sprint2-team9-progress Report

홍순범, 김경호, 나동현, 오준석

1. Loop unrolling Optimization(홍순범)

Loop는 condition, body, incr block으로 구성되어있다. 각각의 block을 넘어갈 때 branch 오퍼레이션과 사용할 변수들을 메모리에서 읽어와야하는 등의 overhead가 발생한다. Spec에 따르면 branch는 최소 30의 cost가 드는 연산으로 꽤 높은 cost가 필요하다. Sprint2에서는 이러한 문제를 해결하기 위한 loop unrolling을 진행하였다.

우선 loop의 condition, body, incr 등의 정보를 잘 볼 수 있도록 canonical form으로 변경하고 몇 번 반복되는지 trip count를 구했다. 그리고 Nested loop도 처리할 수 있게 innermost loop부터 recursive하게 unrolling을 진행하도록 했다. Loop unrolling이 끝난 후에는 불필요하게 생긴 block을 제거하기 위해서 SimplifyCFG를 진행했다.

[Progress]

1. Loop를 canonical form으로 변경

```
FPM.addPass(PromotePass());
FPM.addPass(SROAPass(SROAOptions::PreserveCFG));
FPM.addPass(LoopSimplifyPass());
llvm::LoopPassManager LPM;
LPM.addPass(IndVarSimplifyPass());
FPM.addPass(
    llvm::createFunctionToLoopPassAdaptor(std::move(LPM)));
```

2. Recursive하게 innermost loop부터 시작

```
std::function<void(Loop *)> unrollLoopRecursively = [&](Loop *L) {
  for (Loop *SubLoop : L->getSubLoops()) {
    unrollLoopRecursively(SubLoop);
  }
```

3. tripCount 파악 후 unroll

```
unsigned TripCount = SE.getSmallConstantTripCount(L);
if (TripCount != 0) {
  ULO.Count = std::min(TripCount, unsigned(32));
  ULO.Force = true;
  ULO.Runtime = true;
  ULO.AllowExpensiveTripCount = true;
  ULO.UnrollRemainder = true;
  ULO.ForgetAllSCEV = true;
  LoopUnrollResult result =
      UnrollLoop(L, ULO, &LI, &SE, &DT, &AC, &TTI, &ORE, false);
  if (result != llvm::LoopUnrollResult::FullyUnrolled) {
    simplifyLoopAfterUnroll(L, true, &LI, &SE, &DT, &AC, &TTI);
  }
```

4. unroll 후처리

```
FPM.addPass(SimplifyCFGPass());
```

원래 계획은 기존 llvm에 있던 UnrollLoop를 이용하여 fully unroll을 진행하고 그 위에서 vectorization을 진행하려 하였다. 하지만 UnrollLoop를 사용하는 중에 에러가 많이 나게 되었고 vectorization은 진행하지 못하고 loop unrolling만 구현하였다.

문제1. tripCount를 제대로 읽지 못 함. C코드에서 단순히 ll 코드를 생성하였을 때 constant만큼 iteration하는 경우에도 loop incr 조건에서 변수를 메모리에서 load를 하게되는데 이럴 경우에 컴파일 시간에 tripCount를 읽지 못하는 문제가 있었다. Loop unroll 전에 mem2reg 패스를 적용해서 해결하였다.

문제2. Loop unroll이 끝난 후에 loop body가 하나의 block으로 합쳐지는 게 아니라 모두 각각의 block으로 분리되는 문제가 있었다. SimplifyCFG패스를 적용해서 해결하였다.

[Result]

그리고 모든 벤치마크를 통해 전반적인 성능평가를 진행하였고 cost의 감소를 확인하였다. 특히 반복연산이 많은 matmul같은 경우 큰 효과를 볼 수 있었다.

	before	after
bitcount4	1,127,641	1,106,335
jenkins_hash	1,854,272	1,853,419
matmul4	3,364,236,177	2,135,030,431
All	495,274,070	450,777,229

2. Heap to Stack Optimization(김경호)

[Progress]

이번 스프린트에서는 LLVM IR 코드의 성능 향상을 위해 Heap Allocation을 Stack Allocation으로 대체하는 StackAllocPass 최적화 패스를 구현하였다. 이 패스는 각 함수의 기본 블록을 iterate하며 malloc 호출을 식별하고, allocated memory가 지정된 threshold(1024 바이트) 이하인 경우 malloc 호출을 alloca 연산으로 대체하였다. 이에 대응되는 free 호출도 Stack Allocation의 Undefined Behavior를 피하기 위해 제거하였다. 이러한 변환은 더 낮은 cost를 갖는 Stack Allocation메커니즘을 활용하여 heap allocation오버헤드를 줄였다. 다양한 LLVM IR 테스트 케이스를 사용하여 정확성과 효율성을 검증하였고, malloc 호출이 올바르게 변환되고 의도한 대로 free 호출이 제거되었음을 확인하였다. 테스트는 다음과 같이 구성했다.

[Result]

```
//Test to see if memory is allocated to the stack instead of the heap
define void @test_only_malloc() {
    entry:
        %ptr = call ptr @malloc(i64 16)
        store i8 42, ptr %ptr, align 1
        ret void
}
```

```
define void @test_only_malloc() {
    entry:
      %ptr = alloca i8, i64 16, align 1
      store i8 42, ptr %ptr, align 1
      ret void
}
```

```
//Test to see if free is removed from the IR
define void @test_free_removal() {
    entry:
        %ptr = call i8* @malloc(i64 16)
        call void @free(i8* %ptr)
        ret void
}
```

```
define void @test_free_removal() {
    entry:
      %ptr = alloca i8, i64 16, align 1
      ret void
}
```

```
//Test to see if the Stack Allocation pass isn't invoked
define i8* @large_heap_allocation() {
    entry:
        %size = alloca i64, align 8
        store i64 4, i64* %size, align 8
        %ptr1 = call i8* @malloc(i64 4)
        %ptr2 = call i8* @malloc(i64 8192)
        ret i8* %ptr2
}
```

```
define i8* @large_heap_allocation() {
    entry:
      %size = alloca i64, align 8
      store i64 4, i64* %size, align 8
      %ptr1 = alloca i8, i64 4, align 1
      %ptr2 = call i8* @malloc(i64 8192)
      ret i8* %ptr2
    }
```

test_only_malloc: 이 테스트는 malloc 호출이 힙 대신 스택에 메모리를 할당하는지 확인한다. malloc 호출의 크기가 상수이며 지정된 임계값(1024 바이트) 이하인 경우, alloca 연산으로 대체되어야 한다.

test_free_removal: 이 테스트는 malloc 호출이 스택에 할당된 후 관련된 free 호출이 올바르게 제거되는지 확인한다. malloc 호출이 스택에 할당되면, 해당 메모리를 free하는 free 호출은 필요하지 않으므로 free 호출은 제거되어야 한다.

large_heap_allocation: 이 테스트는 Stack Allocation 패스가 임계값보다 큰 메모리 할당에는 적용되지 않는지 확인한다. malloc 호출의 크기가 상수이며 지정된 임계값보다 큰 경우, alloca 연산으로 대체되지 않아야 한다.

이러한 테스트를 통해 StackAllocPass가 의도한 대로 작동하는지 확인할 수 있었다.

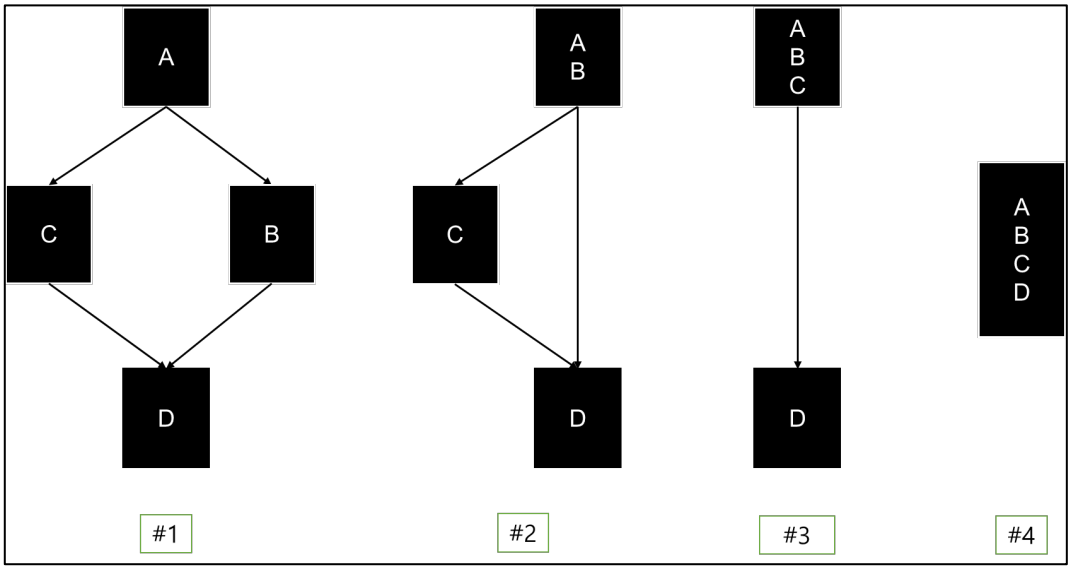
3. Control Flow Graph Optimization(나동현)

[Progress]

제시받은 asmspec에서 machine에서는 branch의 cost가 unconditional, conditional, switch를 가리지

않고 상당히 높아 프로그램의 CFG를 위주로 optimization을 진행할 여지가 많다. Sprint 1에서 이미 다른 팀원분이 loop에서 branch probability를 분석한 뒤 그 값에 따라 True, False를 변경하는 최적화를 진행하였지만, CFG를 직접 변경하는 과정을 통해 더 최적화를 진행할 수 있을 것이라 판단하여, CFGOptPass()를 구현하였다. 또한, 다른 팀원분이 추가하신 LLVM의 InstCombinePass()와 시너지를 낼 수 있을 거 같은, associative한 연산을 단순화가 편한 형태로 만들어주는 ReassociatePass()를 추가했다.(페이지가 부족하므로 이에 대한 내용은 이로 갈음한다).

주로 predecessor가 conditional branch/unconditional branch를 가질 때 BB를 모순되지 않게 predecessor에 병합하는 기능을 구현하였고, 구현된 pass는 다음과 같은 기능을 수행할 수 있다. 간단히 묘사하자면 아래 그림에서 왼쪽에 있는 구조를 오른쪽으로 경우에 따라서 변환시킨다. #1을 #2, #3로 변환시키는 과정과, #3을 #4로 변환시키는 과정이 각각 나누어져 있다.



본래 구현하려고 하던 기능 세 가지를 이를 통해 모두 수행 가능하다.

Reqspec 기준 optimization	방법
If-else if-else to switch	If-else if-else 문은 두 번 conditional branch가 있는 후에 세 block이 한 block으로 모이므로, #2가 두 번 중첩되는 구조이다. #2->#4 과정 두 번으로 최적화가 가능하다.
Merge unconditional branch	#3 to #4. MergeBlockIntoPredecessor()를 이용했다
Diamond to Ternary operation	#1 to #4.

해당 변환을 적용하기 위해 B, C는 single predecessor와 single successor를 가져야 할 필요가 있다. 코드 기능 변형을 막기 위해 Store(B, C에서 서로 다른 값을 저장하면 낭패)가 있는 경우나, A와 B, C가 서로 다른 loop에 속하는 경우 predecessor로의 Merge를 배제하였다. 또한 총합 30 cost를 넘어가는 instruction이 있는 경우에도, branch를 없애는 대신 해당 instruction을 어느 branch로 가든 수행해야 하기 때문에 cost가 줄지 않거나 늘 수 있으므로 수행하지 않았다.

[Benchmarks]

프로그램명	Average cost(before/after)		프로그램명	Average cost(before/after)	
Anagram	294666694	232202702	Gcd	1413	1394

Bitcount1	2109	1704	Jenkins_hash	1854273	1837323
Bitcount2	1772	1756	Matmul1	1831034524	1688368112
Bitcount3	1599	1307	Matmul2	2346869554	2204203044
Bitcount4	1127641	1119931	Matmul3	962970010	958959113
Bitcount5	67652	67652	Matmul4	3364236177	2976789188
Bubble_sort	2957756893	2740870338	Merge_sort	40570258	36917445
Collatz	5028	4969	Rmq1d_naive	1028169961	1028124417
Floyd	8629073	8529989	Rmq1d_sparsetable	99385852	94959846
Friend	111990356	111401529	Rmq2d_naive	20642930	19178616
Game	31589666	30719669	Rmq2d_sparsetable	438810118	428690392

Merge가 일어난 정도에 따라 다양하지만, 전반적으로 많은 프로그램의 cost가 많이 줄어들었다.

4. Upgrade Global Variable Optimization(오준석)

이전 Sprint 1에서는 Global Variable을 Stack으로 바꾸는 방식으로 GVoptPass를 구현하였다. 이런 구현에는 2가지 문제점이 있었다.

1. GV에 해당하는 stack을 할당하고, 함수등을 호출하면 다시 저장하고, 돌아오면 다시 할당하는 등의 overhead가 너무 커져서 cost가 오히려 늘었음
2. stack overflow의 문제가 생길 수 있음

그렇기에, GVoptPass를 다시 설계 및 구현하기로 했다. 변경된 GVoptPass는 다음과 같은 구조를 가진다.

[Progress]

1. overhead대비 확실한 cost 이득을 위해 Loop내에서 사용되는 GV들만 대체한다
2. 해당 GV에 loop 내부에서 store이 있는지 확인한다. store이 안되는 GV들만 optimize한다.
3. 해당 GV를 loop의 header에서 1회 register(기존의 Stack(alloca)과는 다르다)에 load한 후, 이후 loop 내부에서 GV를 load할 때, 대신 해당 register을 사용한다.

(이 과정에서 Load하는 명령어는 삭제, load된 reg를 header에서 load한 reg로 Swap하면 됨)

이러한 좁은 범위에 국한된 구현을 통해, 1. 확실한 cost 감소 2. Error case 감소(or 삭제) 의 2가지 효과를 기대했다.

[Benchmark]

```
total:  109455569663
average:  495274070.8733032
```

- opt ->

```
total:  109427476813
average:  495146953.9049774
```

총 cost 자체는 그렇게 많이 줄어들지 않은 것이 확인 가능하다. 그러나, 이는 GV를 사용하지 않는 testcase들이 많기에 생긴 일이다. GV를 사용하는 testcase를 살펴보자.

```
------- rmq2d_sparsetable
Final Cost : 98219
------- rmq2d_sparsetable
Final Cost : 295300
------- rmq2d_sparsetable
Final Cost : 1257595
------- rmq2d_sparsetable
Final Cost : 3602490
------- rmq2d_sparsetable
Final Cost : 2188170884
```

```
433 ------- rmq2d_sparsetable < input1 -------
434 Final Cost : 98519
435 ------- rmq2d_sparsetable < input2 -------
436 Final Cost : 295950
437 ------- rmq2d_sparsetable < input3 -------
438 Final Cost : 1259345
439 ------- rmq2d_sparsetable < input4 -------
440 Final Cost : 3609740
441 ------- rmq2d_sparsetable < input5 -------
442 Final Cost : 2188787034
```

위는 GV를 사용하는 rmq2d_sparsetable이라는 testcase이다. 확실히 GVoptPass가 적용된(좌측) 경우가 그렇지 않은 경우에 비해 cost가 줄은 것이 확인 가능하다. 주목할 점으로는, input1에서 input5로 감에 따라 입력값과 그 가지수가 커지고, 이에 따라 loop을 도는 횟수가 늘어나며, 그에 따라 GV optimization에 의한 이득이 커지기 때문에, 줄어드는 cost의 값이(300 -> 616150) 점점 커짐을 알 수 있다.

```
----------Running tests for (rmq2d_naive)----------
Test 1 passed.
Test 2 passed.
Test 3 passed.
Test 4 passed.
Test 5 passed.
----------Running tests for (rmq2d_sparsetable)---------
Test 1 passed.
Test 2 passed.
Test 3 passed.
Test 4 passed.
Test 5 passed.
```

또한 추가적으로, benchmark의 주어진 input을 GVoptPass에 의해 optimize된 .s(어셈블리) 파일들에 넣고 interpreter로 실행시켰을 때, 나오는 결과값이 benchmark상 output과 동일한지 확인하는 checker 프로그램을 돌려보았을 때, 동일함을 확인하였다. 비록 cost 감소가 살짝 불만족스러울 수는 있지만, 적절한 구현을 했음을 확인하여 만족스러웠다. 해당 checker 프로그램은 Sprint3부터 배포하여 사용할 예정이다.