Fusion Trees
Part One
Outline for Today

- **Word-Level Parallelism**
  - Harnessing the intrinsic parallelism inside the processor.

- **Word-Parallel Operations**
  - Comparing, tiling, and ranking numbers; adding and packing bits.

- **The Sardine Tree**
  - Unconditionally beating a BST for very small integers.

- **Most-Significant Bits**
  - Finding the most significant bit in O(1) time/space.
Recap from Last Time
Our Machine Model

• We will assume we’re working on a machine where memory is segmented into \( w \)-bit words.
  • Although on any one fixed machine \( w \) is a constant, in general, don’t assume this is the case. 32-bit was the norm until fairly recently, and before that 16-bit was standard.

• We’ll assume C integer operators work in constant time, and won’t assume other integer operations (say, finding most significant bits, counting 1 bits set) are available.

\[
+ \quad - \quad * \quad / \quad \% \quad << \quad >> \quad & \quad | \quad ^ \quad = \quad <=
\]
Integer Ordered Dictionaries

- The **y-Fast Trie** is an ordered dictionary for integers where each operation takes $O(\log w)$ time, amortized/expected.
  - Note that when $\log n = \omega(\log w)$, this is better than a binary search tree!
- Space usage is $\Theta(n)$, where $n$ is the number of elements in the trie.
New Stuff!
A Key Technique: *Word-Level Parallelism*
Word-Level Parallelism

- On a standard computer, arithmetic and logical operations on a machine word take time $O(1)$.

- We can perform certain classes of operations (addition, shifts, etc.) on $\Theta(w)$ bits in time $O(1)$.
  - Think of this as a weak form of parallel computation, where we can work over multiple bits in parallel with a limited set of operations.

- With some creativity, we can harness these primitives to build operations that run in time $O(1)$ but work on $\omega(1)$ objects.

- Let’s see a quick example...
Word-Level Parallelism

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$a_2$</td>
<td>$a_3$</td>
<td>$a_4$</td>
<td>$a_5$</td>
<td>$a_6$</td>
<td>$a_7$</td>
<td>$a_8$</td>
<td></td>
</tr>
<tr>
<td>1101110</td>
<td>0101111</td>
<td>1111000</td>
<td>1001101</td>
<td>0101111</td>
<td>0001101</td>
<td>1110111</td>
<td>1100001</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$b_3$</td>
<td>$b_4$</td>
<td>$b_5$</td>
<td>$b_6$</td>
<td>$b_7$</td>
<td>$b_8$</td>
<td></td>
</tr>
<tr>
<td>0011010</td>
<td>1000101</td>
<td>0010100</td>
<td>0100000</td>
<td>1010000</td>
<td>0100010</td>
<td>1000100</td>
<td>0001000</td>
<td></td>
</tr>
</tbody>
</table>
We’ve performed eight logical additions with a single add instruction!
The Landscape

• Preprocessing/runtime tradeoffs:
  
  “Yes, we have to do a lot of work, but it’s a one-time cost and everything is cheaper after that.”

• Randomization:
  
  “We might have to do a lot of work, but it’s unlikely that we’ll do so.”

• Amortization:
  
  “Yes, we have to do a lot of work every once and a while, but only after a period of doing very little.”

• Word-level parallelism:
  
  “We have to do a lot of work, but we don’t have to perform many operations to do it.”
These actually aren’t called sardine trees. I couldn’t find a name for them anywhere and thought that this title was appropriate. Let me know if there’s a more proper name to associate with them!
The Setup

• Let $w$ denote the machine word size.
• Imagine you want to store a collection of $s$-bit integers, where $s$ is small compared to $w$.
  • For example, storing 7-bit integers on a 64-bit machine would have $s = 7$ and $w = 64$.
• Can we build an ordered dictionary that takes advantage of the small key size?
A Refresher: B-Trees

- A **B-tree** is a multiway tree with a tunable parameter \( b \) called the **order** of the tree.
- Each node stores \( \Theta(b) \) keys. The height of the tree is \( \Theta(\log_b n) \).
- Most operations (**lookup**, **insert**, **delete**, **successor**, **predecessor**, etc.) perform a top-down search of the tree, doing some amount of work per node.
- Runtime of each operation is \( O(f(b) \log_b n) \), where \( f(b) \) is the amount of work done per node.
B-Tree Traversals

- Most B-tree operations work by choosing some subtree to descend into, then descending there.
- **Claim:** The subtree we want is given by the number of keys in the current node less than or equal to the query key $k$. This quantity is the *rank* of $k$.
- For example, in the top node of the B-tree shown below:
  \[ \text{rank}(40) = 0 \quad \text{rank}(74) = 2 \quad \text{rank}(107) = 3 \]
- **Question:** How quickly can we determine the rank of a key in a B-tree node?
B-Tree Traversals

- We can determine \( \text{rank}(k) \) with a linear search in each B-tree node for a total lookup cost of \( O(b \cdot \log_b n) \).
- We can determine \( \text{rank}(k) \) with a binary search in each B-tree node for a total lookup cost of
  \[ O(\log_b n \cdot \log b) = O(\log n). \]
- **Claim:** If we can fit all the keys in a node into \( O(1) \) machine words, we can determine \( \text{rank}(k) \) in time \( O(1) \) for total lookup cost of \( O(\log_b n) \).
How is this possible?
Warmup: Comparing Two Values

- Imagine we have two $s$-bit integers $x$ and $y$ and want to determine whether $x \geq y$.
- How might we do this?

```
    1 1 1 0 1 0
  - 0 0 0 1 1
    1 1 0 0 1
```

This bit tells us whether the first number was as least as big as the second!
Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here’s how we’d compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$a_2$</td>
<td>$a_3$</td>
<td>$a_4$</td>
<td>$a_5$</td>
<td>$a_6$</td>
<td>$a_7$</td>
<td>$a_8$</td>
<td></td>
</tr>
<tr>
<td>1101110</td>
<td>0101110</td>
<td>1111000</td>
<td>1001101</td>
<td>0101111</td>
<td>0001101</td>
<td>1110111</td>
<td>1100001</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>$b_3$</td>
<td>$b_4$</td>
<td>$b_5$</td>
<td>$b_6$</td>
<td>$b_7$</td>
<td>$b_8$</td>
<td></td>
</tr>
<tr>
<td>0011010</td>
<td>1000101</td>
<td>0010100</td>
<td>0100000</td>
<td>1010000</td>
<td>0100010</td>
<td>1000100</td>
<td>0001000</td>
<td></td>
</tr>
</tbody>
</table>
Comparing Multiple Values

- This technique can be extended to work on multiple values in parallel.
- For example, here’s how we’d compare eight pairs of 7-bit numbers by doing a single 64-bit subtraction:

\[
\begin{array}{cccccccc}
1101110 & 10101110 & 11111000 & 11001101 & 10101111 & 10001101 & 1110111 & 1100001 \\
- & 00011010 & 0100101 & 0010100 & 00100000 & 01010000 & 00100010 & 0100100 & 00001000 \\
\hline
11010100 & 01101001 & 11100100 & 10101101 & 01011111 & 01101011 & 10110011 & 11011001
\end{array}
\]

This technique is used in practice, including the glibc version of `strlen`. Thanks to former CS166 student Jane Lange for pointing this out!
Fundamental Primitive: *Parallel Compare*

**Input:** Two machine words. The first holds an array \( x_1, \ldots, x_n \) with one bit of space between each number. The second holds an array \( y_1, \ldots, y_n \) with one bit of space between each number.

**Output:** A machine word with the result of \( x_i \geq y_i \) encoded as a bit in the blank spaces between the numbers in the input array.

**Procedure:**

1. Use an OR operation to place 1s between
2. Pack a list of values \( y_1, \ldots, y_k \) into a machine word \( Y \), separated by 0s.
3. Compute \( X - Y \). The bit preceding \( x_i - y_i \) is 1 if \( x_i \geq y_i \) and 0 otherwise.
Back to B-Trees

- **Recall:** The whole reason we’re interested in making these comparisons is so that we can find how many keys in a B-tree node are less than or equal to a query key $k$.

- **Idea:** Store the (s-bit) keys in the B-tree node in a single (w-bit) machine word, with zeros interspersed:

<table>
<thead>
<tr>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>$y_5$</th>
<th>$y_6$</th>
<th>$y_7$</th>
<th>$y_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>93</td>
<td>103</td>
<td>106</td>
<td>107</td>
<td>109</td>
<td>110</td>
<td>127</td>
</tr>
</tbody>
</table>
Back to B-Trees

- **Recall:** The whole reason we’re interested in making these comparisons is so that we can find how many keys in a B-tree node are less than or equal to a query key $k$.

- **Idea:** Store the (s-bit) keys in the B-tree node in a single (w-bit) machine word, with zeros interspersed:

```
<table>
<thead>
<tr>
<th>y_1</th>
<th>y_2</th>
<th>y_3</th>
<th>y_4</th>
<th>y_5</th>
<th>y_6</th>
<th>y_7</th>
<th>y_8</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101001</td>
<td>01011111</td>
<td>01100111</td>
<td>01101010</td>
<td>01101011</td>
<td>01101101</td>
<td>01101110</td>
<td>01111111</td>
</tr>
</tbody>
</table>
```
Rank in $O(1)$

- To perform a lookup for the key $k$, form a number by replicating $k$ multiple times with 1s interspersed.
- Subtract the B-tree key number from it to do a parallel comparison.
- Count up how many of the sentinel bits in the resulting number are equal to 1. This is the number of keys in the node less than or equal to $k$.

```
1100111 1100111 1100111 1100111 1100111 1100111 1100111 1100111
-  00101001 0101010 01100111 01101010 01101010 01101011 01101110 01111111
```

```
10111110 10001010 10000000 01111111 0111100 01110100 01101000
```

Rank: 3
Suppose you have a one-digit number $m$.

You want to form this base-10 number:

$mmm$

Is there a nice series of arithmetical operations that will produce this?

**Answer:** Compute $m \times 111$.

Why does this work?

\[
m \times 111 = m \ll 2 + m \ll 1 + m \ll 0
= m00 + 0m0 + 00m
= mmm.
\]
Back in Base Ten

• Suppose you have a two-digit number $mn$.
• You want to form this base-10 number:

$$mnmnmn$$

• Is there a nice series of arithmetical operations that will produce this?
• **Answer:** Compute $mn \times 10,101$.
• Why does this work?

$$mn \times 10,101 = mn \ll 4 + mn \ll 2 + mn \ll 0$$
$$= mn0000 + 00mn00 + 0000mn$$
$$= mnmnmnmn.$$
Back in Base Ten

- **Answer:** Compute $mn \times 10,101$.
- Why does this work?

\[
mn \times 10,101 = mn \ll 4 + mn \ll 2 + mn \ll 0 = mn0000 + 00mn00 + 0000mn = mnmnmnmn.
\]
Back in Base Ten

• Suppose you have a three-digit number $mnp$.
• You want to form this base-10 number:

  $\overline{mnp000mnp0mnp}$

• Is there a nice series of arithmetical operations that will produce this?

• **Answer:** Compute $mnp \times 10,000,010,001$.

  $\overline{mnp000mnp0mnp}$

  $= mnp \ll 10 + mnp \ll 4 + mnp \ll 0$

  $= mnp \times 10^{10} + mnp \times 10^4 + mnp \times 10^0$

  $= mnp \times 10,000,010,001$
Computing Rank in \( O(1) \)

```c
const uint64_t kMultiplier = 0b1000000010000000...100000001;

uint64_t tiledK     =  k * kMultiplier;
```

<table>
<thead>
<tr>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
<td>1100111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( y_1 )</th>
<th>( y_2 )</th>
<th>( y_3 )</th>
<th>( y_4 )</th>
<th>( y_5 )</th>
<th>( y_6 )</th>
<th>( y_7 )</th>
<th>( y_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101001</td>
<td>01011101</td>
<td>01100111</td>
<td>01101010</td>
<td>01101011</td>
<td>01101101</td>
<td>01101110</td>
<td>01111111</td>
</tr>
</tbody>
</table>
Fundamental Primitive: *Parallel Tile*

**Input:** A number $k$ much smaller than a machine word.

**Output:** A machine word holding multiple tiled copies of $k$, spread out with gaps between each copy.

**Procedure:**

1. Form a number $M$ with a 1 bit at the end of each location to tile $k$.

2. Compute $M \times k$. 
Computing Rank in O(1)

```cpp
const uint64_t kMultiplier = 0b1000000010000000...100000001;
const uint64_t kOnesMask   = 0b1000000010000000...010000000;

uint64_t tiledK     = (k * kMultiplier) | kOnesMask;
uint64_t comparison = (tiledK - packedKeys) & kOnesMask;
```

How do we count how many of these bits are set?
Summing Up Flags

- After performing our subtraction, we’re left with a number like this one, where the highlighted bits are “interesting” to us.

- **Goal:** Add up these “interesting” values using O(1) word operations.
An Initial Idea

• To sum up the flags, we could extract each bit individually and add the result.

• **The catch:** This takes time $\Theta(r)$, where $r$ is the number of times we tiled our value.

• Can we do better?

  a00000000  b00000000  c00000000  d00000000
A Shifty Solution

• Given this number:

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]

we want to compute \( a + b + c + d \).

• We can’t efficiently isolate \( a, b, c, \) and \( d \).

• **Claim:** We don’t have to!

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]

\[ + \]

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{cccc}
    a & 0 & 0 & 0 \\
    b & 0 & 0 & 0 \\
    c & 0 & 0 & 0 \\
    d & 0 & 0 & 0 \\
\end{array} \]
A Shifty Solution

• Given this number:

\[ \begin{array}{cccc}
  a & b & c & d \\
  a & b & c & d \\
  a & b & c & d \\
  \end{array} \]

we want to compute \( a + b + c + d \).

• We can’t efficiently isolate \( a, b, c, \) and \( d \).

• **Claim:** We don’t have to!

This is a series of shifts and adds. It’s equivalent to multiplying our original number by some well-chosen spreader!
Fundamental Primitive: *Parallel Add*

**Input:** A machine word with “interesting” bits spaced evenly across the word.

**Output:** The sum of those “interesting” bits.

**Procedure:**

1. Perform a *parallel tile* with an appropriate multiplier to place all leading bits on top of one another.

2. Use a bitmask and bitshift to isolate those bits.
Computing Rank in O(1)

```c
const uint64_t kMultiplier = 0b1000000010000000...100000001;
const uint64_t kOnesMask   = 0b1000000010000000...010000000;

uint64_t tiledK     = (k * kMultiplier) | kOnesMask;
uint64_t comparison = (tiledK – packedKeys) & kOnesMask;

const uint64_t kStacker = 0b1000001000001...1000001;
const uint8_t  kShift   = 31;
const uint64_t kMask    = 0b111;

uint64_t rank = ((comparison * kStacker) >> kShift) & kMask;
```

```
+ a0000000 b0000000 c0000000 d0000000
    a0000000 b0000000 c0000000 d0000000
    a0000000 b0000000 c0000000 d0000000
    a0000000 b0000000 c0000000 d0000000
    a0000000 b0000000 c0000000 d0000000

__________________________ sum
```
Fundamental Primitive: **Parallel Rank**

**Input:** An array of integers packed into a machine word with one bit of space between integers, and a key $k$.

**Output:** How many elements of the array are less than or equal to $k$.

**Procedure:**

1. Perform a *parallel tile* to create $n$ copies of the key $k$, prefixed by 1’s.
2. Perform a *parallel compare* of the key $k$ against values $x_1, \ldots, x_n$.
3. Perform a *parallel add* to sum those values into some total $t$.
4. Return $t$. 
The Sardine Tree

- Let $w$ be the word size and $s$ be some (much) smaller number of bits.

- A sardine tree is a B-tree of order $\Theta(w/s)$ where the keys in a node are packed into a single machine word.
  - Get it? The keys are “packed” tightly into a machine word! I’m funny.

- Each node is annotated with several values (the masks and multipliers from the preceding slide), which are updated in time $O(1)$ whenever a key is added or removed.

- Supports all ordered dictionary operations in time $O(\log_b n) = O(\log_{w/s} n)$. 
The Scorecard

- Here’s the performance breakdown for the sardine tree.
- Notice that the runtime performance is strictly better than that of a BST!
- Notice that the space usage is sublinear, since each node stores multiple keys!

The Sardine Tree

- **lookup**: $O(\log_{w/s} n)$
- **insert**: $O(\log_{w/s} n)$
- **delete**: $O(\log_{w/s} n)$
- **max**: $O(\log_{w/s} n)$
- **succ**: $O(\log_{w/s} n)$
- Space: $\Theta(n \cdot s/w)$
For Comparison

- It’s helpful to compare the sardine tree against the y-fast trie.

- To make the comparison fair, I’m treating the y-fast trie as though the machine word size is $s$ rather than $w$.

The y-Fast Trie

- **lookup**: $O(\log s)$
- **insert**: $O(\log s)$*
- **delete**: $O(\log s)$*
- **max**: $O(\log s)$
- **succ**: $O(\log s)$
- Space: $\Theta(n)$

* Expected, amortized
What’s Next

- **Question**: Can we get performance along these lines even if the keys fill full machine words?
- The strategy used in the sardine tree on its own won’t get us there – but many of those same techniques will!
- We’ll see how to do this next time. In the meantime, let’s see some other cool tricks we can do with word-level parallelism.

---

**Mystery Structure?**

- **lookup**: $O(\log_w n)$
- **insert**: $O(\log_w n)$
- **delete**: $O(\log_w n)$
- **max**: $O(\log_w n)$
- **succ**: $O(\log_w n)$
- Space: $\Theta(n)$
Word-Level Parallelism Tricks #2: Most-Significant Bits
Most-Significant Bits

• The *most-significant bit* function, denoted $\text{msb}(n)$, outputs the index of the highest 1 bit set in the binary representation of number $n$.

• Some examples:

  $\text{msb}(0110) = 2$  $\text{msb}(010100) = 4$  $\text{msb}(1111) = 3$

• Note that $\text{msb}(0)$ is undefined.

• Mathematically, $\text{msb}(n)$ is the largest value of $k$ such that $2^k \leq n$. (*Do you see why?*)
Most-Significant Bits

• Although we didn’t have this name earlier in the quarter, you’ve seen a place where we needed to efficiently compute $\text{msb}(n)$.

• Do you remember where?

• **Answer:** In the sparse table RMQ structure, where computing $\text{RMQ}(i, j)$ requires computing the largest number $k$ where $2^k \leq j - i + 1$.

• That’s exactly the value of $\text{msb}(j - i + 1)$!
Most-Significant Bits

- On many architectures, there’s a single assembly instruction that computes $\text{msb}(n)$.
  - on x86, it’s $\text{BSR}$ (bit scan reverse).
- On others, nothing like this exists.
  - Older versions of MIPS, for example.
- **Question:** How would we compute $\text{msb}(n)$ assuming we only have access to the regular C operators?

  ```
  +  -  *  /  %  <<  >>  &  |  ^  ==  <=
  ```
Computing $\text{msb}$

- In Problem Set 1, you (probably) computed $\text{msb}(n)$ by building a lookup table mapping each value of $n$ to $\text{msb}(n)$.

- **The Good:** This takes time $O(1)$ to evaluate.

- **The Bad:** The preprocessing time, and space usage, is $\Theta(U)$, where $U$ is the maximum value we’ll be querying for.

- **The Ugly:** In the worst case $U = 2^w$.

- Can we do better?
Most-Significant Digits

- Can you compute most-significant digits
  - ... in time $O(w)$ using $O(1)$ space?
  - ... in time $O(\log w)$ using $O(1)$ space?
  - ... in time $O(1)$ using $O(1)$ space?
- Remember that the word size $w$ is not a constant and that we can only use C-style operations.
Most-Significant Bits

• There’s a simple $O(w)$-time algorithm for computing $\text{msb}(n)$ that just checks all the bits until a 1 is found:

```c
for (uint8_t bit = 64; bit > 0; bit--) {
    if (n & (uint64_t(1) << (bit - 1))) {
        return bit;
    }
}
flailAndPanic();
```

• Can we do better?
Computing msb

- We can improve this runtime to $O(\log w)$ by using a binary search:
  - Check if any bits in the upper half of the bits of $n$ are set.
  - If so, recursively explore the upper half of $n$.
  - If not, recursively explore the lower half of $n$.
- We can test whether any bit in a range is set by ANDing with a mask of 1s and seeing if the result is nonzero:

  \[
  \begin{array}{cccccccccccc}
    11011100 & 10110111 & 11000100 & 11010101 & 11100110 & 11110111 & 11000010 & 00110010 \\
    \land & 11111111 & 11111111 & 11111111 & 11111111 & 00000000 & 00000000 & 00000000 & 00000000
  \end{array}
  \]

- Can we do better?
Claim: For any machine word size $w$, there is an algorithm that uses $O(1)$ machine operations and $O(1)$ space – independently of $w$ – and computes $\text{msb}(n)$.

This is not obvious!
How is this possible?
Not Starting from Scratch

• We’re not going into this problem blind. We’ve seen a bunch of useful techniques so far:
  • *Parallel compare:* We can compare a bunch of small numbers in parallel in O(1) machine word operations.
  • *Parallel tile:* We can take a small number and “tile” it multiple times in O(1) machine word operations.
  • *Parallel add:* If we have a bunch of “flag” bits spread out evenly, we can add them all up in O(1) machine word operations.
  • *Parallel rank:* We can find the rank of a small number in an array of small numbers in O(1) machine word operations.

• This is an impressive array of techniques. Let’s see if we can reuse or adapt them.
MSBs as Ranks

• *Recall:* $\text{msb}(n)$ is the largest value of $k$ for which $2^k \leq n$.

• *Idea:* Imagine we have an array of all the powers of two that we can represent in a machine word. Then $\text{msb}(n)$ is the rank of $n$ in that array!
The Problem

- We can compute the rank of a value in an array assuming that all the array entries fit into a single machine word.
- This isn’t the case here:
  - $w$ total powers of two to write out.
  - Total bits needed: $\Theta(w^2)$, way too big to fit into a word.
- **Question:** Can we still harness the benefits of this parallel rank operation?
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a \( w \)-bit integer.
  - In what follows, we’ll pick \( w = 64 \), but this works for any \( w \).
- We ultimately want to be finding the MSB of numbers with way fewer than \( w \) bits.
- **Idea:** Split \( w \) into some number of blocks of size \( b \). Then,
  - find the index of the highest block with at least one 1 bit set, then
  - find the index of the highest bit within that block.
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a $w$-bit integer.
  - In what follows, we’ll pick $w = 64$, but this works for any $w$.
- We ultimately want to be finding the MSB of numbers with way fewer than $w$ bits.
- **Idea:** Split $w$ into some number of blocks of size $b$. Then,
  - find the index of the highest block with at least one 1 bit set, then
  - find the index of the highest bit within that block.
A Nice Decomposition

- Imagine we want to compute the most-significant bit of a $w$-bit integer.
  - In what follows, we’ll pick $w = 64$, but this works for any $w$.
- We ultimately want to be finding the MSB of numbers with way fewer than $w$ bits.
- **Idea:** Split $w$ into some number of blocks of size $b$. Then,
  - find the index of the highest block with at least one 1 bit set, then
  - find the index of the highest bit within that block.
A Nice Decomposition

- We will compute the MSB for \( w \)-bit integers by solving MSB for \( b \) and \( w/b \)-bit integers.
- What choice of \( b \) minimizes \( \max\{b, w/b\} \)?
- **Answer:** Pick \( b = w^{1/2} \).
- So now we need to see how to
  - solve \( \text{msb}(n) \) for integers with \( w^{1/2} \) bits, and
  - replace each block with a bit indicating whether that block contains a 1.
MSB for $w^{1/2}$ Bits

- **Recall:** We can compute $\text{msb}(n)$ by counting how many powers of two are less than or equal to $n$.
- If our numbers have size $w^{1/2}$, there are $w^{1/2}$ powers of two to compare against.
- Each of those powers of two has $w^{1/2}$ bits, so all of those powers of two can be packed into a single machine word!
- **Idea:** Use our O(1)-time rank algorithm!
MSB for $w^{1/2}$ Bits

- If our numbers have size $w^{1/2}$, there are $w^{1/2}$ powers of two to compare against, each of which has $w^{1/2}$ bits.
- Our parallel comparison prepends an extra bit to each number to compare.
- That’s barely – just barely – too many bits to fit into a machine word.
MSB for $w^{1/2}$ Bits

- **Claim:** This is an engineering problem at this point.
- **Option 1:** Split the powers of two into two different machine words and do two rank calculations.
- **Option 2:** Special-case the most-significant bit to reduce the number of bits to check.
- Either way, we find that the work done here is $O(1)$ machine operations, with no dependency on the word size $w$!
A Nice Decomposition

• We need to see how to
  • solve $\text{msb}(n)$ for integers with $w^{1/2}$ bits, and
  • replace each block with a bit indicating whether that block contains a 1.
### Identifying Active Blocks

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
<th>$b_5$</th>
<th>$b_6$</th>
<th>$b_7$</th>
<th>$b_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00011001</td>
<td>11110000</td>
<td>10011010</td>
<td>10000000</td>
<td>00011010</td>
<td>11101110</td>
<td>11000010</td>
</tr>
</tbody>
</table>

**Observation**: A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.
Identifying Active Blocks

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
<th>$b_5$</th>
<th>$b_6$</th>
<th>$b_7$</th>
<th>$b_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00011001</td>
<td>11110000</td>
<td>10011010</td>
<td>10000000</td>
<td>00011010</td>
<td>11101110</td>
<td>11000010</td>
</tr>
<tr>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>00000000</td>
<td>10000000</td>
<td>10000000</td>
</tr>
</tbody>
</table>

**Observation:** A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.
Identifying Active Blocks

A number’s lower 7 bits contain a 1 if and only if the numeric value of those bits is at least 1.

**Observation:** A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.
## Identifying Active Blocks

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Observation:** A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.

- **Low bits set?**
  - 00000000 10000000 10000000 10000000 00000000 10000000 10000000 10000000

- **High bit set?**
  - 00000000 00000000 10000000 10000000 10000000 00000000 10000000 10000000
Identifying Active Blocks

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
<th>$b_5$</th>
<th>$b_6$</th>
<th>$b_7$</th>
<th>$b_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00011001</td>
<td>11110000</td>
<td>10011010</td>
<td>10000000</td>
<td>00011010</td>
<td>11101110</td>
<td>11000010</td>
</tr>
<tr>
<td>00000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
<td>10000000</td>
</tr>
</tbody>
</table>

*Observation:* A block contains a 1 bit if its first bit is 1 or its lower 7 bits contain a 1.
Identifying Active Blocks

• We now have a word holding flags telling us which blocks have a 1 bit set.

• We need to find the highest set flag.

• There are only $\sqrt{w}$ flags. If we could compact them into $\sqrt{w}$ adjacent bits, we could use our earlier algorithm to find the highest one set!
Identifying Active Blocks

• **Idea:** Adapt the shifting technique we used to compute ranks.

• Instead of shifting the bits on top of one another, shift the bits next to one another:

```plaintext
a0000000b0000000c0000000d0000000

a0000000b0000000c0000000d0000000

a0000000b0000000c0000000d0000000

+ a0000000b0000000c0000000d0000000

????????????????????????????????????????????abced?????????????????????????????????????????????
**Fundamental Primitive:** *Parallel Pack*

**Input:** A machine word containing several “interesting” bits that are evenly spaced apart.

**Output:** A machine word with those “interesting” bits placed adjacent to one another at the low end of the word.

**Procedure:**

1. Perform a *parallel tile* with an appropriate multiplier to place all leading bits adjacent to one another.

2. Use a bitmask and bitshift to isolate those bits.
Putting It All Together

- Use a bitmask to identify all blocks whose high bit is set.
- Use a *parallel tile* and a *parallel compare* to identify all blocks with a 1 bit aside from the first.
- Use a *parallel pack* to pack those bits together.
- Use a *parallel rank* to determine the highest of those bits set, which gives the block index.
- Use a *parallel rank* to determine the highest bit set within that block.
The Finished Product

• I’ve posted a link to a working implementation of this algorithm for 64-bit integers on the course website.

• Feel free to check it out – it’s really magical seeing all the techniques come together!
What We Covered

- We can use bit-parallel tricks to
  - compare multiple values in parallel,
  - tile a number across a word,
  - sum up evenly-spaced bits in a word,
  - compute ranks in an array,
  - compact evenly-spaced bits in a word, and
  all in $O(1)$ machine word operations!

- Using these techniques, we can modify a B-tree to work strictly faster than a conventional BST, provided that we store tiny keys.

- Using these techniques, can we compute the most-significant bit of a machine word in time $O(1)$, independent of the machine word size.

- And all of this flows from one source: *word-level parallelism* inside of the processor!
What’s Next

- Can we build a data structure for integers that is *strictly better* than a binary search tree?
- The answer is *yes*, and it’s called a fusion tree.
- Today’s exploration provides the techniques we’ll use to build the fusion tree. We just need a few more insights to get us there!
Next Time

- **Patricia Codes**
  - Compressing a small number of big integers into a small number of small integers.

- **Fusion Trees**
  - Combining all these techniques together!