Scientific and Technical Computing

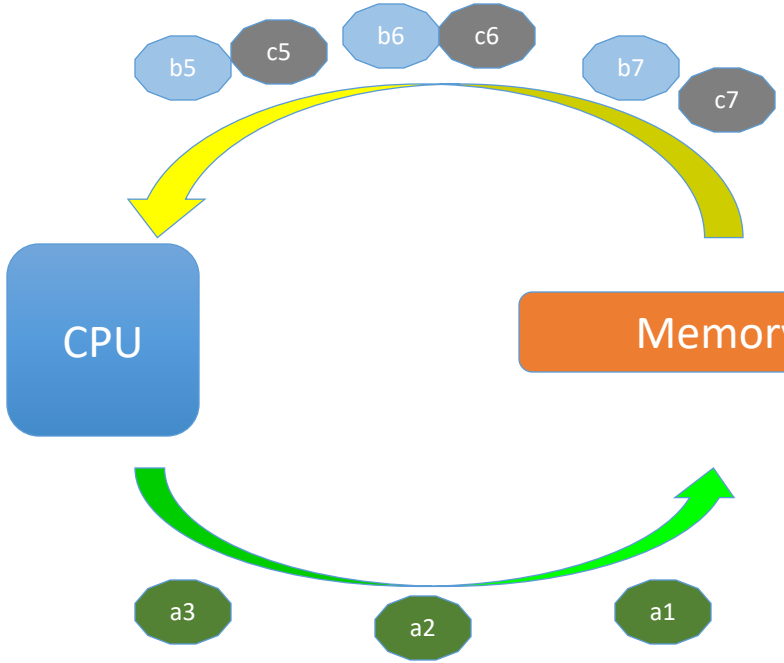
Hardware and Code Optimization

Lars Koesterke

UT Austin, 10/13/20 &

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

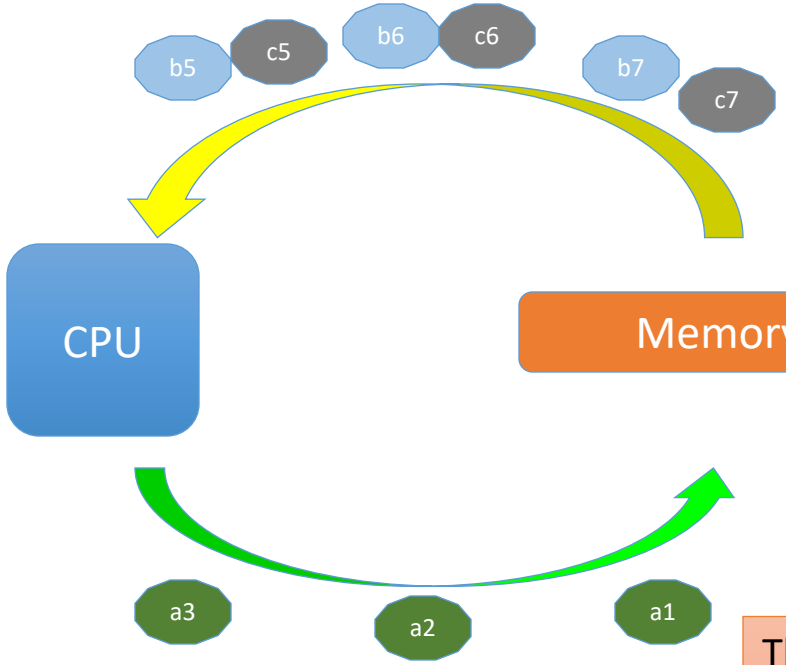
2. How many results (a) do we get per cycle? What is the speedup?

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?

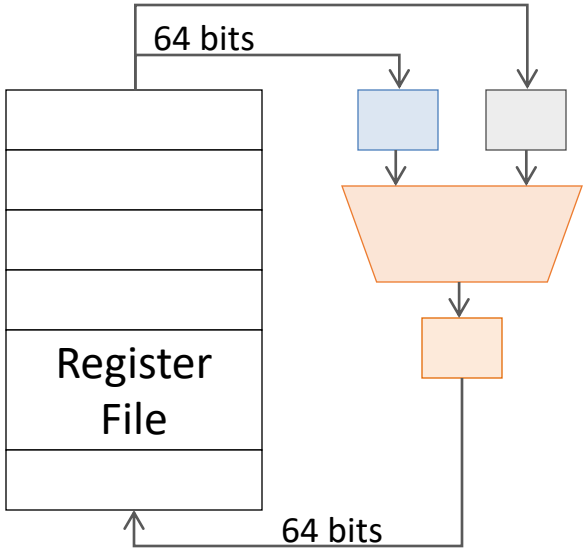
1 result every 3 cycles
Speedup is 200×

What do the designations 'a3', 'b5', 'c7' now 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)



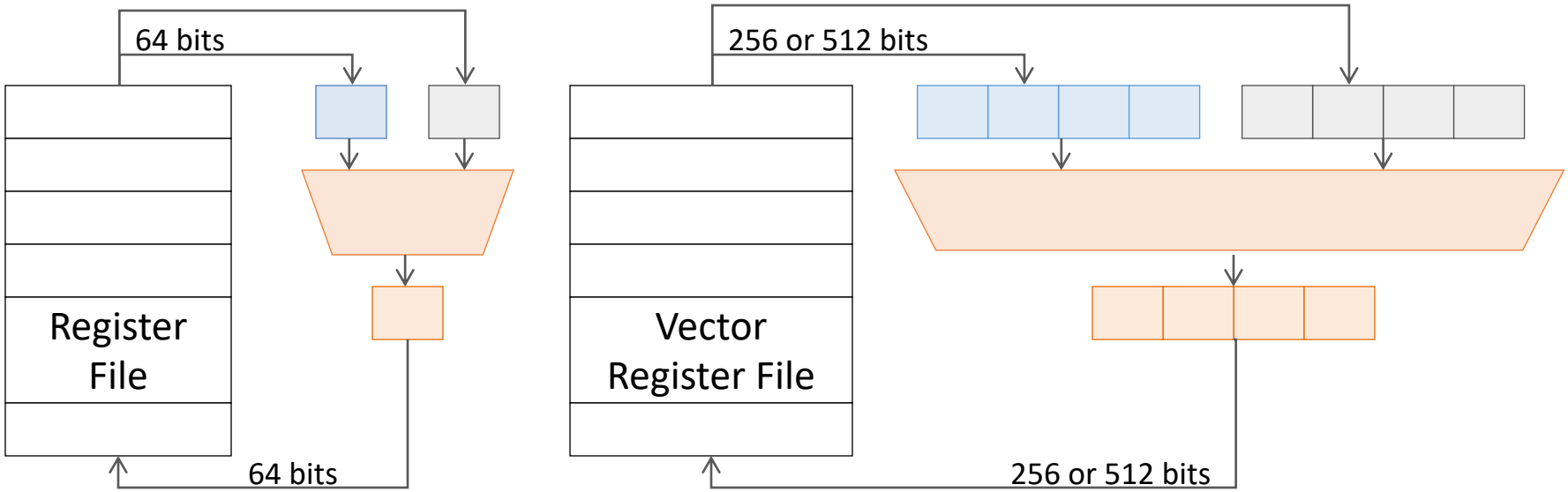
Scalar Unit

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

<u>All 'concurrency' topics come with these questions:</u>

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

<u>Overarching idea</u>

Single transaction is limited in speed

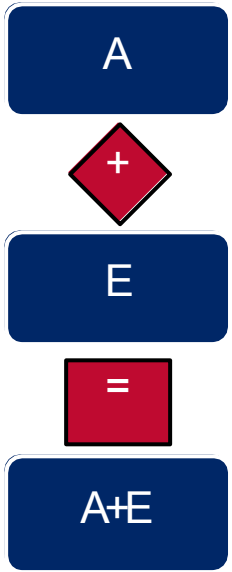
Increasing **concurrency** is our goal

Memory transactions and (floating point) operations

Vectorization

Basics

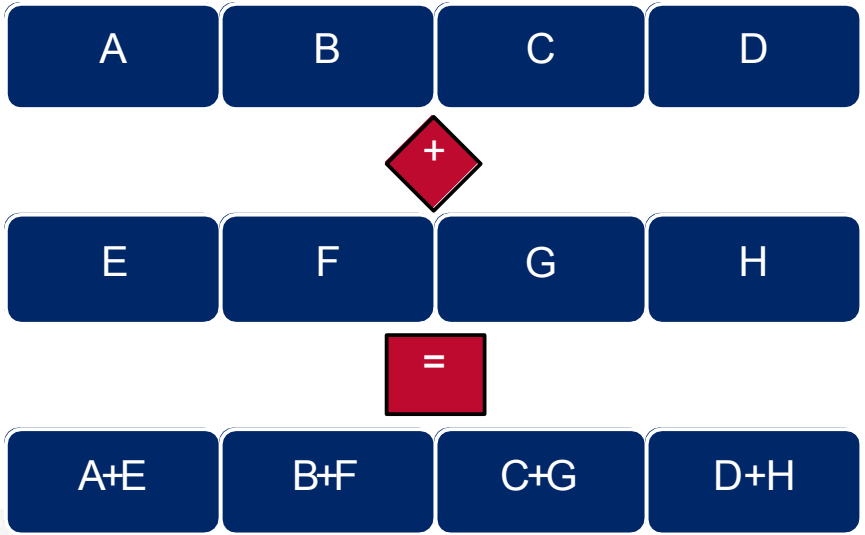
- Cores have registers
- Data must be moved from cache/memory into registers before operating on
- To add 2 floating point (fp) numbers:
  1. Move first fp ‘A’ from cache to a register
  2. Move second fp ‘E’ from cache to a different register
  3. Add fp’s and place result in yet a different register
  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

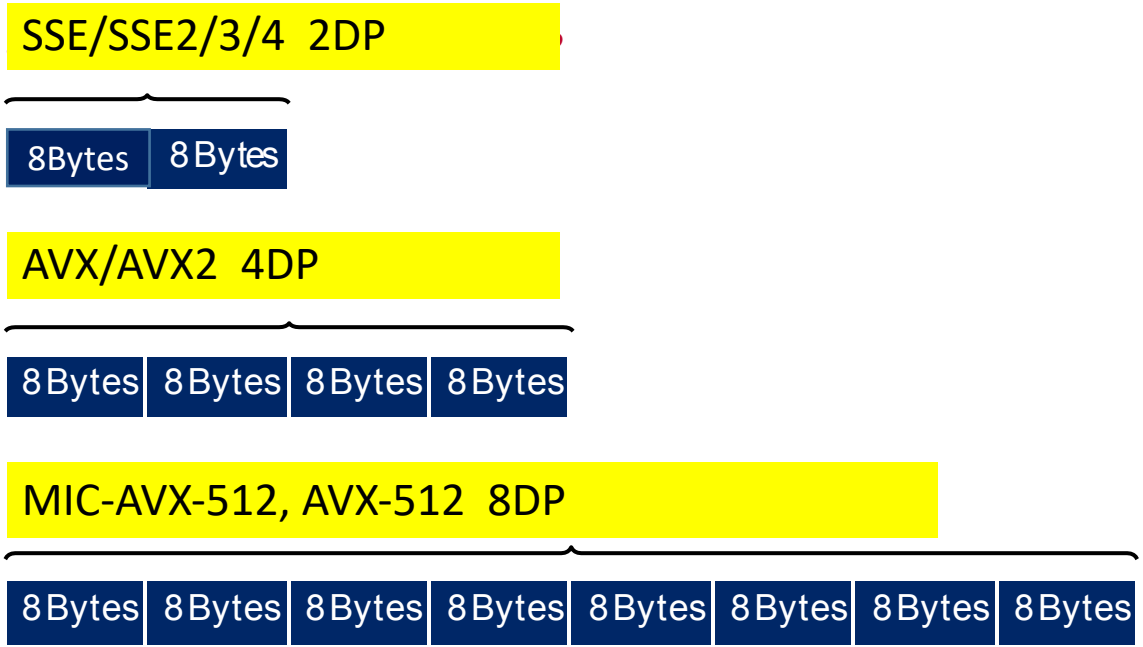
2017: Vector length = length of a cache line

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

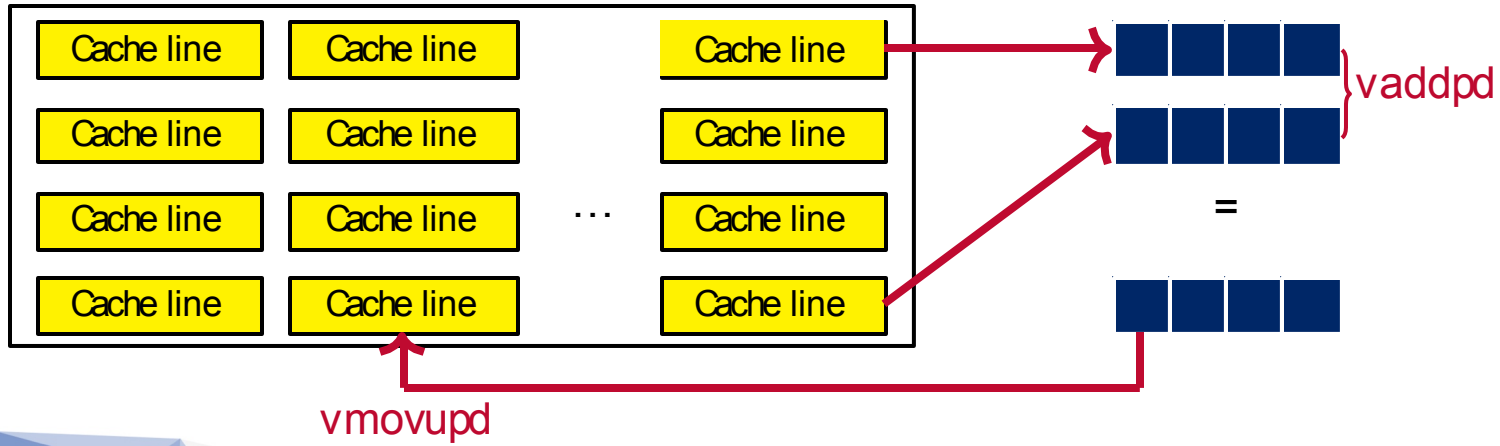
- Different types of processors have different width vector registers
- SP/DP are 32b/64b (4B/8B)
- Vector instructions are required to use vector registers



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

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

Compiler

Start, End, Increment

```
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?