SVE DETAILS BASED ON EXPERIENCES WITH PORTING APPLICATIONS FOR SVE

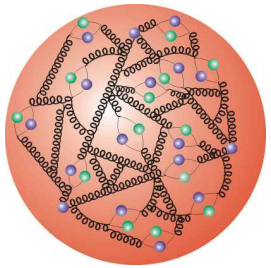
Dirk Pleiter | Arm SVE Hackathon, Dallas | 15.11.2018

JÜLICH
Forschungszentrum

Outline

- First steps
- Auto-vectorisation vs. intrinsics
- Examples: daxpy and zaxpy
- Summary of useful instrinsics
- Wrapping intrinsics in C++ templates

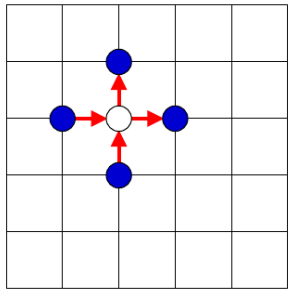
Applications Ported to SVE



[https://github.com/paboyle/Grid]

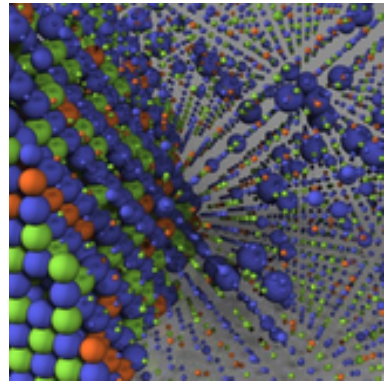
Lattice QCD library and tool suite Grid

- Main numerical task within main kernel: Sparse matrix-vector multiplication
- Software optimised for SIMD instructions, e.g. AVX512
- Credit: Nils Meyer



KKRnano and Quantum Espresso

- Based on Density Functional Theory (DFT) for electronic structure calculations
- For relevant use cases a significant amount of time is spent in zgemm
- Credit: Stepan Nassyr



First Steps

Pre-requisites

- Get access to AArch64-based platform
- Obtain SVE-enabled compiler
  - Here we use Arm Allinea Studio 19.0 [https://developer.arm.com/products/software-development-tools/hpc/arm-allinea-studio/download]
    - Arm C/C++/Fortran Compiler version 19.0 (build number 69) (based on LLVM 7.0.2)
    - gcc (ARM-build-2) 8.2.0
  - Alternatively use compilers as distributed by OS
- Obtain Arm Instruction Emulator (armie) [https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator]

Compiling for SVE

- `gcc -march=armv8-a+sve -O3 -o myprog.x myprog.c`
- `armclang -march=armv8-a+sve -O3 -o myprog.x myprog.c`

Functional simulations

- `armie -msve-vector-bits=512 ./myprog.x`

Auto-Vectorisation vs. Intrinsics

Auto-vectorisation

- Rely on compiler to identify opportunities for vectorisation
- Useful compiler options:
  - `gcc -ftree-vectorizer-verbose=3 -fopt-info-vec-missed` …
  - `armclang -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize` …
- Pro: Portable code
- Con: Compiler still often fail to recognise opportunities for vectorisation

ARM C Language Extensions (ACLE) for SVE

- Makes features of the ARM architecture directly available in C and C++ programs
  [https://static.docs.arm.com/100987/0000/acle_sve_100987_0000_00_en.pdf]
- Pro: Control on code generation
- Con: Non-portable code

Simple Example: daxpy

```
void daxpy(double a, double *restrict x, double *restrict y, int n)
{
  int i;

  for (i = 0; i < n; i++)
    y[i] += a * x[i];
}
```

```
.L3:
        ld1d    z1.d, p0/z, [x1, x3, lsl 3]
        ld1d    z2.d, p0/z, [x0, x3, lsl 3]
        fmla    z1.d, p1/m, z0.d, z2.d
        st1d    z1.d, p0, [x1, x3, lsl 3]
        incd    x3
        whilelo p0.d, x3, x2
        bne     .L3
```

Contiguous load double words to vector

Floating-point fused multiply-add

Contiguous store doublewords

Increment vector

While incrementing unsigned scalar lower than scalar

Increment vector

Complex Arithmetics in SVE

Arithmetics with complex operands u, w, z

- $z \leftarrow u \;\; w$ expands to $(z^{(r)}, z^{(i)}) \leftarrow (u^{(r)} \cdot w^{(r)} - u^{(i)} \cdot w^{(i)}, u^{(r)} \cdot w^{(i)} + u^{(i)} \cdot w^{(r)})$
- AoS vs. SoA memory layout options:
  - $(z_0^{(r)}, z_0^{(i)}, z_1^{(r)}, z_1^{(i)}, \ldots)$ vs. $(z_0^{(r)}, z_1^{(r)}, \ldots), (z_0^{(i)}, z_1^{(i)}, \ldots)$

Structured load

- LD2D: Contiguous load two-doubleword structures to two vectors
- AoS layout in memory, SoA layout in register file

Native instructions for arithmetics with complex operands

- FCADD: Floating-point complex add with rotate
- FCMLA: Floating-point complex multiply-add with rotate

Example: zaxpy with Auto-Vectorization

```
typedef struct { double re; double im; } dcomplex;

void zaxpy(dcomplex a, dcomplex *restrict x, dcomplex *restrict y, int n)
{
  int i;

  for (i = 0; i < n; i++)
  {
    y[i].re += a.re * x[i].re - a.im * x[i].im;
    y[i].im += a.re * x[i].im + a.im * x[i].re;
  }
}
```

Trick to persuade the compiler

```
.LBB0_2:
        lsl     x10, x8, #1
        ld2d    { z2.d, z3.d }, p0/z, [x0, x10, lsl #3]
        ld2d    { z4.d, z5.d }, p0/z, [x1, x10, lsl #3]
        incd    x8
        fmul    z6.d, z0.d, z2.d
        fmul    z7.d, z1.d, z3.d
        fmul    z16.d, z0.d, z3.d
        fmul    z2.d, z1.d, z2.d
        fsub    z3.d, z6.d, z7.d
        fadd    z2.d, z2.d, z16.d
        fadd    z6.d, z4.d, z3.d
        fadd    z7.d, z2.d, z5.d
        st2d    { z6.d, z7.d }, p0, [x1, x10, lsl #3]
        whilelo p0.d, x8, x9
        b.mi    .LBB0_2
```

Example: zaxpy with Intrinsics

```
#include <arm_sve.h>

void zaxpy(double complex a, double complex *restrict x, double complex *restrict y, int64_t n)
{
  svfloat64_t av = svdupq_f64(creal(a), cimag(a));
  svfloat64_t nv = svdup_f64(0.);

  int64_t k = 0;
  svbool_t pg = svwhilelt_b64(2*k, 2*n);
  do
  {
    svfloat64_t xv = svld1(pg, (const double *) &x[k]);
    svfloat64_t yv = svld1(pg, (const double *) &y[k]);

    svfloat64_t zv0 = svcmla_f64_x(pg, nv, xv, av,  0);
    svfloat64_t zv1 = svcmla_f64_x(pg, yv, xv, av, 90);
    svfloat64_t zv  = svadd_f64_x(pg, zv0, zv1);

    svst1(pg, (double *) &y[k], zv);

    k += svcntd() / 2;
    pg = svwhilelt_b64(2*k, 2*n);
  }
  while (svptest_any(svptrue_b64(), pg));
}
```

```
.LBB0_1:
        add     x13, x0, x8
        add     x14, x1, x8
        ld1d    { z2.d }, p0/z, [x13]
        ld1d    { z3.d }, p0/z, [x14]
        mov     z4.d, z1.d
        add     x8, x8, x10
        fcmla   z4.d, p0/m, z2.d, z0.d, #0
        fcmla   z3.d, p0/m, z2.d, z0.d, #90
        fadd    z4.d, p0/m, z4.d, z3.d
        st1d    { z4.d }, p0, [x14]
        whilelt p0.d, x12, x9
        add     x12, x12, x11
        b.ne    .LBB0_1
```

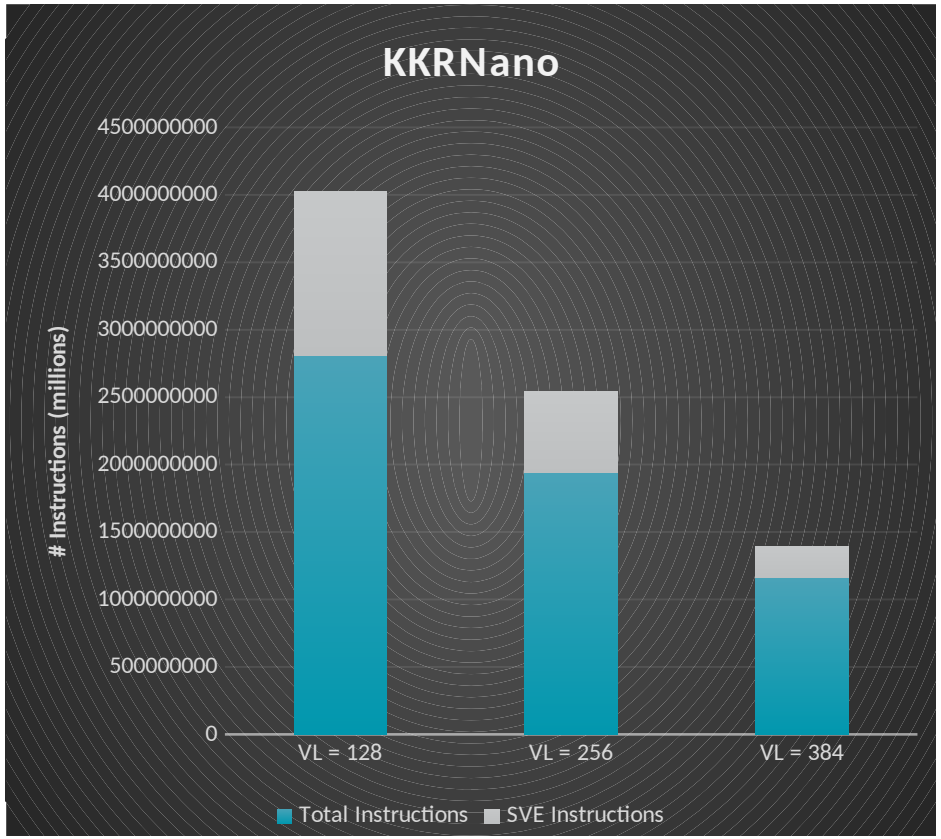
SVE-enabled KKRnano

Status

- Three kernel functions customized:
  - zdtotu(), zaxpy(), zgemm()
- Re-implemented with ARMC language extensions (ACLE) for ARM SVE
- The correct executions are verified with armie

Performance impact

- Unclear as this requires simulator
- Observe significant reduction in number of executed instructions depending on vector length



JÜLICH Forschungszentrum

Summary of Useful Intrinsics (1/2)

Intrinsic	Description
`svfloat64_t svld1[_f64](svbool_t pg, const float64_t *base)`	Load values from memory and store the results in a vector
`svfloat64x2_t svld2[_f64](svbool_t pg, const float64_t *base)`	Load an array of two-element structures into two vectors
`void svst1[_f64](svbool_t pg, float64_t *base, svfloat64_t data)`	Read elements from a vector and store them to memory
`svfloat64_t svadd[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)`	Addition on two floating-point inputs
`svfloat64_t svcadd[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,uint64_t imm_rotation)`	Floating-point complex addition with rotation
`svfloat64_t svmul[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)`	Multiply two floating-point inputs

Summary of Useful Intrinsics (2/2)

Intrinsic	Description
`svfloat64_t svcmla[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3, uint64_t imm_rotation)`	Take and return complex floating-point values, with the real components in even elements and the imaginary components in odd elements
`svbool_t svwhilelt_b64[_u64] (uint64_t op1, uint64_t op2)`	Return a predicate in which element N is active if, for all values M in the range [0, N], adding M to the first input gives a value that is less than the second
`bool svptest_any(svbool_t pg, svbool_t op)`	Test which active elements of the input predicate are true and return a boolean based on the result
`uint64_t svcntd()`	Count the number of active elements in a predicate input

C++ Templates for LQCD: Portable use of Intrinsics

```
template <typename T> struct W;
template <> struct W<double> {
  constexpr static unsigned int c = GEN_SIMD_WIDTH/16u;
  constexpr static unsigned int r = GEN_SIMD_WIDTH/8u;
};

template <typename T>
struct vec {
  alignas(GEN_SIMD_WIDTH) T v[W<T>::r];
};

typedef vec<double>    vecd;
```

```
struct Sum{
  //Complex/Real float
  inline __m512 operator()(__m512 a, __m512 b){
    return _mm512_add_ps(a,b);
  }
  //Complex/Real double
  inline __m512d operator()(__m512d a, __m512d b){
    …
  }
  …
}
```

```
struct Sum{
  inline vecd operator()(const vecd &a, const vecd &b){

    vecd out;
    svbool_t pg1 = svptrue_b64();
    svfloat64_t a_v = svld1(pg1, a.v);
    svfloat64_t b_v = svld1(pg1, b.v);
    svfloat64_t r_v = svadd_x(pg1, a_v, b_v);
    svst1(pg1, out.v, r_v);

    return out;
  }
  inline vecf operator()(const vecf &a, const vecf &b){
    …
  }
  …
}
```

THANK YOU

JÜLICH
Forschungszentrum