CS6241: Project 1 - Part 1

Zhen Li zhenli.craig@gatech.edu 2024-03-26

Contents

I did this alone.

1 Baseline

The statistics are generated with `./gen_baseline_stat.py`.

The llvm bytecode size are measured by python script `os.path.get_size()`.

The time are measured by a `time` based profiling tool `hyperfine`. It runs the program 2 times for warmup and then 10 times to take the average. (macOS seems do not have equivalent to `perf` on Linux.)

Bench		Bytecode Size (byte)	Mean User (ms)	Mean System (ms)	Mean Total (ms)
is	original	22784	16.603	0.558	17.562
	baseline	24800	21.232	0.561	22.153
	ratio	**108.8%**	**127.9%**	**100.4%**	**126.1%**
bfs	original	14384	833.525	27.399	864.265
	baseline	15728	1022.213	27.226	1054.901
	ratio	**109.3%**	**122.6%**	**99.4%**	**122.1%**
dither	original	31984	39.241	2.016	42.365
	baseline	34992	72.203	2.631	76.653
	ratio	**109.4%**	**184.0%**	**130.5%**	**180.9%**
jacobi-1d	original	5840	1270.432	1.748	1286.880
	baseline	6624	9738.391	5.185	9836.787
	ratio	**113.4%**	**766.5%**	**296.6%**	**764.4%**
check_elimination	original	3728	0.261	0.451	1.078
	baseline	4160	0.326	0.452	1.077
	ratio	**111.6%**	**125.0%**	**100.3%**	**99.9%**
check_modification	original	3744	0.207	0.318	0.560
	baseline	4176	0.205	0.209	0.604
	ratio	**111.5%**	**99.3%**	**65.7%**	**107.8%**

Table 1: Performance comparison between original programs and after the `check-ins` pass

2 Gupta's methods

2.1 Bytecode size and performance

The statistics are generated with `./gen_gupta_stat.py`. The measurements are the same as the baseline.

Bench		Bytecode Size (byte)		Mean User Time (ms)		Mean Total Time (ms)	
is	original	22784	100.0%	16.603	100.0%	17.562	100.0%
	baseline	24800	108.8%	21.232	127.9%	22.153	126.1%
	gupta	24272	106.5%	18.645	112.3%	20.106	114.5%
	gupta / baseline		**97.9%**		**87.8%**		**90.8%**
bfs	original	14384	100.0%	833.525	100.0%	864.265	100.0%
	baseline	15728	109.3%	1022.213	122.6%	1054.901	122.1%
	gupta	14992	104.2%	924.663	110.9%	960.220	111.1%
	gupta / baseline		**95.3%**		**90.5%**		**91.0%**
dither	original	31984	100.0%	39.241	100.0%	42.365	100.0%
	baseline	34992	109.4%	72.203	184.0%	76.653	180.9%
	gupta	32656	102.1%	50.717	129.2%	54.459	128.5%
	gupta / baseline		**93.3%**		**70.2%**		**71.0%**
jacobi-1d	original	5840	100.0%	1270.432	100.0%	1286.880	100.0%
	baseline	6624	113.4%	9738.391	766.5%	9836.787	764.4%
	gupta	6368	109.0%	3066.606	241.4%	3085.277	239.7%
	gupta / baseline		**96.1%**		**31.5%**		**31.4%**
check_elimination	original	3728	100.0%	0.261	100.0%	1.078	100.0%
	baseline	4160	111.6%	0.326	125.0%	1.077	99.9%
	gupta	4128	110.7%	0.326	125.0%	0.979	90.8%
	gupta / baseline		**99.2%**		**100.0%**		**90.9%**
check_modification	original	3744	100.0%	0.207	100.0%	0.560	100.0%
	baseline	4176	111.5%	0.205	99.3%	0.604	107.8%
	gupta	4128	110.3%	0.315	152.4%	0.974	173.9%
	gupta / baseline		**98.9%**		**153.4%**		**161.4%**

Table 2: Performance comparison between original programs and after the `check-opt` pass

2.2 Compile time check counts

Compile time check counts are counted during llvm pass. Upperbound and lowerbound checks are counted separately even if they are checking the same GEP.

Bench	After Insertion	After Modification	After Elimination	After Loop Hoisting	Percentage removed
is	56	58	34	23	**-58.9%**
bfs	50	54	20	16	**-68.0%**
dither	186	229	72	67	**-64.0%**
jacobi-1d	22	22	8	7	**-68.2%**
check_elimination	6	8	3	3	**-50.0%**
check_modification	6	6	3	3	**-50.0%**
malloc_1d_array	6	6	6	6	**0.0%**

Bench	After Insertion	After Modification	After Elimination	After Loop Hoisting	Percentage removed
static_1d_array	6	6	6	6	**0.0%**
global_1d_array	6	6	6	6	**0.0%**

Table 3: Bound check counts comparison between `check-ins` and `check-opt` pass (compile time)

2.3 Runtime check counts

The statistics are generated with a stub printing check hits (`BoundCheckWithDump.o`) at runtime and then counting them. `malloc` uses `1 2 3` as arguments. `jacobi-1d` is too large so it's not included.

Bench	Baseline	Gupta's	Percentage removed
is	8086068	6128956	**-24.2%**
bfs	147581344	80999858	**-45.1%**
dither	22984824	8377747	**-63.6%**
check_elimination	4	2	**-50.0%**
check_modification	4	2	**-50.0%**

Table 4: Comparison of bound checks actually performed at runtime

2.4 Static spill code generated in each commenting on causes of performance degradations

The stats are generated using `llc` from another debug build of llvm 17, since my release version of llvm 15 (which is used to compile the passes, and transform code) does not output anything with `--stats`:

```sh
./Ninja-DebugAssert/llvm-macosx-arm64/bin/llc --stats {bench}.bc -o {bench}.s
```

Bench	regalloc - Number of spills inserted		
	Original	Basline	Optimized
is	0	0	1
bfs	0	8	7
dither	0	20	15
jacobi-1d	0	2	0
malloc_1d_array	0	0	0
static_1d_array	0	0	0
global_1d_array	0	0	0
check_elimination	0	0	0
check_modification	0	0	0

Table 5: Spill counts

The script for generating the stats is `./gen_spill_stat.py`.

2.5 Detailed analysis

2.5.1 Co-relate performance degradation with the number of additional spills seen above and the total number of bounds check that are still present in the application

In my `check-opt` pass, for a single moved/modified check, where the subscript/bound values are `%y = A * %x + B`, the pass will `load %x` and create `A * (load %x) + B` from scratch, which introduces 1-3 new instructions (a load, an optional mul, an optional add).

Generally, inserting new variables and instructions can impact the register allocation phase of code gen, because the number of live variables increases to escalate the register pressure. When the live variables exceed the number of available registers, the compiler must spill some variables to memory, storing them on the stack or in global memory locations. Later, these variables may be reloaded back into registers when needed. This instrumentation leads to a more complex interference graph, more conservative allocation and hence increased spill code.

When calling bound check functions, the old alive variables need to be pushed to the stack and the new variables need to be loaded into registers. This can also lead to more spill and reload operations, which can degrade performance.

2.5.2 Why certain benchmarks show a lot of removal opportunities whereas others dont

1. Take `jacobi` as an example. The original code has something like this in the inner loop:

   ```cpp
   for (int i = 1; i < n - 1; i++)
       B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
   ```

   This introduces 8 checks initially, but since B and A has the same size, they can be eventually eliminated into 2 checks: $0 \leq i - 1$ and $i + 1 \leq N - 1$. The benchmarks with similar patterns inside single loop can have more removal opportunities.

2. When hoisting loop checks, I tried to evaluate the MIN/MAX of the possible values inside a loop (by finding all dominating stores to `ptr` or phi nodes). If the incoming value and MIN/MAX are all constants, they are evaluated at compile time and the checks is directly removed (`src/BoundCheckOptimization.cpp`, line 1994). If the incoming value is a variable, the check is kept. This is why `dither` and `jacobi` can have more removal opportunities.

2.5.3 Why removal is co-related to performance improvement in some cases whereas not co-related or less co-related in others and finally comparison of effectiveness

We benefit more from removing checks inside loops that will be executed more times. In `jacobi`, we removed 68.2% checks, and reduced runtime to 31.4% of the original. In `is`, we benefited only ~10% from the removal. This is because `jacobi` spends more time in the loops where we successfully hoisted or removed checks.

3 Run instruction

```sh
(cd stubs && ./build.sh) # build BoundCheck.o
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_INSTALL_PREFIX=./install -
DCMAKE_BUILD_TYPE=Debug -B build -S . -G Ninja
(cd build && ninja install)
# ...
clang++ stubs/BoundCheck.o install/<bench>-transformed.bc -o <bench>
```