Quicksort using SIMD instructions

Jun Makino

Internal Seminar Mar 9, 2022

Talk plan

1. Why quicksort with SIMD?
2. Aren’t there existing implementations?
3. Basics of Quicksort
4. SIMD partition algorithm
5. Current implementation (AVX2)
6. Future Plan

Why quicksort with SIMD?

- **Sorting is used in many, many applications**
- **We use sorting to construct trees in FDPS. This is actually the most time-consuming part of the calculation other than tree traversal and interaction calculation.**
- **However, what are available are not quite the fastest.**
  - **std::sort is a sequential quicksort (actually an introsort, a combination of quicksort and heapsort)**
  - **Parallel sorts are not quite fast**
  - **SIMD instructions are not used**

Aren’t there existing implementations?

- There are many papers on implementation of fast sorting algorithms using SSE, AVX, AVX2, AVX512 and even SVE.
- However, almost all of them cannot be used as a replacement of std::sort.
  - They just sort 32-bit integers
  - What we want to do is to sort, for example, particles using the Morton key.
  - Actually, with usual interface of std::sort [or qsort(3)], we cannot use SIMD sort. They only supply a comparison function.
- We can still make generic interface if we provide a function to generate integer keys from data to be sorted (my samplesortlib `https://github.com/jmakino/sortlib` uses this interface)

Basics of Quicksort

1. We have array a with n elements.
2. Pick one “pivot” value from these n elements
3. divide a into two (or three) parts. The left part contains all values smaller than pivot, the right part larger than, and the middle part equal to.
4. Apply steps 2 and 3 recursively to left and right part. If the number of element is one, do nothing.

An example

```
input:  2 5 1 4 9 5 2 1 6 1

pivot: 2:   1 1 1   2 2   5 4 9 5 6

pivot 1: 1 1 1, (2 2 part finished)  pivot:5  4  5 5  9 6

(1 1 1 part finished)                            pivot 9 : 6 9

result: 1 1 1 2 2  4 5 5 6 9
```

Example Source code

(not quite sure why and how this works, though)

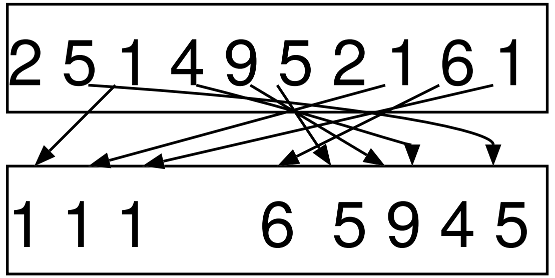
```
void sort_int64_array(int64_t* r, int lo, int up)
{
    int i, j;
    int64_t tempr;
    while ( up>lo ) {
        i = lo;
        j = up;
        tempr = r[lo];
        /*** Split data in two ***/
        while ( i<j ) {
            for ( ; r[j]> tempr; j-- );
            for ( r[i]=r[j]; i<j && r[i]<=tempr; i++ );
            r[j] = r[i];
        }
        r[i] = tempr;
```

Example Source code(continued)

```
        /*** Sort recursively, the smallest first ***/
        if ( i-lo < up-i ) {
            sort_int64_array(r,lo,i-1);  lo = i+1;
        }else{
            sort_int64_array(r,i+1,up);  up = i-1;
        }
    }
}
```

This code requires no additional memory (might have been important in 1960s, when the main memories of computers were small

SIMD partition algorithm



Here, we start from a simple algorithm which require additional working memory.

1. **We have array a with n elements.**
2. **Pick one "pivot" value from these n elements**
3. **prepare an empty array b of size n.**
4. **For each element of array a, if it is smaller than the pivot, put it in the leftmost free location in b. If larger, rightmost. If equal, do nothing**
5. **copy left and right parts of array be back to array a.**
6. **Apply steps 2-5 recursively to the left and right part of array a.**

SIMD partition algorithm(continued)

We can use SIMD instructions to implement the algorithm described in the previous slide. Within an SIMD word,

1. Mark values less than the pivot
2. Move these values to the left side (in a separate SIMD word)
3. Mark values larger than the pivot
4. Move these values to the right side (in yet another separate SIMD word)
5. copy smaller values to leftmost free locations of array b
6. copy larger values to rightmost free locations of array b

All steps can be done using SIMD instructions

Current implementation with AVX2

`https://github.com/jmakino/simdsort` **(contains several bugs...)**

```
register union m256di u, u2;
u.i =  _mm256_cmpgt_epi64(pivotv, pwork[ii]);   // set flags for smaller
int maskl = _mm256_movemask_pd(u.d);            // get bit pattern
u.i =  _mm256_cmpgt_epi64( pwork[ii], pivotv);
int masku = _mm256_movemask_pd(u.d);
register union m256di lower, upper;
int dl = popcount_table_upper[maskl];           // count smaller
int dh = popcount_table_upper[masku];
lower.f =_mm256_permutevar8x32_ps(*((__m256*)(pwork+ii)),  //pack smaller
                          *((__m256i*)(permute_table_lower+maskl)));
mm256_storeu_pd((double*)(data+l+1), lower.d);             //copy smaller
upper.f =_mm256_permutevar8x32_ps(*((__m256*)(pwork+ii)),
                          *((__m256i*)(permute_table_upper+masku)));
mm256_maskstore_pd((double*)(data+h-4), mask_table[masku], upper.d);
l+=dl;
h-=dh;
```

Remarks

- The size of array (in particular during recursion) is not always an integer multiple of the SIMD width. Therefore we need to take care of the residual part.
- For population count and packing, my current implementation uses tables, each with 16 entries. We could make tables of 256 entries to combine left and right parts and reduce the numbers of table lookup and permute operations.

Performance

Time to sort 100K 64bit integer numbers

function	time(sec)
C qsort(3)	0.0169074
non-simd sort	0.00520091
simd sort	0.00274465

- **Six times faster than C qsort(3)**
- **1.9 times faster than a non-simd quicksort (whose speed is about the same as that of std::sort)**
- **Quite reasonable**

Future Plan (or To-do list)

- Implement AVX512 and SVE versions
- Extend to 128-bit key and key-index pair (or just 192-bit key)
- Incorporate into FDPS

Summary

- Implemented a quicksort function for 64-bit integers using AVX2.
- About two times faster than non-simd quick sort or std::sort
- Plan to extend to AVX512, SVE and key-index pair structure.