Quicksort memory optimizations

https://github.com/jmicjm/quicksort_memory_optimizations

Contents

1 Introduction

This document shows series of consecutive quicksort memory optimizations. Optimization of both qsort() and partition() functions starts from implementation equivalent to the textbook implementation and each shown step reduces memory consumption.

1.1 Code

Snippets in this document contains only implementations of each step without benchmark part. You can find complete code with benchmark in this repository. The code contains msvc specific parts(namely _alloca, __forceinline and masm code to obtain stack pointer).

2 qsort() optimizations

2.1 Naive implementation

```
1  template<typename T, T* (*partition)(std::span<T>)>
2  void qsortRecursive(std::span<T> data)
3  {
4      if (data.size() <= 1) return;
5
6      T* p = partition(data);
7
8      qsortRecursive<T, partition>(data.first(p - &*data.begin()));
9      qsortRecursive<T, partition>(data.last(&*data.end() - p));
10 }
```

This implementation serves as starting point to further optimizations.

2.2 Iterative implementation

```
1  template<typename T, T* (*partition)(std::span<T>)>
2  void qsortIterative(std::span<T> data)
3  {
4      std::vector<std::span<T>> ranges;
5      ranges.push_back(data);
6
7      while (ranges.size() > 0)
8      {
9          auto current = ranges.back();
10         ranges.pop_back();
11
12         T* pivot = partition(current);
13
14         std::span<T> left{ &*current.begin(), pivot };
15         std::span<T> right{ pivot, &*current.end() };
16
17         if (left.size() > 1) ranges.push_back(left);
18         if (right.size() > 1) ranges.push_back(right);
19     }
20 }
```

In comparison to recursive implementation this one doesn't 'store' informations necessary to call partition function(begin and end of range) by passing it to next function call. Instead they are stored in **ranges**(lines 5, 17 and 18) that is used as external stack. In the loop range from the top of **ranges** is removed from it and passed to partition function(line 12). After that subranges created by partitioning are inserted into **ranges**(lines 17 and 18) if their sizes are bigger

than one(i.e. they still have to be sorted). The loop continues until **ranges** is empty.
This implementation saves memory by using just single stack frame(plus one(or zero if inlined) used by partition() function). Stack usage is constant in comparison to $O(log(n))$ in recursive implementation. Total(stack + heap) memory usage is still $O(c * log(n))$ however c constant is smaller because amount of informations stored per range has been reduced. The memory used during processing of single range has been reduced from size of qsort() stack frame to sizeof(std::span<T>).

2.3 Omission of redundant informations about ranges

```
1  template<typename T, T* (*partition)(std::span<T>)>
2  void qsortIterativeNoRedundancy(std::span<T> data)
3  {
4      std::vector<T*> ranges;
5      ranges.push_back(&*data.begin());
6      ranges.push_back(&*data.end());
7
8      while (ranges.size() > 1)
9      {
10         auto top = ranges.back();
11         ranges.pop_back();
12
13         if (!top || !ranges.back()) continue;
14
15         T* pivot = partition({ ranges.back(), top });
16
17         if (pivot - ranges.back() >= 2) ranges.push_back(pivot);
18         else if (ranges.back()) ranges.push_back(nullptr);
19
20         if (top - pivot >= 2)
21         {
22             if (!ranges.back()) ranges.push_back(pivot);
23             ranges.push_back(top);
24         }
25     }
26 }
```

Previous implementation greatly reduced memory requirements but there is still room for improvements. We can notice that the end of each range(omitting ranges that are already sorted) stored in **ranges** is the same as the beginning of the next range. In this implementation only the end of range is added to **ranges**(lines 17 and 23) except the case where previous range was already sorted(line 22). Sorted ranges that aren't on top of **ranges** are marked by nullptr(line 18). When sorted range is found during processing it is omitted(line

13).
Omitting size of stack frame and std::vector overhead this implementation reduces memory consumption by half.

2.4 Avoidance of overallocation

```
1  template<typename T, T* (*partition)(std::span<T>)>
2  void qsortIterativeNoRedundancyStack(std::span<T> data)
3  {
4      Stack<T*> ranges;
5      ranges.push(&*data.begin());
6      ranges.push(&*data.end());
7
8      while (ranges.size() > 1)
9      {
10         auto top = ranges.top();
11         ranges.pop();
12
13         if (!top || !ranges.top()) continue;
14
15         T* pivot = partition({ ranges.top(), top });
16
17         if (pivot - ranges.top() >= 2) ranges.push(pivot);
18         else if (ranges.top()) ranges.push(nullptr);
19
20         if (top - pivot >= 2)
21         {
22             if (!ranges.top()) ranges.push(pivot);
23             ranges.push(top);
24         }
25     }
26 }
```

std::vector used in previous implementations is characterized by excessive memory usage due to overallocation. In this case it is replaced by **Stack** class(line 4), which allocates memory on stack so overallocation is no longer required to maintain $O(1)$ insertion at the end. The code of this class is presented below.

```
1  template<typename T>
2  class Stack
3  {
4      constexpr static auto step_size = std::lcm(sizeof(T), 16) / sizeof(T);
5      T* m_data = nullptr;
6      int32_t m_size = 0;
7      int32_t m_capacity = 0;
8
9  public:
10     __forceinline Stack()
11         : m_data(static_cast<T*>(_alloca(step_size * sizeof(T))) + step_size - 1),
12           m_size(0),
13           m_capacity(step_size) {}
14     __forceinline auto push(T value)
15     {
16         if (m_size == m_capacity)
17         {
18             _alloca(step_size * sizeof(T));
19             m_capacity += step_size;
20         }
21         m_data[-m_size] = value;
22         m_size++;
23     };
24     __forceinline auto pop()
25     {
26         m_size--;
27     };
28     __forceinline auto& top()
29     {
30         return m_data[-(m_size - 1)];
31     };
32     __forceinline auto size() const
33     {
34         return m_size;
35     }
36 };
```

The **Stack** class allocates memory on stack using **_alloca** intrinsic which modifies stack pointer. Note the unusual(reversed) access to data in lines 21 and 30 and offsetted m_data address in line 11. It is caused by the fact that the stack on x86 grows towards smaller addresses so we have to reverse order of elements(first element(m_data[0]) is stored at the highest address). All functions that call _alloca are inline functions. It is necessary because stack allocated memory is freed in the function epilogue. The size of allocation growth(**step_size * sizeof(T)**) is not accidental. It is multiply of 16 because _alloca aligns allocation to this size.

3 partition() optimizations

3.1 Naive implementation

```
1  template<typename T>
2  T* partition(std::span<T> data)
3  {
4      T pivot = data[data.size() / 2];
5
6      int i, j;
7      for (i = 0, j = data.size() - 1;; i++, j--)
8      {
9          while (data[i] < pivot) i++;
10         while (data[j] > pivot) j--;
11
12         if (i >= j) break;
13
14         std::swap(data[i], data[j]);
15     }
16
17     return &*data.begin() + i;
18 }
```

This implementation serves as starting point to further optimizations.

3.2 Implementation without copying pivot

```
1  template<typename T>
2  T* partitionNoPivotCopy(std::span<T> data)
3  {
4      T* pivot = &data[data.size() / 2];
5
6      int i, j;
7      for (i = 0, j = data.size() - 1;; i++, j--)
8      {
9          while (data[i] < *pivot) i++;
10         while (data[j] > *pivot) j--;
11
12         if (i >= j) break;
13
14         if (pivot == &data[i]) pivot = &data[j];
15         else if (pivot == &data[j]) pivot = &data[i];
16         std::swap(data[i], data[j]);
17     }
18
19     return &*data.begin() + i;
20 }
```

In comparison to naive implementation **pivot**(line 4) is not a copy of pivot but a pointer to it. When pivot is moved pointer is updated(lines 14 and 15). This reduces memory consumption when T memory use is higher than sizeof(T*) and additionally avoids potentially costly copy constructor call.

3.3 Inlining

Inlining function that is called from non recursive function can reduce memory usage. It saves memory where call arguments would have been stored and removes the calle epilogue and prologue.

4 Measurements

4.1 Measurement method

The tests measure maximum stack and heap usage during execution of each quicksort implementation and its execution time. Manual instrumentation was used to obtain memory usage.

4.2 Test data

Test data consist of five sets(random, sorted asc, sorted desc, equal, interleaved) of strings.

4.3 Results

The figures shown are average of 16 sorts. Each set included 10^6 strings. Tested program was 64bit.

4.3.1 Random dataset

Implementation			stack	heap	% of naive	time
qsortRecursive	partition	not inline	3272B	32B	1.0	242ms
		inline	8602B	32B	2.61	240ms
	partitionNoPivotCopy	not inline	3224B	0B	0.98	245ms
		inline	6256B	0B	1.89	244ms
qsortIterative	partition	not inline	288B	986B	0.39	233ms
		inline	208B	986B	0.36	246ms
	partitionNoPivotCopy	not inline	240B	982B	0.37	239ms
		inline	176B	982B	0.35	252ms
qsortIterativeNoRedundancy	partition	not inline	272B	562B	0.25	239ms
		inline	192B	562B	0.23	260ms
	partitionNoPivotCopy	not inline	224B	560B	0.24	243ms
		inline	144B	560B	0.21	250ms
qsortIterativeNoRedundancyStack	partition	not inline	563B	32B	0.18	243ms
		inline	483B	32B	0.16	236ms
	partitionNoPivotCopy	not inline	515B	0B	0.16	242ms
		inline	435B	0B	0.13	243ms

4.3.2 Sorted ascending dataset

Implementation			stack	heap	% of naive	time
qsortRecursive	partition	not inline	1936B	32B	1.0	127ms
		inline	4928B	32B	2.56	123ms
	partitionNoPivotCopy	not inline	1888B	0B	0.98	123ms
		inline	3584B	0B	1.85	123ms
qsortIterative	partition	not inline	288B	760B	0.54	116ms
		inline	208B	760B	0.50	119ms
	partitionNoPivotCopy	not inline	240B	752B	0.51	114ms
		inline	176B	752B	0.48	120ms
qsortIterativeNoRedundancy	partition	not inline	272B	378B	0.34	119ms
		inline	192B	378B	0.30	118ms
	partitionNoPivotCopy	not inline	224B	376B	0.31	120ms
		inline	144B	376B	0.27	117ms
qsortIterativeNoRedundancyStack	partition	not inline	451B	32B	0.25	121ms
		inline	371B	32B	0.21	115ms
	partitionNoPivotCopy	not inline	403B	0B	0.21	120ms
		inline	323B	0B	0.17	114ms

4.3.3 Sorted descending dataset

Implementation			stack	heap	% of naive	time
qsortRecursive	partition	not inline	1936B	32B	1.0	125ms
		inline	4928B	32B	2.56	124ms
	partitionNoPivotCopy	not inline	1888B	0B	0.98	123ms
		inline	3584B	0B	1.85	124ms
qsortIterative	partition	not inline	288B	760B	0.54	115ms
		inline	208B	760B	0.50	122ms
	partitionNoPivotCopy	not inline	240B	752B	0.51	116ms
		inline	176B	752B	0.48	122ms
qsortIterativeNoRedundancy	partition	not inline	272B	378B	0.34	121ms
		inline	192B	378B	0.30	119ms
	partitionNoPivotCopy	not inline	224B	376B	0.31	120ms
		inline	144B	376B	0.27	117ms
qsortIterativeNoRedundancyStack	partition	not inline	451B	32B	0.25	121ms
		inline	371B	32B	0.21	114ms
	partitionNoPivotCopy	not inline	403B	0B	0.21	117ms
		inline	323B	0B	0.17	117ms

4.3.4 Equal dataset

Implementation			stack	heap	% of naive	time
qsortRecursive	partition	not inline	1424B	32B	1.0	118ms
		inline	3520B	32B	2.49	117ms
	partitionNoPivotCopy	not inline	1376B	0B	0.97	85ms
		inline	2560B	0B	1.80	85ms
qsortIterative	partition	not inline	288B	784B	0.75	112ms
		inline	208B	784B	0.70	116ms
	partitionNoPivotCopy	not inline	240B	752B	0.70	79ms
		inline	176B	752B	0.65	83ms
qsortIterativeNoRedundancy	partition	not inline	272B	408B	0.48	111ms
		inline	192B	408B	0.42	109ms
	partitionNoPivotCopy	not inline	224B	376B	0.42	84ms
		inline	144B	376B	0.37	84ms
qsortIterativeNoRedundancyStack	partition	not inline	448B	32B	0.33	111ms
		inline	368B	32B	0.28	109ms
	partitionNoPivotCopy	not inline	400B	0B	0.28	84ms
		inline	320B	0B	0.22	82ms

4.3.5 Interleaved dataset

Implementation			stack	heap	% of naive	time
qsortRecursive	partition	not inline	1936B	32B	1.0	125ms
		inline	4928B	32B	2.56	126ms
	partitionNoPivotCopy	not inline	1888B	0B	0.98	95ms
		inline	3584B	0B	1.85	98ms
qsortIterative	partition	not inline	288B	544B	0.54	124ms
		inline	208B	544B	0.50	123ms
	partitionNoPivotCopy	not inline	240B	512B	0.51	91ms
		inline	176B	512B	0.48	95ms
qsortIterativeNoRedundancy	partition	not inline	272B	408B	0.34	122ms
		inline	192B	408B	0.30	119ms
	partitionNoPivotCopy	not inline	224B	376B	0.31	94ms
		inline	144B	376B	0.27	97ms
qsortIterativeNoRedundancyStack	partition	not inline	448B	32B	0.25	123ms
		inline	368B	32B	0.21	118ms
	partitionNoPivotCopy	not inline	400B	0B	0.21	93ms
		inline	320B	0B	0.17	94ms