兆松科技（武汉）有限公司
Terapines Technology (Wuhan) Co. , Ltd.

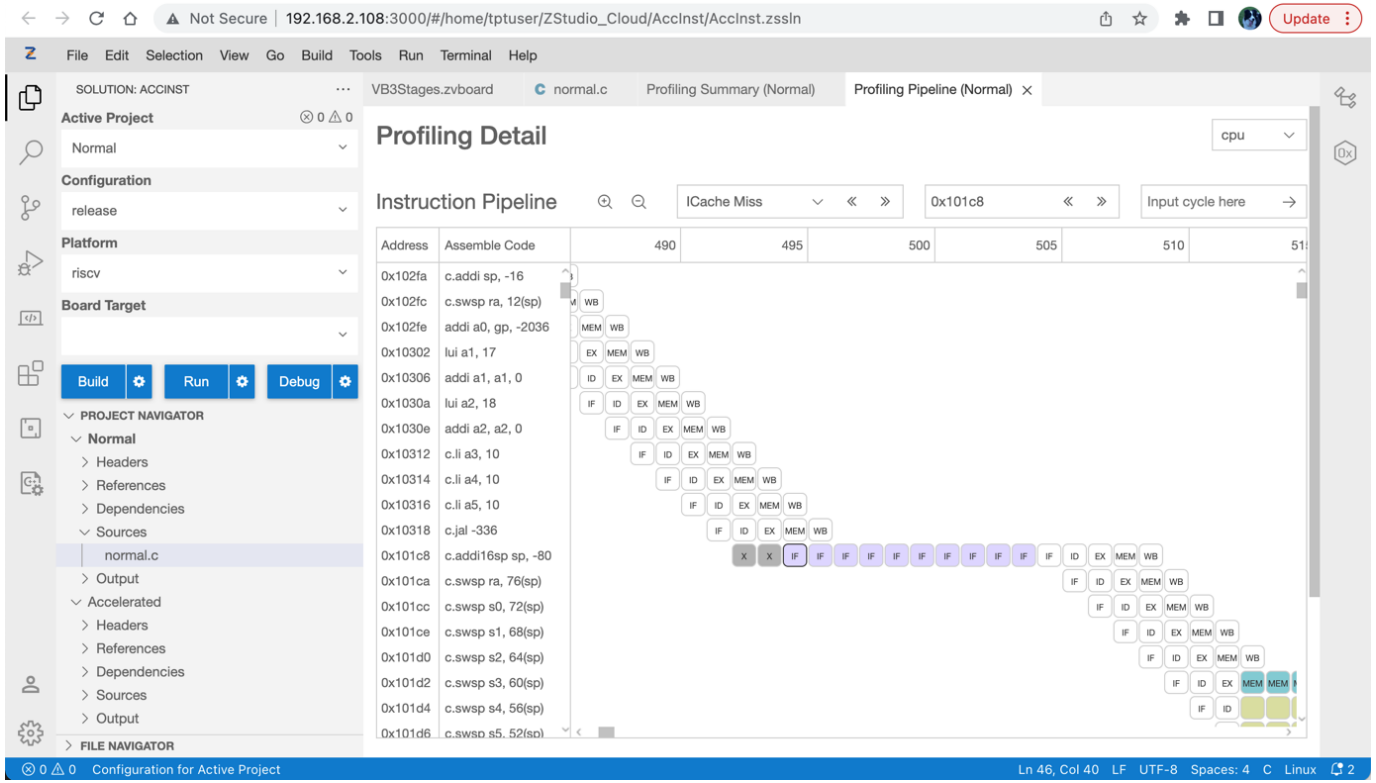
Accelerate HPC and AI applications with RVV auto-vectorization

Terapines Technology (Wuhan)
Hualin Wu, CTO & Co-founder
aries.wu@terapines.com

Terapines – co-design solution provider

- High performance/code density C/C++/Fortran compiler **ZCC**
- Cycle accurate and instruction set simulator **zemu**
- Cloud and cross platform IDE **zstudio**
- Profiling and microarchitecture analysis **zprof**
- Virtual board builder (ESL) **zvboard**
- Rapid architecture exploration and instruction customization and SDK auto-generator **zigen**.
- CIRCT based HDL simulator **zvc**
- ……

Terapines is found at 2019 in Wuhan, China



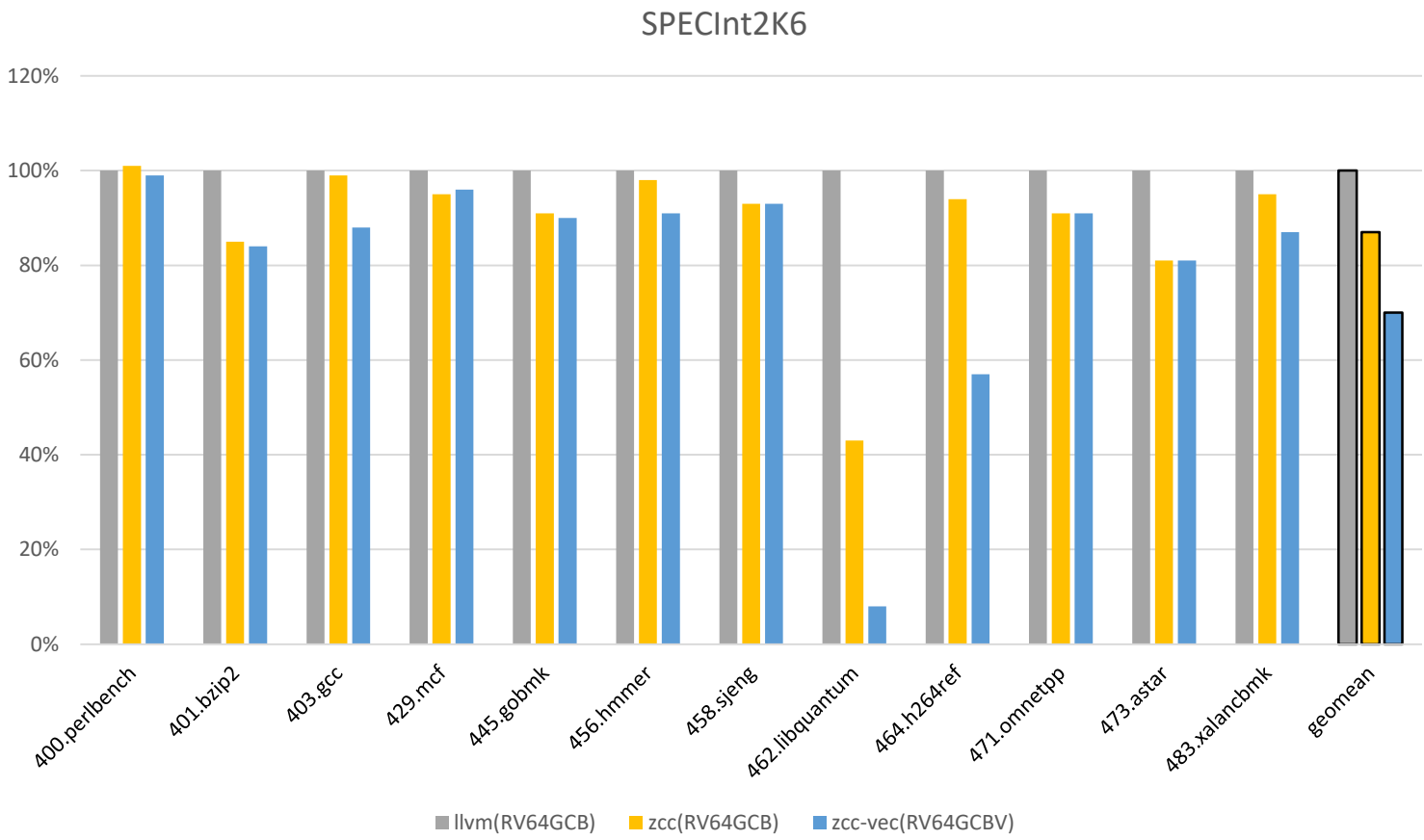
Terapines – founders

Hualin Wu
A LLVM guy since 2008

Tunghwa Wang
Former Andes SW VP

High performance compiler - ZCC

- SPECInt2K6 (Dynamic inst. count)
  - RV64GCB: **13%** better than LLVM16
  - RV64GCBV: **30%** better than LLVM16
- SPECInt2K6 score
  - RV64GCB: **10%** better than LLVM16 (expected)
  - RV64GCBV: **15%~20%** better than LLVM16 (expected)



*relative dynamic instruction count, smaller is better, RV64GCBV, data path: 512bits

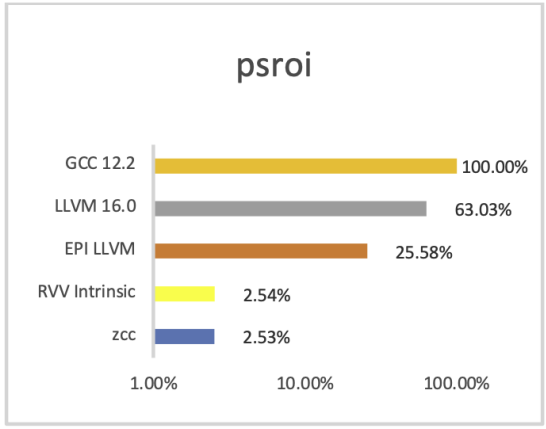
Partner with SOPHGO

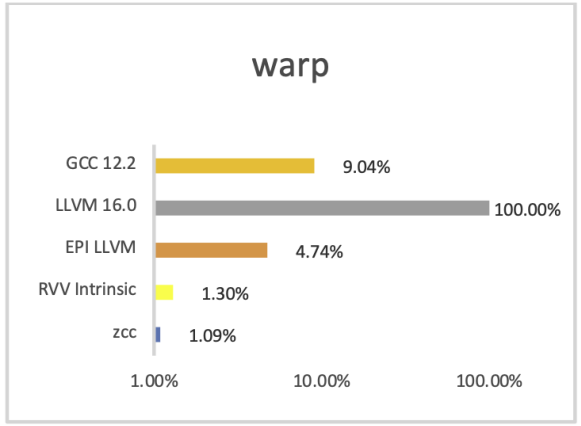
- ISA: RV64GCV (0.71 RVV)
- CPU: SG2042 (64Core@2.0GHz)
- NoC: 4 x 4 2d mesh, 4 cores/cluster
- L1 Cache: I$: 64K, D$: 64K
- L2 Cache: 1M/cluster, 16M total
- L3 Cache: 64M
- TDP: 120W
- DDR: 4 channel 3200MT/s ECC RDIMM/UDIMM/SODIMM
- PCIE: 2 x 16x Gen4
- Optimized by Terapines high performance compiler ZCC
  - **RVV 0.71** auto-vectorization
  - SPECInt2K6 score: **20%** better than stock compiler

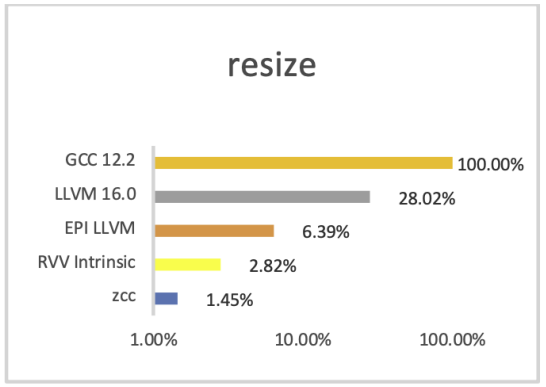
The first 64-core RISC-V CPU in production

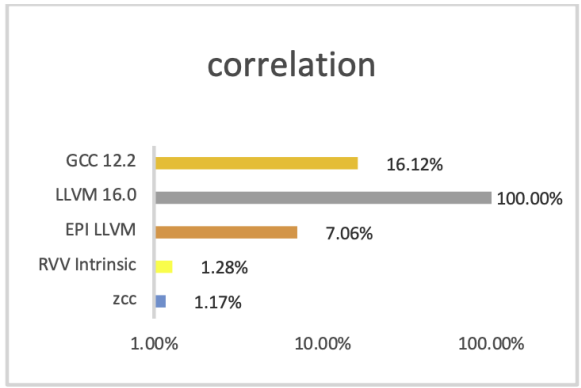
Auto-vectorization on AI kernels - ZCC

- Dynamic instruction count
  - Up to 100 times faster than open source compilers
  - **18%** better than RVV builtins in average
- Cycle count
  - **50%** cycle count improvements in correlation over RVV builtins.









*relative dynamic instruction count, smaller is better

RV32IMCV, data path: 512bits, Kernel source: https://github.com/dodohack/rv_lib

Inner loop auto-vec – reduction sum generation

```
// A[M*N] = B[M*K] * C[K*N]
void mat_mult(float *restrict A,
              float *B, float *C,
              int M, int K, int N) {
  for (int row = 0; row < M; ++row)
    for (int col = 0; col < N; ++col) {
      float sum = 0;
      for (int i = 0; i < K; ++i)
        sum += B[row*K + i] * C[i*N + col];
      A[row*N + col] = sum;
    }
}
```

```
# %bb.5:                                # %vector.body.preheader
                                        #   in Loop: Header=BB0_4 Depth=2
  li  t5, 0
  fmv.s ft1, ft0
.LBB0_6:                                # %vector.body
                                        #   Parent Loop BB0_2 Depth=1
                                        #     Parent Loop BB0_4 Depth=2
                                        # =>    This Inner Loop Header: Depth=3
  sub t6, t0, t5
  vsetvli t6, t6, e32, m8, ta, mu
  mul s0, t5, a5
  add s0, s0, t2
  slli  s0, s0, 2
  add s0, s0, a2
  vlse32.v  v8, (s0), t1
  add s0, t5, t3
  slli  s0, s0, 2
  add s0, s0, a1
  vle32.v v16, (s0)
  vfmul.vv  v8, v16, v8
  vsetivli  zero, 1, e32, m1, ta, mu
  vfmv.s.f  v16, ft1
  vsetvli zero, t6, e32, m8, tu, mu
  vfredosum.vs  v16, v8, v16
  add t5, t5, t6
  vfmv.f.s  ft1, v16
  bne t5, t0, .LBB0_6
.LBB0_7:                                # %for.cond.cleanup7
                                        #   in Loop: Header=BB0_4 Depth=2
```

Stride load (Matrix C) + Unit stride load (Matrix B)

Reduction ordered sum

Inner loop auto-vec – reduction sum generation

```
// A[M*N] = B[M*K] * C[K*N]
void mat_mult(float *restrict A,
              float *B, float *C,
              int M, int K, int N) {
  for (int row = 0; row < M; ++row)
    for (int col = 0; col < N; ++col) {
      float sum = 0;
      for (int i = 0; i < K; ++i)
        sum += B[row*K + i] * C[i*N + col];
      A[row*N + col] = sum;
    }
}
```

```
# %bb.5:                                # %vector.body.preheader
                                        #   in Loop: Header=BB0_4 Depth=2
  li  t5, 0
  fmv.s ft1, ft0
.LBB0_6:                                # %vector.body
                                        #   Parent Loop BB0_2 Depth=1
                                        #     Parent Loop BB0_4 Depth=2
                                        # =>    This Inner Loop Header: Depth=3
  sub t6, t0, t5
  vsetvli t6, t6, e32, m8, ta, mu
  mul s0, t5, a5
  add s0, s0, t2
  slli  s0, s0, 2
  add s0, s0, a2
  vlse32.v  v8, (s0), t1
  add s0, t5, t3
  slli  s0, s0, 2
  add s0, s0, a1
  vle32.v v16, (s0)
  vfmul.vv  v8, v16, v8
  vsetivli  zero, 1, e32, m1, ta, mu
  vfmv.s.f  v16, ft1
  vsetvli zero, t6, e32, m8, tu, mu
  vfredosum.vs  v16, v8, v16
  add t5, t5, t6
  vfmv.f.s  ft1, v16
  bne t5, t0, .LBB0_6
.LBB0_7:                                # %for.cond.cleanup7
                                        #   in Loop: Header=BB0_4 Depth=2
```

Stride load (Matrix C) + Unit stride load (Matrix B)

Not so good

Reduction ordered sum

Good

Outer loop auto-vec – memory access pattern opts

```
// A[M*N] = B[M*K] * C[K*N]
void mat_mult(float *restrict A,
              float *B, float *C,
              int M, int K, int N) {
  for (int row = 0; row < M; ++row)
    #pragma clang loop vectorize(assume_safety)
    for (int col = 0; col < N; ++col) {
      float sum = 0;
      for (int i = 0; i < K; ++i)
        sum += B[row*K + i] * C[i*N + col];
      A[row*N + col] = sum;
    }
}
```

```
.LBB0_4:                                # %vector.body
                                        #   Parent Loop BB0_2 Depth=1
                                        # =>  This Loop Header: Depth=2
                                        #       Child Loop BB0_6 Depth 3
  sub s1, a7, t3
  vsetvli t5, s1, e32, m8, ta, mu
  blez  a4, .LBB0_7
# %bb.5:                                # %for.body863.preheader
                                        #   in Loop: Header=BB0_4 Depth=2
  slli  s1, t3, 2
  add t6, a2, s1
  vsetvli s0, zero, e32, m8, ta, mu
  vmv.v.i v8, 0
  mv  s0, a1
  mv  s1, t0
.LBB0_6:                                # %for.body863
                                        #   Parent Loop BB0_2 Depth=1
                                        #     Parent Loop BB0_4 Depth=2
                                        # =>    This Inner Loop Header: Depth=3
  vsetvli zero, t5, e32, m8, ta, mu
  vle32.v v16, (t6)
  flw ft0, 0(s0)
  vfmacc.vf v8, ft0, v16
  addi  s1, s1, -1
  addi  s0, s0, 4
  add t6, t6, t2
  bnez  s1, .LBB0_6
  j .LBB0_8
.LBB0_7:                                #   in Loop: Header=BB0_4 Depth=2
  vsetvli s1, zero, e32, m8, ta, mu
  vmv.v.i v8, 0
.LBB0_8:                                # %for.cond.cleanup770
                                        #   in Loop: Header=BB0_4 Depth=2
  add s1, t3, t4
  slli  s1, s1, 2
  add s1, s1, a0
  vsetvli zero, t5, e32, m8, ta, mu
  add t3, t3, t5
  vse32.v v8, (s1)
  bne t3, a7, .LBB0_4
```

Unit stride load (Matrix C) + scalar load (Matrix B)

Good

inner loop

outer loop

Correlation kernel – RVV builtins vs auto-vec

```
for (size_t w = width; (vl = vsetvl_e8m2(w)); w -= vl) {
  for (size_t h = 0; h < height; h++) {
    for (size_t d = 0; d < out_channel; d++) {
      src0_ptr = h * width + vl_cnt + d + src0_arr;
      src1_ptr = h * width + vl_cnt + src1_arr;
      out_ptr = h * width + vl_cnt + d * fm_size + out_arr;
      vl0 = vsetvl_e8m2(w - d);
      vint16m4_t acc = vmv_v_x_i16m4(0, vl0);
      vint8m2_t vx, vy;
      for (size_t c = 0; c < in_channel; c++) {
        vx = vle8_v_i8m2(src0_ptr, vl0);
        vy = vle8_v_i8m2(src1_ptr, vl0);
        acc = vwmacc_vv_i16m4(acc, vy, vx, vl0);
        src0_ptr += fm_size;
        src1_ptr += fm_size;
      }
      vint8m2_t acc_sra = vnsra_wx_i8m2(acc, out_shift, vl0);
      vse8_v_i8m2(out_ptr + d, acc_sra, vl0);
    }
  }
  vl_cnt += vl;
}
```

Correlation kernel written in RVV builtins

```
for (size_t i = 0; i < height; ++i) {
  for (size_t d = 0; d < out_channel; ++d) {
    #pragma clang loop vectorize(assume_safety)
    for (size_t j = d; j < width; j++) {
      out_idx = d * width * height + i * width + j;
      int16_t sum_data = 0;
      for (size_t k = 0; k < in_channel; ++k) {
        in_idx1 = k * width * height + i * width + j;
        in_idx2 = k * width * height + i * width + j - d;
        sum_data += (int16_t)src0_arr[in_idx1] * src1_arr[in_idx2];
      }
      out_arr[out_idx] = (int8_t)(sum_data >> out_shift);
    }
  }
}
```

Correlation kernel marked with memory safety #pragma on outer loop

Correlation kernel - assembly code

```
.LBB0_7:
  sub a0, s4, s0
  vsetvli a5, a0, e16, m4, ta, mu
  vmv.v.i v8, 0
  mv  a4, s2
  mv  a0, s5
.LBB0_8:
  add s1, a1, a0
  vsetvli zero, a5, e8, m2, ta, mu
  vle8.v  v12, (s1)
  add s1, a2, a0
  vle8.v  v14, (s1)
  vsetvli zero, zero, e8, m2, tu, mu
  vwmacc.vv v8, v14, v12
  addi  a4, a4, -1
  add a0, a0, a3
  bnez  a4, .LBB0_8
# %bb.9:
  mul a0, s0, a3
  add a0, a0, s6
  add a0, a0, t5
  vsetvli zero, zero, e8, m2, ta, mu
  vnsra.wx  v12, v8, s3
  add a0, a0, s0
  vse8.v  v12, (a0)
  addi  s0, s0, 1
  addi  a1, a1, 1
  bne s0, t3, .LBB0_7
```

ASM of Correlation kernel written in RVV builtins

← Almost identical →

```
.LBB0_7:
  sub a3, t6, a0
  vsetvli a5, a3, e8, m4, ta, mu
  vsetvli a3, zero, e16, m8, ta, mu
  vmv.v.i v8, 0
  mv  a3, a0
  mv  s0, t1
.LBB0_8:
  add s1, a1, a3
  vsetvli zero, a5, e8, m4, ta, mu
  vle8.v  v16, (s1)
  add s1, a2, a3
  vle8.v  v20, (s1)
  vwmacc.vv v8, v20, v16
  addi  s0, s0, -1
  add a3, a3, a4
  bnez  s0, .LBB0_8
# %bb.9:
  add a3, t5, a0
  add a3, a3, t3
  add a3, a3, t4
  vnsra.wx  v16, v8, t2
  add a3, a3, s4
  add a0, a0, a5
  vse8.v  v16, (a3)
  bne a0, t6, .LBB0_7
```

ASM of Correlation kernel marked with memory safety #pragma on outer loop

Summary

- Ease the way to write and maintain high performance libraries with RVV auto-vec.
- ZCC generates pretty good code for inner loop auto-vec.
  - Control flows are flatten into predicted execution.
- ZCC only supports trivial outer loop[1] auto-vec with #pragma.
  - Control flows are kept unchanged.
  - Stride load can be turned into unit stride load by outer loop vectorization.
  - TODO: Simple to Complex loops auto-vec.
- TODO: Polly + auto-vec to improve data locality and performance.

1. https://lists.llvm.org/pipermail/llvm-dev/2017-December/119523.html

THANKS

aries.wu@terapines.com