2024-1 Principles and Practices of Software Development
Seoul National University
Department of Computer Science & Engineering

김경호, 오준석, 나동현, 홍순범

Sprint 3 Documentation

Vectorization (홍순범)

반복되는 instruction 이 나오는 경우 벡터화시키는 패스를 구현했습니다. 스펙에서 벡터연산은 스칼라연산보다 cost 를 2 배 더 소모하는 것을 확인할 수 있습니다. 따라서 2 번 넘게 반복되는 instruction 만 벡터화시킬 수 있어도 cost 감소를 기대할 수 있을 것입니다.

벡터화시키는 조건(1. 같은 instruction 의 묶음이 반복되는 패턴을 찾기, 2. 묶인 instruction 들에서 같은 메모리에 read, write 를 하지 않는다)를 만족하는 instruction 들에 벡터화를 진행했습니다.

Planning 에서는 가능한 모든 패턴의 instruction 들을 찾아서 벡터화시키려 했지만 모든 instruction 들이 나오는 패턴을 찾는 것에 어려움이 있어 가장 흔하게 벡터화시킬 수 있는 패턴인 `c[i] = a[i] + b[i]`의 패턴만 찾아서 vectorization 을 진행했습니다.

[Implementation]

Vectorizable 조건 체크

```
// Check for write-after-write
if (I->mayWriteToMemory() && J->mayWriteToMemory()) {
  if (I->getOperand(0) == J->getOperand(0)) {
    return false;
  }
}
// Check for read-after-write
if (I->mayWriteToMemory() && J->mayReadFromMemory()) {
  if (I->getOperand(0) == J->getOperand(0)) {
    return false;
  }
}
// Check for write-after-read
if (I->mayReadFromMemory() && J->mayWriteToMemory()) {
  if (I->getOperand(0) == J->getOperand(0)) {
    return false;
  }
}
```

벡터화 가능한 패턴 찾기

```
isa<GetElementPtrInst>(pattern[0]) && isa<LoadInst>(pattern[1]) &&
isa<GetElementPtrInst>(pattern[2]) && isa<LoadInst>(pattern[3]) &&
(pattern[4]->getOpcodeName() == instName) &&
isa<GetElementPtrInst>(pattern[5]) && isa<StoreInst>(pattern[6]);
```

벡터화가능한 패턴의 반복 찾기

```
if (isVectorizablePattern(pattern)) {
  size_t repeatCount = 1;
  const char *instName = pattern[4]->getOpcodeName();
  while (i + 7 * repeatCount + 6 < scalarOps.size()) {
    std::vector<Instruction *> nextPattern(
        scalarOps.begin() + i + 7 * repeatCount,
        scalarOps.begin() + i + 7 * repeatCount + 7);

    if (isVectorizablePattern(nextPattern, instName)) {
      repeatCount++;
    } else {
      break;
    }
  }
```

기존 instruction 들을 벡터로 대체

```
Value *vecA = Builder.CreateLoad(VecType, aVecPtr);
Value *vecB = Builder.CreateLoad(VecType, bVecPtr);
Value *vecC = nullptr;
if (strcmp(instName, "add") == 0) {
  vecC = Builder.CreateAdd(vecA, vecB);
} { ... }
Builder.CreateStore(vecC, cVecPtr);
```

[Result]

벤치마크를 통해 확인한 cost 감소는 아쉽지만 확인할 수 없었다. Unrolling 후 연속되는 패턴을 찾을 목적이기 때문에 unrolling 이후 cost 와 비교했지만 cost 감소는 없었다. Cost 감소가 없는 이유로는 찾은 패턴은 벤치마크에서 흔하게 나오는 패턴이었지만 unrolling 등 다른 패스가 적용된 이후 벡터화가능한 패턴이 달라져서 그랬을 것으로 추측한다

	After unrolling	After vectorization
All	439,804,066	439,804,066

Fix: add runtime loop unrolling on sprint2 unrolling pass (홍순범)

Sprint2 에서 loop unrolling 패스를 만들었습니다. Trip count 를 확인할 때 컴파일 타임에 확인할 수 있는 횟수만큼 unrolling 을 시켰는데 따라서 런타임에 결정되는 반복은 unrolling 하지 못했습니다. 컴파일때 확인할 수 없는 반복도 4 개 단위로 묶어서 unrolling 을 할 수 있도록 작업했습니다.

[Implementation]

`ULO.Count = TripCount != 0 ? std::min(TripCount, unsigned(32)) : 4;`

위와 같이 tripCount 를 파악할 수 없는 경우에도 4 의 tripCount 로 unrolling

[Result]

그리고 모든 벤치마크를 통해 전반적인 성능평가를 진행하였고 cost 의 감소를 확인하였다. 특히 반복연산이 많은 matmul 같은 경우 큰 효과를 볼 수 있었다.

	before	after (sprint2)	after (sprint3)
All	495,274,070	450,777,229	439,804,066

Progression & Explanation of Sprint 3 (나동현)

원본 LLVM IR 코드상에서는 다른 변수로 표현되더라도 실질적인 값이 같은 경우가 생길 수 있다. 이러한 경우를 포착하여 다른 변수로 표현된 use 들을 모두 하나의 변수에 대한 use 로 대체하면, 같은 operands 를 가지는 연산을 최대한 많이 만들어낼 수 있다. 같은 operand 를 가진다는 조건 하에서는 많은 instruction 들을 더 cost 가 적은 다른 instruction 이나 어떤 변수, 상수값으로 바꾸어 cost 를 줄이는 최적화를 진행할 수 있다. 이러한 논리 하에 세 가지 구현을 수행하였다.

먼저 첫 번째로는 CFG 상에서 terminator 가 conditional branch 일 때 condition 의 종류가 “icmp eq/ne ixx %a, %b”일 경우, 각각 true/false branch 의 edge 에 지배되는 곳에서 %b 를 %a 로 변환하는 propagation 을 구현하였다(=icmp_propagation 패스). 이때 dominate 의 개념만으로는, 수학적으로 어떤 BB 에서 %a == %b 가 성립하지만 이 BB 를 향하는 predecessor 가 여러 개인 경우에 propagation 이 가능함에도 이를 하지 못하므로, 이 경우도 찾아서 빠짐없게 propagation 이 일어나도록 하였다.

두 번째로는 CSE(Common Subexpression Elimination)를 큰 주제로, 전체 instruction 중 실제로 같은 값을 계산하거나 이름만 다르고 같은 instruction 인 경우를 추려 가장 상위의 한 instruction 만을 참조하도록 바꾸고자 했는데, 이 경우 EarlyCSE, NewGVN 이라는 기존의 패스가 이 최적화를 이미 충분히 진행하여, 개발한 custom pass 앞뒤에 추가하였다. 반면 이들은 두 Instruction 의 Nearest common dominator 로 한 Instruction 을 Hoisting 시키고, 이 instruction 이 use 를 대체하게 하는 기능은 수행하지 않아, 해당 기능을 구현하고 두 패스를 앞뒤로 붙여, 최대한 많은 Instruction 의 operand 를 같게 만드는 것에 중점을 둔 패스 make_same_operand 패스를 구현하였다. 세 번째로는 이제 이 구현한 패스들에 의해 operand 가 같은 연산이 최대한 생겼을 때 이들을 적절히 교체하는 optimization 을 기존의 arith_opt 에 구현하였고 그 과정에서 함께 기존의 매우 알아보기 난해했던 코드를 보다 간편한 구조로 refactoring 하였다.

[Benchmarks]

아래는 이번 Sprint3 에서 구현한 세 가지 패스를 모두 적용했을 때의 average cost 의 차이를 보여준다. Icmp_propagation, make_same_operand 패스는 사실상 같은 operand 를 가지는 연산 최적화의 효과를 극대화하기 위한 준비작업이기에 따로 구분하지 않고 한 번에 포함시켜 cost 를 측정하였다. Gcd 를 제외한 모든 프로그램에서 약간의 개선을 보였다. Refactoring 및 bugfix 과정에서 AShr->SDiv optmization 이 incorrect transformation 임을 찾아내어 수정하였기에 개선되는 값의 크기는 sprint 1 의 arith_opt 만을 적용했을 때보다 더 줄었지만, 더욱 안정성이 높은 패스가 되었다.

프로그램명	Average cost(before/after)		프로그램명	Average cost(before/after)	
Anagram	294666694	290209762	Gcd	1413	1413
Bitcount1	2109	2023	Jenkins_hash	1854273	1796491
Bitcount2	1772	1744	Matmul1	1831034524	1812091062
Bitcount3	1599	1514	Matmul2	2346869554	2323267803
Bitcount4	1127641	1125840	Matmul3	962970010	921320987
Bitcount5	67652	67629	Matmul4	3364236177	3322004839
Bubble_sort	2957756893	2881897343	Merge_sort	40570258	40193239
Collatz	5028	4964	Rmq1d_naive	1028169961	1028157010
Floyd	8629073	8612003	Rmq1d_sparsetable	99385852	98830423
Friend	111990356	111796392	Rmq2d_naive	20642930	20629622
Game	31589666	31575232	Rmq2d_sparsetable	438810118	435414463

Stack to Heap Allocation Extension (김경호)

이번 Sprint 에서는 이전 Sprint 2 에서 작성한 Stack to Heap Allocation Pass 에 recursion detection 을 추가해서 pass 의 잠재적 undefined behavior 를 잡는 작업으로 진행했다. 패스는 각 함수의 기본 블록을 iterate 하며 malloc 호출을 식별하고, 할당된 메모리가 지정된 threshold(1024 바이트) 이하이고 recursive function 이 아닌

것을 확인해서 malloc 호출을 alloca 연산으로 대체했다. 이러한 변환은 더 안전하게 낮은 cost 를 갖는 Stack Allocation 메커니즘을 활용하여 heap allocation 오버헤드를 줄였다.

최대한 deep recursion 만 식별해서 optimization pass 를 skip 하는 구현을 구성해 보려고 노력했지만 기준점을 잡기가 힘들었다. 그러므로 최대한 보수적인 방법으로 recursion 이 detect 되는 경우, optimization 을 진행되지 않게 pass 를 구성했다. 추가적으로, recursive 한 function 이 있을 경우 pass 가 implementation 작동하지 않는것을 확인하기 위해서 Recursion 이 있는 LLVM IR 코드가 변환하지 않는 test case 를 작성했다.

```
recurse:
  %ptr = call ptr @malloc(i64 16)
  ; CHECK: call ptr @malloc(i64 16)
```

Delete Last Free/Function Inlining Optimization (오준석)

[Implementation & Progress]

Delete Last Free Optimization 은, 마지막으로 등장한 malloc 이후의 free 를 전부 삭제함으로써 free 에 드는 비용을 줄이며, heap 에 allocate 최대 사이즈를 늘리지 않는 Optimization 이다. 이는 다음과 같은 알고리즘으로 구현했다.

1) Function 내부를 돌면서 마지막으로 등장하는 malloc 을 찾아 lastMalloc 에 저장한다.

2) Free 가 나올 때 마다 해당 inst 가 lastMalloc 에 의해 Dominate 되고, lastMalloc 이 해당 inst 에 의해 Post Dominate 된다면, 해당 free inst 를 freesToDelete vector 에 push 한다. 해당 vector 은 lastMalloc 이 업데이트 될 때마다 초기화 된다.

2-1) 또 하나의 조건으로, malloc - free 가 하나의 loop 안에서 반복적으로 일어난다면 그 free 들을 지우면 allocate 된 heap 의 최대 크기가 늘어날 수 있음으로 해당 free 들은 지우지 않는다.

3) freesToDelete 에 있는 free inst 를 삭제한다.

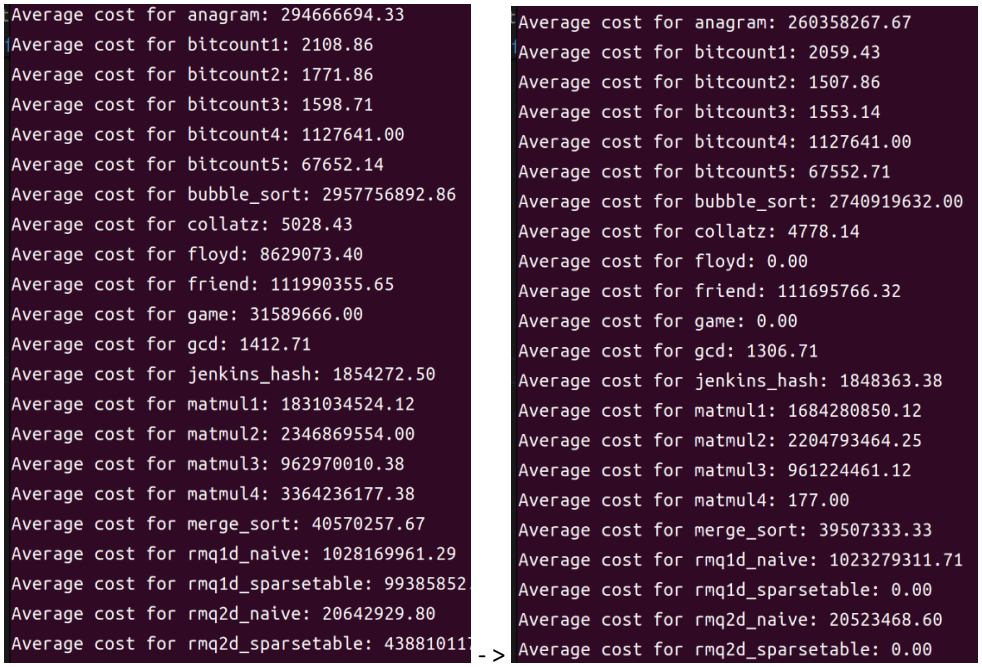
Function Inlining 의 경우, inline 할 function 의 조건으로 다음 3 가지를 고려했다.

1) 너무 많은 register 을 사용하지 않을 것. (총 inst 수)/(총 block 수) 로 해당 수치를 heuristic 하게 알아낸 후, 적절한 상한선인 20 을 설정했다. 이는 register 이 너무 많으면 stack 영역으로 가 문제가 생기기 때문이다.

2) 재귀함수가 아닐것 : 무한반복의 문제가 생김.

3) NoInline attribute 가 없을 것

아래는 Benchmark 에 대해 function inlining 을 적용한 결과이다.



위 결과를 비교해 보면, inlining 을 통해 모든 경우 cost 가 감소함을 알 수 있다.

다만 matmul4/floyd/game/rmq_sparsetable 의 경우, 컴파일 및 실행시 오류가 생겨 제외했다. 해당 오류는 Sprint3 시점에서는 아직 해결하지 못했으며, 현재 해결하기 위해 다방면으로 알아보는 중이다.

Delete last Free 의 경우, 아쉽게도 주어진 benchmark 에서는 cost 감소가 전혀 일어나지 않았다. 애초에 benchmark 에서 free 를 사용한 case 가 *jenkins_hash* 와 *merge_sort* 밖에 없었으며. 해당 케이스를 살펴본 결과, malloc 이 다른 function 의 내부에서 불렸기에 서로 다른 function 끼리 domination 을 확인하지 못하고, 삭제하지 못한 것으로 보인다.

한 function 내부에서 제대로 free 를 삭제하는 경우(github 상의 free_opt@3.ll) 에는 cost 가 감소함이 확인가능했다.

`Final Cost : 618070` -> `Final Cost : 617920` 으로 딱 150 만큼 줄었다.