Superword Level Parallelism in LLVM

https://15745-SLP-Project.github.io

Ashwin Adulla (aadulla@andrew.cmu.edu), Li Shi (lishi@andrew.cmu.edu)

1 Introduction

1.1 Problem

Nowadays, many computer architectures have added support for multimedia extensions in the form of SIMD instructions. SIMD instructions allow the same operation to be performed on multiple pieces of data simultaneously, thus exploiting the inherent parallelism present in the application. Compilers employ various auto-vectorization strategies to recognize this parallelism and utilize the SIMD instructions. However, many of these strategies are very limited in their scope and sometimes require explicit indication by the user of what to parallelize through the use of library functions or annotated pragmas. As a result, the user often holds the burden of writing and formatting their code in such a way that the compiler can easily determine the inherent parallelism.

1.2 Approach

Superword level parallelism (SLP) is one approach to expanding the compiler's auto-vectorization scope. SLP packs independent but similar data and operations into "superwords" that can then be executed using SIMD instructions. It does not require any explicit hints from the user, and it can recognize the dependency between SIMD operations through following def-use chains.

1.3 Related Work

SLP was first introduced by [1] where an auto-vectorization algorithm was made to identify isomorphic (meaning they perform the same operation i.e. add, subtract, etc.) statements and merge them together into packs. These packs corresponded to vectors of data that could be directly lowered into SIMD instructions. The algorithm begins by identifying adjacent array accesses as the basis for these packs and then extends these packs by following their use-def and def-use chains. [1] tested out the SLP algorithm on various tests from the SPEC95fp benchmark suite, and found speedups ranging from 1.24 to 6.70.

SLP was further optimized in [2] where the authors came up with a modified algorithm that reduced the overhead with packing/unpacking. If the operands in superwords are already in arrays, there is little overhead required to load portions of an array into vector registers. If however, these operands are in other scalar registers, are constants, or are in other parts of memory, significant effort needs to be spent to load each of these operands into a vector register, and then store them back into the original locations. [2] specifically addresses this issue by extending the scope of the SLP algorithm to look at an entire basic block. By considering a more global view that spans the whole basic block, the algorithm proposed in [2] can determine what operands will have to be packed/unpacked, and thus optimize the data layout of these operands in memory so as to reduce individual loads/stores.

Currently, SLP vectorization has been widely used in modern compilers, e.g., LLVM [3], which will automatically exploit potential instruction-level parallelism in the program and pack operations into SIMD instructions.

1.4 Contributions

In our project, we drew inspiration from [1], one of the first works on an SLP compiler optimization, and implemented a similar SLP pass in LLVM. Our pass works by first focusing on array accesses, and then following the def-use chains to construct the superwords. We then replace these superwords with vectorized packs of operations that utilize the LLVM vector IR type. During code-generation, the LLVM vector IR type is automatically lowered into SIMD instructions.

With our SLP pass, we saw performance improvements across a variety of benchmarks from simple linear algebra routines to more complicated programs like matrix multiplication. We compiled these benchmarks using the ARMv8-A ISA with the NEON SIMD extension, and then ran the executables through the gem5 simulator to gather performance statistics.

2 Algorithm

2.1 Overview

Our implementation of SLP in LLVM borrows heavily from the algorithm discussed in [1]. We begin by first performing loop unrolling to open up the opportunity for adjacent memory accesses to existing within a single basic block. These adjacent memory accesses form the basis for a “pack”, where a “pack” is a collection of independent, isomorphic statements. From this initial pack, we follow the use-def and def-use chains to form a chain of packs known as a “packset”. Once the packset can no longer be extended, it is then scheduled to respect data dependencies and is finally lowered into vectorized LLVM IR that eventually emits SIMD instructions.

2.2 SLP Identification

```
SLP_extract: BasicBlock B → BasicBlock
    PackSet P ← EmptySet
    P ← find_adj_refs(B, P)
    P ← extend_packlist(B, P)
    P ← combine_packs(P)
    return schedule(B, [], P)
```

Figure 1. Algorithm to identify SLP.

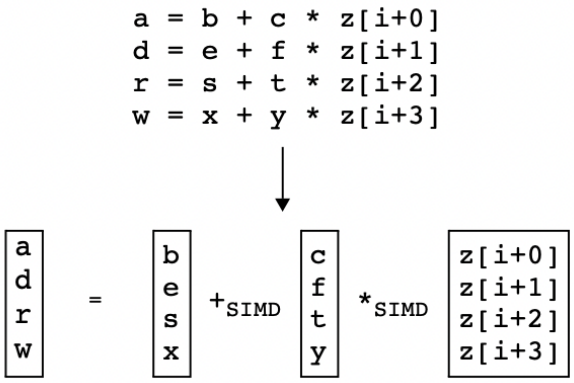

Figure 2. Example of identifying SLP packs.

The overall algorithm to perform SLP identification is shown in Figure 1, while the details can be found in [1]. We will also use an example (Figure 2) from [1] to explain the process to identify potential SLP packs. In Figure 2, the algorithm would first invoke `find_adj_refs` function and identify the consecutive array references to array **`z`**. Two adjacent memory reference instructions will be packed into a pair. In `extend_packlist`, following the def-use chain, it would then recognize the independent, isomorphic multiplication operations of **`c * z[i+0]`** & **`f * z[i+1]`**, **`f * z[i+1]`** & **`t * z[i+2]`**, and **`t * z[i+2]`** & **`y * z[i+3]`**, which would form more pairs. We keep iterating along the def-use chain to find more dependent instructions until there is no change. Next, in `combine_packs`, isomorphic pairs

would then be combined together into a pack, e.g., the aforementioned pairs will be combined into a pack (`c * z[i+0]`, `f * z[i+1]`, `t * z[i+2]`, `y * z[i+3]`), and packs will be packed in a pack set. Finally, the algorithm would then apply the list scheduling algorithm to find a correct sequence of packs according to the data dependencies between packs.

2.3 Code Generation

Once the packs have been identified and scheduled, they can then be lowered to SIMD instructions, consisting of 2 phases: (1) User Code to LLVM IR, (2) LLVM IR to ARMv8-A.

2.3.1 Phase (1): User Code to LLVM IR

In Phase (1), packs are converted to LLVM vectors that are either a) explicitly packed/unpacked if they need to gather/scatter their operands or are b) implicitly packed/unpacked through bitcasts if their operands are adjacent array references.

Figure 3 shows the lowering from user code to LLVM IR. We can see the explicit packing of operands a, b, c, d into a vector `<a, b, c, d>`, and then the explicit unpacking of `<e, f, g, h>` into operands e, f, g, h. We can also see the implicit packing of operands `A[i], A[i+1], A[i+2], A[i+3]` into vector `A[i:i+3]` through a bitcast.

Figure 4 shows another example of the lowering from user code to LLVM IR. Here, all operands are found in adjacent array references, so no explicit packing/unpacking is needed. Array pointers to A and B are simply bitcast into vectors, the addition operation is performed, and then the result is stored in C.

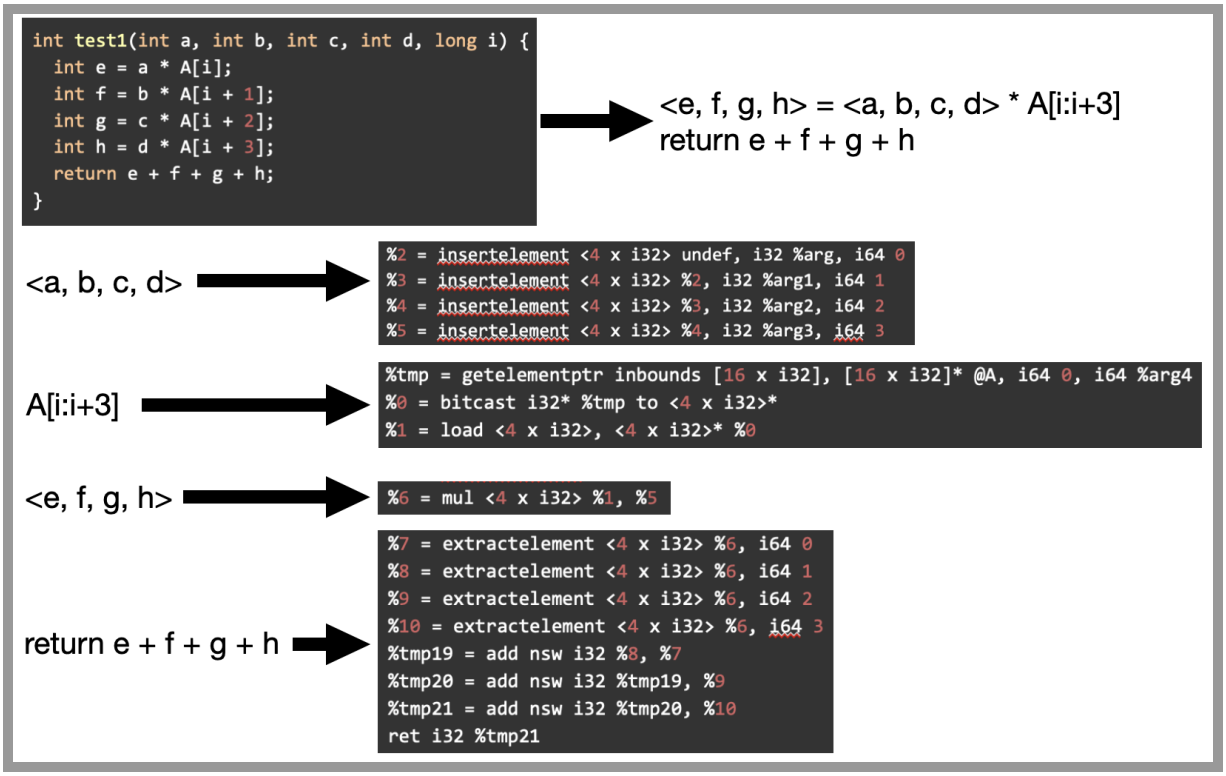

Figure 3. Example 1 of lowering from user code to LLVM IR.

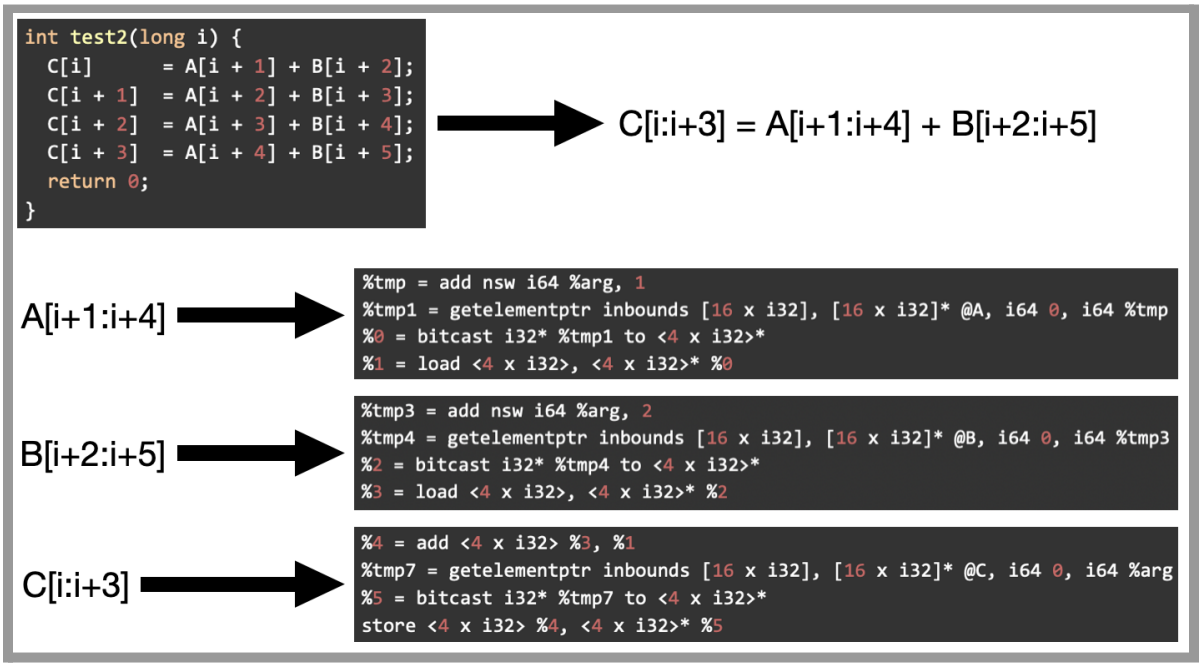

Figure 4. Example 2 of lowering from user code to LLVM IR.

2.3.2 Phase (2): LLVM IR to ARMv8-A

As we target the ARMv8-A backend, the LLVM IR vectors are lowered into NEON syntax SIMD instructions.

Figure 5 shows the emitted assembly code for the LLVM IR code shown in Figure 3. To perform the vector multiplication, **`A[i:i+3]`** is loaded into the **`q1`** SIMD register, and operands **`a, b, c, d`** are packed into the **`v0`** SIMD register. The vector multiplication is then performed (**`mul v0.4s, v1.4s, v0.4s`**), and the result is then unpacked into scalar registers **`s2, s3, s4, s5`** representing **`e, f, g, h`**, respectively.

Figure 6 shows the emitted assembly code for the LLVM IR code shown in Figure 4. First **`A[i+1:i+4]`** and **`B[i+2:i+5]`** are loaded into **`q0`** and **`q1`** SIMD registers, respectively. The vector addition is then performed (**`add v0.4s, v1.4s, v0.4s`**), and the resulting vector is stored directly into the **`C[i:i+3]`** array.

3 Experimental Setup

3.1 Environment

We develop and test our SLP algorithms in LLVM version 10.0.0 in Ubuntu 20.04.3. We compile the benchmark programs with clang to generate LLVM IR and perform SLP on IR. After that, llc and aarch64-linux-gnu-gcc will be invoked to emit machine code and executables, targeting aarch64 platform with gcc toolchain version 9.4.0. The programs are executed in Gem5 simulator (commit: 141cc37c2d4b93959d4c249b8f7e6a8b2ef75338), with CPU type `O3_ARM_v7a_3`, default L1 and L2 cache, to acquire the runtime performance data.

```
test1:                                  // @test1
      adrp   x8, A
      add    x8, x8, :lo12:A
      mov    w9, #4
      mov    w10, w9
      mul    x10, x10, x4
      ldr    q1, [x8, x10]              // q1 = A[i:i+3]
      mov    v0.s[0], w0                // v0.s[0] = a
      mov    v0.s[1], w1                // v0.s[1] = b
      mov    v0.s[2], w2                // v0.s[2] = c
      mov    v0.s[3], w3                // v0.s[3] = d
      mul    v0.4s, v1.4s, v0.4s        // v0 = v1 * v0 = A[i:i+3] * <a, b, c, d>
      mov    s2, v0.s[0]                // s2 = v1[0] = e
      mov    s3, v0.s[1]                // s3 = v1[1] = f
      mov    s4, v0.s[2]                // s4 = v1[2] = g
      mov    s5, v0.s[3]                // s5 = v1[3] = h
      fmov   w9, s3
      fmov   w11, s2
      add    w9, w9, w11
      fmov   w11, s4
      add    w9, w9, w11
      fmov   w11, s5
      add    w0, w9, w11
      ret
```

Figure 5. Emitted assembly code from IR in Figure 3.

```
test2:
      adrp   x8, A
      add    x8, x8, :lo12:A
      adrp   x9, B
      add    x9, x9, :lo12:B
      adrp   x10, C
      add    x10, x10, :lo12:C
      mov    w11, wzr
      add    x12, x0, #1
      mov    w13, #4
      mov    w14, w13
      mul    x12, x14, x12
      add    x15, x0, #2
      mul    x15, x14, x15
      mul    x14, x14, x0
      ldr    q0, [x8, x12]              // q0 = A[i+1:i+4]
      ldr    q1, [x9, x15]              // q1 = B[i+2:i+5]
      add    v0.4s, v1.4s, v0.4s        // v0 = v1 + v0 = A[i+1:i+4] + B[i+2:i+5]
      str    q0, [x10, x14]             // C[i:i+3] = q0
      mov    w0, w11
      ret
```

Figure 6. Emitted assembly code from IR in Figure 4.

3.2 Benchmark Programs

We measure the effectiveness of SLP algorithms on 5 benchmark programs:

1. memcpy: Simple data movement in memory.
2. axpy: Calculate `a * X + Y`, where `a` is a scalar, `X` and `Y` are vectors.
3. dotprod: Calculate the dot product of two vectors.
4. mmm: Matrix multiplication.
5. arithmetic: More complicated floating-point arithmetic operations.

We obtain the experimental results in terms of runtime and code size with 4 compilation setups: O1, O1 + unroll, O1 + unroll + SLP, and O2. Runtime data is obtained by the `clock()` system call and refers to the execution time of the computation kernel function without data initialization and correctness checking. Code size refers to the instruction counts of only the computation kernel function, as ARM is a RISC ISA and all instructions are encoded in 4 bytes. In some programs (mmm and memcpy), auto loop unrolling will lead to unexpected results, and we need to unroll the loop (e.g., unroll only the inner loop in mmm) manually by inserting clang pragma.

4 Experimental Evaluation

We measure the performance and code size of the 4 compilation setups. Table 1 and Figure 7 show the runtime of the computation kernel function in the 5 benchmark programs. The results match our expectation that by exploiting SLP vectorization in the program, it will utilize the SIMD computing units in the CPU and accelerate the computation, compared with simple O1 and O1 with loop unrolling. After SLP vectorization, the programs run 1.31*x* faster than O1 and 1.17*x* faster than O1 with loop unrolling in geometric mean. As O2 will perform more aggressive

optimizations besides vectorization, only performing SLP vectorization results in a relatively lower performance than O2.

	O1	O1+unroll	O1+unroll+SLP	O2
memcpy	14833	14031	13768	7085
axpy	5011	4056	2628	1995
dotprod	5675	4443	3582	2969
mmm	40158	37575	37212	33945
arithmetic	20073	19752	17934	10102

Table 1. Execution time of 5 benchmark programs in microseconds.

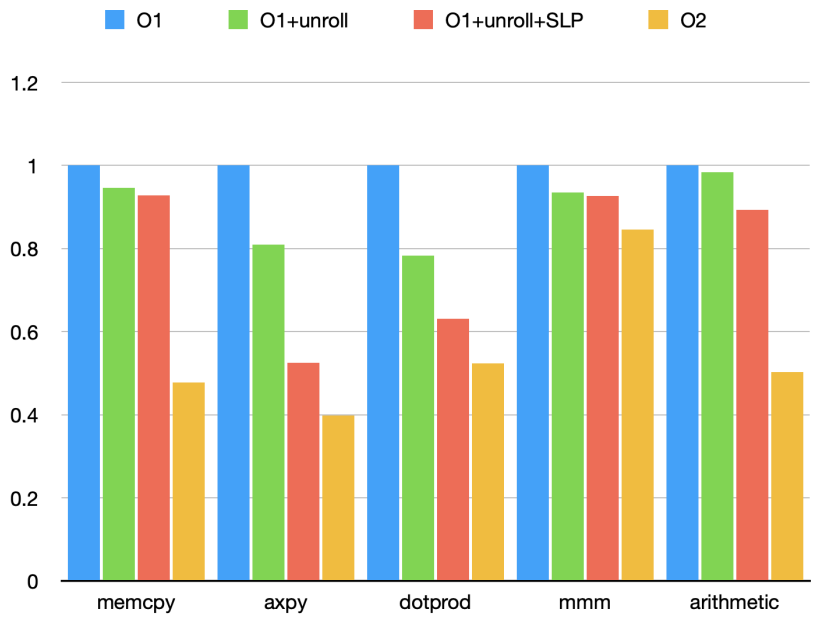

Figure 7. Execution time in different compilation setups, O1 is 1*x*.

Table 2 and Figure 8 show the code size of the computation kernel function in the 5 benchmark programs. While both direct loop unrolling and SLP vectorization aim to utilize the instruction-level parallelism of loops, SLP vectorized programs usually have a lower code size as

it reduces instruction counts by packing isomorphic operations into vectorization instructions. In our experiment results, SLP results in lower instruction counts in 4 of the 5 short programs, even though it requires further variable packing or unpacking. The code size of the arithmetic program after SLP is the largest mainly because there is frequent register spilling and reloading, and a lot of load instructions are added. In summary, SLP vectorization leads to a 94% increase in code size compared to O1 and a 4% decrease in code size compared to O1 with direct loop unrolling.

	O1	**O1+unroll**	**O1+unroll+SLP**	**O2**
memcpy	20	64	59	71
axpy	25	43	43	90
dotprod	42	70	59	90
mmm	82	110	108	123
arithmetic	50	131	146	50

Table 2. Instruction counts of computation kernels in 5 benchmark programs.

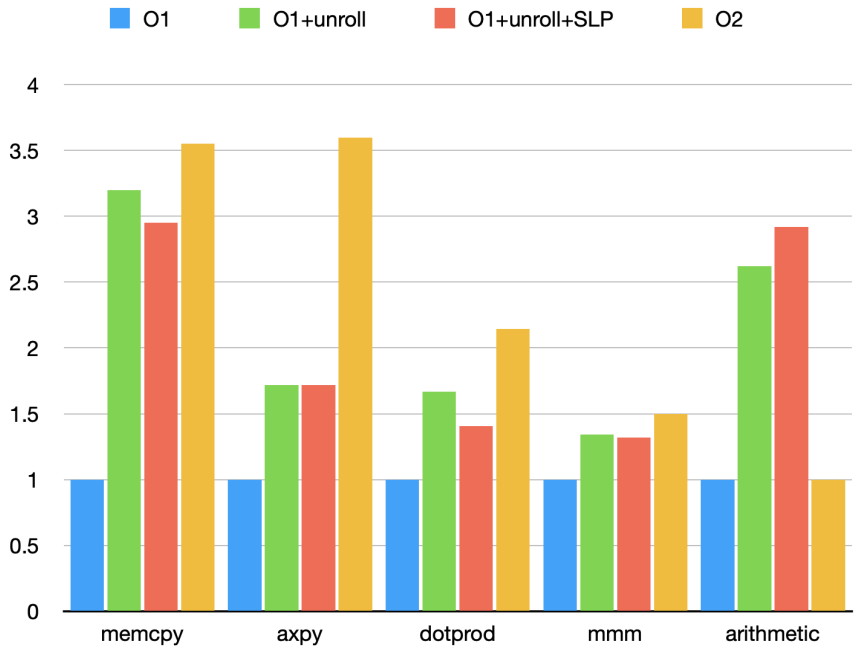

Figure 8. Instruction counts of computation kernels in different compilation setups, O1 is 1*x*.

5 Conclusions, Lessons Learned & Future Work

In this project, we implement the SLP algorithms from [1] in LLVM compiler infrastructure and measure the effectiveness on multiple benchmark programs. In particular, our algorithms are able to automatically identify superword level parallelism within a basic block and pack isomorphic and independent operations into SIMD instructions to improve the performance. Our experiment results show that basic SLP algorithms will improve the runtime performance by 1.31 times compared to O1 and 1.17 times compared to O1 with loop unrolling with a smaller program size compared with direct loop unrolling.

During the project, we learned the basic techniques to exploit instruction level parallelism in the programs and the effectiveness and necessity of utilizing SIMD units in modern processors. We also got more familiar with the development workflow on a modern compiler, including using the APIs and optimizing programs on intermediate representations.

Due to time limitations, our algorithms are able to run on simple vectors and matrices without complicated transformations on array indices. The algorithms can be further improved by handling more complex cases, introducing a cost model to evaluate the improvement of SLP vectorization and explore the design space, etc.

6 Distribution of Total Credit

Ashwin Adulla: 50%

Li Shi: 50%

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.