15-745 Project Minestone - Superword Level Parallelism in LLVM

Ashwin Adulla, Li Shi

Ashwin Adulla, aadulla@andrew.cmu.edu
Li Shi, lishi@andrew.cmu.edu
https://15745-SLP-Project.github.io

1 Major Changes

We have not had any major changes since our project proposal.

2 What You Have Accomplished So Far

We have implemented most of the original SLP algorithm described in [1]. We have been able to identify opportunities for SLP, we are able to emit the vectorized LLVM code that correctly performs the same operations. Here are some examples of the SLP algorithm in action:

Example 1:

User Code

```
int test1(int a, int b, int c, int d, long i) {
  int e = a * A[i];
  int f = b * A[i + 1];
  int g = c * A[i + 2];
  int h = d * A[i + 3];
  return e + f + g + h;
}
```

Emitted LLVM Code:

```
define dso_local i32 @test1(i32 %arg, i32 %arg1, i32 %arg2, i32 %arg3, i64 %arg4)
local_unnamed_addr #1 {
bb:
  %tmp = getelementptr inbounds [16 x i32], [16 x i32]* @A, i64 0, i64 %arg4
  %0 = bitcast i32* %tmp to <4 x i32>*
  %1 = load <4 x i32>, <4 x i32>* %0
  %2 = insertelement <4 x i32> undef, i32 %arg, i64 0
  %3 = insertelement <4 x i32> %2, i32 %arg1, i64 1
  %4 = insertelement <4 x i32> %3, i32 %arg2, i64 2
  %5 = insertelement <4 x i32> %4, i32 %arg3, i64 3
  %6 = mul <4 x i32> %1, %5
  %7 = extractelement <4 x i32> %6, i64 0
  %8 = extractelement <4 x i32> %6, i64 1
  %9 = extractelement <4 x i32> %6, i64 2
  %10 = extractelement <4 x i32> %6, i64 3
  %tmp19 = add nsw i32 %8, %7
  %tmp20 = add nsw i32 %tmp19, %9
  %tmp21 = add nsw i32 %tmp20, %10
  ret i32 %tmp21
}
```

Description:

1. SLP algorithm recognizes that we are accessing 4 consecutive locations within array A
2. Perform 4 wide bitcast of A[i:i+4] into a vector (%0, %1)
3. Create a vector of <a, b, c, d> (%2, %3, %4, %5)
4. Perform element-wise vector multiplication (%6)
5. Unpack vector in <e, f, g, h> (%7, %8, %9, %10)

Example 2:

User Code:

```
int test2(long i) {
  C[i]     = A[i + 1] + B[i + 2];
  C[i + 1] = A[i + 2] + B[i + 3];
  C[i + 2] = A[i + 3] + B[i + 4];
  C[i + 3] = A[i + 4] + B[i + 5];
  return 0;
}
```

Emitted LLVM Code:

```
define dso_local i32 @test2(i64 %arg) local_unnamed_addr #2 {
bb:
  %tmp = add nsw i64 %arg, 1
  %tmp1 = getelementptr inbounds [16 x i32], [16 x i32]* @A, i64 0, i64 %tmp
  %0 = bitcast i32* %tmp1 to <4 x i32>*
  %1 = load <4 x i32>, <4 x i32>* %0
  %tmp3 = add nsw i64 %arg, 2
  %tmp4 = getelementptr inbounds [16 x i32], [16 x i32]* @B, i64 0, i64 %tmp3
  %2 = bitcast i32* %tmp4 to <4 x i32>*
  %3 = load <4 x i32>, <4 x i32>* %2
  %4 = add <4 x i32> %3, %1
  %tmp7 = getelementptr inbounds [16 x i32], [16 x i32]* @C, i64 0, i64 %arg
  %5 = bitcast i32* %tmp7 to <4 x i32>*
  store <4 x i32> %4, <4 x i32>* %5
  ret i32 0
}
```

Description:

1. SLP algorithm recognizes that we are accessing 4 consecutive locations within array A
2. Perform 4 wide bitcast of A[i+1:i+5] into a vector (%0, %1)
3. SLP algorithm recognizes that we are accessing 4 consecutive locations within array B
4. Perform 4 wide bitcast of B[i+2:i+6] into a vector (%2, %3)
5. Perform element-wise vector addition (%4)
6. SLP algorithm recognizes that we are storing into 4 consecutive locations within array C
6. Perform 4 wide bitcast of addition result into int* pointer (%5)
7. Store into array C (store)

Example 3 (Example 1 + Example 2):

User Code:

```
int test3(int a, int b, int c, int d, long i) {
  C[i]     = A[i + 1] + B[i + 2] * a;
  C[i + 1] = A[i + 2] + B[i + 3] * b;
  C[i + 2] = A[i + 3] + B[i + 4] * c;
  C[i + 3] = A[i + 4] + B[i + 5] * d;
  return 0;
}
```

Emitted LLVM Code:

```
define dso_local i32 @test3(i32 %arg, i32 %arg1, i32 %arg2, i32 %arg3, i64 %arg4)
local_unnamed_addr #2 {
bb:
  %tmp = add nsw i64 %arg4, 1
  %tmp5 = getelementptr inbounds [16 x i32], [16 x i32]* @A, i64 0, i64 %tmp
  %0 = bitcast i32* %tmp5 to <4 x i32>*
  %1 = load <4 x i32>, <4 x i32>* %0
  %tmp7 = add nsw i64 %arg4, 2
  %tmp8 = getelementptr inbounds [16 x i32], [16 x i32]* @B, i64 0, i64 %tmp7
  %2 = bitcast i32* %tmp8 to <4 x i32>*
  %3 = load <4 x i32>, <4 x i32>* %2
  %4 = insertelement <4 x i32> undef, i32 %arg, i64 0
  %5 = insertelement <4 x i32> %4, i32 %arg1, i64 1
  %6 = insertelement <4 x i32> %5, i32 %arg2, i64 2
  %7 = insertelement <4 x i32> %6, i32 %arg3, i64 3
  %8 = mul <4 x i32> %3, %7
  %9 = add <4 x i32> %8, %1
  %tmp12 = getelementptr inbounds [16 x i32], [16 x i32]* @C, i64 0, i64 %arg4
  %10 = bitcast i32* %tmp12 to <4 x i32>*
  store <4 x i32> %9, <4 x i32>* %10
  ret i32 0
}
```

Description:

1. Create vector A[i+1:i+5] (%tmp, %tmp5, %0, %1)
2. Create vector B[i+2:i+6] (%tmp7, %tmp8, %2, %3)
3. Create vector <a, b, c, d> (%4, %5, %6, %7)
4. A[i+1:i+5] + B[i+2:i+6] * <a, b, c, d> (%8, %9)
5. Store into C[i:i+4] (%tmp12, %10, store)

We also set up the environment to evaluate the performance of optimized code in gem5 microarchitecture simulator. As the ARM instruction set architecture provides good support for SIMD instructions and at the same time is easy to understand, we mainly focus on aarch64 to perform the evaluation. For example, the command we use to execute a test program is

```
all:
 ./build/ARM/gem5.opt configs/example/se.py \
      --cpu-type O3_ARM_v7a_3 \
      --caches --l2cache \
      --interp-dir /usr/aarch64-linux-gnu \
      --redirects /lib=/usr/aarch64-linux-gnu/lib \
      --cmd ../15745/SLP-Project/tests/tests.slp.out
```

The result is

```
...
Setting the interpreter path to: /usr/aarch64-linux-gnu
For dynamically linked applications you might still need to setup the --redirects so
that libraries are found

Global frequency set at 1000000000000 ticks per second
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
build/ARM/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting
simulation...
build/ARM/sim/mem_state.cc:443: info: Increasing stack size by one page.
Exiting @ tick 105852500 because exiting with last active thread context
```

We may also refer to m5out/stats.txt for more information during simulation, e.g., cache hit/miss rate, etc. In the next step, we'll also try to gain more accurate data, including clock cycle counts in each functions and perform evaluation and comparison between original code and optimized code.

3 Meeting Your Milestone

We are little behind meeting the milestone we had set out in our proposal. While we do have most of the SLP algorithm and code generation working and are able to verify that it is producing the correct output, we do not yet have the infrastructure setup to gather performance data. Specifically, we were hoping to have a simulator such as Gem5 up and running by now such that we would be able to actually profile the compiled code with and without SLP to determine what are the actual performance improvements. One of the portions of the algorithm described in [1] that we have omitted is the cost function to determine whether it is worthwhile to perform the packing/unpacking of elements into vectors. We were hoping to gain some insight on the performance impacts from first simulating vectorized code before spending time designing and tuning the cost function in our implementation.

4 Surprises

We have not encountered any significant surprises that have affected our progress. One surprise however that is worth mentioning is that given the simplicity of the programs currently in our test suite, LLVM is able to optimize with anything -O2 or higher such that there will not be any opportunity for SLP to perform optimizations. To work around this, we are first compiling with -O1, then running our SLP pass, and then performing further optimizations such that the resulting LLVM bytecode will be comparable to having compiled the program initially with -O2.

5 Revised Schedule

- Write more test cases to identify corner cases and bugs in SLP (Ashwin)
- Create benchmark suite (Ashwin)
- Perform code generation optimizations to not perform redundant packing/unpacking (Ashwin)
- Further implement the algorithm details, e.g., functions to estimate the benefits of performing SLP (Li)
- Port over Gem5 to gather performance data (Li)

6 Resources Needed

In terms of implementation, we developed SLP algorithms in LLVM and we are able to utilize the API and IR provided by LLVM to generate vectorized code. In terms of performance evaluation, we execute benchmark programs and collect related data in gem5, mainly because we need a microarchitecture simulator to evaluate the cost of vectorized program.
In the meantime, we tried to acquire the original benchmark programs in [1]. Two program sets were used: multimedia and scientific programs, but unfortunately both of them are not available. The latest SPEC benchmark suites require licenses. Thus, we will create our own benchmark

suites, but we would also do further searching on open source CPU benchmark programs, especially focusing on SIMD.

7 References

[1] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. PLDI, 2000.

[2] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. PLDI, 2012.

[3] Auto-vectorization in LLVM. https://llvm.org/docs/Vectorizers.html