Autovectorization in Graal
Graal Workshop @ CGO 2020

Our journey implementing Autovectorization in
Graal CE on the #TwitterVMTeam

David Degazio (@elucentdev) University of Michigan, USA
Niklas Vangerow (@nvgrw) Imperial College London, UK

February 22, San Diego, USA

Talk Outline

1. Autovectorization recap
2. Implementation
3. Challenges
4. Examples and Benchmarks
5. Conclusion

Autovectorization Recap

Autovectorization Recap

- Modern processors feature registers that hold vectors of values and have vector arithmetic operations.
- Analyze a program and find the vectors.
- Generate vector instructions.
- Usually found in loops.

Autovectorization Recap: An Example

```
void addMyArrays(int[] a, int[] b, int[] c) {
 for (int i = 0; i < 100; i++) {
   c[i] = a[i] + b[i];
 }
}
```

Autovectorization Recap: An Example

```
void addMyArrays(int[] a, int[] b, int[] c) {
 for (int i = 0; i < 100; i+=4) {
   c[i:i+4] = a[i:i+4] + b[i:i+4];
 }
}
```

Autovectorization Recap: An Example

```
...
add [rax], 1
add [rax + 4], 2
add [rax + 8], 3
add [rax + 12], 4
```

```
...
movdqu xmm1, 0xf12937b0
movdqu xmm0, [rax]
paddd  xmm0, xmm1
movdqu [rax], xmm0
```

Scalar → **Vector**

Implementation

The Algorithm

Exploiting Superword Level Parallelism
with Multimedia Instruction Sets

Samuel Larsen and Saman Amarasinghe
MIT Laboratory for Computer Science
Cambridge, MA 02139
{slarsen,saman}@lcs.mit.edu

Abstract

Increasing focus on multimedia applications has prompted the addition of multimedia extensions to most existing general purpose microprocessors. This added functionality comes primarily with the addition of short SIMD instructions. Unfortunately, access to these instructions is limited to in-line assembly and library calls. Generally, it has been assumed that vector compilers provide the most promising means of exploiting multimedia instructions. Although vectorization technology is well understood, it is inherently complex and fragile. In addition, it is incapable of locating SIMD-style parallelism within a basic block.

In this paper we introduce the concept of *Superword Level Parallelism (SLP)*, a novel way of viewing parallelism in multimedia and scientific applications. We believe SLP is fundamentally different from the loop level parallelism exploited by traditional vector processing, and therefore demands a new method of extracting it. We have developed a simple and robust compiler for detecting SLP that targets basic blocks rather than loop nests. As with techniques designed to extract ILP, ours is able to exploit parallelism both across loop iterations and within basic blocks. The result is an algorithm that provides excellent performance in several application domains. In our experiments, dynamic instruction counts were reduced by 46%. Speedups ranged from 1.24 to 6.70.

1 Introduction

The recent shift toward computation-intensive multimedia workloads has resulted in a variety of new multimedia extensions to current microprocessors [6, 10, 16, 18, 20]. Many new designs are targeted specifically at the multimedia domain [3, 7, 11]. This trend is likely to continue as it has been projected that multimedia processing will soon become the main focus of microprocessor design [8].

While different processors vary in the type and number of multimedia instructions offered, at the core of each is a set of short SIMD or superword operations. These instructions operate concurrently on data that are packed in a single register or memory location. In the past, such systems could accommodate only small data types of 8 or 16 bits, making them suitable for a limited set of applications. With the emergence of 128-bit superwords, new architectures are capable of performing four 32-bit operations with a single instruction. By adding floating point support as well, these extensions can now be used to perform more general purpose computation.

It is not surprising that SIMD execution units have appeared in desktop microprocessors. Their simple control, replicated functional units, and absence of heavily-ported register files make them inherently simple and extremely amenable to scaling. As the number of available transistors increases with advances in semiconductor technology, datapaths are likely to grow even larger.

Today, use of multimedia extensions is difficult since application writers are largely restricted to using in-line assembly routines or specialized library calls. The problem is exacerbated by inconsistencies among different instruction sets. One solution to this inconvenience is to employ vectorization techniques that have been used to parallelize scientific code for vector machines [5, 14, 15]. Since a number of multimedia applications are vectorizable, this approach promises good results. However, many important multimedia applications are difficult to vectorize. Complicated loop transformation techniques such as loop fission and scalar expansion are required to parallelize loops that are only partially vectorizable [2, 4, 17]. Consequently, no commercial compiler currently implements this functionality. This paper presents a method for extracting SIMD parallelism beyond vectorizable loops.

We believe that short SIMD operations are well suited to exploit a fundamentally different type of parallelism than the vector parallelism associated with traditional vector and SIMD supercomputers. We denote this parallelism *Superword Level Parallelism (SLP)* since it comes in the form of superwords containing packed data. Vector supercomputers require large amounts of parallelism in order to achieve speedups, whereas SLP can be profitable when parallelism is scarce. From this perspective, we have developed a general algorithm for detecting SLP that targets basic blocks rather than loop nests.

In some respects, superword level parallelism is a restricted form of ILP. ILP techniques have been very successful in the general purpose computing arena, partly because of their ability to find parallelism within basic blocks. In the same way that loop unrolling translates loop level parallelism into ILP, vector parallelism can be transformed into SLP. This realization allows for the parallelization of vector-

PLDI 2000, Vancouver, British Columbia, Canada.

Available from:
http://groups.csail.mit.edu/cag/slp/SLP-PLDI-2000.pdf

The Algorithm

For each basic block:

1. Pair isomorphic instructions containing adjacent memory references.
2. Extend set of pairs by finding instructions using def-use chains and def-use chains of step 1 pairs.
3. Combine the pairs into a set of adjacent packs, until the set no longer changes.
4. Schedule packs for execution.

c	=	a	+	b
d	=	e	+	f

✔️ Isomorphic

✔️ Independent

c	=	a	+	b
d	=	c	+	f

✔️ Isomorphic

❌ Independent

The Algorithm

For each basic block:

1. **Pair isomorphic instructions containing adjacent memory references.**
2. Extend set of pairs by finding instructions using def-use chains and def-use chains of step 1 pairs.
3. Combine the pairs into a set of adjacent packs, until the set no longer changes.
4. Schedule packs for execution.

Program

```
(1) b = a[i+0]
(2) c = 5
(3) d = b + c
(4) e = a[i+1]
(5) f = 6
(6) g = e + f
(7) h = a[i+2]
(8) j = 7
(9) k = h + j
```

Packed Set

```
{ (1, 4),
  (4, 7),
}
```

The Algorithm

For each basic block:

1. Pair isomorphic instructions containing adjacent memory references.
2. **Extend set of pairs by finding instructions using def-use chains and def-use chains of step 1 pairs.**
3. Combine the pairs into a set of adjacent packs, until the set no longer changes.
4. Schedule packs for execution.

Program

```
(1) b = a[i+0]
(2) c = 5
(3) d = b + c
(4) e = a[i+1]
(5) f = 6
(6) g = e + f
(7) h = a[i+2]
(8) j = 7
(9) k = h + j
```

Packed Set

```
{ (1, 4),
  (4, 7),
  (3, 6), // u-d
  (6, 9), // u-d
  (2, 5), // d-u
  (5, 8), // d-u
}
```

E.g. (1, 4) defines b and e, which are used by (3, 6).
(3, 6) uses c and f, which are defined by (2, 5).

The Algorithm

For each basic block:

1. Pair isomorphic instructions containing adjacent memory references.
2. Extend set of pairs by finding instructions using def-use chains and def-use chains of step 1 pairs.
3. **Combine the pairs into a set of adjacent packs, until the set no longer changes.**
4. Schedule packs for execution.

Program

```
(1) b = a[i+0]
(2) c = 5
(3) d = b + c
(4) e = a[i+1]
(5) f = 6
(6) g = e + f
(7) h = a[i+2]
(8) j = 7
(9) k = h + j
```

Packed Set

```
{ (1, 4, 7),
  (3, 6, 9),
  (2, 5, 8)
}
```

The Algorithm

For each basic block:

1. Pair isomorphic instructions containing adjacent memory references.
2. Extend set of pairs by finding instructions using def-use chains and def-use chains of step 1 pairs.
3. Combine the pairs into a set of adjacent packs, until the set no longer changes.
4. **Schedule packs for execution.**

Program

```
(1) b = a[i+0]
(2) c = 5
(3) d = b + c
(4) e = a[i+1]
(5) f = 6
(6) g = e + f
(7) h = a[i+2]
(8) j = 7
(9) k = h + j
```

Vectorized Program

```
d       a[i+0]       5

g   =   a[i+1]   +   6

k       a[i+2]       7
```

Algorithm Alternatives

Why SLP?

- A starting point.
- Tried and tested, implemented in C2 and LLVM to different extents.
- Later literature builds on top of SLP.
  - Bottom-up SLP
  - VW-SLP
  - etc

The Algorithm: Implementation

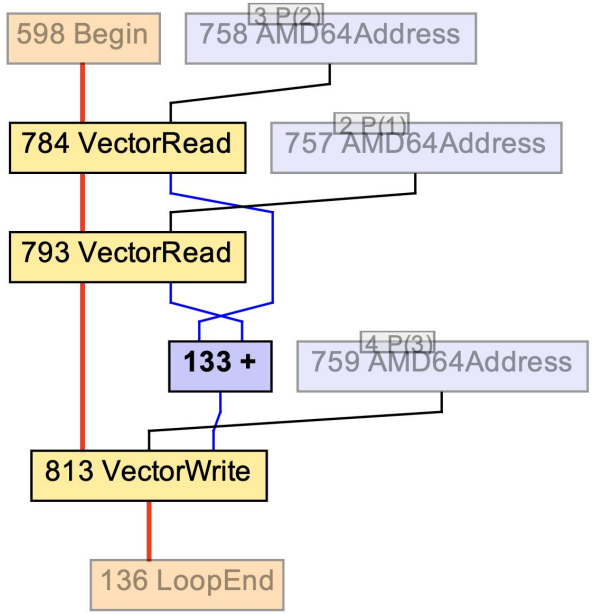
IsomorphicPackingPhase

Vector Stamps

VectorReadNode / VectorWriteNode

VectorExtractNode / VectorPackNode

Scheduling + Canonicalization



Phase Ordering

IsomorphicPackingPhase invoked in LowTier

Low tier features general memory ops.

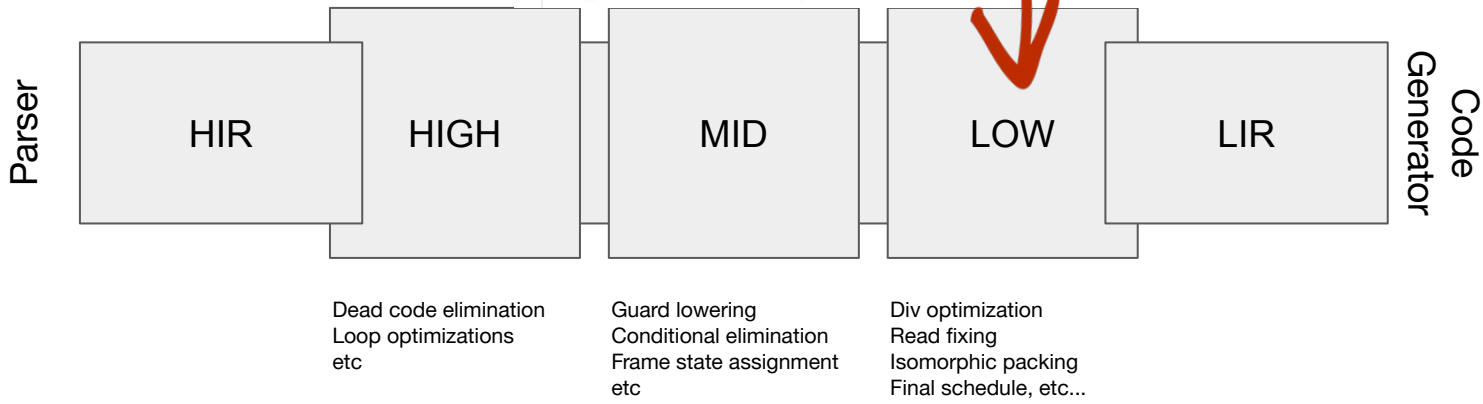
Low tier means fewer passes need to support vector stamps.

```
appendPhase(new UseTrappingNullChecksPhase());

if (Options.Autovectorize.getValue(options)) {
    final String listValue = Options.AVList.getValue(options).trim();
    List<String> list = new ArrayList<>();
    if (!listValue.isEmpty()) {
        list = Arrays.asList(listValue.split( regex: ","));
    }

    appendPhase(new IsomorphicPackingPhase(
            new SchedulePhase(SchedulingStrategy.EARLIEST),
            createAutovectorizationPolicies(),
            new MethodList(list, Options.AVWhitelist.getValue(options)),
            NodeView.DEFAULT));
}

appendPhase(canonicalizer);
```



Interface to Code Generator

Three new methods in LIRGeneratorTool

- emitPackConst
- emitPack
- emitExtract

Otherwise, we reuse existing emits

- emitAdd
- emitXor
- etc...

Supported Operations

Arithmetic

Integer

Add
Subtract
Multiply
And
Or
Xor
Negate
Not

Float

Add
Subtract
Multiply
Divide
Remainder
Negate

Packing

Load
Store
Constant Pack
Insert
Extract

Stack Packing

```
uint32_t foo1( __m256i vec , int j)
{
   return (uint32_t)_mm256_extract_epi32( vec, j );
}
```

Clang 5.0 intrinsic.
(viewed using godbolt.org)

Aligns stack pointer and allocates space.

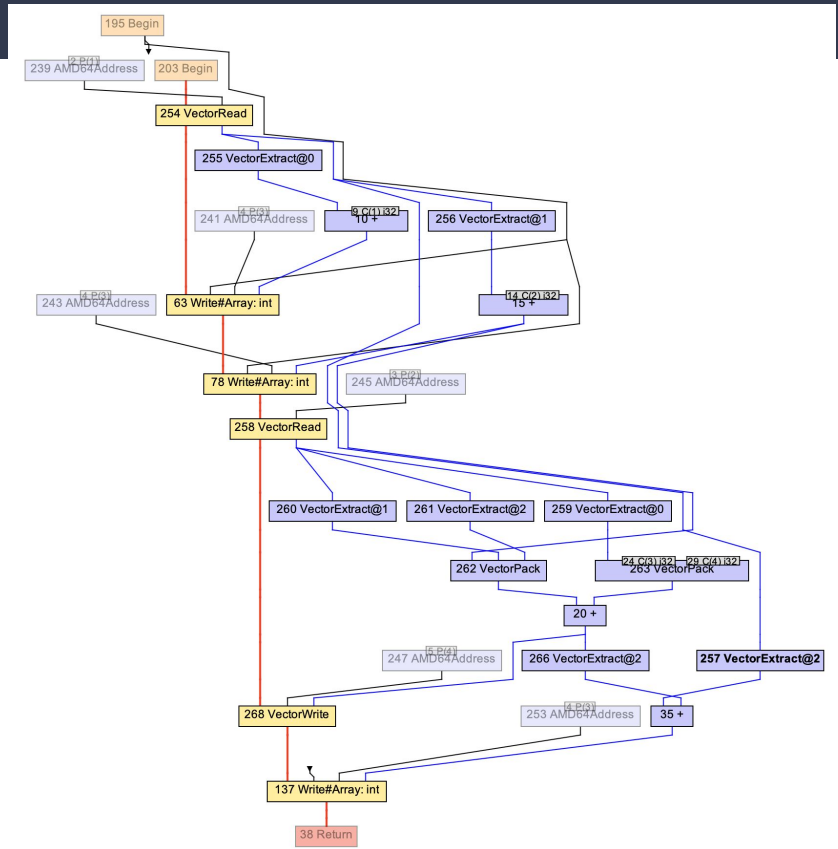
Extracts a single element from the vector.

```
and rsp, -32
sub rsp, 64
and edi, 7
vmovaps ymmword ptr [rsp], ymm0
mov eax, dword ptr [rsp + 4*rdi]
mov rsp, rbp
```

Challenges

Heuristics

- Too much vectorization is bad.
- Heuristics let us know if a part of the code should be vectorized.



Loop Unrolling & Bounds Check Elimination

- Graal features bounds checks for array accesses.
- Disables loop unrolling!
- Hacks around this:
  - Unroll earlier, before guards are lowered.
  - Remove bounds checks.

Bug: String Hashcode

The Problem: Memory corruption in strings, lots of NoSuchMethodErrors.

Location: Traced to a hashCode() method for Latin1 strings.

Cause: Emitting instructions using the stack kind instead of platform-specific kind.

Solution: Use platform-specific kind instead.

```
switch (kind) {
    case BYTE:
        masm.movb(dst, src);
        break;
    case WORD:
        masm.movw(dst, src);
        break;
    case DWORD:
        masm.movl(dst, src);
        break;
    case QWORD:
        masm.movq(dst, src);
        break;
```

Bug: Stack Pointer Addressing

The Problem: More memory corruption in strings!

Location: Traced to part of the regex matcher implementation.

Cause: Using the previously-mentioned stack loads and stores, rsp-relative addressing gets broken.

Solution: Replace dynamic use of stack pointer with a stack slot.

```
while (i + elements * 2 <= numElements && movKind.ge
    movKind = twice(movKind);
    elements *= 2;
}

// We compute the addresses we're going to move betw
final AMD64Address src = displace(getSourceAddress(c
final AMD64Address dst = displace(getDestinationAddr

// Perform the move.
addr2reg(crb, masm, movKind, asRegister(scratch), sr
reg2addr(crb, masm, movKind, dst, asRegister(scratch

// Increment both stack offset and number of element
// proceed with the loop.
stackOffset += movKind.getSizeInBytes();
i += elements;
```

Bug: Floating-Point Extraction

The Problem: Data corruption, invalid results from vectorized floating point operations.

Location: AMD64Assembler.

Cause: VPEXTRQ does not support XMM destination registers.

Solution: Check the register type and emit VPSHUFD if the destination register is of XMM type.

```
VPEXTRQ.emit(masm, XMM, asRegister(result), asRegister(vector), selector);
final Register resultRegister = asRegister(result);
// VPEXTRQ does not support xmm result registers
if (resultRegister.getRegisterCategory().equals(AMD64.XMM)) {
    VPSHUFD.emit(masm, XMM, resultRegister, asRegister(vector), selector);
} else {
    VPEXTRQ.emit(masm, XMM, asRegister(result), asRegister(vector), selector);
}
```

Examples and Benchmarks

Sample Code

```
for (int i = 0; i < count; i++) {
    tmp[i] = buf[i];
}
```

Excerpt from regex.Pattern#atom.

```
vmovdqu ymm0,YMMWORD PTR
[r11+r10*4+0x10]
vmovdqu YMMWORD PTR
[r14+r10*4+0x10],ymm0
```

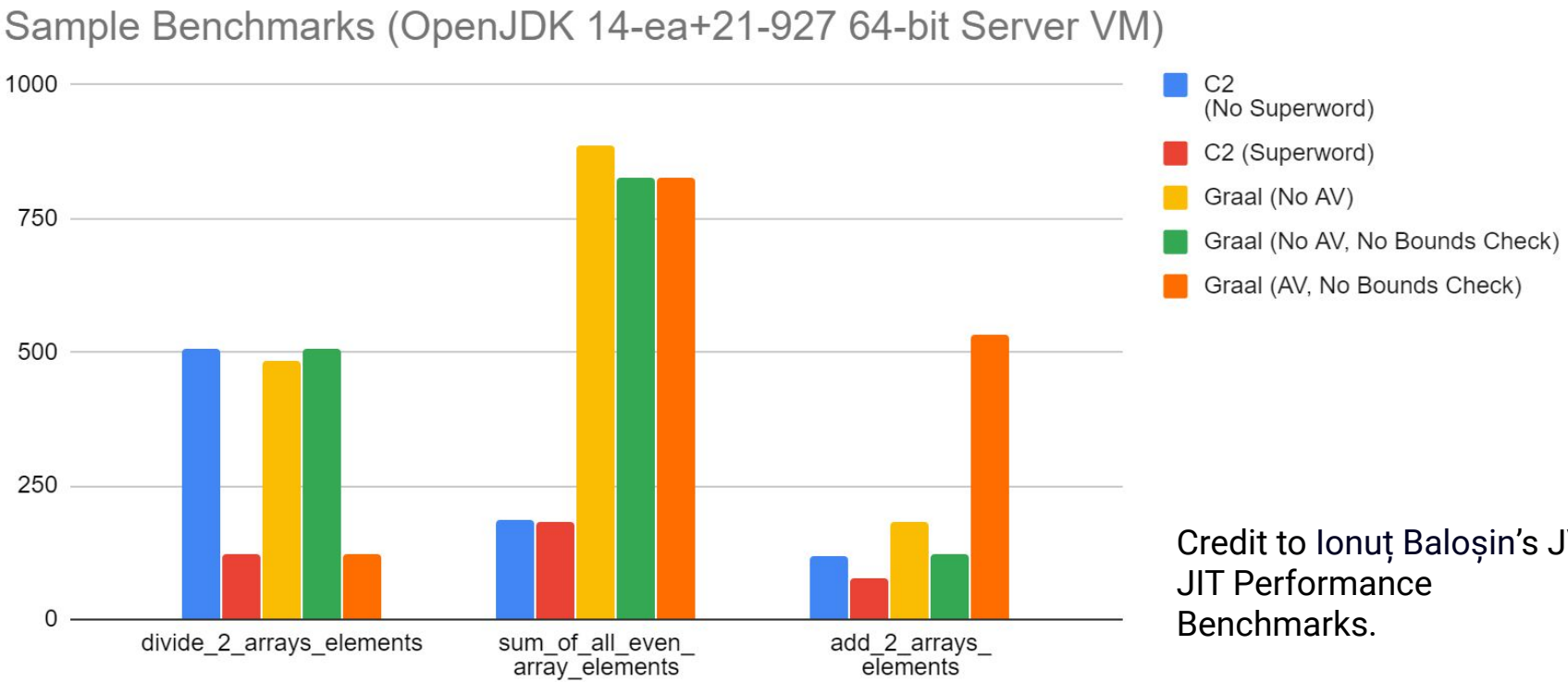
Pitfalls

```
for (int t = 16; t <= 79; t++) {
    int temp = W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16];
    W[t] = (temp << 1) | (temp >>> 31);
}
```

Excerpt from SHA#implCompress0.

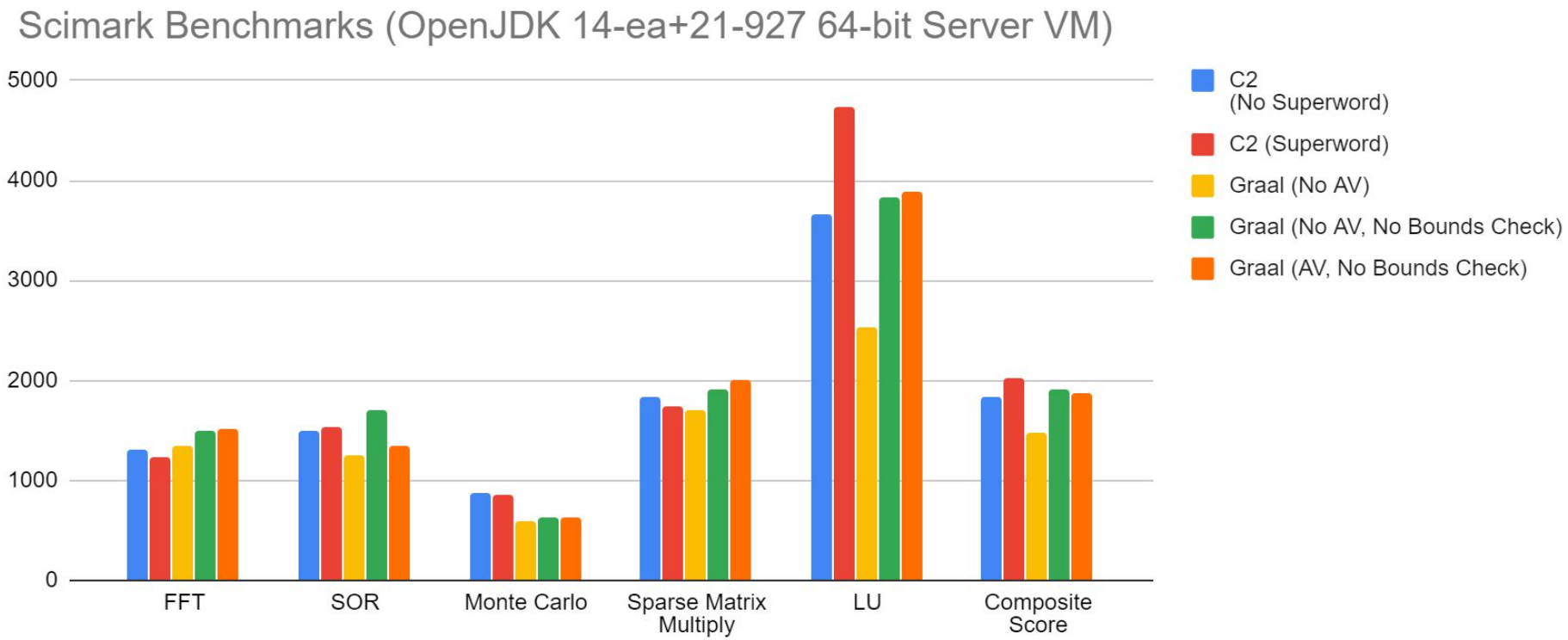
```
vextracti128 xmm1,ymm0,0x1
vpextrd eax,xmm1,0x2
xor    esi,eax
vextracti128 xmm0,ymm0,0x1
vpextrd eax,xmm0,0x3
vpextrd ebp,xmm3,0x2
vpextrd r13d,xmm3,0x3
vextracti128 xmm0,ymm3,0x1
vmovd  r14d,xmm0
mov    DWORD PTR
mov    DWORD PTR [rsp+0x144],r13d
mov    DWORD PTR [rsp+0x148],r14d
vmovdqu ymm0,YMMWORD PTR [rsp+0x140]
vpxor  xmm0,xmm4,xmm0
```

Ionut Benchmarks



Credit to Ionuț Balosin's JVM JIT Performance Benchmarks.

Scimark


Scimark Benchmarks (OpenJDK 14-ea+21-927 64-bit Server VM)
5000
4000
3000
2000
1000
0
FFT
SOR
Monte Carlo
Sparse Matrix Multiply
LU
Composite Score
C2 (No Superword)
C2 (Superword)
Graal (No AV)
Graal (No AV, No Bounds Check)
Graal (AV, No Bounds Check)

Conclusion

Overall

- Autovectorization is **hard.**
- **Heuristics** make or break an implementation.
- Not all **requirements** are necessarily implemented already or foreseeable.
  - Bounds check elimination is really important.

Overall

The pull request:
https://github.com/oracle/graal/pull/1692

Contribute: https://github.com/usrinivasan/graal

David Degazio (@elucentdev)

Niklas Vangerow (@nvgrw)