[SWPP 202001 Project]

Team 1 Requirement & Specification Document

Team 1(이진우, 정덕인, 김도현, 임지규)

0. Project plan summary

1. GV Optimization & Other global-level preprocessing
   - (InterProcedural) Sparse Conditional Constant Propagation + Simplify CFG
   - GV optimizer
   - Dead Global Elimination
   - GV to malloc in main
   - Mem2Reg
   - Global Value Numbering
2. Function Call Optimization
   - Function Merging
   - Tail Call Elimination
   - Function Inline
3. Memory Operation Optimization
   - Vectorize Load/Store

   3-1. Stack/Heap Optimizations
     - Heap to Stack Lowering
     - Interprocedural Alloca Sinking
     - Stack/Heap Access Grouping
     - malloc in main to GV + Dead Argument Elimination
     - Mem2Reg
4. Computative Instruction Optimization Pipeline
   - Reassociate
   - Constant Propagation
   - Combine Instructions
   - Code Sinking
   - SCCP + Simplify CFG
5. Loop Optimization Pipelines

   5-1. LICM&Roting&Unswitching
     - Reassociate
     - Loop Simplify
     - Loop-Closed SSA Form(Terminal phi node)
     - Loop Invariant Code Motion

- Loop Rotate
- Loop Unswitching

5-2. Loop vectorization

- Reassociate
- Loop Simplify
- Canonicalize induction variables
- Loop Unroll and Jam + *Vectorize Load & Store*

6. Backend Optimization

- Liveness Analysis
- Direct Access to Parent Local Registers(Parameter Reduction)
  ⇒ Register Spilling
- LLVM IR to Assembly Static Translation

1	GV ⇔ malloc() + Embedding GV optimization
2	Modifying Combine Insturction
3	Embedding Function opt. + Instruction opt.
4	Embedding Loop Opt Pipeline 1
5	Loop unroll + Vectorized load&store
6	Embedding Loop Opt Pipeline 2
7	Heap to Stack&Argument lowering
8	Interprocedural Alloca Sinking
9	Heap/Stack grouping
10	Liveness Analysis + Translation
11	DAPLR ⇒ Register Spilling
12	Register Allocation + Translation

I. Sprint I

1. GV to malloc in main & malloc in main to GV

1-1. Basic description of the task

This chunk has two goals: one is to implement the GV to malloc pass and embed it in our project with standard GV optimizing passes, and other is to implement malloc in main to GV.

GV to malloc pass moves all the global variables to the *malloc* heap assignment in *main()*. Naive assembler of our project translates GV into the *malloc* instruction in *main()*. Why not before? Our project specification ensures that there exists main() in the IR, allowing this optimization in any cases. This pass will be activated later than the LLVM bundle passes which find and optimize GV uses, but before the heap allocation optimization passes(3-1 from the index above) for maximum efficiency.

However, some heap allocations originating from GV may not be further modified into stack allocation or register by other passes till the end. These leftovers should be converted to GV to reduce cost of *call* instruction. If the time is enough, we also aim to implement this pass also in this sprint.

1-2. GV to malloc in main

1-2-1. Algorithm and Pseudocode

This task is to change Global variable to malloc heap allocation. In llvm, both of the declaration of GV and heap allocated variable returns the pointer of the variable. This means What we have to do is getting the GV and make new malloc variable set to the size of the original GV. And finally change all uses of the GV to new variable In main function. But In case of other functions which used the original GV, it is necessary to give same variables to the functions. According to our spec, main function always exists and it is starting function. So after we make new malloc heap allocated variable, we have to give those variables to functions as arguments. And later we will eliminate the unused arguments from functions by using passes like dead argument elimination.

The following is the pseudocode of this algorithm.

```
for GV in Module
  size = GV.size()
  m = MakeNewMalloc(size)
  for f in functions
    f.GiveArguments(m)
  for every call in module:
    rewrite calls to match signature
  GV.setUses(m)
```

To summarize the effect, these consequences can be observed in the source IR program.

1. Global value are replaced into malloc function inside the main.
2. The malloc variable was put as the argument of function foo.(this means other
functions except main function in real code.)
3. The use of the original GV is changed to the new malloc variable pointer.

It might be dangerous to increase the number of arguments a lot because our specification restricts the maximum number of arguments to 16. There are three significant steps which reduce the number of function arguments.

1. Malloc to alloca & Interprocedural Alloca Sinking: sinks the malloc/alloca into subprocedure if the alloca is only used as a parameter in the function and is not passed to multiple functions. Calling Dead Argument Elimination(DAE) will easily reduce the function arguments.
2. malloc to GV pass: This pass(specifications written in 1-3) implements the exact inverse procedure of this pass. This transforms mallocs in main to GV, so that function arguments containing the malloc instruction in main() is not used anywhere. DAE pass will be applied to remove redundant dummy arguments.
3. Direct Access to Parent Local Register pass: this pass from the backend side moves arguments into normal registers. Invariant values like pointers are likely to sent through arg registers while other registers ride on normal ones.

1-2-2. IR transformation examples

1. LLVM IR input
@GV = global i32 1, align 4 define i32 @main() #0 {

```
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @foo()
  ret i32 %2
}

define i32 @foo() #0 {
  %1 = load i32, i32* @GV, align 4
  %2 = mul nsw i32 2, %1
  ret i32 %2
}
```

2. LLVM IR output

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 1, i32* %GV
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @foo(i32* %GV)
  ret i32 %2
}

define i32 @foo(i32* %GV) #0 {
  %1 = load i32, i32* %GV, align 4
  %2 = mul nsw i32 2, %1
  ret i32 %2
}
```

1. LLVM IR input

```
@GV = global i32 2, align 4

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo()
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* @GV, align 4
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo() #0 {
  %1 = load i32, i32* @GV, align 4
  %2 = load i32, i32* @GV, align 4
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

2. LLVM IR output

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 2, i32* %GV
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo(i32* %GV)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %GV, align 4  // @a -> %a
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo(i32* %GV) #0 {  // add argument %a
  %1 = load i32, i32* %GV, align 4  // @GV -> %GV
  %2 = load i32, i32* %GV, align 4  // @GV -> %GV
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

1. LLVM IR input

```
@GV = global i32 3, align 4

define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* @GV, align 4
  %2 = load i32, i32* @GV, align 4
  ret i32 %2
```

}
2. LLVM IR output

```
define i32 @main() #0 {
  %GV = call i32* @malloc(i64 4)
  store i32 2, i32* %GV
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* %GV, align 4
  %2 = load i32, i32* %GV, align 4
  ret i32 %2
}
```

1-3. malloc in main to GV

1-3-1. Algorithm and Pseudocode

After the GV to malloc pass is executed and optimizations like DAE, stack lowering happen, some malloc operations will still reside in the main function. For instance, allocating a large array in the stack may cause stack overflow, so it should remain on the heap memory. Since we pass transformed GV to all functions by parameters, it may be better to not pass GV and just treat them as global variables. If the malloc to GV pass is excuted, every malloc operation found in main() function are transformed into GV.

```
for mallocInst instM in main:
  create GV
  instM = find callInst(malloc()) in main
  perform Memory Alias Analysis with instS globally
  find all arguments which share the value with malloc()
  for all argument that share value with instM:
    change ptr ⇒ GV
  add StoreInst(GV, (initial value)) to main

Afterwards:
perform Dead argument elimination, Dead code elimination
```

This pass(+DCE) should be a exact inverse of the GV to malloc pass, so LLVM IR examples are omitted.

4. Verifying Loop Optimization Pipeline 1(김도현)

This chunk is to embed the loop optimization pipeline of LLVM standard to our project. The sequence of existing passes which we will bring from LLVM is as following. The order here is the same as the order we are going to apply. Dependencies between passes are found from various sources; LLVM documentations, clang -O1 tests, and the comments in the source code.

- Loop Simplify
- Loop-Closed SSA Form(Terminal phi node)
- Loop Invariant Code Motion
- Loop Rotate
- Loop Unswitching

These optimizations reduce complexity inside the loop by moving invariant values outside the loop(LICM), restructuring the loop into if-nested do-while loops(Loop Rotate), or reducing the branch instruction in the loop(Loop Unswitching). The task is to create a pass which wraps all these passes, so a single run through the pass will optimize loops as desired. Also, we should ensure that no invalid IR components such as intrinsics appear after the optimization.

5. Loop unrolling + Vectorize Load & Stores(정덕인)

5-1. Basic Description of the task

The pass implemented here receives the loop-unrolled functions, and vectorizes the load & store. Our system has 64-bit register, which means that it can load at most 2 *int32*, 4 *int16* registers at once. We may load multiple variables as an *int64* type and then split it using *udiv* & *urem* if a sequencial load happens, and vice versa(sequential store will use *or* and *mul* instructions). This reduces the cost of load&store especially when the data type is small.

To perform vectorized load & store, we should find that consequent load and store are happening in the code, However, most consequent memory access is done by looping, so Loop Unrolling pass compatible with the Vectorize pass is required. Loop unrolling pass unrolls and jams the loop, and then rearrange the load/store instructions to let Vectorize pass perform.

5-2. Algorithm and Pseudocode

As we mentioned before, we will use int64 type to vectorize. One 64-bit register can save 2 32-bit registers, 4 16-bit registers, and 8 8-bit registers. According to this, if we unroll loop in bunches of 8, we can handle every case easily. For example, if target IR has loop with consequent memory access by

16-bit, then it can be unrolled in bunches of 8 and every bunch of loop would use two 64-bit registers, which have 4 16-bit values individually.

We can modify the parameters of the existing Loop Unrolling pass in LLVM standard to unroll loops by 8. Also, unrolling loops that number of iterations are not determined in the compile time is initially inactivated in the LLVM standard loop unrolling because it consumes more time while compiling. However, we will force the pass to perform unrolling even in these cases.

This is the pseudocode of this algorithm below.

```
Prerequisites:
all loop optimizers until Loop unrolling performed
Combine instructions performed(for easy comparing of GEP offsets)
  ⇒ (must perform add combination)

VectorizeLoadNStore::run():
set sequentialGEPs
for BB in Function
  if GEP instructions are sequential:
    sequentialGEPs.add( { (GEP0, GEP1), (GEP2, GEP3) ... } )
  for GEPs in sequentialGEPs:
    for (User, nextUser) w.r.t. GEPs:
      type := inst.getType()
      if type == i8, i16, i32 then
        groupAndSplit(User, nextUser)

groupAndSplit() performs the splitting of operations using urem+udiv /
or+umul.
- e.g. If 'load' operations comes in:
Adds a i64 load and spares it before the first usage, and distributes all
values to new registers. usages(loads) are all swapped to these new
distributed values.
```

First, LoopUnrollPass from LLVM standard is included to unroll the loop. We will assign a value to LoopUnrollOptions, so we can unroll loop with unknown iteration. After loop unrolling, we check if there are any sequential loads inside the unrolled loop body. By the effects of loop simplification and unswitching, most body codes are going to be written in one basic block, so we restrict the searching scope into a basic block. Sequential memory accesses are defined as sequential if two pointers(should be *getelementptr*, the GEP instruction) are sequential. Two GEP instructions are considered to be sequential if,

- Types of *getelementptr* are equal.
- Base address pointers are equal.

- Offset instruction has 1 difference.
  e.g.) “%x = i32..” and “add i32 %x, 1” has 1 difference.
  “add i32 %x, 2” and “add i32 %x, 3” has 1 difference.

If there exists 8(=unroll factor) sequential GEPs in total, they are grouped and their uses are transformed into vectorized memory accesses.

In the second sprint, the compatibility between this pass and other LLVM standard loop optimizing passes will be investigated. Also, we may modify the algorithm finding consecutive GEPs considering the behaviors of the loop unrolling pass. More detail about possible further implementations will be described in section 6, Verifying loop optimization pipeline 2.

5-3. IR transformation examples

We supposed unroll by 8 bunches, but this leads to unrealistic length of code to show here. Because of this reason, following examples are slightly different(unroll factor = 4, not 8) as we mentioned above.

```
define void @_Z1fv() {
  %1 = alloca [5 x i32], align 16
  %2 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  br label %3

3:
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 5
  br i1 %5, label %6, label %16

6:
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %2, align 4
  %9 = mul nsw i32 %7, %8
  %10 = load i32, i32* %2, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 %11
  store i32 %9, i32* %12, align 4
  br label %13

13:
  %14 = load i32, i32* %2, align 4
```

```
define void @_Z1fv() {
  %1 = alloca [5 x i32], align 16
  %2 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 0
  %3 = mul nsw i64 0, 4294967296
  %4 = add nsw i64 %3, 1
  store i64 %4, i32* %2, align 8
  %5 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 2
  %6 = mul nsw i64 4, 4294967296
  %7 = add nsw i64 %6, 9
  store i64 %7, i32* %5, align 8
  %8 = getelementptr inbounds [5 x i32], [5 x i32]* %1, i64 0, i64 4
  store i32 16, i32* %8, align 4
  ret void
}
```

```
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %2, align 4
  br label %3

16:
  ret void
}
```

```
define i32 @_Z1fv() {
%1 = alloca [100 x i32], align 16
br label %2

2:
%.04 = phi i32 [ 0, %0 ], [ %6, %2 ]
%3 = mul nsw i32 %.04, %.04
%4 = sext i32 %.04 to i64
%5 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %4
store i32 %3, i32* %5, align 4
%6 = add nuw nsw i32 %.04, 1
%7 = icmp ult i32 %6, 100
br i1 %7, label %2, label %.preheader

.preheader:
%.13 = phi i32 [ %12, %.preheader ], [ 0, %2 ]
%.012 = phi i32 [ %11, %.preheader ], [ 0, %2 ]
%8 = sext i32 %.13 to i64
%9 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %8
%10 = load i32, i32* %9, align 4
%11 = add nsw i32 %.012, %10
%12 = add nuw nsw i32 %.13, 1
%13 = icmp ult i32 %12, 100
br i1 %13, label %.preheader, label %14

14:
%.01.lcssa = phi i32 [ %11, %.preheader ]
ret i32 %.01.lcssa
}
```

```
define i32 @_Z1fv() {
%1 = alloca [100 x i32], align 16
br label %2

2:
%.04 = phi i32 [ 0, %0 ], [ %22, %2 ]
%3 = mul nsw i32 %.04, %.04
%4 = sext i32 %.04 to i64
%5 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %4
%6 = add nuw nsw i32 %.04, 1
%7 = mul nsw i32 %6, %6
%8 = sext i32 %3 to i64
%9 = mul nsw i64 %8, 4294967296
%10 = sext i32 %7 to i64
%11 = add nuw nsw i64 %9, %10
store i64 %11, i64* %5, align 8
%12 = add nuw nsw i32 %6, 1
%13 = mul nsw i32 %12, %12
%14 = sext i32 %12 to i64
%15 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %14

%16 = add nuw nsw i32 %12, 1
%17 = mul nsw i32 %16, %16
%18 = sext i32 %13 to i64
%19 = mul nsw i64 %18, 4294967296
%20 = sext i32 %17 to i64
%21 = add nuw nsw i64 %18, %20
store i64 %21, i64* %15, align 8
%22 = add nuw nsw i32 %16, 1
%23 = icmp ult i32 %22, 100
br i1 %23, label %2, label %.preheader.preheader
```

	.preheader.preheader: br label %.preheader .preheader: %.13 = phi i32 [0, %.preheader.preheader], [%39, %.preheader] %.012 = phi i32 [0, %.preheader.preheader], [%38, %.preheader] %24 = sext i32 %.13 to i64 %25 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %24 %26 = load i64, i64* %25, align 8 %27 = rem i32 %26, 4294967296 %28 = div i32 %26, 4294967296 %29 = rem i32 %28, 4294967296 %30 = add nsw i32 %.012, %27 %31 = add nsw i32 %30, %29 %32 = add nuw nsw i32 %.13, 2 %33 = sext i32 %32 to i64 %34 = getelementptr inbounds [100 x i32], [100 x i32]* %1, i64 0, i64 %33 %35 = load i64, i64* %34, align 8 %36 = rem i32 %35, 4294967296 %37 = div i32 %36, 4294967296 %38 = rem i32 %37, 4294967296 %39 = add nsw i32 %31, %36 %40 = add nsw i32 %39, %38 %41 = add nuw nsw i32 %32, 2 %42 = icmp ult i32 %41, 100 br i1 %42, label %.preheader, label %43 43: %.01.lcssa = phi i32 [%38, %.preheader] ret i32 %.01.lcssa }

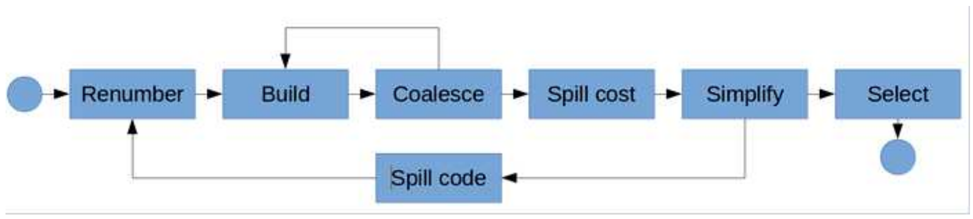
10. Liveness Analysis(이진우)

10-1. Basic description of the task

This task aims to build the Liveness Analysis module. It dynamically determines the live range of all the registers, and performs O($L \cdot N^2$) analysis to search through an overlapping pair among all N IR registers and in code length of L. This analysis is globally done, however it is ensured that registers in different functions are considered to be not overlapping in any cases. i.e. it is a FunctionPass wrapped within the ModulePass. This assumption will break by the Direct Access to Parent Local Register pass (DAPLR) which is going to be implemented in sprint II.

Also, we plan to implement the interface that receives the result of Liveness Analysis and the pass for sprint II and generates the symbol table by graph coloring. According to Briggs, Cooper, & Torczon 1992, this tasks covers the Renumber~Coalesce in the whole register allocation procedure.

Several liveness analysis pass with LLVM IR are present online, however for the compatibility with the DAPLR pass, we decided to implement it again.



10-2. Interference Graph Construction

10-2-1 Algorithm & Pseudocode

All N symbols(arguments and instructions, but not labels) are mapped by a unsigned integer. All L instructions in the target LLVM source are mapped to the boolean array length of N. For each N symbols, the pass recursively traces every usage of it from the definition and marks the flag array true if the variable is alive. Postorder search is used to track the liveness in the measure of instructions, not basic blocks. The results are converted into the graph form(N*N adjacency matrix) and all results are preserved.

Pseudocode of this analysis pass can be written as follows.

```
LivenessAnalysis::run()

  //Counts all symbols
  map<unsigned int, argument+instructions> symbols
  N = 0
  for symbol in arguments + instructions:
    symbols[N++] = symbol
  //Recursively searches and marks liveness
  map liveness: instruction ↦ bool[N]
  for i in range(N):
    for user in all uses of i^th Instruction:
      RecursiveLiveIntervalSearch(user, i, symbols, liveness)
  //Turn the result into graph
  bool[][] adjacencyMatrix
  for I in instructions:
    overlap = Liveness[instruction]
    for (i, j) from (0, 0) to (N, N), i < j:
      if liveness[I][i] && liveness[I][j] :
        adjacencyMatrix[i][j] = adjacencyMatrix[j][i] = true

LivenessAnalysis::RecursiveLiveIntervalSearch(curr, inst, &symbol, &liveness)

  currBlock = curr.parent
  for i = curr to currBlock.begin()
    liveness[i][inst] = true;
  for PB in currBlock.predecessors():
    RecursiveLiveIntervalSearch(PB.terminator, inst, symbol, liveness)
```

EDIT: we modified the algorithm as an backward full search from the user to the definition. For every uses, the definition dominates the user in SSA. This indicates that if we traverse in the direction of being farther from the user, we will definitely reach the definition. The definition should be live in every instructions lying on every paths.

10-3. Graph Coloring

10-3-1. Algorithm

*Most mathematical and algorithmic approach of this section comes from here[1]. This document includes pseudocodes as well as the algorithms which we would like to follow.

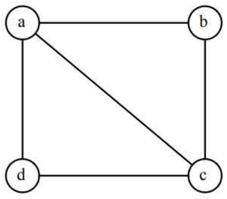
Graph coloring with minimal set of colors is generally a NP-complete problem, so it would be wise to reduce the compile time with efficient algorithms. We can prove that in SSA form, interference graphs become 'chordal', so that no minimal loop larger than size 3 is formed. This proves that a Perfect(Simplicial)

1) http://laure.gonnord.org/pro/research/ER03_2015/SSABasedRA.pdf

Elimination Ordering exists. Precise definition for SEO is that:

A PEO of the graph G is a permutation of every N vertices(V_1, ..., V_N) where for every V_i and the subgraph G_i of G induced from {V_1, V_2, ..., V_i}, neighbours of V_i in G_i form a clique.

For example, in the graph drawn below, A⇒C⇒B⇒D can be a instance for the PEO of G.



There are various linear time algorithms which find PEO from chordal graphs. Maximum Cardinality Search and Lexicographic BFS are two algorithms available for PEO search.

Also, when the graph has an PEO, it is proved that the graph can be colored with C colors with greedy algorithm in polynomial time where C, chromatic number of the graph, is smaller than (size of the biggest clique) + 1. Greedy algorithm is simple; it colors all nodes in order of the PEO. The color is the smallest number not used in the clique of the vertices' neighbours.

10-3-1. LLVM IR Examples

1. LLVM IR input

```
define i32 @main() #0 {
entry:
  %r0 = alloca [10 x i32], align 16
  br label %BB1
BB1:
  %r1 = phi i32 [ 0, %entry ], [ %r6, %BB3 ]
  %r2 = icmp slt i32 %r1, 5
  br i1 %r2, label %BB2, label %BB4
BB2:
  %r3 = sub nsw i32 %r1, 2
  %r4 = sext i32 %r1 to i64
  %r5 = getelementptr inbounds [10 x i32], [10 x i32]* %r0, i64 0, i64 %r4
  store i32 %r3, i32* %r5, align 4
  br label %BB3
BB3:
  %r6 = add nsw i32 %r1, 1
  br label %BB1
BB4:
  %r7 = add nsw i32 5, 3
  %r8 = ashr i32 %r7, 1
  %r9 = sext i32 %r8 to i64
  %r10 = getelementptr inbounds [10 x i32], [10 x i32]* %r0, i64 0, i64 %r9
  %r11 = load i32, i32* %r10, align 4
  ret i32 %r11
}
```

2. Register coloring result

```
main 3 colors needed
color 0 : r11 r2 r4 r9 r5 r6 r7 r8 r10
color 1 : r3
color 2 : r1
```

1. LLVM IR input

```
define i32 @main() #0 {
entry:
  br label %BB1
BB1:
  %r0 = phi i32 [ 16, %entry ], [ %r8, %BB5 ]
  %r1 = phi i32 [ 0, %entry ], [ %r9, %BB5 ]
  %r2 = icmp sgt i32 %r0, 1
  br i1 %r2, label %BB2, label %BB6
BB2:
  %r3 = srem i32 %r0, 2
  %r4 = icmp eq i32 %r3, 0
  br i1 %r4, label %BB3, label %BB4
BB3:
  %r5 = sdiv i32 %r0, 2
  br label %BB5
BB4:
  %r6 = mul nsw i32 %r0, 3
  %r7 = add nsw i32 %r6, 1
  br label %BB5
BB5:
  %r8 = phi i32 [ %r5, %BB3 ], [ %r7, %BB4 ]
  %r9 = add nsw i32 %r1, 1
  br label %BB1
BB6:
  ret i32 %r1
}
```

2. Register coloring result

```
main 5 colors needed
color 0 : r6 r4 r3 r2 r9
color 1 : r8 r0
color 2 : r7
color 3 : r5
color 4 : r1
```

1. LLVM IR input

```
define i32 @main() {
  %a0 = add i32 3, 5
  %b0 = add i32 4, 2
  %c0 = add i32 8, 3
  %d0 = add i32 6, 5
  %e0 = call i32 @foo(i32 %a0, i32 %b0, i32 %c0, i32 %d0)
  %f0 = call i32 @bar(i32 %a0, i32 %c0, i32 %b0, i32 %d0)
  %g0 = add i32 %e0, %f0
  ret i32 %g0
}

define i32 @foo(i32 %a1, i32 %b1, i32 %c1, i32 %d1) {
  %e1 = add i32 %a1, %b1
  %f1 = add i32 %c1, %d1
  %g1 = add i32 %e1, %f1
  %h1 = call i32 @bar(i32 %a1, i32 %b1, i32 %c1, i32 %d1)
  %i1 = add i32 %g1, %h1
  ret i32 %i1
}

define i32 @bar(i32 %a2, i32 %b2, i32 %c2, i32 %d2) {
  %e2 = mul i32 %a2, %b2
  %f2 = mul i32 %c2, %d2
  %g2 = mul i32 %e2, %f2
  ret i32 %g2
}
```

2. Register coloring result

```
main 5 colors needed
color 0 : f0 d0 g0
color 1 : e0
color 2 : c0
color 3 : a0
color 4 : b0

foo 2 colors needed
color 0 : e1 h1 i1
color 1 : f1 g1

bar 2 colors needed
color 0 : f2 g2
color 1 : e2
```

II. Sprint 2

2. Instruction Optimization (김도현)

2-1. Basic description of the task

This pass will provide instruction level optimizations. Our target machine has some unnatural properties about register level instructions, and we tend to optimize instructions in a target-specific manner. We plan to run this pass among other instruction level optimizer passes such like Reassociate or InstCombine. The most efficient order of different transformations aligned will also be searched.

2-2. Algorithm & Pseudocode & 2-3. LLVM IR examples

In this section, we write the target instruction forms with their corresponding IR examples. More examples could be added in further development process. The following pseudocode is the summary of what is going to be done. An important point is that the simplifyOperand() should be a postorder recursive, so that in each phase the operands are already optimized prior to the current instruction.

```
(Prerequisites: Run InstCombine pass)

function simplifyOperand(I) //simplify I
  if !isBinaryOperator(I) then
    return
  else
    simplifyOperand(I.operand[0]) //recursive
    simplifyOperand(I.operand[1]) //recursive
    if isAddOperator(I) and hasSameOperands(I) then
      convertToMultiply(I)
    if isShiftOperator(I) then
      convertToMultiplyOperator(I)
    if isAndOperator(I) && Type(I) == i1 then
      convertToMultiplyOperator(I)
    if isMoveOperator(I)
      convertToMultiplyOperator(I)
```

- Adding equal operands to multiplication

1. LLVM IR input
%y = add %x, %x
2. LLVM IR output
%y = mul %x, 2

Actually, this transformation is done by the standard InstCombine pass. However, we should validate that the result is stable; standard pass optimizes this as a shift operator again, while in our target machine it should not. Maybe it can be combined with the following pass, but we mention this feature here to clarify that we are concerned about this optimization.

- Arithmetic/logical shifts to signed&unsigned divisions

1. LLVM IR input
%s1 = ashr %x, 3 %s2 = lshr %x, 3 %s3 = shl %x, 3
2. LLVM IR output
%s1 = sdiv %x, 8 %s2 = udiv %x, 8 %s3 = mul %x, 8

- Move instructions

1. LLVM IR input
%y = add %x, 0 %y2 = shl %x, 0
2. LLVM IR output
%y1 = mul %x, 1 %y2 = mul %x, 1

Move instructions are required in the backend of the compiler to remove SSA form by coallocating all operands of *phi*. Since the most effective form of move instruction is the multiplication(cost=0.8), other move forms with higher costs(add, shift, logical copy...) are reduced to multiplications.

- Logical 1-bit operations

1. LLVM IR input
%z = and i1 %x, %y
2. LLVM IR output
%z = mul i1 %x, %y

- Remainder calculated by bitwise logical operators(LSB bitmasks),

1. LLVM IR input
%y1 = and %x, 3 %y2 = and %x, 15
2. LLVM IR output
%y1 = urem %x, 4 %y2 = urem %x, 16

6. Verifying Loop Optimization Pipeline 2 (정덕인)

6-1. Basic description of the task

This pass is continued version of task 4 from our project. In task 4, we used some loop related passes, which are belonging to -O1 optimization level of standard clang compiler. They are loop-simplify, lcssa(loop-closed ssa), licm(loop invariant code motion), loop-rotate, and loop-unswitch. We wish to determine which pass is required to correctly and efficiently unroll loops and vectorize memory operations by referring to documentations and testing multiple IR inputs.

In this pass, we will use loop-unroll pass before applying the Vectorized LoadAndStore pass. Loop unrolling is method that unrolls loop to reduce the number of operations, especially branches. If loop is unrolled, then the number of operations and condition branches are decreased, so program can run faster. Also unrolled loop has lots of possibility to get consecutive load/store instructions, so loops become easily to vectorize load/store instructions. We are going to tune the parameters of loop unrolling so it behaves exactly as we want. Loop unroll factors and whether to toggle the unrolling pass on if the loop iteration count is undetermined but the variable is in the correct induction form. For example, the default option of clang loop unroller only unrolls loops like (a). However, we wish to apply loop unrolling in cases even like (b). Hints about how to assign tuning parameters could be find on the LLVM standard documentations, namely here[2] and here[3].

Case (a)	Case (b)	Case (c)
for(int i = 0; i < 10; ++i)	for(int i = 0; i < n; ++i)	for(int i = n; i > 0; i>>=1)

Also, since the last sprint's developer did not implement any framework suitable for grouping and controlling the optimizer passes(both existing and custom), we plan to implement these in this sprint too. We may develop them via

2) https://llvm.org/docs/LangRef.html#llvm-loop
3) https://llvm.org/docs/TransformMetadata.html#loop-unrolling

the PassManager class, or using the clang/opt optimizer via user-friendly interfaces such as shell files or python.

7. Heap to Stack & Argument lowering (임지규)

7-1. Basic description of the task

This pass lowers the heap variable to the stack variable. *i.e.* malloc function to alloca instruction.

Because accessing the heap memory is more costly than with stack, this lowering benefits in reducing the total execution cost. Static sized variables may be used. After this heap to stack lowering is finished, we would like to further proceed with more aggresive version of mem2reg pass if possible.

7-2. Algorithm & Pseudocode

This task is to change malloc allocation(heap allocation) to alloca instruction (stack allocation). Combination of malloc + bitcast is semantically equal to the alloca instruction, However because stack memory disappears after the function is returned, we have to carefully identify if the lowering does not cause any problem.

Another important difference is that the size of heap allocation may be determined during run-time, unlike static stack allocations. So we have to care about the size of malloc variable and we only change the allocation location when the size is determined by the constant value.

```
for function in Module
  entry = function.getEntryBlock()
  for I in entry
    cI = dyn_cast<CallInst>(I)
    if cI != null && cI->getCalledFunction()->getName() == "malloc“
      checkCondition();
      alloca = ReplaceToAlloca(cI) // make alloca set to the malloc variable.
      cI->replaceAllUsesWith(alloca)
```

First of all, we have to get the type and size of malloc allocated value. And make new alloca instruction set to the malloc variable's type and size.

Important conditions like if the size is constant, if the cI is not looped(which can dynamically assign the number of malloc instructions while the size of individual instructions are constant), and if the value of the pointers are still used after the function is returned must be checked before the lowering part.

After we create an alternative alloca instruction for malloc, we have to replace the uses of original malloc value to new alloca value.

Some details might be changed since there could be more conditions-to-check revealed in additional implementation progress.

7-3. LLVM IR examples

The change between input and output file is very simple. It is only about the way of allocation(malloc -> alloca). Since the way it is used is same(pointer type variable), all we have to do is judge whether replacement is possible and change the allocation.

1. LLVM IR input

```
define i32 @main() #0 {
  %ma0 = call i32* @malloc(i64 4)
  store i32 1, i32* %ma0
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = load i32, i32* %ma0
  ret i32 %2
}
```

2. LLVM IR output

```
define i32 @main() #0 {
  %ma0 = alloca i32, align 4  // malloc -> alloca
  store i32 1, i32* %ma0
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = load i32, i32* %ma0
  ret i32 %2
}
```

1. LLVM IR input

```
define i32 @main() #0 {
  %ma0 = call i32* @malloc(i64 4)
  store i32 2, i32* %ma0
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo(i32* %ma0)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %ma0, align 4
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo(i32* %GV) #0 {
  %1 = load i32, i32* %GV, align 4
  %2 = load i32, i32* %GV, align 4
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

2. LLVM IR output

```
define i32 @main() #0 {
  %ma0 = alloca i32, align 4 // malloc -> alloca
  store i32 2, i32* %ma0
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %3 = call i32 @foo(i32* %ma0)
  store i32 %3, i32* %2, align 4
  %4 = load i32, i32* %ma0, align 4
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %4, %5
  ret i32 %6
}

define i32 @foo(i32* %GV) #0 {
  %1 = load i32, i32* %GV, align 4
  %2 = load i32, i32* %GV, align 4
  %3 = mul nsw i32 %1, %2
  ret i32 %3
}
```

11. Register Spilling (이진우)

11-0. NOTICE: Plans changed

Originally, we planned to implement a pass which can exploit computable registers(*r1~16*) as a function argument, however, we found out that it is almost impossible to implement the feature(it costs NP-time in most cases). We give up the feature and focus on efficient register spilling, which was the subtask of our original implementation plan.

11-1. Basic description of the task

This process spills registers to stack memory, regard to the cost of spilling the register. Using the result of LivenessAnalysis, which provides a minimal coloring of registers, register spiller assigns IR register to hardware registers and transform leftovers into *alloca, load, store* instructions. Spilling once does not ensure that the program uses less than designated amount of registers, so LivenessAnalysis is performed once more. If it is concluded that the number of required registers exceed 16, spilling process is repeated.

11-2. Algorithm & Pseudocode

11-2-1. Calculating Spilling Cost

If the total color of registers are less than 16, we do not need to consider the spilling process. However if not, we must.

Spilled registers should introduce three operations to the IR, which are *alloca+store* and *load*. Since LLVM IR is a SSA form language, we are ensured that one *alloca* operation[4] is performed per one color, and one *store* operation is performed per one IR register. So the cost of spilling registers of same color can be estimated by the equation:

$$Cost(c) \simeq Cost(alloca) + n_c \bullet Cost(store) + \sum_{i=1}^{n_c} Use_i \bullet Cost(load)$$

The dominant part of the equation is the *load* part. If an IR register is used a lot, then the total cost of spilling is dynamically increased. Say if 10 registers are colored to 0 but each are used only once and a single register colored to 1 is used a hundred time, it would be more efficient to spill the color 0.

Calculating the number of uses is very complicated since loops may iterate through the basic block random times(*i.e.* unknown in compile-time). So estimating the iteration numbers of a loop is very hard-assuming. If we can

4) We do not have any *alloca* operation in our specification. Instead, we increase the sp for the sum of all *alloca* size. This cost is almost zero(=1.2/(total colors-15)), compared to other costs. We do not add this cost to the actual implementation(check the pseudocode). However, for the consistency of IR code, *alloca* instructions are inserted temporarily.

easily figure out the loop time via loop optimizations and analyses, we can use the value. However, if not(argument-recieved or determined by user input), we assume the repeat count to be a specific constant. The optimal value is to be decided. If the loop is nested, the use count is multiplied by every layer it resides in. Note that this is just a heuristic, it does not guarantee any mathematical optimum of total spilling costs.

We search the If we ensure this condition, we need to spill only once per function in SSA form programs.

```
RegisterSpiller::SpillCost(RegisterGraph) //total colors are ordered.

  //RG: RegisterGraph, result of LivenessAnalysis
  for c in {colors of F}:
    cost[c] = 0;
    for v in {values colored as c}:
      cost[c] += Cost(store);
      for u in {user of v}:
        costLoad = Cost(load);
        for L in {nestedLoops of u}:
          costLoad *= loopCount of L(or default value if undetermined)
        cost[c] += costLoad;

  sort <color, cost>
  return
```

Maximum $N<16(=N_R)$ colors are selected to be on the register memory, and others are all spilled. If 16 registers are all on, we do not have a spare to use for loading and storing the registers. The criteria of finding N is that, for every instruction which takes N_c spilled operands, $N = N_R - \max\{N_C\}$.

To illustrate this procedure more vividly, consider the example. Let's say that the machine has 4 registers and program requires 6 colors. Then, we know that at least 3 should be spilled. They must be reloaded via 1(=4-3) buffer. However, if there is an operation which requires 2 spilled operands, we should need 2 buffers. To achieve the condition, we spill one more registers so now only 2 are permanently loaded on the register file, leaving two buffers. We repeat the process until the buffer is enough to embrace all spilled operands.

11-2-2. Spilling

This part of the register spilling is a definitely a transform pass. We have to add new instructions, such as *alloca*, *store*, and *load*. Several interesting observations could be made in the spilling process. First is that, as noted in the previous section, we need only one *alloca* for one color. This is very similar to coallocating in register level, which assigns one register for one color. This proves that we can unify the operations to add within the color. We just assign the pointer P_c(may be expressed by *sp* + constant) to each color, and we can call *load* P_c and *store* P_c every time we need to fetch or store the value of it.

Second observation is that load is not mandatory for every time some value is used. Temporary buffer registers store the most recently loaded value. In consequence, if we use a value twice in a row, we do not need to flush the value back to the memory which causes additional delay. We have to store and load buffer only if the used color is changed. We call this phenomenon as the color-switching.

The precise spilling process is introduced below.

1) Define the set of to-be-spilled registers. If empty, do nothing from below.

2) For memory variables(result of *load* instructions), insert different loads which share the pointer value in front of all uses.

3) Create *alloca* instruction for each spilled colors in the *entry* block of the function.

4) For all new definitions in each color, create *store*.

5) For all BB through the function in preorder,

    If a color is permanently loaded, it is also in the beginning of BB.
    If not, 1. BB has a single predecessor: same color loaded with prev. BB
            2. BB has multiple predecessor: buffers are all empty.
    For all instruction, if the operands' color is not loaded,

5-1) if use is *phi*-instruction, create *load* at the end of the previous block(before terminator inst).

5-2) if use is non-*phi* instruction, create *load* right before the instruction.

6) Replace the operand register(use) to the load created in (4).

7) if needed, update the set of currently loaded colors. We plan to use the LRU replacement policy, even though Look-ahead Use has a better performance.

8) After everything, run the Dead Store Elimination Pass.

After the spilling process, the following condition should be checked.

Total color of registers should remain under N_R.

If this condition fails, we cannot assign single buffer register for memory variables, which violates the main purpose of spilling. This property will be checked by assertions.

11-3. LLVM IR examples

To simplify and shorten the code, we assumed that the target machine has total 4 registers, while in practice it would be 16.

1. LLVM IR input

```
define i32 @main() {
  %a = ...
  %b = ...
  %c = ...
  %d = ...
  %e = ...
  %f = add i32 %a, %b ; alive variables: a, b, c, d, e : max-clique
  %g = add i32 %c, %d
  %h = add i32 %e, %f
  %i = add i32 %h, %e
  ret i32 %i
}
```

2. Register coloring result

```
main() : 5 colors needed
color 0: a, f
color 1: b
color 2: c, g
color 3: d, h, i
color 4: e
```

3. Register Spilling

Spill order = 4, 1, 0, 2, 3

We need to spill 2 registers. Spilling only 4 is not enough because to define %e, 4 must be loaded. We so spill one more to make space. => Spill 4, 1

```
define i32 @main() {
  %color1 = alloca i64, align 8
  %color4 = alloca i64, align 8
  %a = ...
  %b = ...
  store %b, %color1
  %c = ...
  %d = ...
  ; current loaded color = [0, 2, 3 / 1]
  %e = ...
  ; current loaded color = [0, 2, 3 / 4] (0, 2, 3 are permanently loaded)
  store %e, %color4
  load %b.0, %color1
  ; current loaded color = [0, 2, 3 / 1]
  %f = add i32 %a, %b
  %g = add i32 %c, %d
  load %e.0, %color4
  ; current loaded color = [0, 2, 3 / 4]
  %h = add i32 %e.0, %f
  %i = add i32 %h, %e
  ret i32 %i
}
```

4. Coloring after Register Spilling

```
main() : 4 colors needed
color 0: a, f
color 1: b, e, b.0, e.0
color 2: c, g
color 3: d, h, i
```

1. LLVM IR input

```
define i32 @main() {
  %a = ...
  %b = ...
  %c = ...
  %d = ...
  %e = call i32 @foo(%a, %b, %c, %d)
  %f = call i32 @foo(%d, %c, %b, %a) ;alive: a, b, c, d, e : max-clique
  %g = add i32 %e, %f
  ret i32 %g
}
```

2. Register coloring result

```
main() : 5 colors needed
color 0: a, f, g
color 1: b
color 2: c
color 3: d
color 4: e
```

3. Register Spilling

Spilling order = 4, 1, 2, 3, 0
We need to spill 2 colors to memory, namely 4, 1.

```
define i32 @main() {
  %color1 = alloca i64, align 8
  %color2 = alloca i64, align 8
  %a = ...
  %b = ...
  store %b, %color1
  %c = ...
  %d = ...
  ; current loaded color = [0, 2, 3 / 1]
  %e = call i32 @foo(%a, %b, %c, %d)
  ; current loaded color = [0, 2, 3 / 4]
  store %e, %color4
  load %b.0, %color1
  ; current loaded color = [0, 2, 3 / 1]
  %f = call i32 @foo(%d, %c, %b.0, %a)
  load %e.0, %color4
  ; current loaded color = [0, 2, 3 / 4]
  %g = add i32 %e.0, %f ;we do not need load.
  store %g, %color0
  ret i32 %g
}
```

4. Coloring after Register Spilling

main() : 4 colors needed

color 0: a, f, g

color 1: b, e, b.0, e.0

color 2: c

color 3: d

1. LLVM IR input

```
define i32 @main() {
  %a = ...
  %b = ...
  %c = ...
  %cmp = icmp eq i32 %a, %b
  br i1 %cmp, label %BB1, label %BB2
BB1:
  %d = ...
  %e = ...
  %f = call i32 @foo(i32 %b, i32 %c, i32 %d, i32 %e)
  br label %BB2
BB2:
  %g = phi i32 [%d, %BB1], [0, %entry]
  %h = phi i32 [%e, %BB1], [0, %entry]
  %i = call i32 @foo(i32 %a, i32 %b, i32 %g, i32 %h)
  %j = call i32 @foo(i32 %c, i32 %b, i32 %g, i32 %h)
  ret i32 0
}
```

2. Register coloring result

```
main() : 6 colors needed
color 0 : f h cmp d i j
color 1 : e
color 2 : c
color 3 : b
color 4 : a
color 5 : g
```

3. Register Spilling

Spill order = 1, 4, 5, 2, 3, 0

To satisfy all call insts, we should have at least 2 buffers. We spill 1, 2, 4, 5.

```
define i32 @main() {
  %color1 = alloca i64, align 8
  %color2 = alloca i64, align 8
  %color4 = alloca i64, align 8
  %color5 = alloca i64, align 8
  %a = ...
  store %a, %color4
  %b = ...
  %c = ...
  store %c, %color2
; current loaded color = [0, 3 / 4, 2]
  %cmp = icmp eq i32 %a, %b
  br i1 %cmp, label %BB1, label %BB2
BB1:
; current loaded color = [0, 3 / 4, 2]
  %d = ...
  %e = ...
  store %e, %color1
; current loaded color = [0, 3 / 1, 2]
  %f = call i32 @foo(i32 %b, i32 %c, i32 %d, i32 %e)
  br label %BB2
BB2:
; current loaded color = [0, 3 / ]
```

```
  %g = phi i32 [%d, %BB1], [0, %entry]
  %h = phi i32 [%e, %BB1], [0, %entry]
  store %g, %color5
; current loaded color = [0, 3 / 5]
  load %a.0, %color4
; current loaded color = [0, 3 / 5, 4]
  %i = call i32 @foo(i32 %a.0, i32 %b, i32 %g, i32 %h)
  load %c.0, %color2
; current loaded color = [0, 3 / 2, 4]
  load %g.0, %color
; current loaded color = [0, 3 / 2, 5]
  %j = call i32 @foo(i32 %c.0, i32 %b, i32 %g.0, i32 %h)
  ret i32 0
}
```

4. Coloring after Register Spilling

```
main() : 4 colors needed
color 0 : f h cmp d i j
color 1 : a e g c.0
color 2 : c a.0 g.0
color 3 : b
```