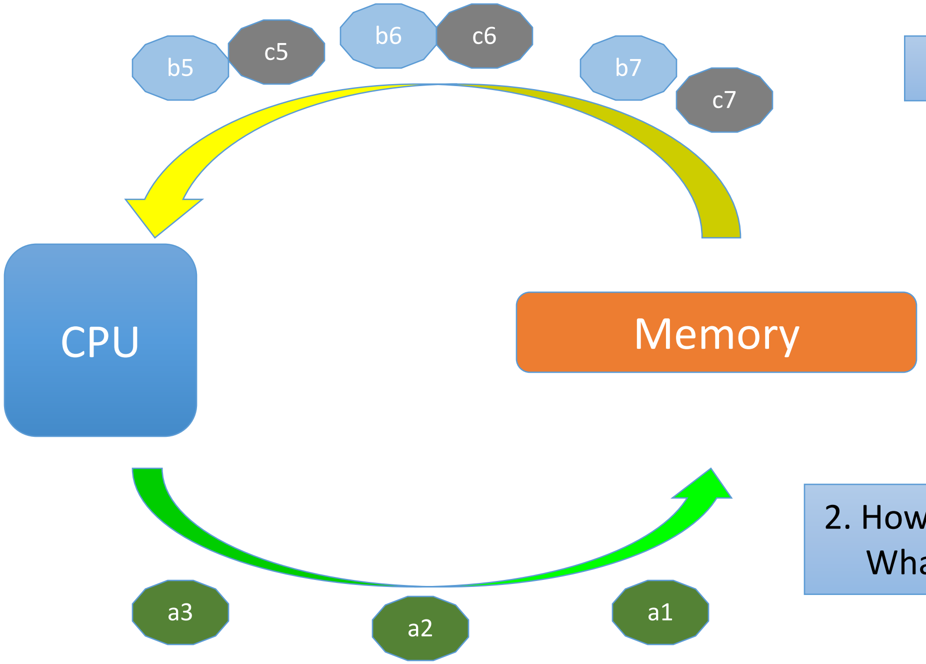
Hardware and Code Optimization

Lars Koesterke

UT Austin, 10/7/21 & 10/12/21

Remember this? Data streams

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



1. How much data has to be 'en route'?

Operation:	3 cycles
Data transfer:	300 cycles
Ratio:	1/100

100 elements of a, b, and c

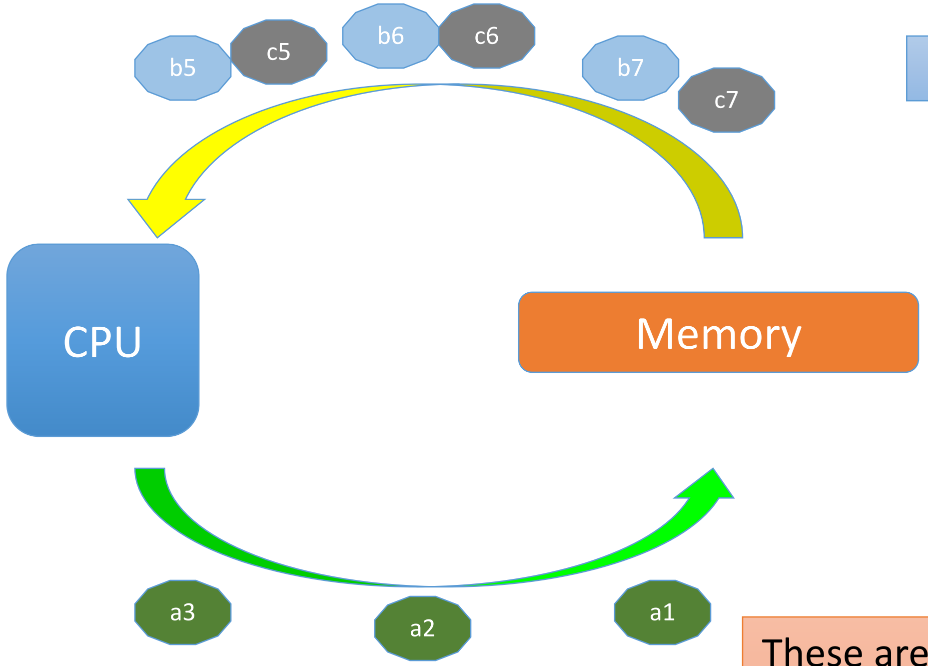
2. How many results (a) do we get per cycle?
What is the speedup?

1 result every 3 cycles
Speedup is 200×

What do the designations 'a3', 'b5', 'c7' now stand for?

Remember this? Data streams

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



1. How much data has to be 'en route'?

Operation: 3 cycles
Data transfer: 300 cycles
Ratio: 1/100

100 elements of a, b, and c

2. How many results (a) do we get per cycle?
What is the speedup?

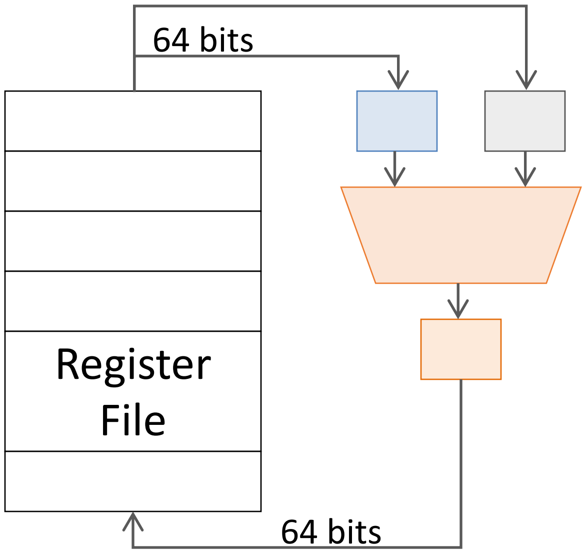
1 result every 3 cycles
Speedup is 200×

What do the designations 'a3', 'b5', 'c7' <u>now</u> stand for?

These are now 'names/designations' of whole cache lines (512 bit)
And we will use this to our advantage to **increase computational speed**

Scalar Hardware

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



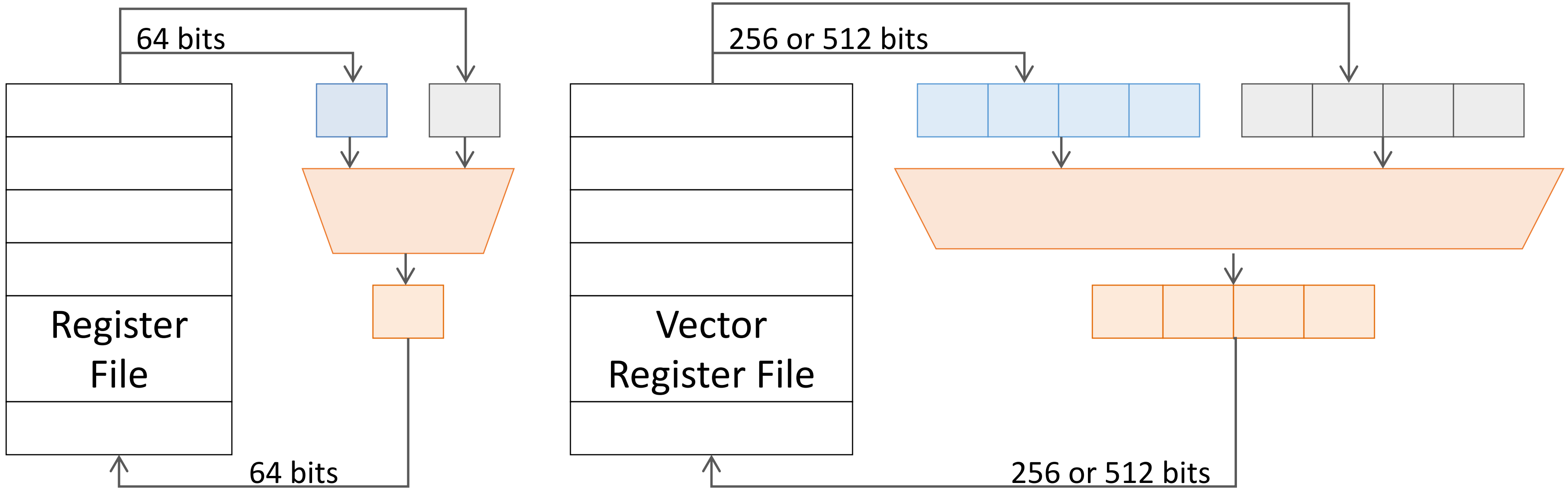
Scalar Unit

Input: 2 words (single or double precision)
Output: 1 word
Operations: 1 operation ➔ 1 result

Vector Hardware

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

Example for vector length = 4 words



Scalar Unit

Input: 2 words (single or double precision)
Output: 1 word
Operations: 1 operation ➔ 1 result

Vector Unit

Input: 2 cache lines
Output: 1 cache line
Operations: 1 operation ➔ 8/16 results (dp/sp)

Vectorization

The whole process is called vectorization

- Vector registers
- Vector instructions
- Loading a cache line into vector register with one instruction
- Operating on vector registers with one instruction

Using vector instructions

- Compiler creates vector instructions when possible (and
- Compiler creates scalar instructions when necessary
- Programmers help by
  - Writing vectorizable code
  - Adding hints to the source code (OpenMP)

Status

- Cache line: 512 bits
- Vector width/length: 512 bits (width of the vector register)
- In the past: vector length shorter than cache line

Reminder

All 'concurrency' 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?

Overarching idea

Single transaction is limited in speed

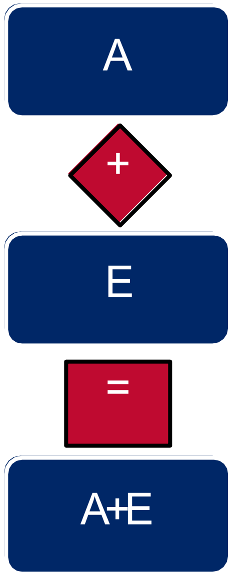
Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

Vectorization

Basics

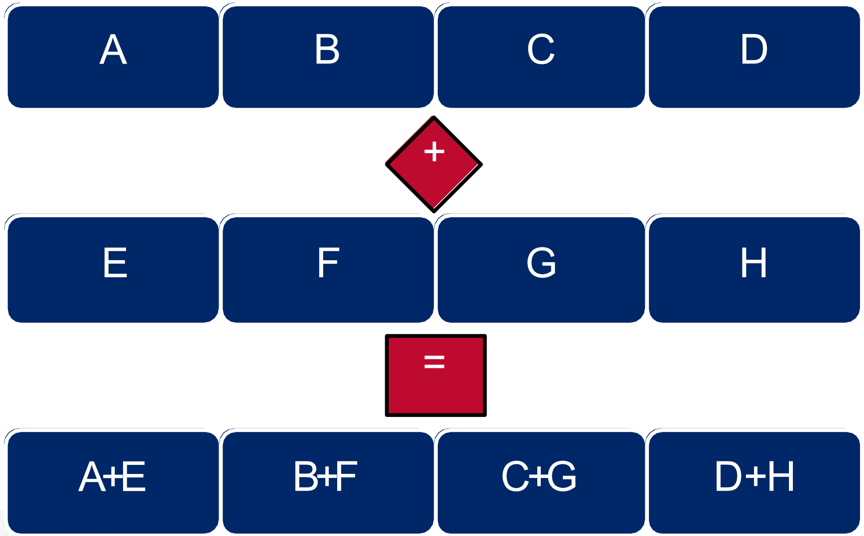
- Cores have registers
- Data must be moved from cache/memory into registers before operating on
- To add 2 floating point (fp) numbers:
    1. Move first fp ‘A’ from cache to a register
    2. Move second fp ‘E’ from cache to a different register
    3. Add fp’s and place result in yet a different register
    4. Move result to cache/memory
- Frequencies are limited so a single fp operation can only go so fast
- Why not move and add several fp’s simultaneously?
    - SIMD: Single Instruction Multiple Data



Vectorization

Make registers wider → vector registers

- Same motivations as multicore: more compute at comparable power
- Increase # computations by increasing number of ops per cycle



Evolution of SIMD Hardware

2001 – 2017: Vector length has increased by a factor of 4

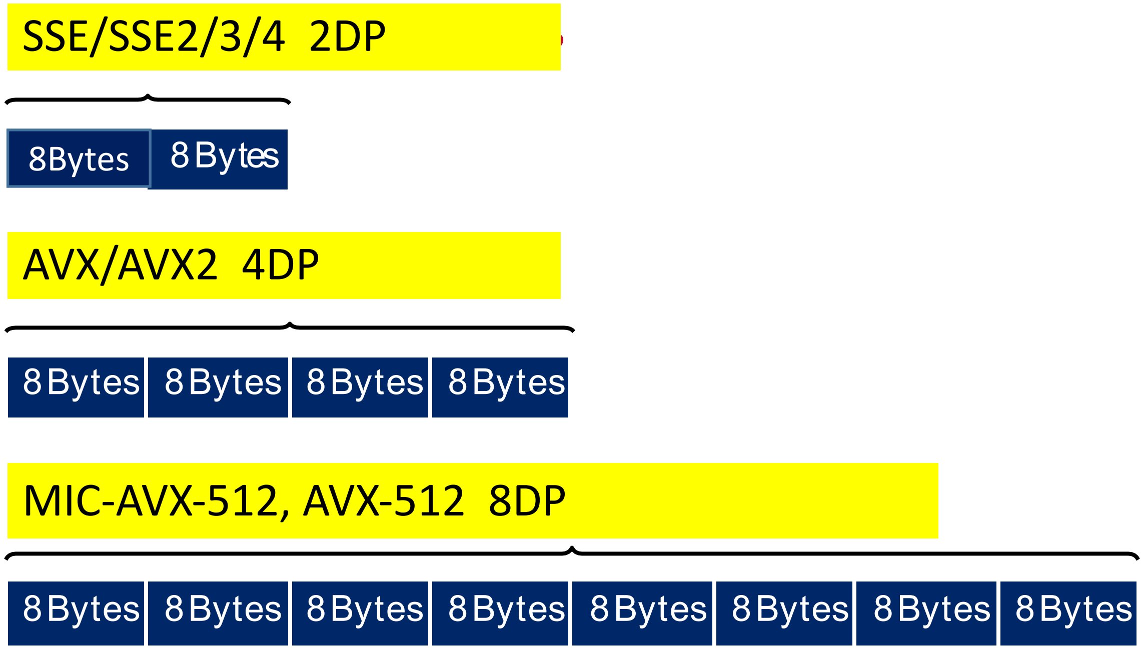
2017: Vector length = length of a cache line

Year	Width (bits)	ISA Name	Data (register names)
1997	80	x87 + MMX	float+Integer
1999	128	SSE1	SP FP SIMD (xMM0-8)
2001	128	SSE2	DP FP SIMD (xMM0-8)
2004	128	SSE3	DP FP SIMD (xMM0-8)
2006	128	SSE4	DP FP SIMD (xMM0-8)
2010	256	AVX	DP FP SIMD (yMM0-16)
2013	256	AVX2	DP FP SIMD (yMM0-16)
2016	512	MIC-AVX512, COMMON-AVX512	DP FP SIMD (zMM0-32)
2017	512	CORE-AVX512	DP FP SIMD (zMM0-32)

Stampede2 and Frontera

Vector Registers

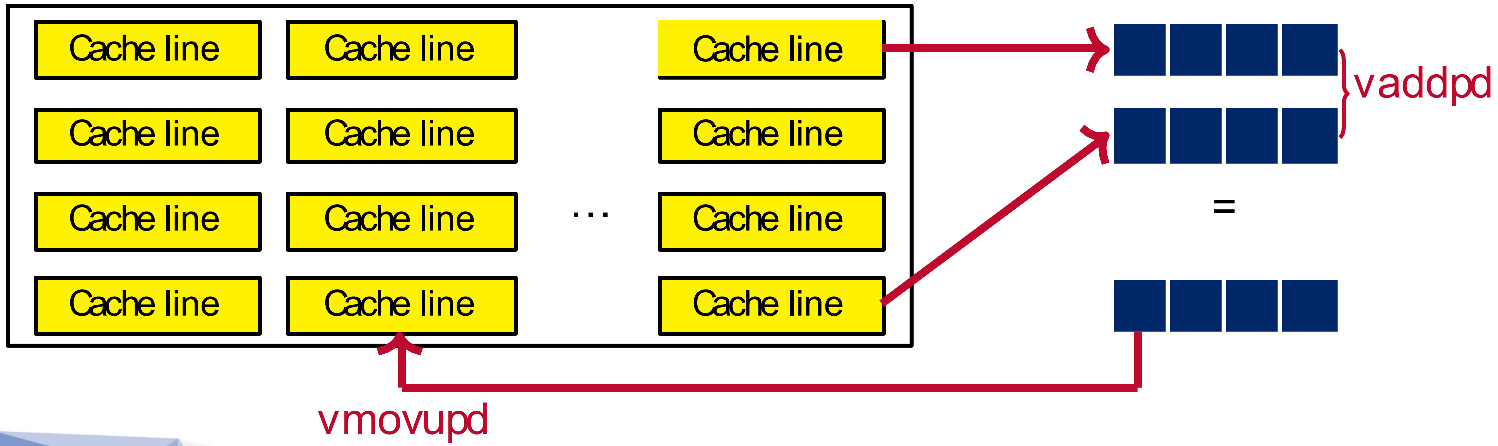
- Different types of processors have different width vector registers
- SP/DP are 32b/64b (4B/8B)
- Vector instructions are required to use vector registers



Vector Instructions

Vectorized code uses vector instructions

- Vector instructions act on multiple data elements
- Vector instructions exist for moving data and operating on data



Vectorization

Example: vector length=4

Start, End, Increment

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

Compiler →

```
do i=1, n, 4
  a(i+0) = b(i+0) + c(i+0)
  a(i+1) = b(i+1) + c(i+1)
  a(i+2) = b(i+2) + c(i+2)
  a(i+3) = b(i+3) + c(i+3)
enddo
```

Compiler unrolls the loop

Vectorization

Example: vector length=4

```
do i=1, n
   a(i) = b(i) + c(i)
   d(i) = e(i) + f(i)
enddo
```

Compiler

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   d(i+0) = e(i+0) + f(i+0)
   a(i+1) = b(i+1) + c(i+1)
   d(i+1) = e(i+1) + f(i+1)
   a(i+2) = b(i+2) + c(i+2)
   d(i+2) = e(i+2) + f(i+2)
   a(i+3) = b(i+3) + c(i+3)
   d(i+4) = e(i+3) + f(i+3)
enddo
```

Compiler unrolls the loop

The compiler can change the order of statements if it's correct to do so

```
do i=1, n, 4
   a(i+0) = b(i+0) + c(i+0)
   a(i+1) = b(i+1) + c(i+1)
   a(i+2) = b(i+2) + c(i+2)
   a(i+3) = b(i+3) + c(i+3)
   d(i+0) = e(i+0) + f(i+0)
   d(i+1) = e(i+1) + f(i+1)
   d(i+2) = e(i+2) + f(i+2)
   d(i+3) = e(i+3) + f(i+3)
enddo
```

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes

But why exactly?

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes
Loop iterations are independent

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

```
do i=2, n-1
  a(i) = a(i-1) + a(i+1)
end do
```

Yes

Loop iterations are independent

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes
Loop iterations are independent

```
do i=2, n-1
  a(i) = a(i-1) + a(i+1)
end do
```

No
Loop iterations are not independent

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes
Loop iterations are independent

```
do i=2, n-1
  a(i) = a(i-1) + a(i+1)
end do
```

No
Loop iterations are not independent

```
for ( int i=0; i<n; i++ ) {
  temp = a[i] + 2.;
  a[i] = b[i];
  b[i] = temp;
}
```

How about this one?
There is, sort of, a dependency

TACC

Vectorizable code?

```
for ( int i=0; i<n; i++ ) {
  a[i] = b[i] + c[i];
}
```

Yes
Loop iterations are independent

```
do i=2, n-1
  a(i) = a(i-1) + a(i+1)
end do
```

No
Loop iterations are not independent

```
for ( int i=0; i<n; i++ ) {
  temp = a[i] + 2.;
  a[i] = b[i];
  b[i] = temp;
}
```

Yes
Compiler resolves dependencies for scalars like temp, and also takes care of unnamed constants

TACC

Efficiency of Vectorization

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

Assume double precision
Assume efficient data streaming and pipelining

How many cycles per operation?
(Assuming pipelining)

What is the total bandwidth?
What is the 'effective' total bandwidth?
How many results per cycle?

TACC

Efficiency of Vectorization

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

Assume double precision
Assume efficient data streaming and pipelining

How many cycles per operation?	1
What is the total bandwidth?	3×8 wpc
What is the 'effective' total bandwidth?	3×4 wpc
How many results per cycle?	4

Efficiency of Vectorization

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

Assume double precision
Assume efficient data streaming and pipelining

How many cycles per operation? 1

What is the total bandwidth? 3×8 wpc
What is the 'effective' total bandwidth? 3×4 wpc
How many results per cycle? 4

So what happens to the 4 elements of the vector unit that we don't need?

Vectorization and Masks

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

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)

+

c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)

=

a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

Vectorization and Masks

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

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)	
+								
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)	
0	1	0	1	0	1	0	1	Mask
=								
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)	a(1), a(3), a(5), a(7) unchanged

Vectorization and Masks

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

Complete cache lines are loaded

Unwanted results are either not calculated or not stored back to register

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)

+

c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)

0	1	0	1	0	1	0	1	Mask

=

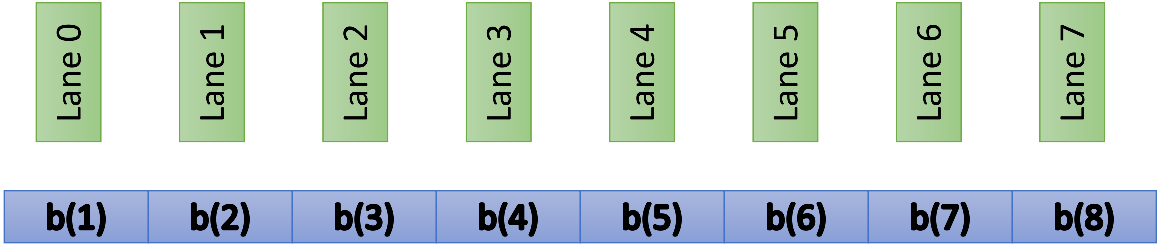
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)	a(1), a(3), a(5), a(7) unchanged

Vector Lanes

Double precision: 8 vector lanes

Single precision: 16 vector lanes

Load 8 consecutive elements in memory
to
8 consecutive elements in memory



Vector Lanes

All elements!

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

Complete cache lines are loaded
Unwanted results are not stored back to register

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)

+

c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)

1	1	1	1	1	1	1	1

=

a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

Load b(1:8)
Load c(1:8)
Compute: add
Store a(1:8) – full mask

Total: 4 instructions
8 results

Mask: All results copied back

TACC

Vector Lanes

Every other element!

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

Complete cache lines are loaded

Unwanted results are either not calculated or not stored back to register

How many instructions and results?

	b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)
+								
	c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)
Mask	0	1	0	1	0	1	0	1
=								
	a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

a(1), a(3), a(5), a(7) unchanged

TACC

Vector Lanes

Every other element!

```fortran
do i=1, n
  a(2*i) = b(2*i) + c(2*i)
end do
```

Complete cache lines are loaded

Unwanted results are either not calculated or not stored back to register

Load b(1:8)
Load c(1:8)
Compute: add – with mask
Store a(1:8)

Total: 4 instructions
4 results

	b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)
+								
	c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)
Mask	0	1	0	1	0	1	0	1
=								
	a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

a(1), a(3), a(5), a(7) unchanged

Vector Lanes

Extreme case: every 8th element!

```
do i=1, n
  a(8*i) = b(8*i) + c(8*i)
end do
```

<u>Complete</u> cache lines are loaded

Unwanted results are either not calculated or not stored back to register

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)
+							
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)
0	0	0	0	0	0	0	1
=							
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

Load b(1:8)
Load c(1:8)
Compute: add – with mask
Store a(1:8)

<u>Total: 4 instructions</u>
<u>1 results</u>

Mask

a(1), a(2), … unchanged

Vector Lanes

Extreme case: every 8th element!

Why is ’a’ also loaded, even if we only write to it?

```
do i=1, n
  a(8*i) = b(8*i) + c(8*i)
end do
```

Complete cache lines are loaded
and written back

Unwanted results are either not calculated or not stored back to register

b(1)	b(2)	b(3)	b(4)	b(5)	b(6)	b(7)	b(8)
+							
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)	c(7)	c(8)
0	0	0	0	0	0	0	1
=							
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)	a(8)

Load b(1:8)
Load c(1:8)
Load a(1:8)
Compute: add – with mask
Store a(1:8)

Total: 5 instructions
1 results

a(1), a(2), … unchanged
a(1) to a(8) is written back

Strided Memory Access

```
do i=1, m
  a(1*i) = b(1*i) + c(1*i)
end do
```

Stride 1 access

```
do i=1, m
  a(2*i) = b(2*i) + c(2*i)
end do
```

Stride 2 access

Which access is best?

Which one is worse?

```
do i=1, m
  a(8*i) = b(8*i) + c(8*i)
end do
```

Stride 8 access

```
do i=1, m
  a(n*i) = b(n*i) + c(n*i)
end do
```

Stride ‘n>8’ access

Strided Memory Access

```
do i=1, m
  a(1*i) = b(1*i) + c(1*i)
end do
```

Stride 1 access

```
do i=1, m
  a(2*i) = b(2*i) + c(2*i)
end do
```

Stride 2 access

```
do i=1, m
  a(8*i) = b(8*i) + c(8*i)
end do
```

Stride 8 access

```
do i=1, m
  a(n*i) = b(n*i) + c(n*i)
end do
```

Stride ‘n>8’ access

Which access is best?

Lower strides are better

Stride-1 is best

Which one is worse?

Higher strides are worse

Stride-8, and stride-n are worst

Lower ‘effective’ memory bandwidth

Lower number of results per numerical vector operation

Strided Memory Access

```
do i=1, m
  a(1*i) = b(1*i) + c(1*i)
end do
```

Stride 1 access

```
do i=1, m
  a(2*i) = b(2*i) + c(2*i)
end do
```

Which access is best?

But things are a bit more complicated

Again, we have to do back to ‘data streams’

Higher strides are worse
Stride-8, and stride-n are worst

```
do i=1, m
  a(8*i) = b(8*i) + c(8*i)
end do
```

Stride 8 access

```
do i=1, m
  a(n*i) = b(n*i) + c(n*i)
end do
```

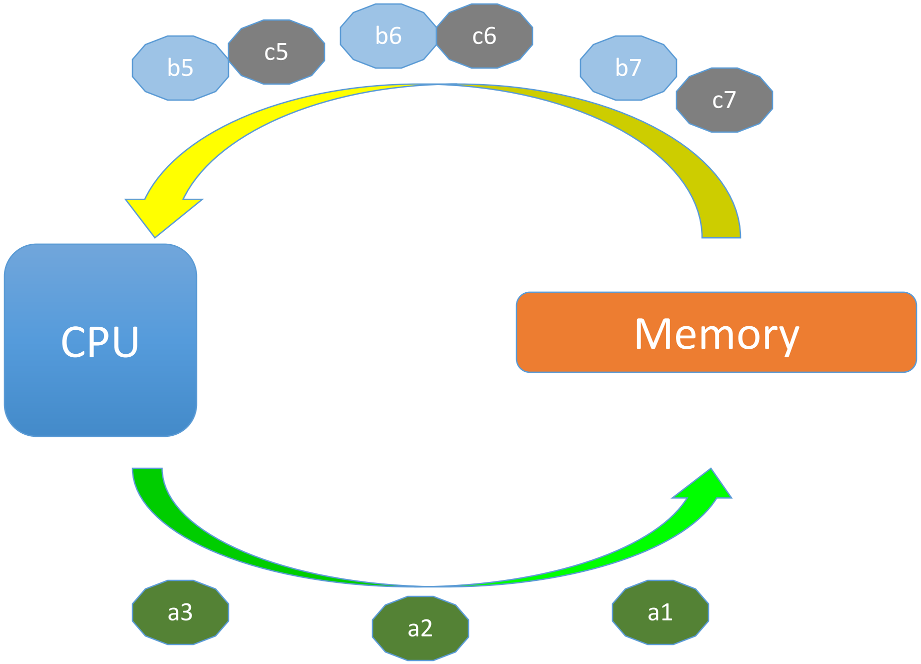
Stride ‘n>8’ access

Lower ‘effective’ memory bandwidth

Lower number of results per numerical operation

Remember this? Data streams

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



Let's consider input streams only

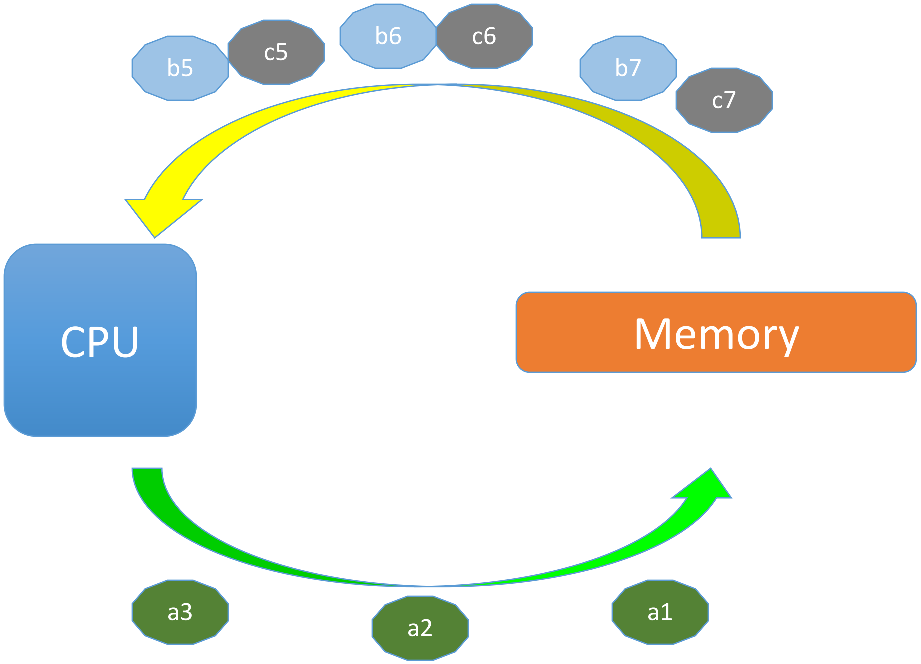
Data is streaming between Memory and CPU

Goal: Results are calculated every cycle

How do 'we' know what to put in the data stream?

Remember this? Data streams

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



Let's consider input streams only

Data is streaming between Memory and CPU

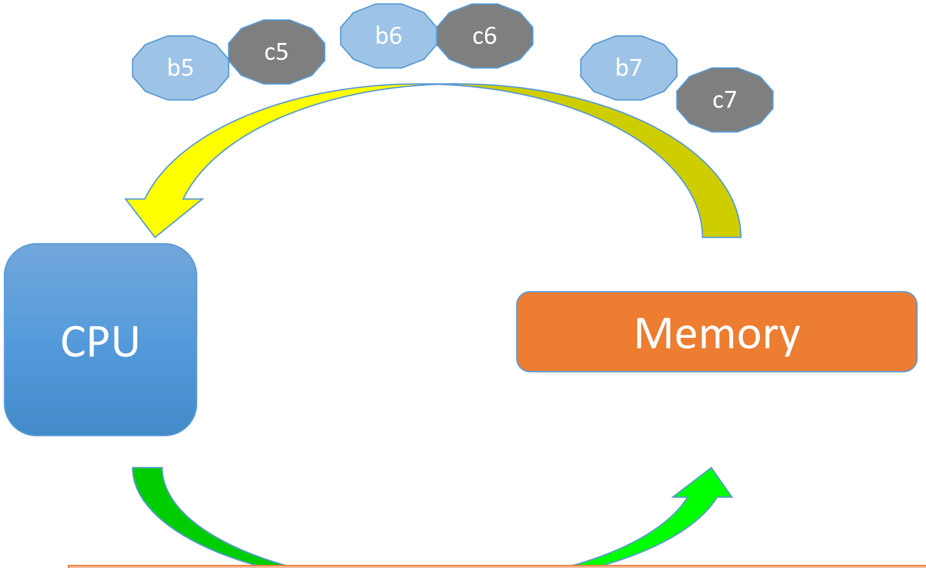
Goal: Results are calculated every cycle

How do 'we' know what to put in the data stream?

How does the 'computer' know what to put into the data stream?

Remember this? Data streams

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



Let's consider input streams only

Data is streaming between Memory and CPU

Goal: Results are calculated every cycle

How do 'we' know what to put in the data stream?

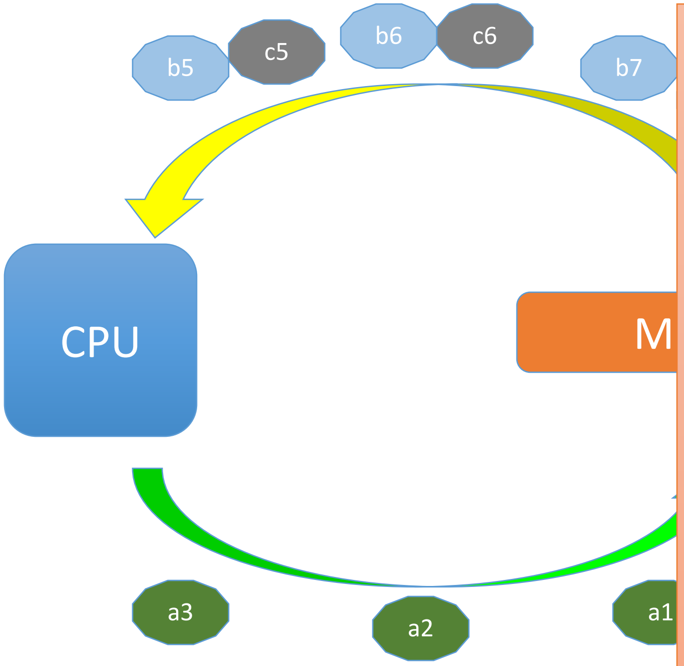
How does the 'computer' know what to put into the data stream?

The computer does not know the code (or the intention of the code) and cannot infer anything from the pattern in the code.

However, it can analyze the pattern from previous memory/data access.
For example, data is requested cache line by cache line.

Prefetching

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



The computer does not know the code and cannot infer anything from the pattern in the code.

However, it can analyze the pattern from previous memory access. For example, data is requested cache line by cache line.

This is called prefetching
Prefetch instructions are added either

- during execution by the hardware (hardware prefetching)
- or to the assembly code by the compiler (software prefetching)

Typical defaults (x86 architecture):

- Software prefetching is off
- Hardware prefetching is on

Prefetching fills the data streams
Unwanted data (that is not useful) is ignored

TACC

Random Memory Access

```
do i=1, m
  a(1*i) = b(1*i) + c(1*i)
end do
```

```
do i=1, m
  a(2*i) = b(2*i) + c(2*i)
end do
```

```
do i=1, m
  a(8*i) = b(8*i) + c(8*i)
end do
```

```
do i=1, m
  a(n*i) = b(n*i) + c(n*i)
end do
```

```
do j=1, m
  i    = index(j)
  a(i) = b(i) + c(i)
end do
```

Can we have prefetching in this situation?

Random Memory Access

```
do i=1, m
  a(1*i) = b(1*i) + c(1*i)
end do
```

```
do i=1, m
  a(2*i) = b(2*i) + c(2*i)
end do
```

```
do i=1, m
  a(8*i) = b(8*i) + c(8*i)
end do
```

```
do i=1, m
  a(n*i) = b(n*i) + c(n*i)
end do
```

```
do j=1, m
  i    = index(j)
  a(i) = b(i) + c(i)
end do
```

Random access is the worst
Data streams cannot be constructed from access pattern

Data access ‘suffers’ the full latency
~300 cycles between request and arrival of cache line

Pitfalls: Multi-dimensional arrays

Fortran and C behave differently
Fortran: Column major
C: Row major

Stride-1 data access

```
for ( int j=0; j<n; j++ )   {
  for ( int i=0; i<n; i++ ) {
    a[j][i] = b[j][i] + c[j][i];
}}

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

Stride-n data access

```
for ( int i=0; i<n; i++ )   {
  for ( int j=0; j<n; j++ ) {
    a[j][i] = b[j][i] + c[j][i];
}}

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

Design decision on day 1
malloc/allocate your arrays such that later in the ‘hot’ loops the access is low-stride

‘hot’ = where you spend most of the execution time

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = (double *)malloc(sizeof(double)*N);
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

Can this code be vectorized?

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = (double *)malloc(sizeof(double)*N);
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

Can this code be vectorized?

Yes, because the loop iterations are independent

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = a+1;
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

Can this code be vectorized?

What is the problem here?

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = a+1;
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

```
a[i] = a[i+1] + c[i]
```

Can this code be vectorized?

What is the problem here?

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = a+1;
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

```
a[i] = a[i+1] + c[i]
```

Can this code be vectorized?

What is the problem here?

a and b are pointing to the same memory addresses

Pitfall: Pointers

C code: Guarantee non-overlapping pointers

```
  a = (double *)malloc(sizeof(double)*N);
  b = (double *)malloc(sizeof(double)*N);
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float * restrict a, float * restrict b, float * restrict c, float d, int
n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

For pointers with the 'restrict' keyword, the compiler can assume that there is no overlap

Or compile with option '`-fno-alias`'

```
icc -fno-alias source.c
```

Pitfall: Pointers

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = (double *)malloc(sizeof(double)*N);
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

Assume: no ‘restrict’ keyword and no special compiler option

When compiling source file with function ‘add’
How can the compiler be sure whether there is overlap or not?

TACC

Pitfall: Pointers

```
Overlap: if (b>a+n or b+n<a) then 'no-overlap'
```

C code

```
  a = (double *)malloc(sizeof(double)*N);
  b = (double *)malloc(sizeof(double)*N);
  c = (double *)malloc(sizeof(double)*N);

  add(a,b,c,N);

void add(float *a, float *b, float *c, float d, int n)
{for (int i=0; i<n; ++i) { a[i] = b[i] + c[i]; }}
```

When compiling source file with function 'add'
How can the compiler be sure whether there is overlap or not?

Compiler may create 2 versions, with and without vectorization
Upon entry it checks whether intervals [a,a+n], [b,b+n], and [c,c+n] do 'overlap'
When there is no overlap at run-time, the vectorized version may be used

Recap: Vectorization & Prefetching

Vectorization: When, how, why?
Under what circumstances can loops be vectorized?
Vector lanes
Strided data access
Vectorization efficiency
Data transfer efficiency
Prefetching
Multi dimensional arrays: strice-1 v. stride-n
Pointers and 'array overlap'; mostly in C

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?

Overarching idea

Single transaction is limited in speed

Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

TACC

Example

Assume ‘n’ is a large number

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

Can we vectorize this loop?

TACC

Example

Assume ‘n’ is a large number

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

do i=2, n
  b(i-1) = b(i-1) * c(i)
enddo
```

Can we vectorize these loops?

Yes, now both loops have independent Loop iterations

Example

Assume ‘n’ is a large number

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

do i=2, n
   b(i-1) = b(i-1) * c(i)
enddo
```

Can we vectorize these loops?

Yes, now both loops have independent Loop iterations

But what else is going on

Have we changed our data streams?

Example

```
n=1000; ns=10

do iout=1, n/ns
  is = …
  ie = …

  do i=is, ie
    a(i)   = b(i) + c(i)
  enddo

  do i=is, ie
    b(i-1) = b(i-1) * c(i)
  enddo
enddo
```

Can we vectorize these loops?

Yes, now both loops have independent
Loop iterations

<u>Cache blocking</u>

Re-use elements of ‘b’ before
they are evicted from the cache

TACC

Code Tuning: When and What?

Tuning your code manually

- When, where, how?
- How to balance this: readable/maintainable code vs. fast code

'Hot' routines vs. 'rest of the code' --- 'inner' vs. 'outer' routines

1. 'hot' routines: emphasis on execution speed
    1. Do what is necessary, even if it leads to code bloat und less readable code
       Some transformations are better left to the compiler, though
    2. Example: Manual cache blocking
2. 'rest' of the code': emphasis on readable/maintainable code

The 'inner' and 'outer' routines are linked

Some optimizations in the inner routines can only be done if the outer part is properly designed

Good memory layout will lead to

1. Memory-access friendly 'hot' loops
2. Code where hardware features like caches, vectorization, and pipelining (etc.) can be applied

Bad memory layout cannot be changed

1. Design process: think through your code from top-to-bottom, and bottom-to-top