SIMD Scans

CSC 490: Lock-free, GPU & vectorization

Assignment 2

Victor Kamel
vkamel@uvic.ca
University of Victoria
Victoria, BC, Canada

1 INTRODUCTION

In this report, we will recreate Figure 6 of Zhou and Ross [1], and attempt further performance enhancements using multi-threading.

2 PROBLEM STATEMENT

Suppose that we have an internal B-tree node, stored as specified in [1]. That is, with a sorted, contiguous array of floating point keys and a separate array of pointers to child nodes. In order to find the index of the pointer to the correct child node containing some search key t, we must search through the keys.

In this report, we will use algorithms that produce a result equivalent to `std::lower_bound` in order to calculate the index, although this differs somewhat from the paper. That is to say, the index is the index of the first key $k, k \geq t$. Notably, this is equivalent to counting the number of keys with values strictly less than t.

We will use instructions in the SSE family, with 4-wide SIMD in order to better replicate what the authors of the paper were working with at the time, although our machine does support AVX2. B-tree internal nodes will contain 512 keys.

3 ALGORITHMS

Several algorithms were presented in the paper. We will focus on their implementation rather than the theory here, since the paper does not include source code.

3.1 Sequential Scans

Since the number of keys in a B-tree internal node is relatively small, it is reasonable to expect that a sequential algorithm will perform well. Two variations are presented.

```
uint32_t sequential_search_1(const float array[], float target, std::
    ↪ size_t n)
{
  __m128i vsum = _mm_setzero_si128();

  for (std::size_t i = 0; i < n; i += 4)
    {
      __m128 m = _mm_cmplt_ps(_mm_load_ps(&array[i]), _mm_set_ps1(
      ↪ target));
      vsum = _mm_add_epi32(vsum, _mm_and_si128(_mm_set1_epi32(1),
      ↪ _mm_castps_si128(m)));
    }

    // Coalesce
    vsum = _mm_hadd_epi32(vsum, vsum);
    vsum = _mm_hadd_epi32(vsum, vsum);
    return _mm_cvtsi128_si32(vsum);
}
```

In the first variation, we traverse the whole array, comparing groups of 4 32-bit floating point values at a time. The mask is then converted to a vector of 1's or 0's, and added to a running total. We end with a horizontal add. The advantage of this approach is that we do not require a branch in the main loop.

```
uint32_t sequential_search_2(const float array[], float target, std::
    ↪ size_t n)
{
  __m128i vsum = _mm_setzero_si128();

  for (std::size_t i = 0; i < n; i += 4)
    {
      __m128 m = _mm_cmplt_ps(_mm_load_ps(&array[i]), _mm_set_ps1(
      ↪ target));
      vsum = _mm_add_epi32(vsum, _mm_and_si128(_mm_set1_epi32(1),
      ↪ _mm_castps_si128(m)));
      // only need to check one
      if (_mm_cvtsi128_si32(_mm_castps_si128(m)) == 0) { break; }
    }

    // Coalesce
    vsum = _mm_hadd_epi32(vsum, vsum);
    vsum = _mm_hadd_epi32(vsum, vsum);
    return _mm_cvtsi128_si32(vsum);
}
```

In the second variation, we compare groups of 4 as in the previous, but if we see that the least significant floating point value is greater than or equal to the target, we have reached the block that covers the boundary, and we can return early.

We expect the first variation to be faster than the second with fewer keys, and the second version to become faster as we increase the number of keys.

We note that if we write the first version of the algorithm to be scalar, the compiler (`gcc 13.2.1 20231011`) is able to auto-vectorize and emit AVX2 instructions (compares 8 32-bit floats at a time). This renders it faster than the version that utilizes SSE intrinsics.

```
// Should be autovectorized
uint32_t sequential_search_scalar(const float array[], float target,
    ↪  std::size_t n)
{
    uint32_t c = 0;
    for (std::size_t i = 0; i < n; c += array[i++] < target);
    return c;
}
```

The second version, however, does not get auto-vectorized by default. This is likely due to the branch. In the performance evaluation section, we will not benchmark the auto-vectorized code.

3.2 Binary Search

Binary search is the standard technique to solve this problem. We present two variations, a standard implementation, and the implementation provided by the standard library.

```cpp
// Binary search implementation from the standard library
uint32_t std_binary_search(const float array[], float target, std::
      ↪ size_t n)
  { return std::lower_bound(array, array + n, target) - array; }

// Standard no-frills binary search implementation
uint32_t normal_binary_search(const float array[], float target, std
      ↪ ::size_t n)
{
  uint32_t l = 0;
  uint32_t r = n;

  // Search
  while (l < r) {
    uint32_t c = std::midpoint(l, r);
    if (array[c] < target) { l = c + 1; } // Go right
    else                   { r = c;     } // Go left
  }

  return l;
}
```

Zhou and Ross [1] present a "naive" SIMD optimization to this algorithm, where groups of 4 keys are compared at a time. If the resulting mask is all zeros, we search left, if the mask is all ones, we search right, and otherwise, the boundary is within the current 4-block. The advantage is that we can return early, since looking at a single value is not sufficient to make this determination, but a group of 4 is (as long as it does not fall on a boundary). This is because we are looking for a transition point between elements, rather than a specific element.

```cpp
uint32_t simd_binary_search(const float array[], float target, std::
      ↪ size_t n)
{
  uint32_t l = 0;
  uint32_t r = n / 4;

  while (l < r) {
    uint32_t c = std::midpoint(l, r);

    __m128 m = _mm_cmplt_ps(_mm_load_ps(array + c * 4), _mm_set_ps1(
      ↪ target));
    if      (_mm_test_all_ones(_mm_castps_si128(m)))
      ↪       { l = c + 1; }
    else if (_mm_test_all_zeros(_mm_castps_si128(m),
      ↪ _mm_castps_si128(m))) { r = c;     }
    else {
      // Find the index
      __m128i vsum = _mm_and_si128(_mm_castps_si128(m),
      ↪ _mm_set1_epi32(1));
      vsum = _mm_hadd_epi32(vsum, vsum);
      vsum = _mm_hadd_epi32(vsum, vsum);
      return c * 4 + _mm_cvtsi128_si32(vsum);
    }
  }

  return l * 4;
}
```

This implementation was compared with a version closer to `normal_binary_search` with only a single check in the branch (`test_all_ones` is cheaper than `test_all_zeros` since the latter is combined with bitwise AND), however returning early performed better.

3.3 Hybrid Techniques

Zhou and Ross [1] note, based on the asymptotic analysis, that a hybrid binary search that divides an array into L-length segments and binary searches among the segments may perform faster for some values of L based on the hardware. Since the original paper did not mention whether this was done using the naive SIMD algorithm or the regular one, we have implemented both.

```cpp
uint32_t hybrid_binary_search(const float array[], float target, std
      ↪ ::size_t n, std::size_t lim)
{
  assert(lim % 4 == 0);
  uint32_t l = 0;
  uint32_t r = n / lim;

  // Search
  while (l < r) {
    uint32_t c = std::midpoint(l, r);
    if (array[std::min((c * lim) + lim - 1, n - 1)] < target) { l =
      ↪ c + 1; } // Go right
    else                                                      { r =
      ↪ c;     } // Go left
  }

  std::size_t pos = l * lim;
  return pos + sequential_search_1(array + pos, target, std::min(lim,
      ↪  n - pos));
}
```

In this case, we have to compare the right-most value in each block only, then finish with a sequential search. Since the array may not be perfectly divisible by L, we must clamp the right edge of the array during indexing.

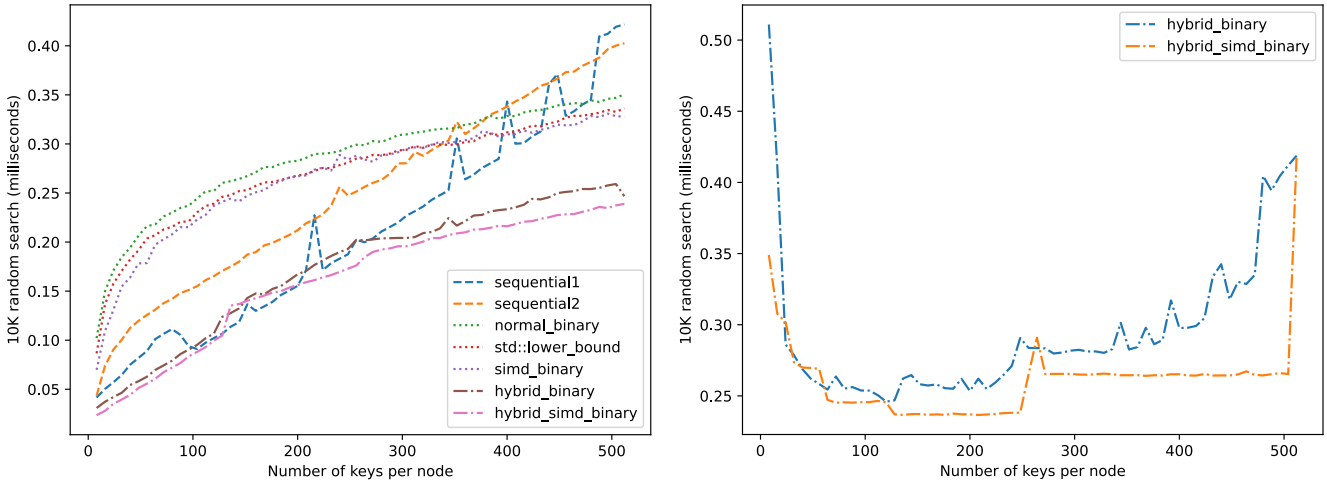
Finally, we hybridize with the SIMD variant.

```cpp
// Hybrid binary search (switch to sequential at lim)
uint32_t hybrid_simd_binary_search(const float array[], float target,
      ↪  std::size_t n, std::size_t lim)
{
  uint32_t l = 0;
  uint32_t r = n / 4;

  while (r - l > (lim / 4)) {
    uint32_t c = std::midpoint(l, r);

    __m128 m = _mm_cmplt_ps(_mm_load_ps(array + c * 4), _mm_set_ps1(
      ↪ target));
    if      (_mm_test_all_ones(_mm_castps_si128(m)))
      ↪       { l = c + 1; }
    else if (_mm_test_all_zeros(_mm_castps_si128(m),
      ↪ _mm_castps_si128(m))) { r = c;     }
    else {
      // Find the index
      __m128i vsum = _mm_and_si128(_mm_castps_si128(m),
      ↪ _mm_set1_epi32(1));
      vsum = _mm_hadd_epi32(vsum, vsum);
      vsum = _mm_hadd_epi32(vsum, vsum);
      return c * 4 + _mm_cvtsi128_si32(vsum);
    }
  }

  assert(l <= r);

  return (l * 4) + sequential_search_1(array + (l * 4), target, (r -
      ↪  l) * 4);
}
```

Notably, here we do not split the array into segments of length L, but rather we search until the search space is sufficiently narrow. This allows us to retain the ability to stop the search early.

4 PERFORMANCE EXPERIMENTS

We test in the same way as in the original paper, with an array of 512 32-bit floating point values as keys, aligned to the 16-byte boundary.



(a) Internal Node Traversal

(b) Hybrid Search: Choice of L value

Figure 1: Comparison of Search Methods

Algorithm	*Min*	*Avg*	*Max*
`sequential_search_1`	0.827	1.654	2.993
`sequential_search_2`	0.867	1.262	2.351
`std::lower_bound`	1.033	1.058	1.180
`simd_binary_search`	1.013	1.087	1.464
`hybrid_binary_search`	1.339	1.923	4.068
`hybrid_simd_binary_search`	1.454	2.124	5.348

Figure 2: Speed up as compared to `normal_binary_search`.

4.1 Algorithm Comparison

To test the algorithms' relative performance, We pre-compute 10, 000 random search keys. For each algorithm, for each value N from 4 to 512, we will generate a random array of N sorted keys, and time how long it takes to complete all 10, 000 searches. This is repeated 15 times, and the average time is recorded. The results are given in Figure 1a.

We note that the results are consistent with Zhou and Ross [1]. Of notable difference, is that the algorithms are in general faster, and the intersection point where `sequential_search_1` becomes slower than `sequential_search_2`, and likewise the sequential algorithms become slower than the binary search algorithms has moved further right. This is likely both due to the difference in hardware, and the difference in the compiler.

As we expected, the SIMD-optimized binary search performed better than the normal binary search, although the difference is less pronounced compared to `std::lower_bound`.

We also see that the hybrid SIMD-optimized algorithm performs better than the regular hybrid algorithm.

As seen in Table 2, can see that compared to `normal_binary`, the average speed up of the best-performing algorithm was at just over a two times improvement.

4.2 Choosing an L-Value

In order to determine the value of L for the hybrid algorithms, we time the 10,000 searches with a fixed array length of 512, while varying L from 4 to 512.

As shown in Figure 1b, the result for the hybrid binary search algorithm is consistent with the paper. The result for the SIMD-optimized algorithm is better, and the "shelves" are expected, since we are not searching among L-length segments, but rather finishing the search once we have divided the search space sufficiently. According to the graph, we chose a value of 128 as the L value for the hybrid algorithms.

4.3 Perf Measurements

The command line tool `pref` was used to count SIMD instructions retired. Due to our CPU architecture, however, the only performance counter that is accessible is `fp_ret_sse_avx_ops.all`. However, this counts *floating point* SSE/AVX operations. Although we are working with floating point data, we avoid floating point operations, and thus none of the instructions are counted. To circumvent this "issue", we re-implemented `sequential_search_1` using floating point operations:

```
1  static uint32_t sequential_search_1ps(const float array[], float
       ↪ target, std::size_t n)
2  {
3    __m128 vsum = _mm_setzero_ps();
4
5    for (std::size_t i = 0; i < n; i += 4)
6      {
7        __m128 m = _mm_cmplt_ps(_mm_load_ps(&array[i]), _mm_set_ps1(
       ↪ target));
8        vsum = _mm_add_ps(vsum, _mm_and_ps(_mm_set1_ps(1), m));
9      }
10
11     // Coalesce
12     vsum = _mm_hadd_ps(vsum, vsum);
13     vsum = _mm_hadd_ps(vsum, vsum);
14     return _mm_cvtss_f32(vsum);
15 }
```

Thus, we run `pref` on code that runs 10,000 pre-calculated queries on pre-generated node of size 512:

```
$ perf stat -e cycles:u,instructions:u,fp_ret_sse_avx_ops.all:u ./
    ↪ simd

Performance counter stats for './simd':

     5,906,585      cycles:u
    11,874,977      instructions:u          #  2.01  insn per cycle
     5,200,000      fp_ret_sse_avx_ops.all:u

   0.004008594 seconds time elapsed

   0.002440000 seconds user
   0.001626000 seconds sys
```

As we can see, a significant number of SIMD instructions are being retired.

5 MULTI-THREADING

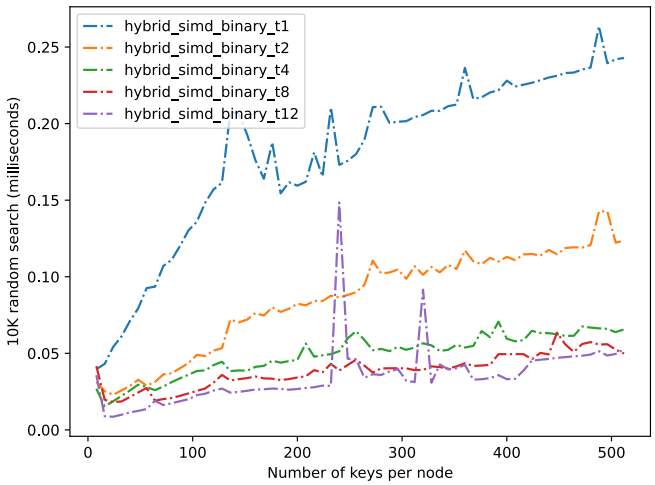


Figure 3: Hybrid Search: Multi-threading

We will now layer on multi-threading using OpenMP's `parallel for` compiler directive (with a static schedule) onto the best-performing hybrid algorithm in order to attain a greater speed-up. We note that we are parallelizing across invocations of the search function. Thus, a single search is not made any faster by using this method.

The parameters of the experiment are identical to §4.1, but we average across 5 runs.

From Table 4, we can see that compared to our baseline of the algorithm running one a single thread, achieve an average speed up of up to about 6 times, with a maximum of just over 8 times.

Algorithm	*Min*	*Avg*	*Max*
`hybrid_simd_binary_t2`	1.265	2.211	3.204
`hybrid_simd_binary_t4`	1.474	3.651	5.413
`hybrid_simd_binary_t8`	0.950	4.678	6.445
`hybrid_simd_binary_t12`	1.110	5.689	8.547

Figure 4: Speed up as compared to `hybrid_simd_binary_t1`.

6 CONCLUSION

Results from Zhou and Ross [1] were replicated. It was shown that some measure of additional performance can be gained by running scans in parallel.

REFERENCES

[1] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/564691.564709