Data Processing on Modern Hardware
Parabix with asmJIT – Final Report

1 Problem Statement

The traditional approach of building a Finite State Automata for Regular Expression matching runs asymptotically in linear time, but is only able to process a single Byte at a time and is hard to parallelize. Parallel Bitstreams is a technique for regular expression matching that transforms regular expressions in a sequence of bitwise operations that can be performed on the entire input at the same time (assuming arbitrarily long streams). icGrep applies code generation to compile the sequence of Bitwise operations to efficient code, removing the overhead of interpreting those operations.

However, the chosen code compilation framework (LLVM) has too high compilation times which is specially problematic for small inputs, where most of the overall execution is spent in compiling the code. This motivates the search for better techniques that can speed up compilation time and close the gap between JIT compiled code and interpreted code.

2 Applied Techniques for Optimization

Coat is a code generation framework that supports asmJIt and LLVM backends. AsmJIT directly emits x86 Assembly code with focus on low latency. It performs no optimization passes and has a simple built-in linear-time register allocator. This results in compilation times which are usually in the range of 100µs. Since it has no optimization pass, the user is responsible for writting efficient code.

The proposed approach is to replace LLVM with asmJIT as code generation framework to decrease compilation time, the basic assumption is that asmJIT is potentially slower, but the smaller compilation time compensates for it.

3 Evaluation

Due to time constrains, we avoid SIMD instructions as they complicate code generation and instead perform the experiments on a scalar version of the algorithm that uses a vector size of 64. We perform 2 experiments. We compare 4 regular expression implementations. The first is a DFA-based approach, the second a handwritten C++ program that uses parallel bitstreams, the third is parallel bitstreams with asmJIT and the last is parallel bitstreams with LLVM. We are mostly interested in the performance numbers of the last 2 approaches, the first two serve as baselines. All performance numbers are independent of the content of the input array. To avoid using SIMD in generated code, we assumed that the input array is already transposed and measure the cost for transposing the input separately.

The experiments are performed on a single thread of an intel Xeon machine with 12MB L3 cache and up to 4GHz frequency.

	Cycles	Inst	L1-misses	LLC-misses	branch-misses	IPC	Time
Overall	725483457	2265136327	14174610	13006936	1118	3.12	3.82ms
Cost/Byte	2.7	8.43	0.052	0.048	0,000004	-	

Table 1: We report for 256MB input the total number of cycles, instructions, cache and branch misses and average IPC. The we normalize it to a Per-Byte Cost.

	Cycles	Inst.	L1-misses	LLC-misses	branch-misses	IPC	Time 256MB
DFA	10.1	8	0.03	0.05	0.0	0.79	630ms
Parabix C++	0.49	0.92	0.03	0.02	0.0	1.89	32ms

Table 2: Although we varied the input size, we noticed that the Cost per Byte was kept constant for both approaches, thus we compare only the cost per Byte.

3.1 Bit transposition time

This experiment measures the number of Bytes per cycle needed to transpose some input by transposing a large array of 256MB. The transposing algorithm uses SSE instructions, thus processing only 16 Bytes at a time. The results are summarized on table 1

The transposing takes an average of 2.7 Cycles per Byte. There are nearly no cache and branch misses and the IPC count is high meaning we can fully utilize the CPU. It is expected that with increasing vector size, the Per-Byte costs will decrease.

3.2 DFA vs parallel bitstreams

This experiment compares the performance of a DFA-based approach with a parallel bitstreams approach. We fix the same previous regular expression 130-[0:9]*-140 (starting with 130, matching range 0 to 9 multiple times, then matching 140) and vary input size from 1024KB to 256MB by constantly multiplying by 4. We verify that there is no change in the Cost/Byte for any of the metrics. Table 2 reports the numbers for both approaches. We see that both approaches have no cache and branch misses. We also see that Parabix is 20 times faster than the DFA approach. This is super linear with the number of Bytes per Vector (8) and can be explained by the also higher IPC, since that in the DFA method the next operation is dependent on the result of the current, the CPU can't take advantage of out-of-order execution.

3.3 C++ vs asmJIT vs LLVM runtimes

This experiment compares the runtime of each implementation, disconsidering the compilation time. We check how much LLVM performance matches that of a hand-written C++ program and how much asmJIT's performance degrades when compared to LLVM. Based on the results from the previous experiments, we only evaluate for 256MB inputs. The results are displayed in 3. We see that LLVM with optimization level 0 performs very poorly. Furthermore, LLVM has a higher count of instructions per cycle, this is most likely due to computing many times the same expression for different matches, which are indenpendent actions and can benefit from out-of-order execution. Overall we see that asmJIT is about 50% slower than LLVM with optimization level greater than 0.

3.4 LLVM vs asmJIT compile times

This experiment compares the compilation times for asmJIT and different optimization levels for LLVM for our simple RegEx. The results can be seen on table 4. We notice that compilation with asmJIT is

	Cycles	Inst.	L1-misses	LLC-misses	branch-misses	IPC	Time 256 MB
C++	0.49	0.92	0.03	0.02	0.0	1.89	32ms
AsmJIT	0.82	2.53	0.04	0.02	0.0	3.08	53.6ms
LLVM (0)	2.16	3.42	0.06	0.02	0.0	1.59	136.9ms
LLVM (1)	0.47	0.83	0.03	0.02	0.0	1.78	29.5ms
LLVM (2)	0.47	0.83	0.03	0.02	0.0	1.78	29.5ms
LLVM (3)	0.46	0.83	0.03	0.02	0.0	1.79	29.5ms

Table 3: Cost per Cycle for each method. LLVM achieves better performance than hand-written C++

	Compilation Time (µs)
AsmJIT	60
LLVM (0)	7511
LLVM (1)	8751
LLVM (2)	8886
LLVM (3)	9241

Table 4: Compilation time is reported in microsencods. asmJIT is orders of magnitude faster than LLVM

more than 100 times faster than LLVM even with -O0.

4 Learnings & Conclusion

My expectations for the project we 2:

- First, verify the speed up of Parabix over DFA methods. I had the suspicion it would be super-linear due to better IPC count
- Second, verify that asmJIT might be a more suitable framework for compilation due to the smaller compilation time. There is still some significant difference between LLVM and asmJIT so that one can't say it for sure. However, I believe that further methods to optimize the generated assembly code and more careful code generation could reduce the total runtime to about 40ms.

All in all, the results fully match my expectations. Throughout the project I learned a lot about code generation, specially asmJIT as I tried using it before switching to coat. I also spotted a few problems with coat. Even though I had some previous experience with coat and code generation because of my thesis, I think I also improved my skills to write code for JIT compilation.

5 Questions:

I would like to participate on the ACM Student Competition

- **Is this topic sufficient for the competition?** Somehow it seems like it's nothing really new, just the same old optimizations techniques, so I'm a bit unsure.
- **If the topic is sufficient, what would be the next steps**

6 Future Work

The most intuitive direction to expand the project is to use SIMD instructions which should give almost linear speed up on execution time, apart from some extra overhead to maintain the carry queues and

some decrease in the CPU frequency. Assuming that AVX-512 brings a linear speed up, this would imply for 256MB an execution time of $30/8 = 3.75$ms for LLVM and $55/8 = 6.6$ms for AsmJIT. If compile times stay constant (60 µs for AsmJIT and 8ms for LLVM), the code for LLVM will spend almost one third of its execution on compilation and AsmJIT will be done with the workload before LLVM could even finish compiling.

There is still the problem of generating the bitwise equations from the regular expression. The hardest equations are those for single characters and ranges (ex: [a-zA-Z]) which will either generate inefficient C++ equations with too many operations, or incur some cost on optimizing the Boolean expression. On AVX-512 one can match single characters or character classes before transposing the input, by directly using the `_mm512_cmp_epi8_mask` function, avoiding completely the burden of generating those equations. To match a character `a` we just compare `vector==a` and to match a class `[b-c]` we compare `b <= vector <= c`.

On the other hand, keeping bitwise operations could have the advantage of reusing intermediate computations. Listing 1 shows the code for matching the numbers 0 and 1. One can see that A5 and A3 are equal for both operations, so that the intermediate results can be shared among both functions, saving almost 50% of the operations. In coat this simplification is (likely) already performed by LLVM. For AsmJIT one could implement a map containing all intermediate results.

Due to problems with Coat, it wasn't possible to analyze the generated Assembly for the AsmJIT backend. Since AsmJIT has no optimization passes, one must write code very carefully to make sure that efficient instructions are emitted. For example we found that loading the basis streams has a significant performance impact depending on how it's written (see listing 2).

Nevertheless, even if the above optimizations bring AsmJIT's performance more close to LLVM's, there would still be some differences which get expensive for large workloads. To cope with it one can use a hybrid-execution approach, where vectors are fetched in morsels and the execution function is switched from AsmJIT to LLVM when the compilation is done. This should be nearly as fast as hand-written C++ code.

As a last comment, it's important to notice that with growing regular expression complexity, the number of required operations increase and therefore also the total execution time. However, parallel bitstreams don't suffer from the problem of exploding state size, common on larges Regexes. So there is no clear winner for this case and further analysis would be necessary.

```
template<typename Vector>
inline Vector match_0(..) {
  auto A1 = B7 | B6;
  auto A2 = B5 | B4;
  auto A3 = B3 | B2;
  auto A4 = B1 | B0;

  auto A5 = A1 | A2;
  auto A6 = A3 | A4;

  Vector ret = ~(A5 | A6);
  return ret;
}
```

```
template<typename Vector>
inline Vector match_1(...) {
  Vector B0_copy = B0;
  auto A1 = B7 | B6;
  auto A2 = B5 | B4;
  auto A3 = B3 | B2;
  auto A4 = B1 | ~B0_copy;

  auto A5 = A1 | A2;
  auto A6 = A3 | A4;

  Vector ret = ~(A5 | A6);
  return ret;
}
```

Listing 1: Code for matching 0 and 1 shares many common intermediate computations.

```
coat::loop_while(fn, count < size, [&]{
  auto B0 = *(B0_ptr + count);
  auto B1 = *(B1_ptr + count);
  auto B2 = *(B2_ptr + count);
  auto B3 = *(B3_ptr + count);
  auto B4 = *(B4_ptr + count);
  auto B5 = *(B5_ptr + count);
  auto B6 = *(B6_ptr + count);
  auto B7 = *(B7_ptr + count);

  // Code ...

  ++output;
  count += 1;
});
```

```
coat::loop_while(fn, count < size, [&]{
  auto B0 = *B0_ptr;   auto B1 = *B1_ptr;
  auto B2 = *B2_ptr;   auto B3 = *B3_ptr;
  auto B4 = *B4_ptr;   auto B5 = *B5_ptr;
  auto B6 = *B6_ptr;   auto B7 = *B7_ptr;

  // Code ...

  ++output;
  count += 1;
  ++B0_ptr;   ++B1_ptr;
  ++B2_ptr;   ++B3_ptr;
  ++B4_ptr;   ++B5_ptr;
  ++B6_ptr;   ++B7_ptr;
});
```

Listing 2: For 256MB input for AsmJIT, the loop on the left runs in 50ms and the loop on the right in 60ms. Overall, the inefficient loop resulted in a 20% slower code