CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 15: Data-Level Parallelism

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements

- Exam prep sections continue
  - RISC-V Datapath: Thursday 5-7pm, Friday 1-3pm (Cory 540AB)
- HW4 due today 7/19
- Labs 5 and 6 due tomorrow 7/20
- Project 3 (CPU) releases soon!

Agenda

- Why Parallelism?
- Amdahl’s Law
- SIMD Instructions
- Applying SIMD to Matrix Multiplication

Why Parallelism?

Why Parallelism?

- How do we make computers faster?
- To answer this question, it's useful to take a look at modern supercomputers
- Often large warehouse-scale systems, which when used properly can handle extremely fast/large computations
  - Google/Amazon computing warehouses
  - Supercomputers, like Fugaku in Kobe, Japan
    - Fastest in the world from 2020 to 2022

What’s the biggest difference between my computer and the Fugaku supercomputer?

1. Higher clock speed?
2. Faster memory access?
3. Higher transistor density?
4. More complicated instructions (more work in one clock cycle)?
5. Nothing; a supercomputer is just a bunch of regular computers wired together.

Higher Clock Speed?

My Computer	Fugaku
3.2 GHz clock frequency	2.2 GHz clock frequency

Faster memory access?

My Computer	Fugaku
3.2 GHz clock frequency	2.2 GHz clock frequency
LPDDR5 memory, 200 GB/s bandwidth	HBM2 memory, 1024 GB/s bandwidth
NVMe SSD for storage	NVMe SSD for storage

- More bandwidth, but not enough to account for millions of times difference in performance.

More transistors?

My Computer	**Fugaku**
3.2 GHz clock frequency	2.2 GHz clock frequency
LPDDR5 memory, 200 GB/s bandwidth	HBM2 memory, 1024 GB/s bandwidth
NVMe SSD for storage	NVMe SSD for storage
Manufactured on TSMC 5nm process; 180M transistors per mm^2	Manufactured on TSMC 7nm process; 100M transistors per mm^2

More complicated instructions?

My Computer	**Fugaku**
3.2 GHz clock frequency	2.2 GHz clock frequency
LPDDR5 memory, 200 GB/s bandwidth	HBM2 memory, 1024 GB/s bandwidth
NVMe SSD for storage	NVMe SSD for storage
Manufactured on TSMC 5nm process; 180M transistors per mm^2	Manufactured on TSMC 7nm process; 100M transistors per mm^2
64-bit ARM ISA, with up to 128-bit vector instructions	64-bit ARM ISA, with up to 512-bit vector instructions

- Fugaku’s processor does have wider vector instructions (more to come)

More computers?

My Computer	**Fugaku**
3.2 GHz clock frequency	2.2 GHz clock frequency
LPDDR5 memory, 200 GB/s bandwidth	HBM2 memory, 1024 GB/s bandwidth
NVMe SSD for storage	NVMe SSD for storage
Manufactured on TSMC 5nm process; 180M transistors per mm^2	Manufactured on TSMC 7nm process; 100M transistors per mm^2
64-bit ARM ISA, with up to 128-bit vector instructions	64-bit ARM ISA, with up to 512-bit vector instructions
10 independent CPUs (“cores”)	7,000,000+ independent CPUs (“cores”)

More computers?

- This is the biggest difference.
- In order to gain any benefits from using a supercomputer, we need to know how to get many computers to work together on the same problem
- Fugaku:

- Side note: there is very detailed documentation publicly available on the architecture of the Fujitsu A64FX (the CPU used by Fugaku).

Recall: New School Machine Architecture

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

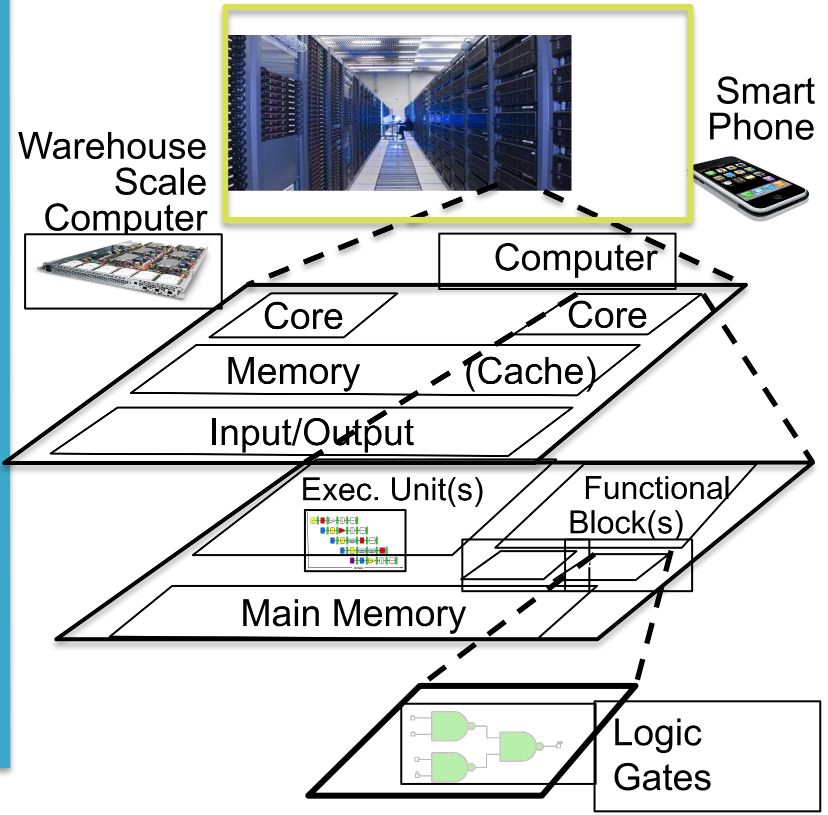
Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Hardware



So far…

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

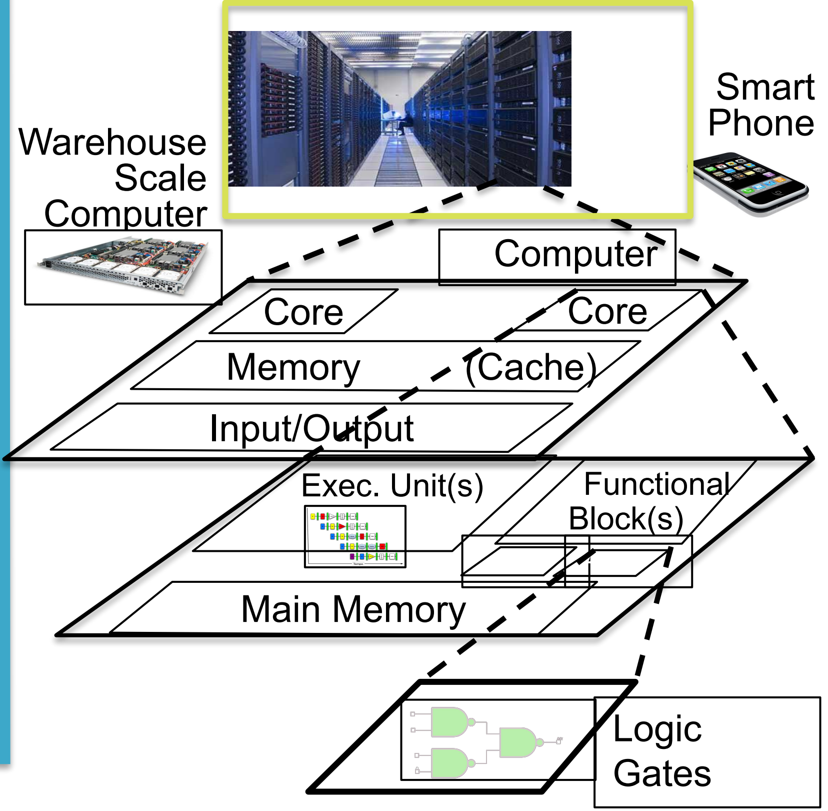
Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

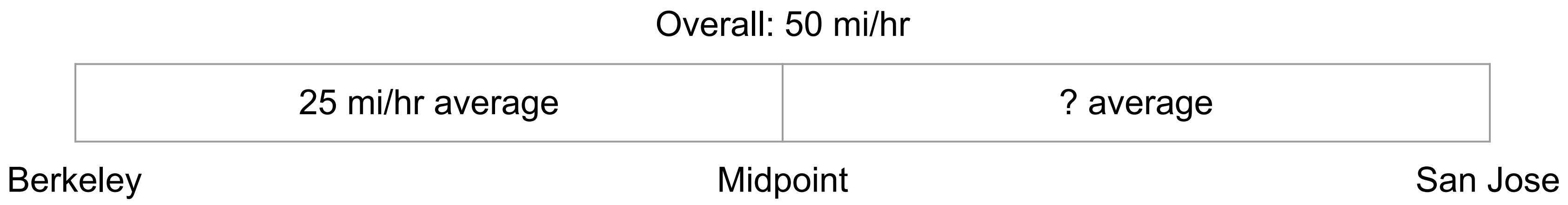
Hardware descriptions
All gates work in parallel at same time

Hardware



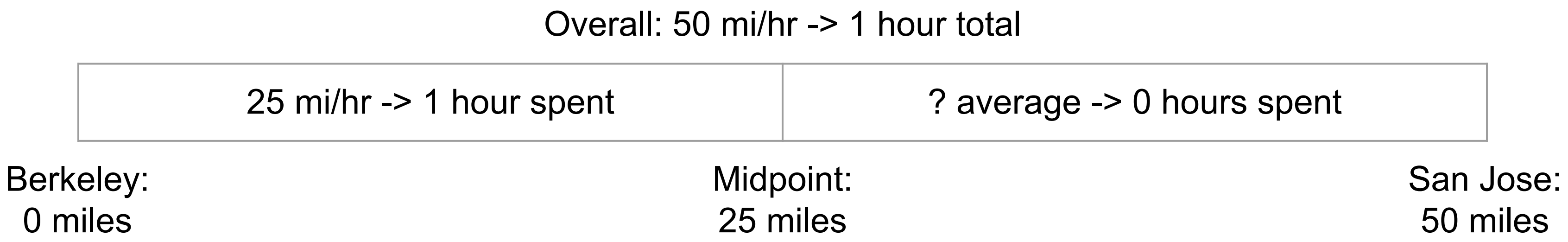
Amdahl’s Law

Amdahl's Law: Analogy



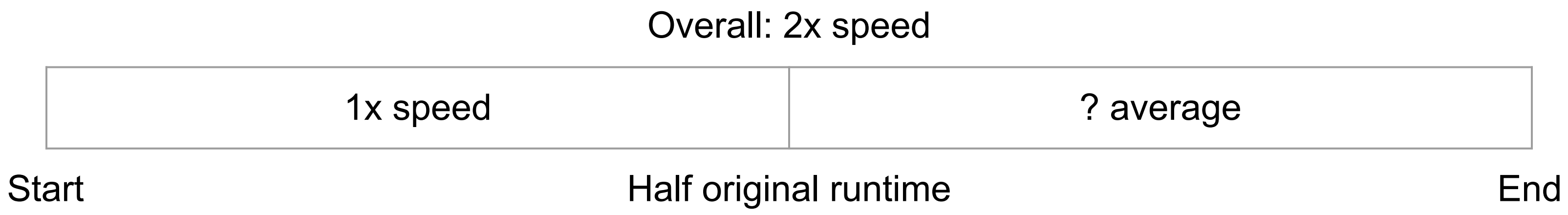
- You're driving from Berkeley to San Jose.
- For the first half of the distance, you average 25 miles/hour.
- How fast do you need to travel for the second half, in order to average 50 miles/hour overall?

Amdahl's Law: Analogy



- Assume Berkeley-San Jose is 50 miles
- First half: 25 miles at 25 mph = 1 hour
- Overall: 50 miles at 50 mph = 1 hour
- Time left for second half: 25 miles in 0 hours -> We need to travel at ∞ mph

Amdahl's Law



- You're trying to speed up a program.
- The first half of the code can't be sped up.
- How many times faster do you need to make the second half of the code, if you want to overall get a 2x speedup?
    - Infinitely faster!

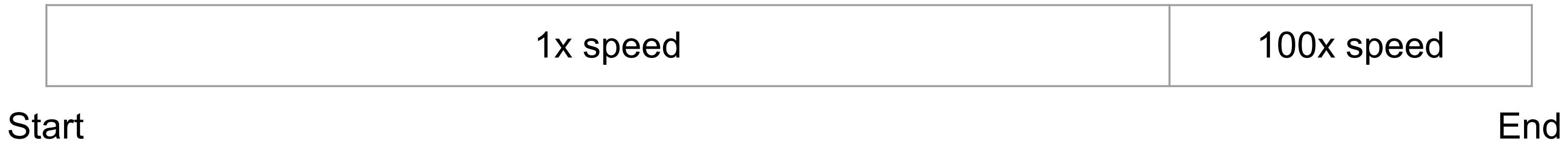
Amdahl's Law

- AKA the bane of performance programming
- "The maximum speedup we can attain with our code is limited by the fraction that cannot be sped up"
  - If we speed up 95% of our code infinitely, we can reduce runtime to 5% of what it originally was
- Formal equation:

$$Speedup = \frac{1}{(1-p) + p/N}$$

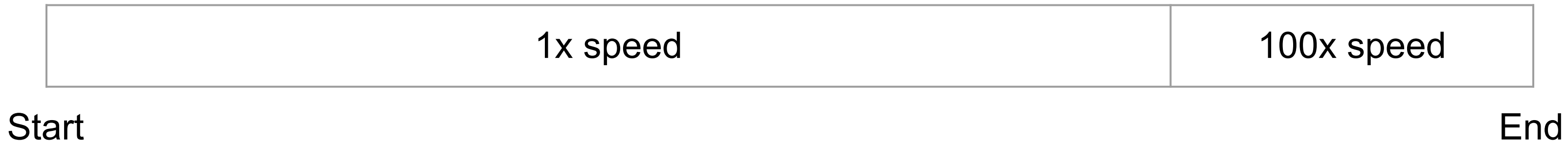
where p is % of code that you speed up and N is how many times faster you make that part.

Amdahl's Law: Example



- You have an optimization that speeds up the foo function by 100x
- Unfortunately, the runtime of foo was only 25% of our original code's runtime. How many times faster have we made our code overall?

Amdahl's Law: Example



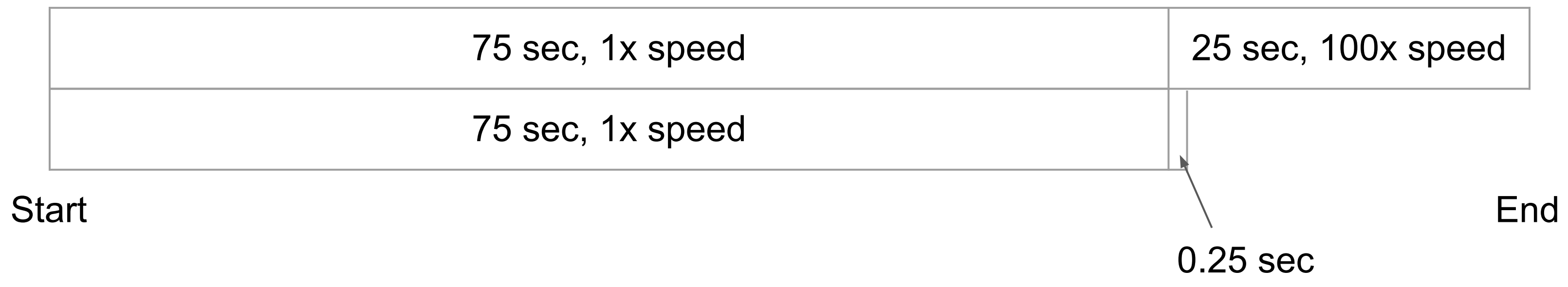
- Strategy one: Use the formula

  1/((1-p)+p/N)

  = 1/((1-0.25)+0.25/100)

  = 1/(0.7525) ≈ 1.33x speedup

Amdahl's Law: Example



- Strategy two: Assign values

  Let's say that our original code took 100 seconds to run

  Total runtime is now 75 seconds for part 1, .25 seconds for part 2 = 75.25 seconds

  Speedup = 100 seconds / 75.25 seconds ≈ 1.33x speedup

Amdahl's Law: Consequences

- In order to properly speed up your code, you need to know which parts of your code are taking the runtime
- Test your code to help analyze where your runtime's going
    - Check multiple sizes, check repeatedly (since there might be variation between runs)
    - Check each component independently
        - Our autograders will only give you overall speedup, which doesn't help much in determining where you can speed things up further.

Iron Law and Parallelism (1/2)

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Recall the "Iron Law" of Processor Performance. In order to speed up our code, we need to improve one of:
- Instructions/Program
- Cycles/Instruction
- Time/Cycle

Iron Law and Parallelism (2/2)

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions/Program
  - At some point, we will have reduced our program down to the basic operations that must be carried out.
  - CISC instructions increase the complexity of hardware.
- Cycles/Instruction
  - Memory instructions have high cycles/instruction, so we can improve this via caches (next week).
  - Pipelining and reordering instructions also help.
- Time/Cycle
  - Up until recently, done by improving manufacturing to increase frequency (doubled every ~2 years according to Moore's Law).
  - Moore's Law has slowed down!
- Need to parallelize
  - Increase the number of processors that get used by a single program, or
  - **Increase the work that gets done per instruction.**

Today: Parallel Data

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

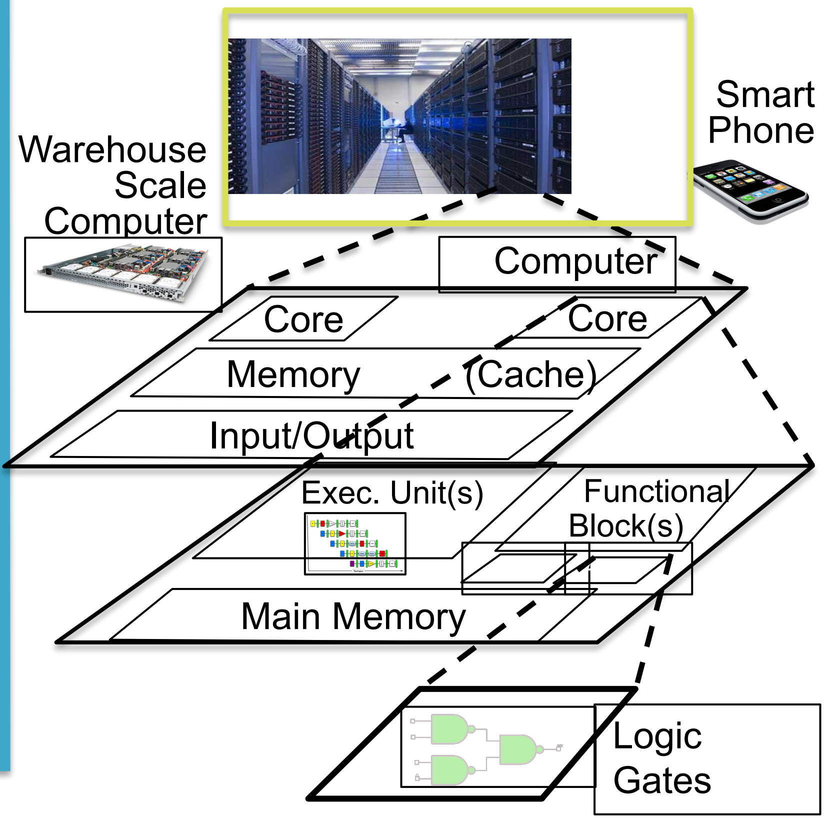
Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Hardware



SIMD Instructions

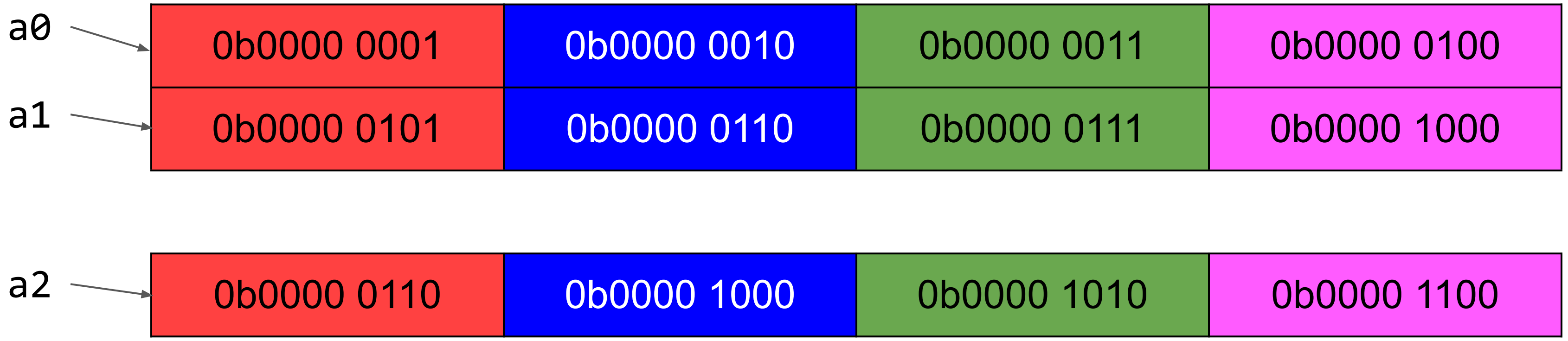
Toy Example: Vector Sum

0b0000 0001	0b0000 0010	0b0000 0011	0b0000 0100
0b0000 0101	0b0000 0110	0b0000 0111	0b0000 1000

0b0000 0110	0b0000 1000	0b0000 1010	0b0000 1100

- We have two 4-D vectors whose components are 8-bit numbers
- Goal: Determine the sum of the two vectors
- For the following example, inputs stored at 0(a0) and 0(a1), and output saved to 0(a2)

Toy Example: Vector Sum: Naive



```
lb t0 0(a0)
lb t1 0(a1)
add t0 t0 t1
sb t0 0(a2)
```

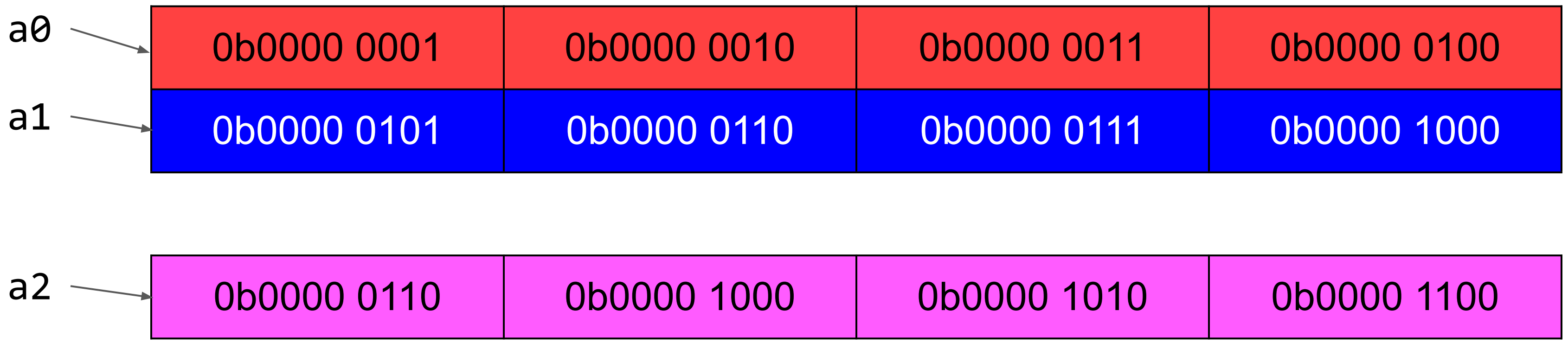
```
lb t0 1(a0)
lb t1 1(a1)
add t0 t0 t1
sb t0 1(a2)
```

```
lb t0 2(a0)
lb t1 2(a1)
add t0 t0 t1
sb t0 2(a2)
```

```
lb t0 3(a0)
lb t1 3(a1)
add t0 t0 t1
sb t0 3(a2)
```

- 16 total instructions. Can we do better?

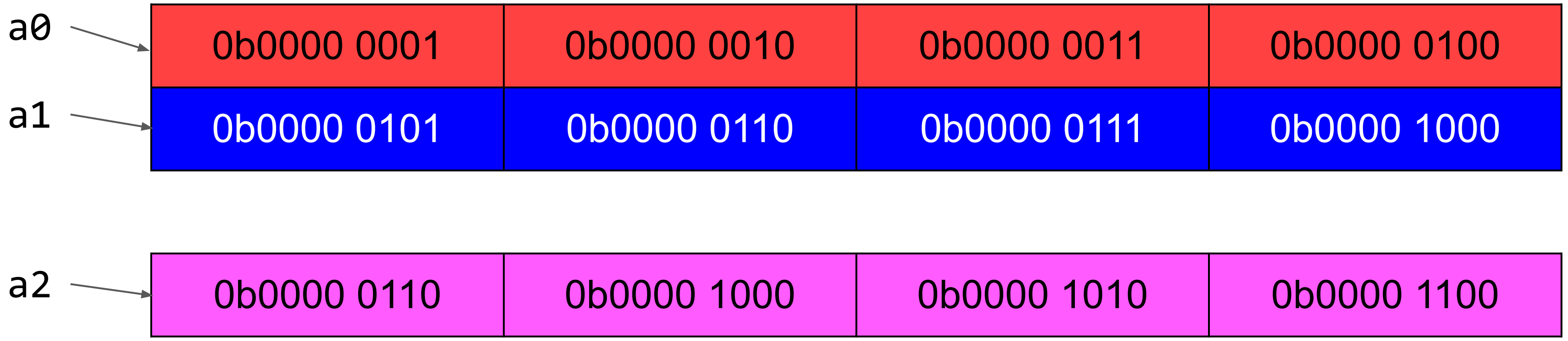
Toy Example: Vector Sum: Single Add



- Solution: If we treat these arrays as 32-bit integers, we can add with one operation, and do this in 4 instructions.

```
lw t0 0(a0)
lw t1 0(a1)
add t0 t0 t1
sw t0 0(a2)
```

Toy Example: Vector Sum: Vectorized Add



- This doesn't quite work, because overflow on one element affects other elements. So we need to create a slightly different instruction that ignores overflow every 8th bit.

```
lw t0 0(a0)
lw t1 0(a1)
vec_add t0 t0 t1
sw t0 0(a2)
```

SIMD Instructions (1/2)

- Instead of working on one number at a time, we can instead work on multiple numbers at a time, in a single clock cycle
- Known as **SIMD instructions (Single Instruction, Multiple Data)** or **vector instructions**
  - Tomorrow: MIMD (Multiple Instruction, Multiple Data)
- Use specialized **vector registers** which store 128, 256, or even 512 bits
- SIMD instructions act as extensions to the base instruction set, with different systems supporting different SIMD instructions.

SIMD Instructions (2/2)

- Each instruction needs its own circuitry, so we're limited to the set of instructions that came with the CPU
- **Important:** we use **Intel** SIMD instructions in 61C.
  - We can run these programs on our x86 Intel servers
  - Arithmetic syntax looks similar to RISC-V
  - Side note: RISC-V vector extension now official!
- Only one PC, so we can't vectorize branch or jump instructions
  - Workarounds exist (see lab 7)
- Can't do different operations to different vector components
- We can only easily load consecutive blocks of memory to a vector
  - Or with some kind of “pattern”

Implications of SIMD

- Large vectors require more circuitry
  - Take up more chip area
  - SIMD instructions have higher cycles/instruction than standard instructions
- Most of the speedup comes not from doing four math operations at a time, but instead from doing a large memory load/store at a time.

Intel Intrinsics

- SSE library
  - 64- and 128-bit registers: 4 32-bit integers at a time or 2 doubles at a time
- AVX library
  - 256-bit registers: 8 32-bit integers at a time or 4 doubles at a time
- AVX-2 library
  - Extension of the AVX library, with more supported instructions
- AVX-512 library
  - 512 bit registers
  - For most use cases, does not actually provide much speedup over 256-bit vectors
  - Not covered in this class, and not available on hive machines

- Intel's Ark database lists which libraries are supported on which Intel CPU
  - Can also type `` `lscpu` `` in Linux terminal

Intel Intrinsics: Instructions

- Generally of the format:

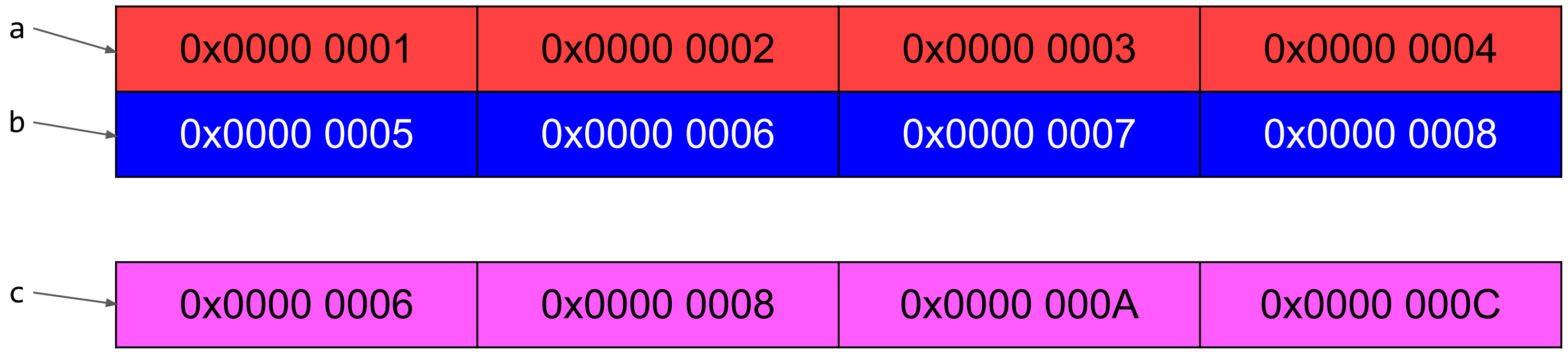
```
_<register size>_<instruction>_<component_type>
```

- Ex. `_mm256_add_epi32` adds two 256-bit vectors, treating the vectors as arrays of 32-bit integers.
- Ex. `_mm_load_ps` loads 4 floats into a 128-bit register from the given memory address. The memory address must be aligned to a 16-byte boundary (loadu allows for nonaligned addresses, but is slower)
- More instructions: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- Can be inlined into C code and looks like C functions, but programming with them feels more like assembly

Intel Intrinsics: Types

- `__m256`
  - 256 bit register for storing floats
- `__m256d`
  - 256 bit register for storing doubles
- `__m256i`
  - 256 bit register for storing 32-bit integers
- `__m128, __m128d, __m128i`
  - 128 bit registers
- Each type corresponds directly to a type of SIMD register (note that x86 has different sets of registers for floats, doubles, and integers).

Vector Sum (128-bit registers, 32-bit integers)

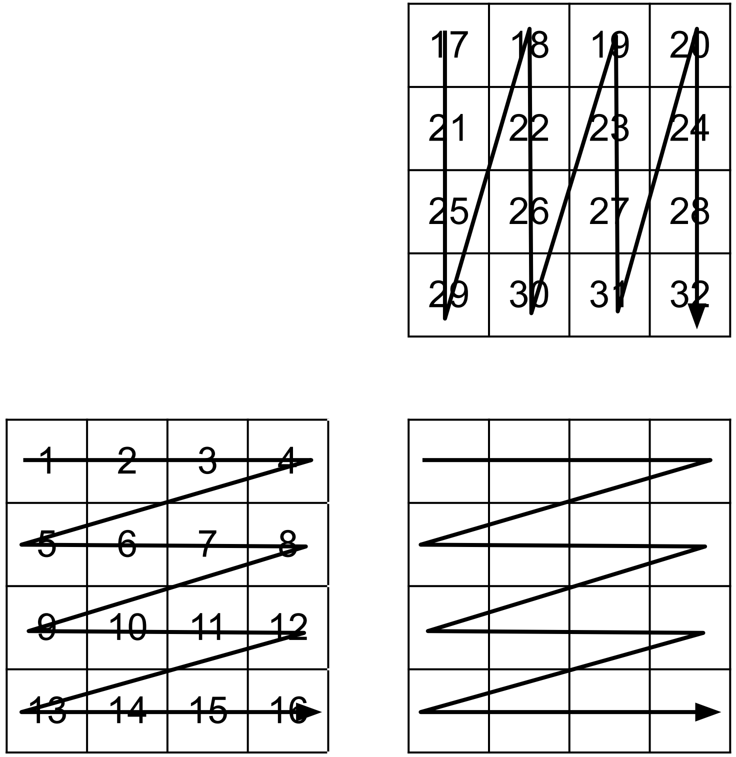


```
__m128i avec = _mm_load_si128(a);
__m128i bvec = _mm_load_si128(b);
__m128i sum = _mm_add_epi32(avec, bvec);
_mm_store_si128(c, sum);
```

Applying SIMD to Matrix Multiplication

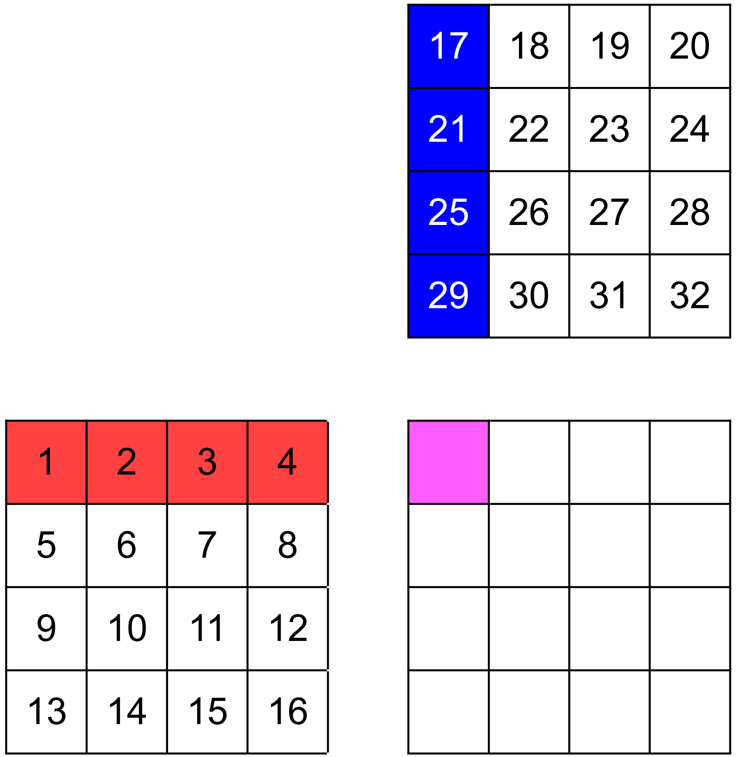
Matrix Multiplication

- In a naive matrix multiplication, we take the dot product (element-wise multiply, then sum) of each row of one matrix with each column of the other matrix
- Say matrices are stored in a row-major format: consecutive elements in one row are consecutively stored in memory



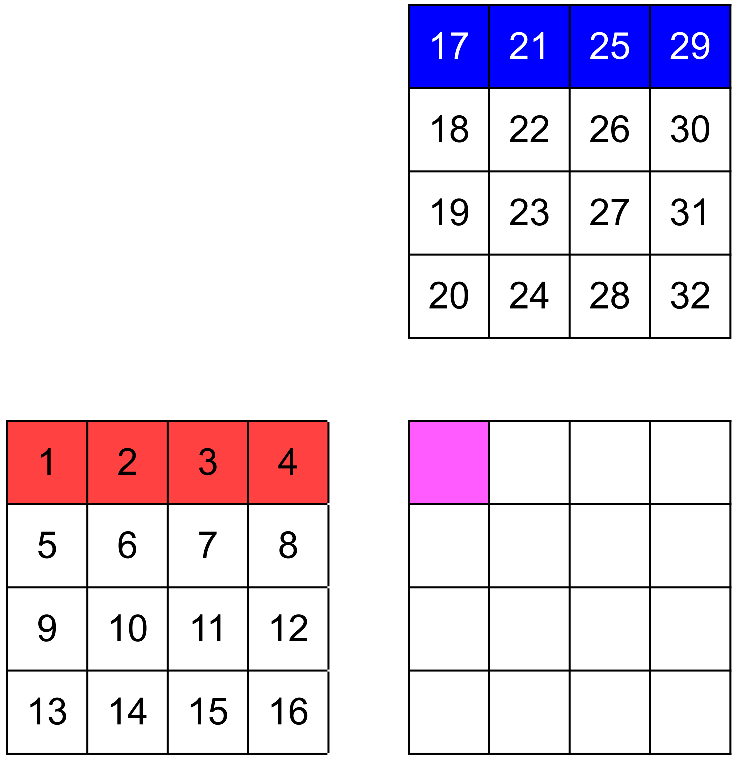
Applying DLP to Matrix Multiply

- Would be useful to compute one whole dot product at a time; however, it’s easier to load data that's consecutive in memory
- Therefore, start by transposing (flipping diagonally) the second matrix, turning the columns into rows



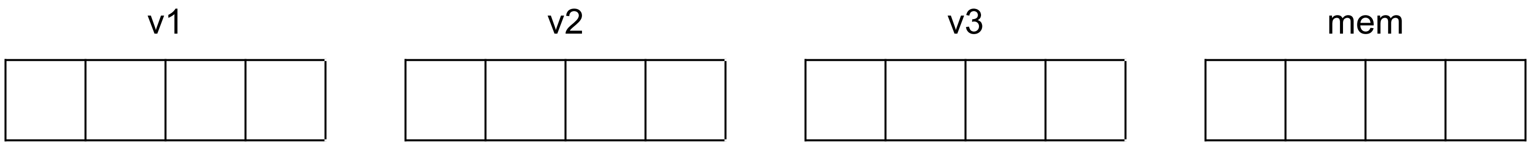
Applying DLP to Matrix Multiply

- How do we compute the dot product quickly?



Applying DLP to Matrix Multiply

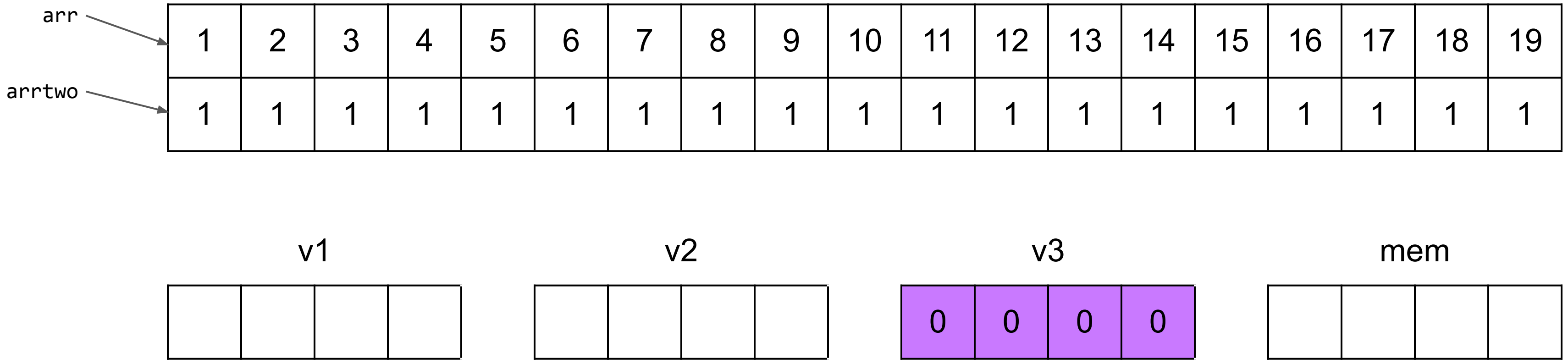
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1



- Let's say we want to find the dot product of two 19-element rows of single-precision (32-bit) floats. Assume the arrays are aligned.
- Step 1: Load 0s into v3:

```
__m128 v3 = _mm_set1_ps(0);
```

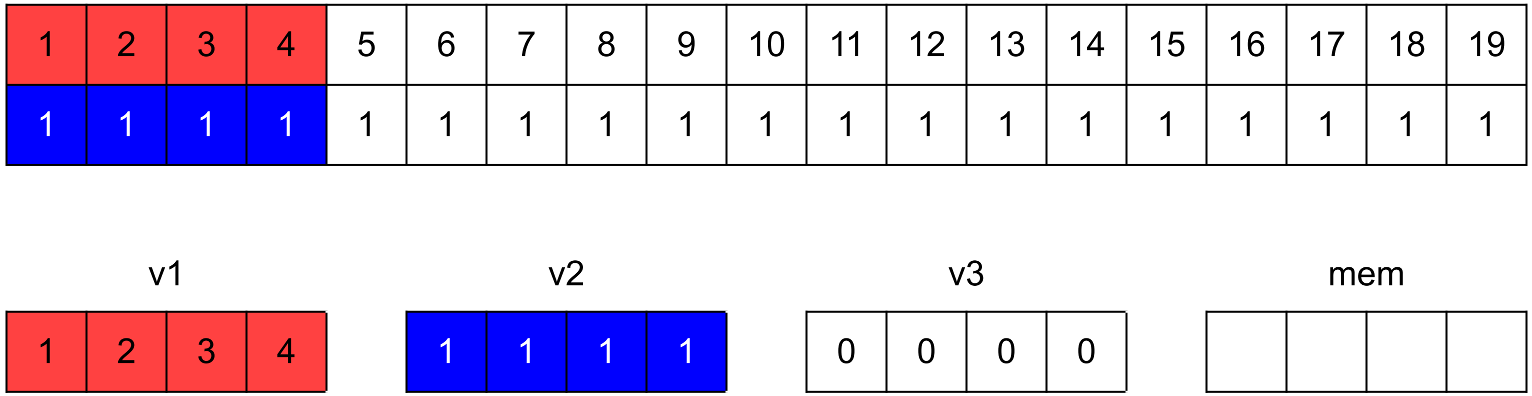
Applying DLP to Matrix Multiply



- Step 2: Load the first four numbers of each input into v1 and v2, respectively

```
__m128 v1 = _mm_load_ps(&arr);
__m128 v2 = _mm_load_ps(&arrtwo);
```

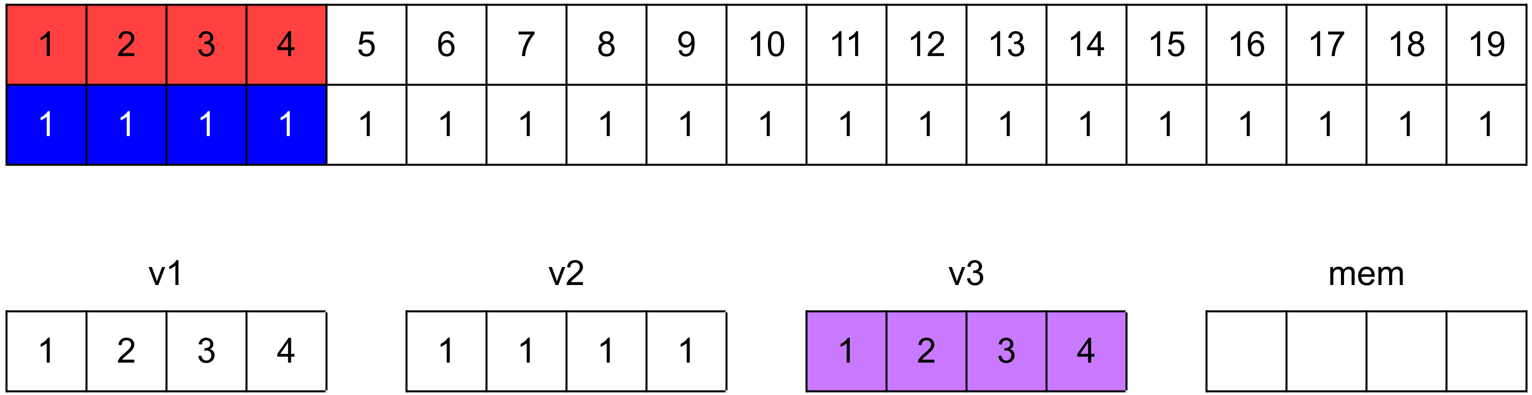
Applying DLP to Matrix Multiply



- Step 3: Multiply v1 and v2 together, and add that to v3
  - This procedure is so common, there's a single instruction to do this! (Well, for floats and doubles only)

```
v3 = _mm_fmadd_ps(v1, v2, v3);
```

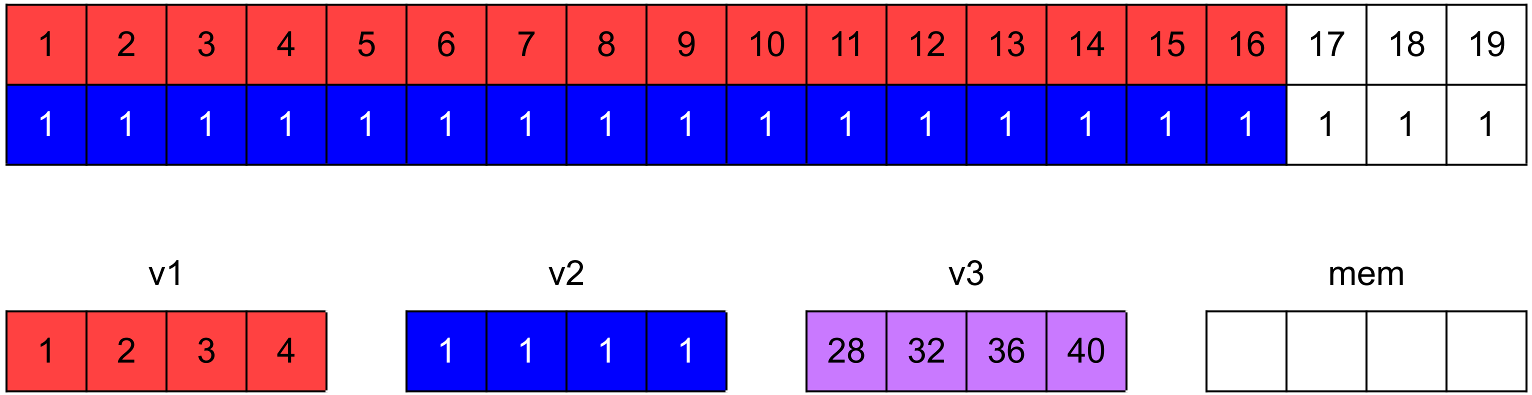
Applying DLP to Matrix Multiply



- Step 4: Repeat for the majority of the array

```
int i;
for(i = 0; i < arrlen/4*4;i+=4) {
    __m128 v1 = _mm_load_ps(arr+i);
    __m128 v2 = _mm_load_ps(arrtwo+i);
    v3 = _mm_fmadd_ps(v1, v2, v3); }
```

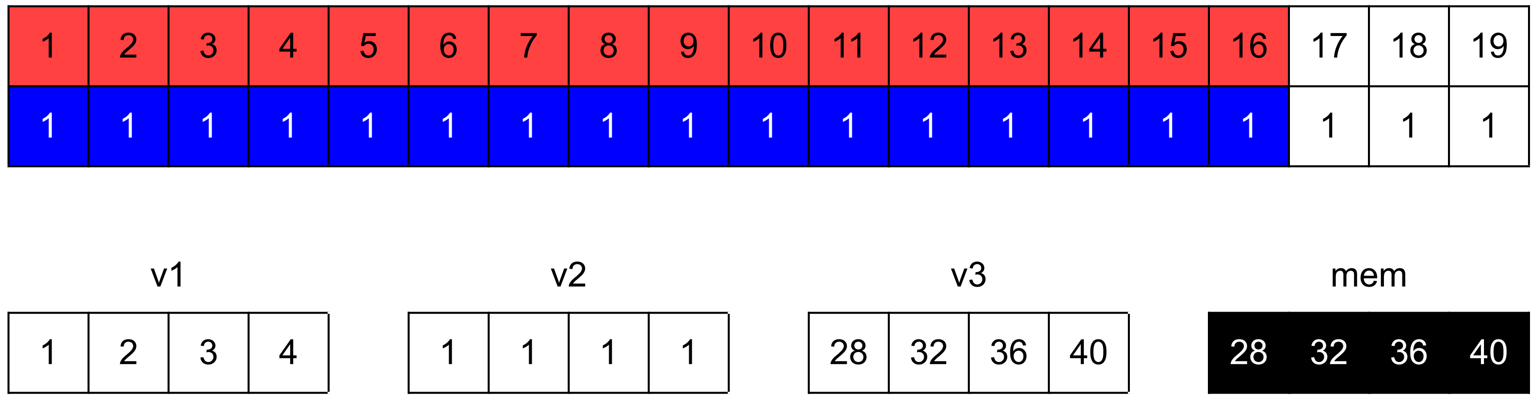
Applying DLP to Matrix Multiply



- Step 5: Store the results in memory somewhere.

```
//Force alignment
float mem[4] __attribute__ ((aligned (16)));
_mm_store_ps(mem, v3);
```

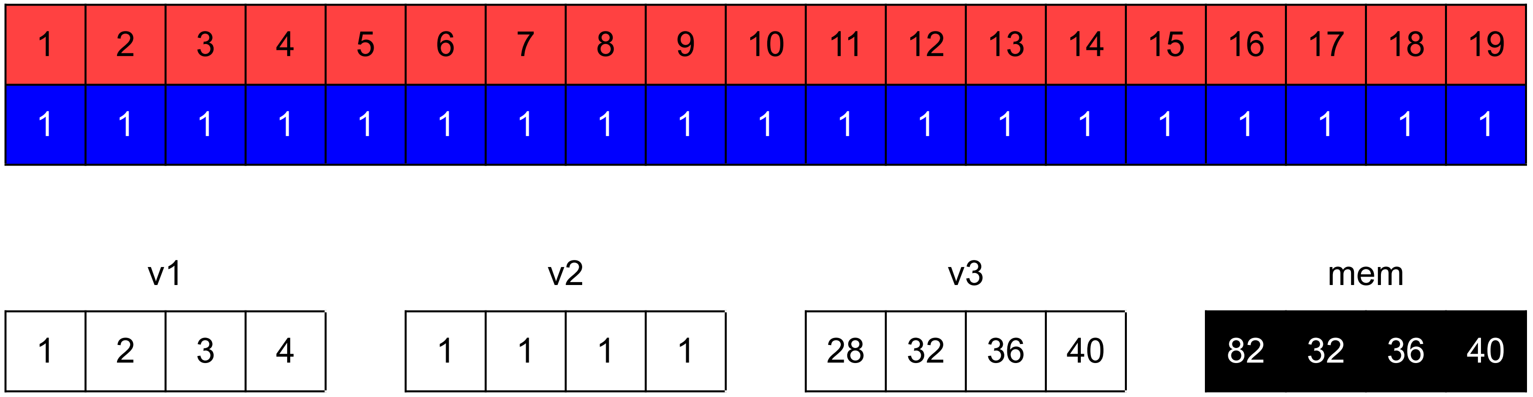
Applying DLP to Matrix Multiply



- Step 6: Resolve the tail case

```
for(i=arrlen/4*4; i<arrlen; i++) {
    mem[0]+=arr[i]*arrtwo[i];
}
```

Applying DLP to Matrix Multiply



- Step 7: Return the sum of mem

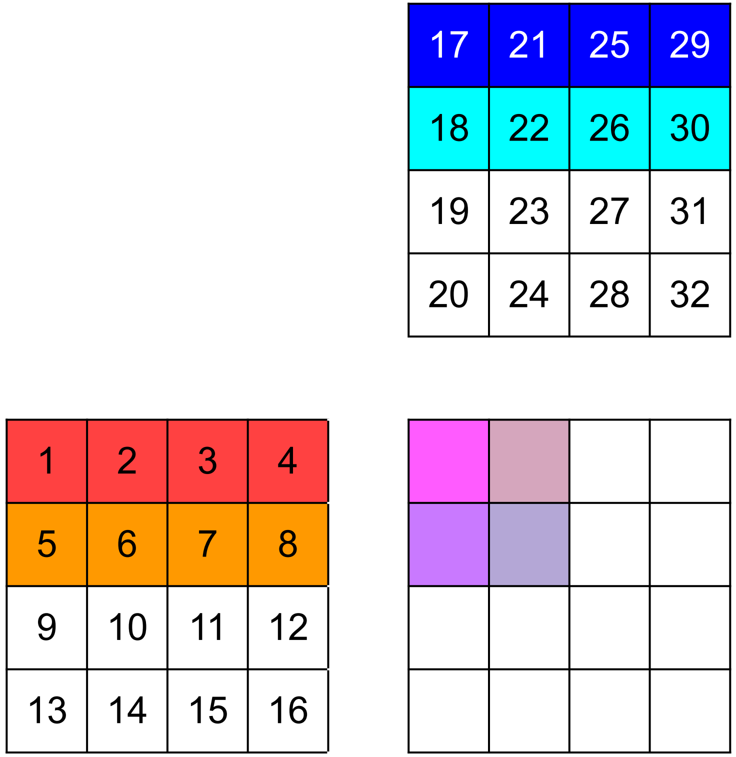
```
return mem[0]+mem[1]+mem[2]+mem[3];
```

Applying DLP to Matrix Multiply

```
__m128 v3 = _mm_set1_ps(0);
int i;
for(i = 0; i < arrlen/4*4;i+=4) {
    __m128 v1 = _mm_load_ps(arr+i);
    __m128 v2 = _mm_load_ps(arrtwo+i);
    v3 = _mm_fmadd_ps(v1, v2, v3);
}
float mem[4] __attribute__ ((aligned (16)));
_mm_store_ps(mem, v3);
for(;i<arrlen;i++) {
    mem[0]+=arr[i]*arrtwo[i];
}
return mem[0]+mem[1]+mem[2]+mem[3];
```

Applying DLP to Matrix Multiply

- One final optimization: right now, we load two rows to yield one value. Each row gets loaded n times.
- With enough vector registers, we can do this simultaneously to several cells at once
- This uses 8 vector registers, but computes 4 cells with 4 loads: 2x fewer loads!
- Requires tail case for odd n



Common mistakes when working with SIMD instructions

- Trying to directly access a 32-bit chunk of a SIMD vector (such as through typecasting)
  - Need to do an explicit load/store, since registers are different from memory
- Trying to _mm_load or _mm_store with unaligned addresses
  - Use loadu or storeu, or try to get addresses aligned
  - For mallocs, aligned_alloc gives you an aligned address
  - For local variables, you can set an attribute (example shown in slides)
- Forgetting the tail case
  - If data length not a multiple of your vector size: handle the last iterations of your dataset one-by-one instead of 4 at a time.
- Using too many vectors (or creating a large array of vectors)
  - Ends up throttling your code because the compiler ends up trying to load/store SIMD vectors to the stack a bunch of times.

Conclusion

- Without parallelism, the most powerful computers in the world would not exist.
- There are several types of parallelism; **data-level parallelism (DLP)** is where we act on multiple pieces of data at once.
- A common way to implement DLP is using single instructions that perform the same operation on multiple instructions (**SIMD/vector instructions**).
- This can help speed up programs that have many repeated operations across arrays.
- Tomorrow: **thread-level parallelism (TLP)**.