Recitation 1.2

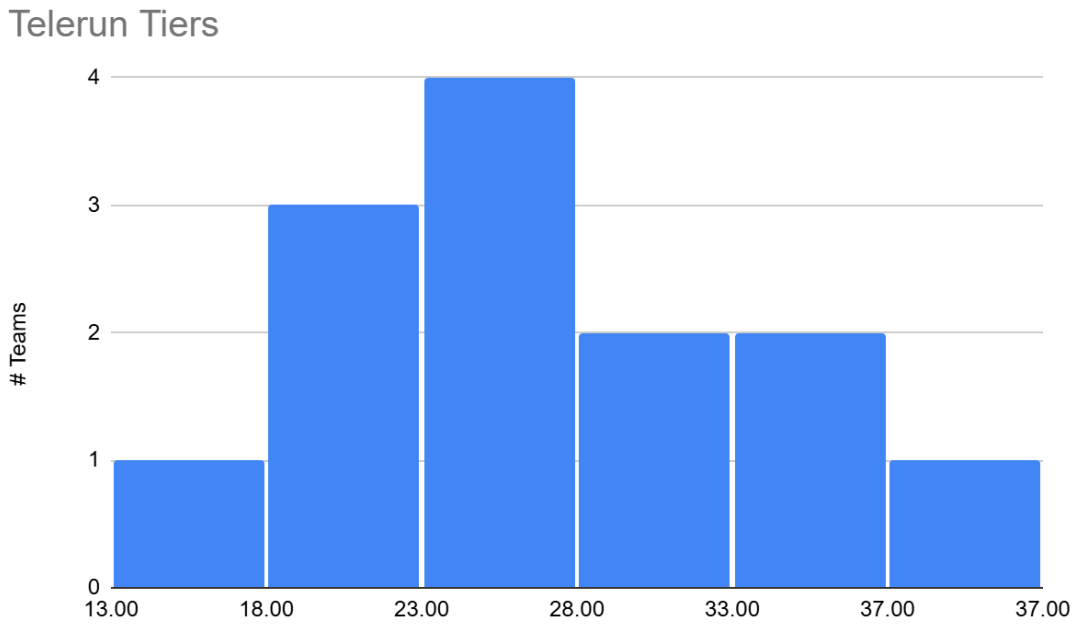
Recitation 1.1 Feedback [Project 1 Beta help]

- Was Recitation 1.1 helpful?
- Any improvements to suggest?

Recitation 1.2

- A very short introduction to what is available for Project 1 Final
- After that, you are free to work on HW3 or get started on Project 1 Final
- Rest of the class I'll be going around asking if I could help with anything that has happened so far in the course

Project 1 Beta



Vectorization

Basic Vector Instructions

- Vector Register Sizes: xmm: 128, ymm: 256, ~~zmm: 512~~ bits
- Packing: Fit many values in one register
  - xmm fits 2x uint64_t, 4x int32_t, 4x float, etc… (in general = `(128/8) / sizeof(T)`)
- Packed Arithmetic Operations
  - add, subtract, multiply, divide, reciprocate, max, min, sqrt, reciprocal of sqrt
- Packed Comparison Operations
- Packed Logical Operations
- Packed Type Conversions
  - int to float etc

Overlapping Memory Regions

- Difference between memcpy and memmove
- When `A[]` and `B[]` overlap, `memcpy(B, A, n)` can be wrong
  - because the memcpy can overwrite data from the source, which it would need to use later
- Same thing can happen with vector operations
- To vectorize codes that deal with two arrays, need to know they don't overlap
- We can mark this using the restrict keyword
- `int* restrict A, int* restrict B`
  - The only way to obtain pointers from A's memory region is by offsetting from A
  - Thus A and B cannot overlap because there are no (non UB) offsets into each other

What can a compiler auto-vectorize?

- “Pure” (not interdependent) loop iterations
  - Dependence between loop iterations, e.g. prefix sum
- Reductions
  - Sum or product of contiguous memory: can detect the result variable is used for reduction
- Inductions
  - `A[i] = i;`
- If Statements
  - Certain if statements can be converted to branchless code
- Stride
  - `A[i] += B[i * 4];`
- When restrict is not known
  - Generates both vector and scalar code paths
  - Tests for overlap at runtime

https://llvm.org/docs/Vectorizers.html

Associativity of Reductions

- `1 + (2 + 3) == (1 + 2) + 3`
- `0.1 + (0.2 + 0.3) != (0.1 + 0.2) + 0.3`

- Floating point addition is not associative. This means order matters.
- -O3 has to produce code that behaves exactly the same as -O0
- Vector operations do not add the values in the same order as a scalar loop.
- For float: Need the `-ffast-math` flag to tell the compiler we're ok with this.

Good Reading:
"What Every Computer Scientist Should Know About Floating-Point Arithmetic"

Ben Bitdiddle wants to optimize the following piece of working code:

```
/* The elements of A and B are known to be nonnegative and not
   aliased */
/* k is known only at runtime */
static const int N = (1 << 30);
for (int i = 0; i < N; i++) {
  A[i] = k * B[i];
}
```

Using what he learned in this class, he rewrites the code as:

```
/* The elements of A and B are known to be nonnegative and not
   aliased */
/* k is known only at runtime */
static const int N = (1 << 30);
A[N - 1] = -1;
int *a = A , *b = B;
while (*a >= 0) {
  *(a++) = k * *(b++) ;
}
*a = k * *b;
```

Ben compiles both codes with GCC using optimization `-O3`. When he runs the two codes, however, Ben finds that his new code is several times slower than his old code because it no longer vectorizes. Why does it fail to vectorize?

Given a matrix `M` in row-major format, we wish to compute the row sums (i.e., the sums across the rows).

```
 1 #define ROWS 2048
 2 #define COLS 3072
 3
 4 void compute_row_sums(int M[ROWS][COLS], int row_sums[ROWS]) {
 5   for (int i = 0; i < ROWS; i++) {
 6     row_sums[i] = 0;
 7     for (int j = 0; j < COLS; j++) {
 8       row_sums[i] += M[i][j];
 9     }
10   }
11 }
```

Can the code in `compute_row_sums()` be transformed to use vector hardware effectively? If yes, explain briefly where and how to vectorize `compute_row_sums()`. If no, explain briefly why the function cannot be vectorized.

Useful Intrinsics

Alignment Hint

```
void *__builtin_assume_aligned(const void *arg, size_t align);
```

Returns its first argument and allows the compiler to assume that the returned pointer is at least `align` bytes aligned.

```
// x is at least 16 byte aligned
void *x = __builtin_assume_aligned(arg, 16);


// (char *) x - 8 is 32 byte aligned
void *x = __builtin_assume_aligned(arg, 32, 8);
```

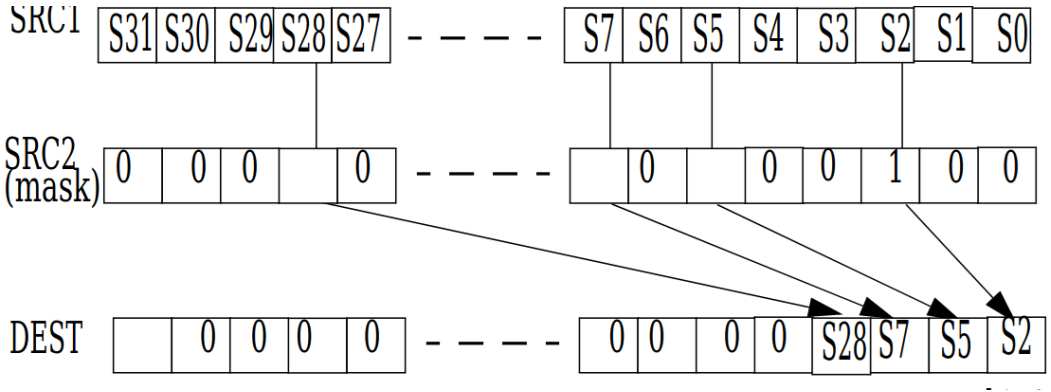
Other Bit Manipulations

- `uint64_t __builtin_rotateleft64(uint64_t x, bits_t amt);`
- `uint64_t __builtin_rotateright64(uint64_t x, bits_t amt);`
- `uint64_t __builtin_bitreverse64(uint64_t x);`
- `uint64_t __builtin_bswap64(uint64_t x);`
- `bits_t __builtin_popcountll(uint64_t x);`
- `bits_t __builtin_clzll(uint64_t x);`
- `bits_t __builtin_ctzll(uint64_t x);`

Parallel Bit Extraction

```
#include <immintrin.h>
uint64_t _pext_u64(uint64_t s1, uint64_t mask);
```

Transfer either contiguous or non-contiguous bits in the first source operand to contiguous low order bit positions in the destination according to the mask values.



Packed Shift

```
#include <immintrin.h>
__m128i _mm_sll_epi64(__m128i reg, __m128i count);
```

Shift the two **64** bit numbers packed in `reg` left by `count`.

Pre Fetching!

```
__builtin_prefetch (const void *addr[, rw[, locality]])
```

Takes:

- Addr to prefetch from
- Read mode or write mode
- Locality (L1 /L2/ L3/ auto)

Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide