LLVM-CTH @ ISC 2020

SVE in LLVM

Will Lovett – Principal Technical Product Owner

will.lovett@arm.com

@hpc_will

26th June 2020

Supercomputer Fugaku: Fastest Supercomputer in the World

TOP 500 The List.

CERTIFICATE

Supercomputer Fugaku - A64FX 48C 2.2GHz, Tofu interconnect D

RIKEN Center for Computational Science, Japan

is ranked

No. 1

among the World's TOP500 Supercomputers

with 415.53 Pflop/s Linpack Performance

in the 55th TOP500 List published at the ISC 2020 Digital

Conference on June 22nd, 2020.

Congratulations from the TOP500 Editors

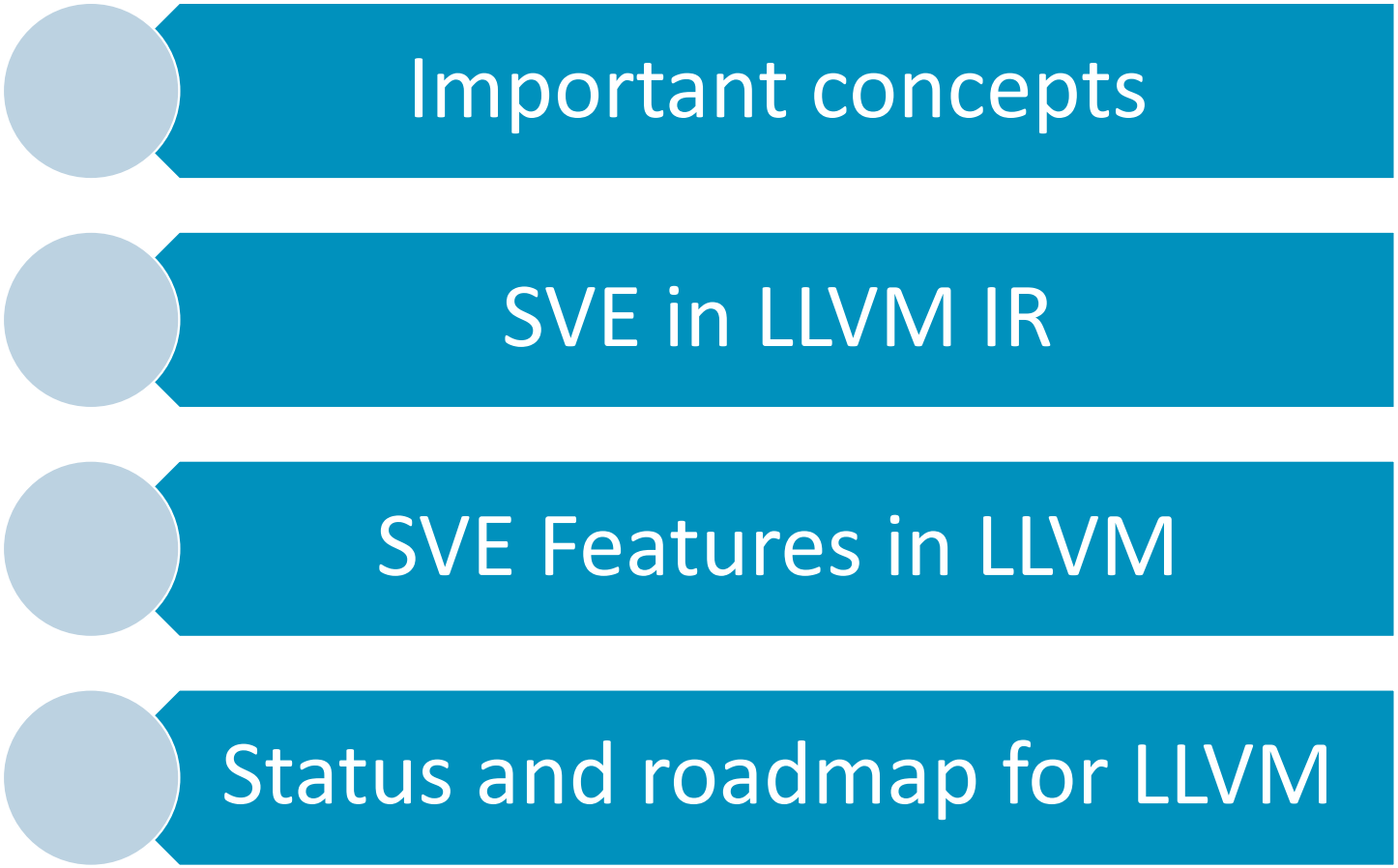
Erich Strohmaier
NERSC/Berkeley Lab

Jack Dongarra
University of Tennessee

Horst Simon
NERSC/Berkeley Lab

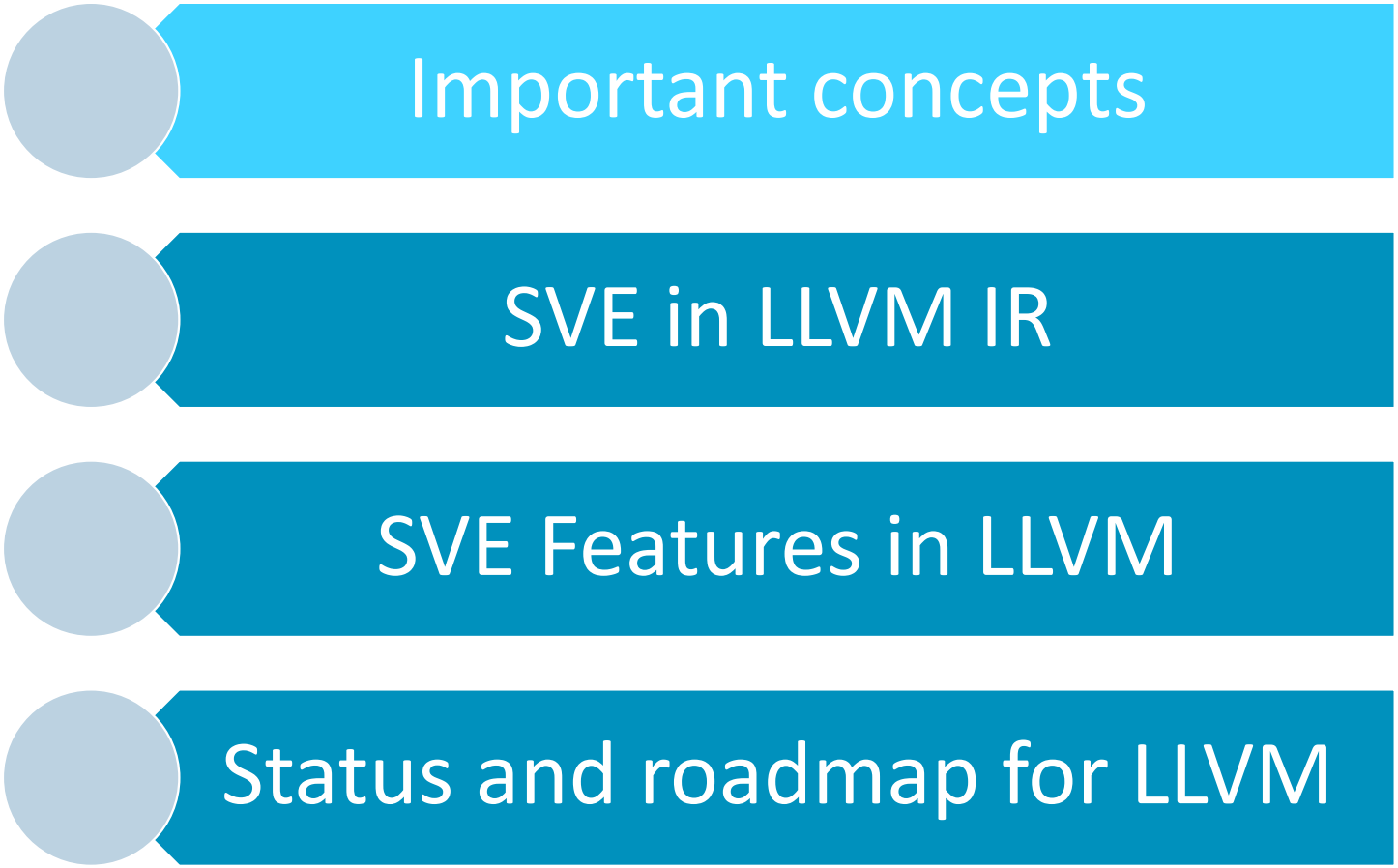
Martin Meuer
Prometeus

Overview



arm

Overview



arm

Important concepts and further reading

SVE

- Scalable vector extension to the Arm v8-a architecture, designed for HPC workloads
- Vector length can be any multiple of 128 bits, up to 2048 bits
- Predicate registers allow conditional execution of individual lanes within the vector
- Find out more

SVE2

- Builds upon SVE for usecases outside of HPC
- Adds NEON™-style fixed-point DSP/multimedia plus other new features.
- Tackles further obstacles to compiler auto-vectorization.
- Find out more

ACLE

- Arm C Language Extensions
- C intrinsics that map (roughly) 1-1 with Arm instructions
- ACLE for SVE covers support for SVE, SVE2 and Arm v8.6-a extensions to SVE
- Find out more

Arm v8-a

- The Arm 64-bit architecture
- Find out more

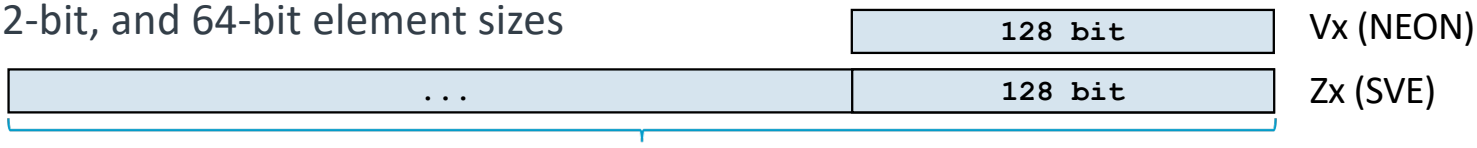
Arm v8.6-a

- The most recent annual update to the Arm v8-a architecture
- Adds support for bfloat16 and high performance matrix multiplication
- Find out more

arm

Scalable Vectors and Predication

- 32 scalable vector registers (Z0-Z31):
  - 128-2048 bits vector length is decided by **implementation**
  - Bottom 128 bits are overlaying with the Floating-point & NEON vector register bank (V0-V31)
  - Supporting 8-bit, 16-bit, 32-bit, and 64-bit element sizes

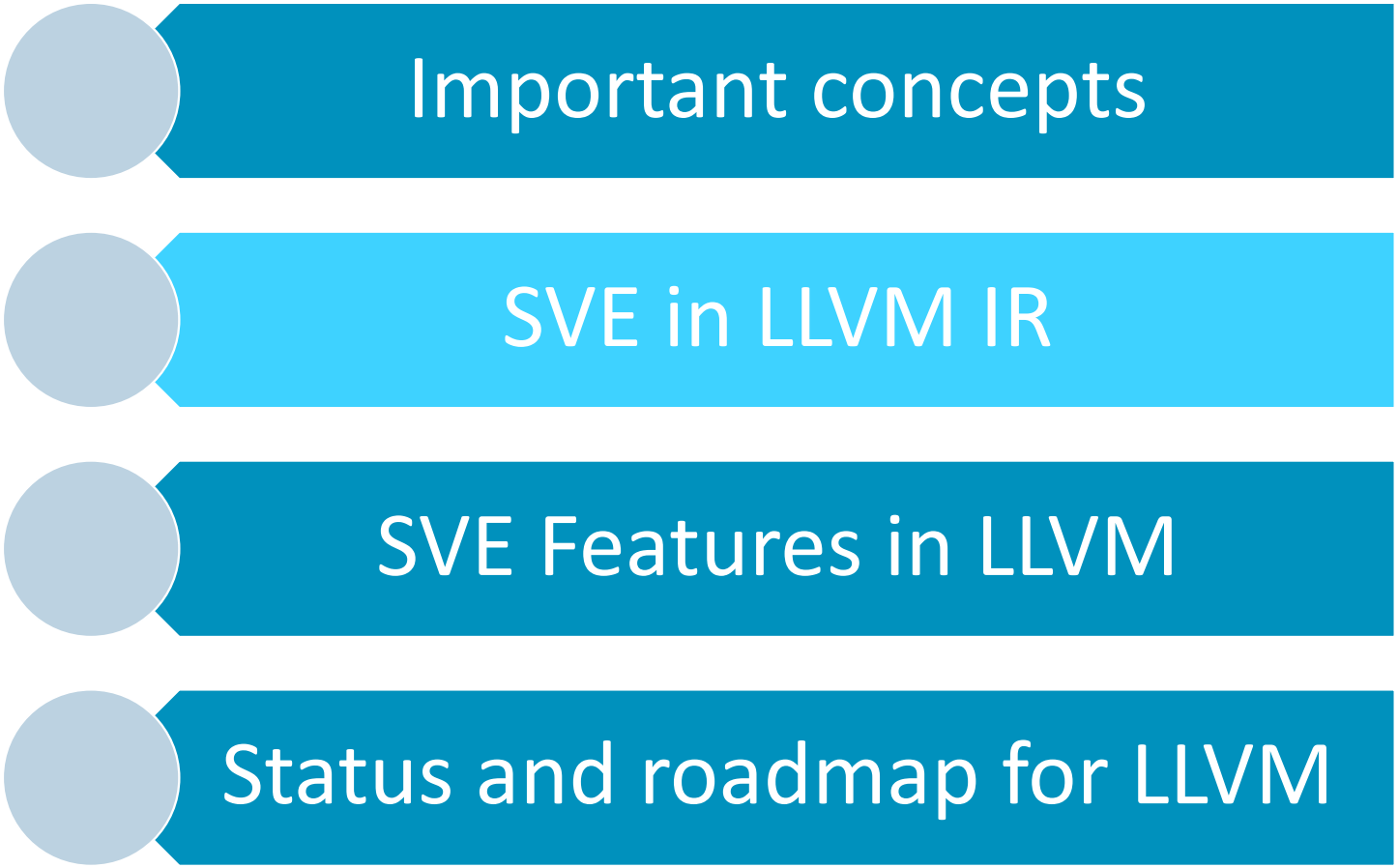


- 16 Scalable predicate registers P0-P15:
  - Flags that represent whether each vector lane is active
  - 1/8th size of vector registers, so can represent the lanes of 8-bit elements
  - For vector elements > 8-bit, only the least-significant bit is used

Zx	8-bit	8-bit	8-bit	8-bit	…	8-bit	8-bit	8-bit	8-bit	8-bit element
Px	1	1	1	1	…	1	1	1	1	32 lanes on A64FX
Zx	32-bit				…	32-bit				32-bit element
Px				1	…				1	16 lanes on A64FX

arm

Overview



arm

Teach LLVM IR to understand scalable vectors

How long is a piece of (scalable) string?

1. Add the concept of scalable vectors
2. Fix up `VectorType::getNumElements()` et al
3. Stack frame layout with scalable sizes
4. Representing shuffles of scalable vectors

arm

LLVM IR 1/4: Add the concept of scalable vectors

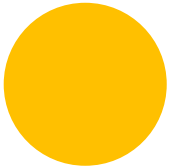
The problem

- Traditionally, vector size was known at compile-time
  - eg. `<4 x i32>` is a 128-bit vector of 4 32-bit integer values
- With SVE, vector size is now unknown at compile-time
  - Some unknown multiple of 128-bits, up to 2048 bits

The solution

- LLVM IR chooses to assume that this value is unknown but constant
  - If you break that assumption, you've broken the SVE calling convention and you're on your own
- Extend vector type to include vscale, which represents this unknown constant
  - eg. `<vscale x 4 x i32>` is a vector with a multiple of 4 32-bit integer values
  - On a 128-bit implementation of SVE, vscale = 1, and there will be 4 elements
  - On a 512-bit implementation of SVE, vscale = 4, and there will be 16 elements

LLVM IR 2/4: Fix up VectorType::getNumElements() et al



The problem

```
unsigned getNumElements() const {return VectorTypeBits.NumElements;}
```

- This, and other similar accessors are used **everywhere**

The solution

- Split `VectorType` into `FixedVectorType` and `ScalableVectorType`
- Methodically go through all 787 calls to `getNumElements()`
- Convert them to use `FixedVectorType` or to correctly support scalable vectors
- Kudos to Christopher Tetreault from Qualcomm!

LLVM IR 3/4: Stack frame layout with scalable sizes

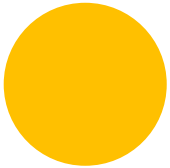
The problem

- Traditionally, all values on the stack have a known size in bytes
- So the offset from the stack base of every stack element is a compile-time constant
- We now need to support spilling/filling scalable vectors

The solution

- Add a scalable area to each function's stack frame
- This area only contains scalable values
- Address calculation becomes
  - `pointer + offset + offset2 * vscale`

LLVM IR 4/4: Representing shuffles of scalable vectors



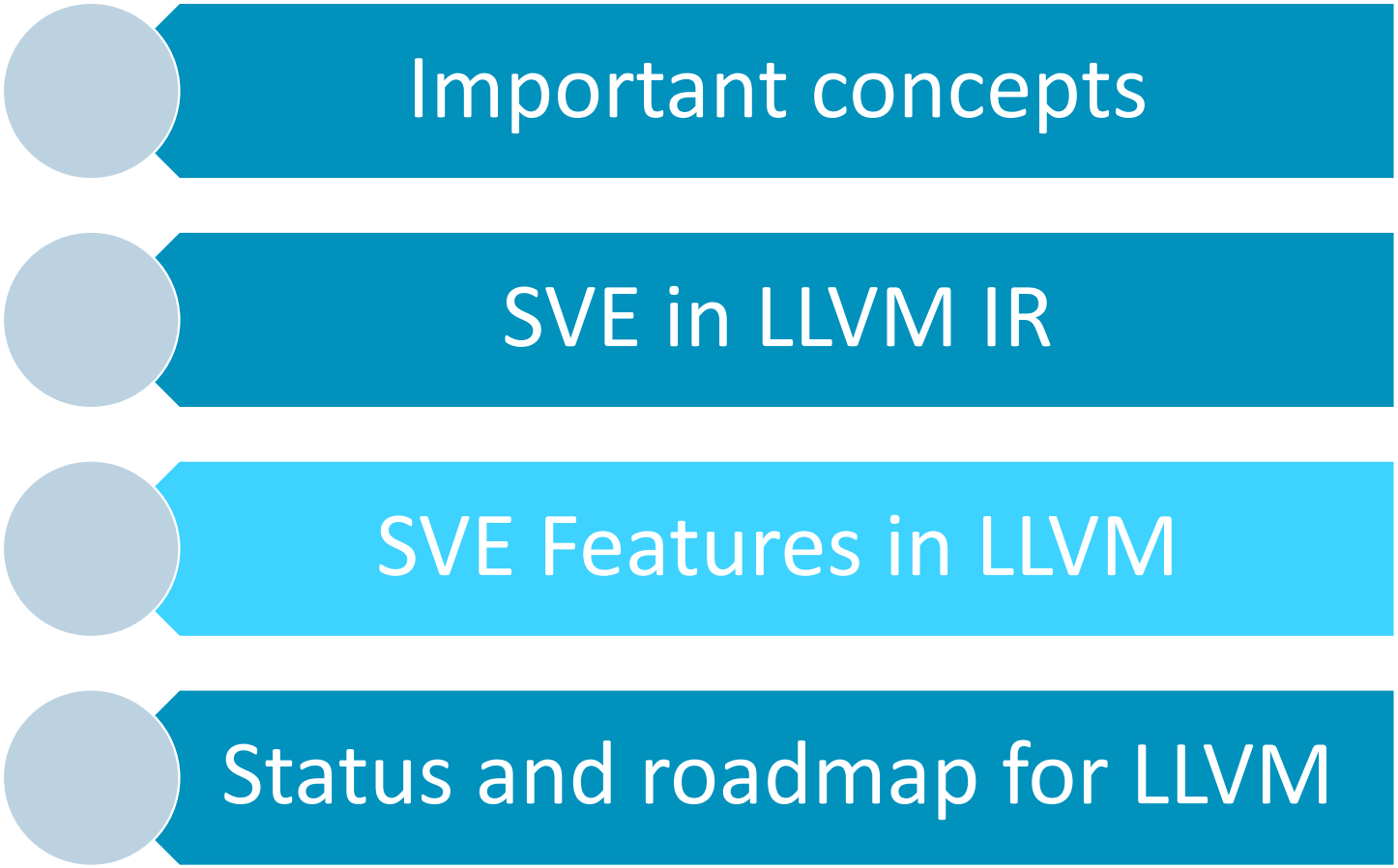
The problem

- LLVM needs to represent permute the elements within a vector
- Eg a zip of the even-numbered elements of two `<4 x i32>` vectors looks like this:
  - `<result> = shufflevector <4 x i32> %v1, <4 x i32> %v2, <4 x i32> <i32 0, i32 4, i32 1, i32 5> ;`
- But how do we construct an index into a vector when we don't know its length?
- Other uses of shufflevector have similar issues
  - splat, reverse, concat, split_lo/hi, zip_lo/hi, unzip_even/odd, splice

The solution

- Undecided upstream!
  - Current discussion is around using named shuffle patterns and/or pattern-generating IR intrinsics
- Downstream, we have added additional constants:
  - We use constant stepvector and vscale to derive all the access patterns for shufflevector

Overview



arm

Teach LLVM vectorizer to use SVE features

Leveraging new instructions for fun and profit

1. Per-lane predication
2. Scalar tail removal
3. Multiple exits and unknown tripcounts
4. Loops with gather/scatters
5. Reductions
6. Complex number operations

SVE features 1/6: Per-lane predication

Allows efficient conditional execution within loops

```
void foo(char* mask,
         double *result,
         int n)
{
  for (int x = 0; x < n; x++)
    if(mask[x] == 0)
      result[x] += 2.0;
}
```

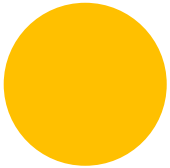
Today

- Supported in LLVM today using `select` to filter out inactive lanes in results
- This is incompatible with FP operations that have side-effects, such as raising an exception on FP divide-by-zero

Future

- First-class predication support in LLVM IR is a work is in progress in the community
- Will make it easier to support fully compliant IEEE FP exception handling

SVE features 2/6: Scalar tail removal



Introducing WHILELO and friends

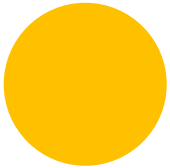
- SVE has instructions such as `whilelo` which test the content of a predicate vector and set condition codes
- These condition codes can then control conditional branches
- `whilelo p1.d, x8, x9`

```
for (i = 0; i < n; ++i)
```

INDEX i	n+1	n	n-1	n-2
WHILELO	0	0	1	1

- This allows us to eliminate scalar tail blocks

SVE features 2/6: Scalar tail removal



SVE example

Setup

Vector loop

```
foo:
// <setup snipped>
.LBB0_2:
  ld1d { z0.d }, p1/z, [x0, x8, lsl #3]
  fadd z0.d, p0/m, z0.d, #1.0
  st1d { z0.d }, p1, [x0, x8, lsl #3]
  incd x8
  whilelo p1.d, x8, x9
  b.first .LBB0_2
.LBB0_3:
  ret
```

```
void foo (double* a,
          unsigned n) {
  for (unsigned i=0; i<n; i++)
    a[i] += 1.0;
}
```

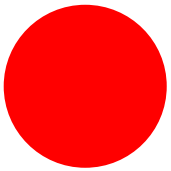
Advantages

- Scalar tail is removed
- Great for code density/icache

Disadvantages

- Adds a `whilelo` -> `b.first` dependency

SVE features 3/6: Multiple exits and unknown tripcounts



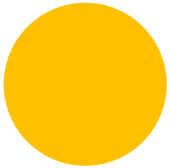
Allows vectorization with early exits

```
void strcpy(char *a, char *b)
{
  int i=0;
  while (a[i] != 0)
    b[i] = a[i];
}
```

- Loops with unknown tripcounts are traditionally hard to vectorize
- Vector loads or stores past the end of either array could trigger a page fault which could never occur in a scalar implementation
- First-faulting loads and stores in SVE allow a safe vector implementation

- This is unsupported by LLVM's vectorizer today
- Easy to implement for library functions using the ACLE

SVE features 4/6: gather/scatters and structured accesses



Allows indirect memory accesses within vectorized loops

```
double lookup_and_sum(double* a,
                      int* index,
                      int length) {
  double res = 0;
  for (int i = 0; i < length; ++i)
    res += a[index[i]];
  return res;
}
```

- SVE supports vector loads from a vector of (calculated) addresses
- Supported in LLVM today
- Cost modeling is overly simplistic

- InterleavedAccessPass replaces gathers/scatters with interleaved accesses, which are commonly faster
- Work is needed to update this pass to handle indexing of scalable vector types

SVE features 5/6: Horizontal reductions

Allows efficient reduction loops

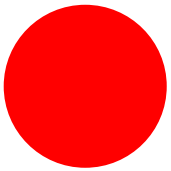
```
double sum(double* a, int length) {
  double res = 0;
  for (int i = 0; i < length; ++i)
    res += a[i];
  return res;
}
```

Two options, depending on FP requirements:

1. **Strict ordering**
   - Scalar partial result is updated in each vector loop iteration
   - Significant cost to cross-lane operation within the vector body
2. **Loose ordering**
   - `-fassociative-math -fno-signed-zeros -fno-trapping-math`
   - Allows fast vertical reductions within loop body
   - Relatively fast recursive pairwise reduction of final vector

Both options are fully supported in LLVM

SVE features 6/6: Complex number operations

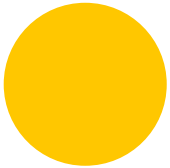


Allows efficient operations on complex numbers

```
double _Complex foo (double _Complex *v,
                     double _Complex *w,
                     int n) {
  double _Complex result = 0;
  for (int i = 0; i < n; ++i)
    result += v[i] * w[i];
  return result;
}
```

- SVE supports operations on complex numbers, operating on pairs of (real, imaginary) lanes
- Can be identified using SLP-style vectorization with fixed width vectors
- This is more difficult with scalable vectors
- Some work done downstream (not yet productised)

SVE features 7*/6: Vector-length Specific SVE support



Purpose

- Allow LLVM to assume a specific SVE vector width
- Behaviour with a different vector width is undefined

Usage:

- `-msve-vector-bits=<length>`

ACLE

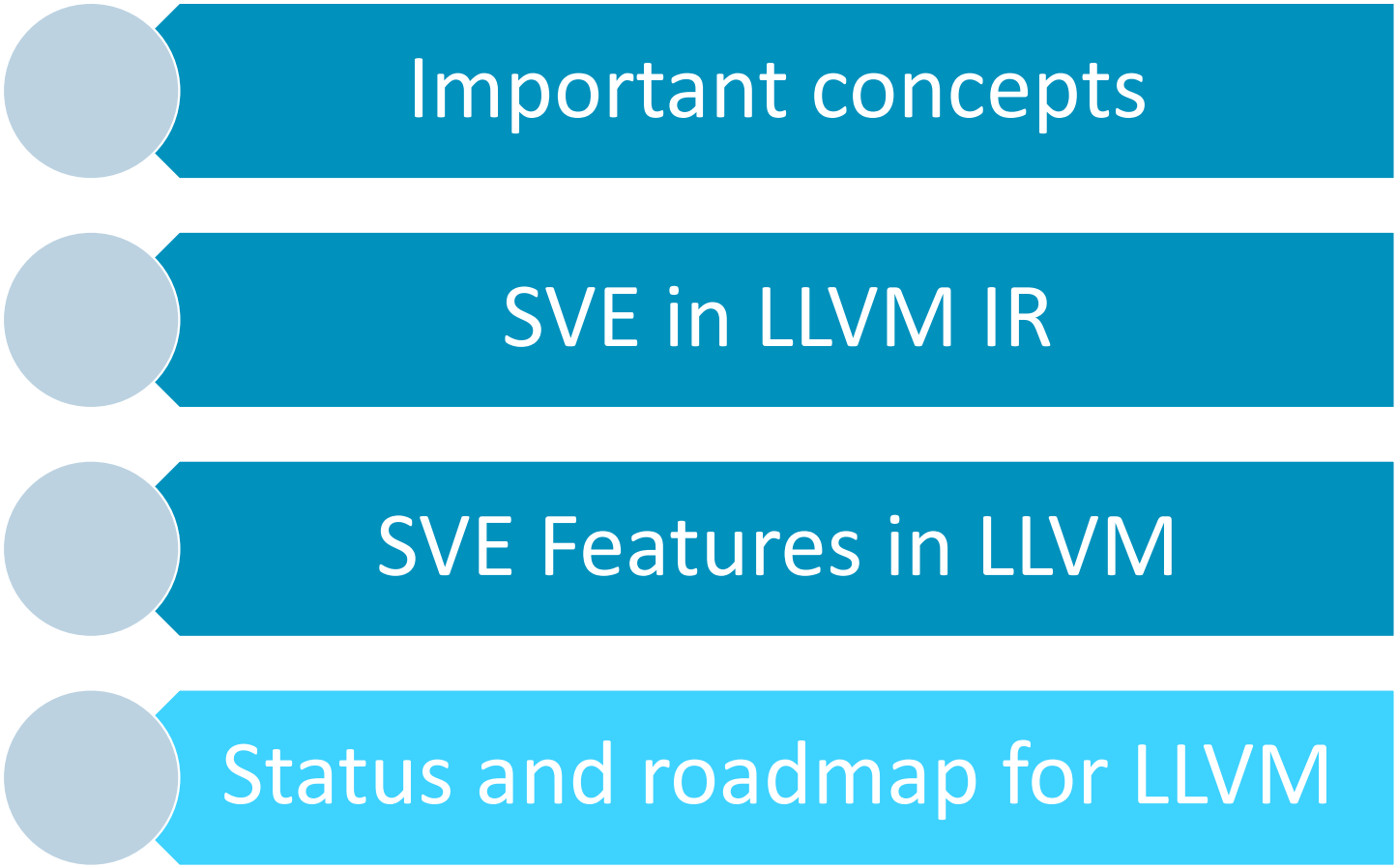
- Allows LLVM to transparently cast between SVE ACLE types and GNU C/C++ vectors
- This is necessary for storing values within structs and classes
- Likely to be used in some HPC frameworks such as Grid and Kokkos SIMD

Autovec

- Allows LLVM vector IR to codegen to SVE
- Likely to be used by some downstream LLVM compilers

* This one isn't really a feature of SVE!

Overview



arm

Status and roadmap in LLVM

LLVM 9 (September 2019)

- SVE/SVE2 assembly and disassembly

LLVM 11 (September 2020)

- SVE/SVE2 intrinsic support
- Stabilisation, stack unwinding, debuginfo
- Armv8.6-a support (bfloat16, matmul)
- Vector-length specific SVE codegen

LLVM 12 (March 2021)

- ACLE code quality improvements (let us know!)
- Vector-length agnostic SVE vectorization of a few simple loops

LLVM 13 (September 2021)

- Full support for Vector-length agnostic SVE vectorization

arm

Contributions welcome!

- Huge thanks to the Arm partners and the wider LLVM community, especially
  - Qualcomm
  - HPE/Cray
  - Fujitsu
  - Linaro
  - Huawei

- Get involved!
  - SVE/SVE2 bi-weekly sync up signup sheet
  - Meeting notes from recent calls
  - Phabricator patches
  - List of identified work items

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה