sort.cpp

```
#include <stdint.h>
void sort(uint32_t *a, int n)
{
 int split=0;
 if( n < 32 ){
   int i, j;
   uint32_t t;
   for (i = 1; i < n; i++) {
    t = a[i];
    for (j=i; j>0&&t<a[j-1]; j--){
      a[j] = a[j - 1];
    }
    a[j] = t;
 }else{
   uint32_t pivot=a[n/2];
   int i,j;
   for(i=0, j=n-1;; i++, j--){
     while(a[i]<pivot)
     i++;
     while(pivot<a[j])
       j--;
     if(i>=j)
      break;
     uint32_t tmp=a[i];
     a[i]=a[j];
     a[j]=tmp;
   }
   split=i;
   r_sort(a,       split);
   r_sort(a+split, n-split);
 }
}
```

A S B C

mark_sort.mlir

```
module attributes {
  func.func @_Z6_sortPji(%arg0, %arg1 \
  state_base = 0 : i32, state_finish = 3 : i32, state_region0 = 3 :
i32} {
    %c1 = arith.constant 1 : index
     . . .
    %2 = arith.cmpi slt, %arg1, %c32_i32
    scf.if %2 {
       .
    } else {
      . . .
      %5:3 = scf.while (%arg2 = %1, %arg3 = %4, \
        %arg4 = %c0_i32, %arg5 = %true) {
        scf.condition(%arg5) %arg4, %arg2, %arg3
      } do {
      ^bb0(%arg2, %arg3, %arg4):
        . . .
        %12:3 = scf.if % {
          %13 = arith.index_cast %9
          %14 = memref.load %arg0[%13]
          %15 = arith.index_cast %10
          %16 = memref.load %arg0[%15]
          memref.store %16, %arg0[%13]
          memref.store %14, %arg0[%15]
          %17 = arith.addi %9, %c1_i32
          %18 = arith.addi %10, %c-1_i32
          scf.yield %14, %18, %17
        } else {
          scf.yield %arg3, %10, %9
        }
        scf.yield %12#0, %12#1, %12#2, %11
      }
      func.call @_Z6r_sortPji(%arg0, %5#0) {carry_value = 1 : i32,
"operand#1" = 1 : i32, state_base = 0 : i32, state_finish = 1 : i32}
      %6 = arith.index_cast %5#0
      %7 = "polygeist.subindex"(%arg0, %6)
      %8 = arith.subi %arg1, %5#0
      func.call @_Z6r_sortPji(%7, %8) {state_base = 1 : i32,
state_finish = 2 : i32}
    } {state_base = 0 : i32, state_finish = 3 : i32, state_region1 =
2 : i32}
    return}}
```

C1 D1 B1

sort_opt.mlir

```
module attributes {
    func.func @_Z6r_sortPji(%arg0, %arg1){
    . . .
    %alloca = memref.alloca()
    %alloca_0 = memref.alloca()
    %alloca_1 = memref.alloca()
    memref.store %c0_i32, %alloca_1[%c0]
    memref.store %arg0, %alloca[%c0]
    memref.store %arg1, %alloca_0[%c0]
    %0 = llvm.mlir.undef : i32
    %1:2 = scf.while (%arg2 = %c0_i32, %arg3 = %true, %arg4 = %c0_i32) : . . .
{
      scf.condition(%arg3) %arg2, %arg4
    } do { ^bb0(%arg2: i32, %arg3: i32):
      %2 = arith.index_cast %arg2
      %3 = memref.load %alloca_1[%2]
      %4 = arith.index_cast %3
      %5 = memref.load %alloca[%2]
      %6 = memref.load %alloca_0[%2]
      %7 = arith.index_cast %6
      %8 = arith.cmpi slt, %6, %c32_i32
      %9:3 = scf.index_switch %4 -> i32, i1, i32
      case 0 {
        %10:3 = scf.if %8 -> (i32, i1, i32) {
          . . .
          %11 = arith.cmpi eq, %arg2, %c0_i32
          %12 = arith.cmpi ne, %arg2, %c0_i32
          %13 = scf.if %11 -> (i32) {
            scf.yield %arg2 : i32
          } else {
            %14 = arith.addi %arg2, %c-1_i32
            scf.yield %14
          } scf.yield %13, %12, %arg3
        } else {
          memref.store %c1_i32, %alloca_1[%2]
          %11 = arith.addi %arg2, %c1_i32
          %12 = arith.index_cast %11
          memref.store %c0_i32, %alloca_1[%12]
          . . .
          memref.store %5, %alloca[%12]
          memref.store %21#0, %alloca_0[%12]
          scf.yield %11, %true, %21#0
        } scf.yield %10#0, %10#1, %10#2
      }
      case 1 {
        %10 = arith.index_cast %arg3
        . . .
        memref.store %c0_i32, %alloca_1[%14]
        memref.store %11, %alloca[%14]
        memref.store %12, %alloca_0[%14]
        scf.yield %13, %true, %arg3
      } case 2 { . . . }
      . . .
}return}}
```

A1 B2 A2 S2 S3 C2 B3

① cgeist-opt

② mark-recursive pass

③ optimize-recursion pass

④ Lower: MLIR->LLVM

⑤ sort.ll

llc

Function Verification

⑥ RTL