Vectorizing with RVV

Tools, Metrics, Tips and tricks

Objectives of the lab

- Learn how to autovectorize codes using the RISC-V Vector Extension (RVV)

LLVM-based compiler

COMPILER EXPLORER

Online tool to compile snippets of code

RAVE

RISC-V Analyzer of Vector Executions

- Evaluate the quality of the vectorization

- Learn common code techniques that improve vectorization

Organization

- First part → We show you how to use the tools
- Break
- Second part → We give you a set of challenges

Some reminders

From this morning's classes:

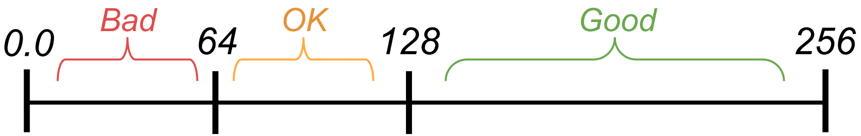
- The **Vector-Length** changes during execution → It's important to keep it high!
- RVV instruction can be **SEW8**, **SEW16**, **SEW32**, **SEW64**
- There's three memory access modes: **Unit Strided, Strided, Indexed**
- **Vector Masks** for conditionals are possible, but expensive.

Some vectorization metrics

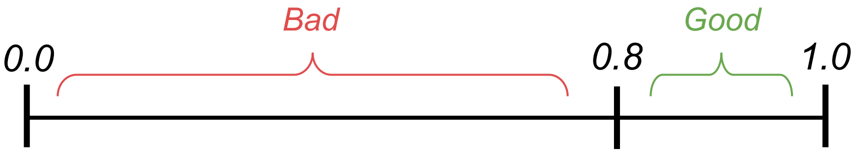
- We mainly look at four metrics:

Vector Mix: $\dfrac{\#\text{Vec. Instr}}{\#\text{Tot. Instr}}$ →

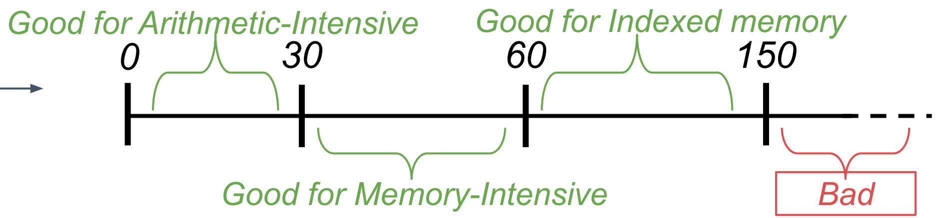
Avg. VL: $\dfrac{\#\text{Vec. Elem}}{\#\text{Vec. Instr}}$ →



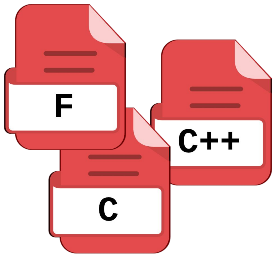
Vector Activity: $\dfrac{\#\text{Vec. Cyc}}{\#\text{Tot. Cyc}}$ →



Vector CPI: $\dfrac{\#\text{Vec. Cyc}}{\#\text{Vec. Instr}}$ →



LLVM-Based compiler



- BSC develops a vectorizing compiler that supports C, C++, and Fortran

- The Compiler can vectorize on its own, with assistance, or using intrinsics:

```c
for (int i=0; i<N; ++i){
  B[i] = A[i];
}
```

```c
#pragma clang loop vectorize(enable)
for (int i=0; i<N; ++i){
  B[i] = A[i];
}
```

```c
for (int i=0; i<N; ){
  long vl = __builtin_epi_vsetvl(N-i,e64,m1);
  __epi_1xf64 va = __builtin_epi_vload_1xf64(&A[i],vl);
  __builtin_epi_vstore_1xf64(&B[i], va, vl);
  i += vl;
}
```

```
src/friendly.c:33:2: remark: vectorized loop (vectorization width: vscale x 1, interleaved count: 1) [-Rpass=loop-vectorize]
src/friendly.c:43:5: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]
          for(long j=block_j; j<block_j+BLOCK_DIM_X; ++j){
          ^
src/friendly.c:43:5: remark: vectorized loop (vectorization width: vscale x 1, interleaved count: 1) [-Rpass=loop-vectorize]
src/friendly.c:56:2: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]
        for(int ij=0; ij<N*M; ++ij){
        ^
src/friendly.c:56:2: remark: vectorized loop (vectorization width: vscale x 1, interleaved count: 1) [-Rpass=loop-vectorize]
src/friendly.c:65:22: remark: loop not vectorized: call instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]
                array[i*M + j] = (rand()%1024) / 1024.0;
```

Compiler versions

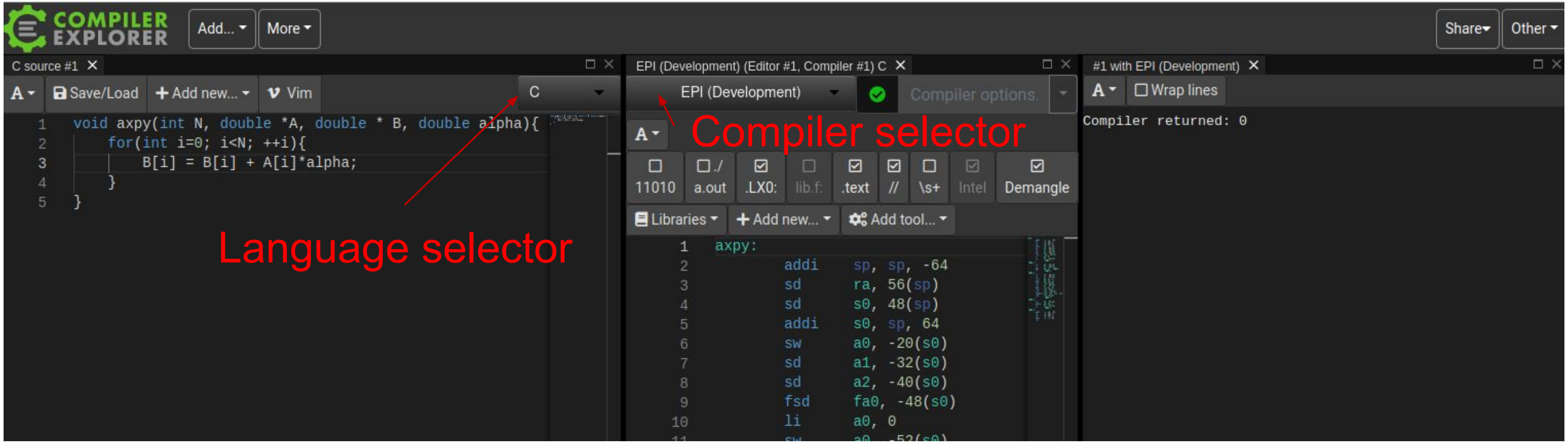
- Currently, two specs of the RVV extension coexist in our systems:

Spec	Runs in emulation (VEHAVE / RAVE)	Runs in hardware (FPGA)	Compiles C/C++	Compiles Fortran
RVV0.7	✅	✅	✅	❌
RVV1.0	✅	🏗️	✅	✅

- The compiler for RVV1.0 is more mature
- But the spec is not fully supported in hardware
- For compilation tests and emulation, we will mostly use RVV1.0

Using the compiler explorer

- Let's jump into this compiler explorer https://repo.hca.bsc.es/epic/z/xRyX3A
- You will find something like this:



Code

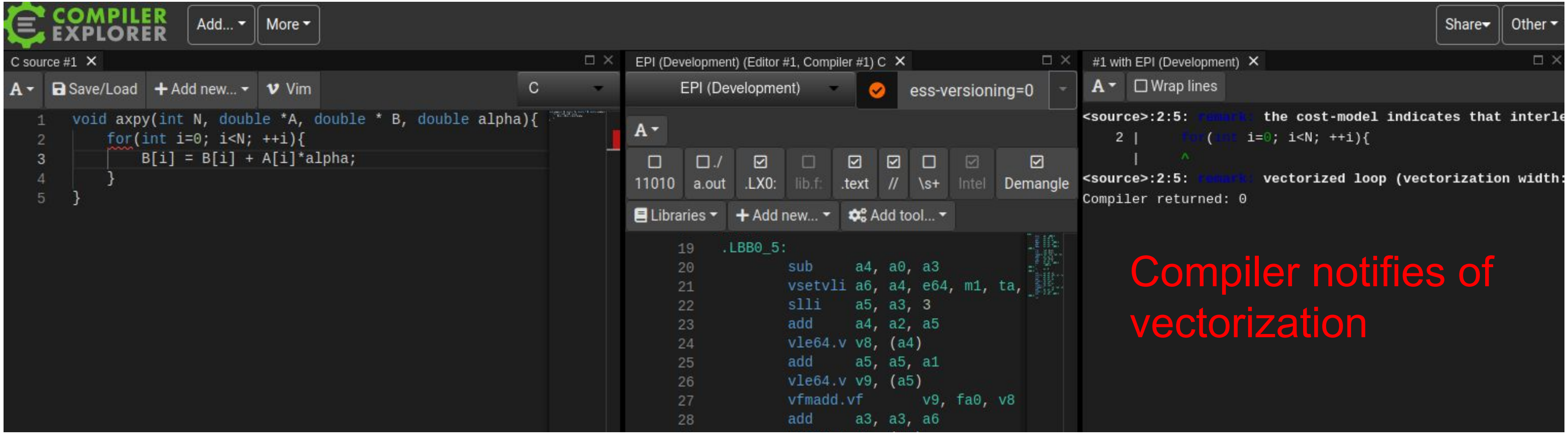
Compiled Assembly

Compiler messages

Autovectorizing the code

- We recommend using these flags:

```
-O3 -ffast-math -mepi -mcpu=avispado -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize
-mllvm -vectorizer-use-vp-strided-load-store -mllvm -disable-loop-idiom-memcpy
-mllvm -combiner-store-merging=0 -fno-slp-vectorize -mllvm -enable-mem-access-versioning=0
```



Vec instructions in the assembly

Helping the compiler

- Some loops might not vectorize even with flags: https://repo.hca.bsc.es/epic/z/krL65-
- But adding the pragma #pragma clang loop vectorize (assume_safety) may help:

```
#1 ✕
Save/Load   + Add new...   Vim

void indirect(int N, double *A, double * B, int * index){
    for(int i=0; i<N; ++i){
        A[i] = B[index[i]];
    }
}

void indirect_fix(int N, double *A, double * B, int * index){
    #pragma clang loop vectorize(assume_safety)
    for(int i=0; i<N; ++i){
        A[i] = B[index[i]];
    }
}
```

```
indirect:
        blez    a0, .LBB0_3
        slli    a0, a0, 3
        add     a0, a0, a1
.LBB0_2:
        lw      a4, 0(a3)
        slli    a4, a4, 3
        add     a4, a4, a2
        fld     fa5, 0(a4)
        fsd     fa5, 0(a1)
        addi    a1, a1, 8
        addi    a3, a3, 4
        bne     a1, a0, .LBB0_2
.LBB0_3:
        ret
indirect_fix:
        blez    a0, .LBB1_3
        li      a6, 0
.LBB1_2:
        sub     a5, a0, a6
        vsetvli a5, a5, e64, m1, ta, ma
        slli    a4, a6, 2
        add     a4, a4, a3
        vle32.v v8, (a4)
        vsext.vf2       v9, v8
        vsll.vi v8, v9, 3
        vluxei64.v      v8, (a2), v8
        slli    a4, a6, 3
```

```
#1 with EPI (Development) ✕
A   Wrap lines
<source>:3:16: remark: loop not vectorized:
    3 |         A[i] = B[index[i]];
      |                ^
<source>:9:5: remark: the cost-model indicat
    9 |     for(int i=0; i<N; ++i){
      |     ^
<source>:9:5: remark: vectorized loop (vecto
Compiler returned: 0
```

Vectorizing manually

- You can also use the Compiler Explorer to test manual vectorization.
- Intrinsics for RVV 0.7.1 and 1.0 differ slightly:
  - 0.7: https://admin.hca.bsc.es/epi/ftp/doc/intrinsics/EPI-0.7/epi-intrinsics.html
  - 1.0: https://admin.hca.bsc.es/epi/ftp/doc/intrinsics/EPI/epi-intrinsics.html

```
void axpy(int N, double * X, double * Y, double alpha){
    for(int i=0; i<N; ++i){
        Y[i] += X[i]*alpha;
    }
}
```

https://repo.hca.bsc.es/epic/z/a_VvQK

```
void axpy_intrinsics(int N, double * X, double * Y, double alpha){
    for(int i=0; i<N;){
        long gvl = __builtin_epi_vsetvl(N-i, __epi_e64, __epi_m1);
        __epi_1xf64 vx = __builtin_epi_vload_1xf64(&X[i], gvl);
        __epi_1xf64 vy = __builtin_epi_vload_1xf64(&Y[i], gvl);
        __epi_1xf64 va = __builtin_epi_vfmv_v_f_1xf64(alpha, gvl);
        __epi_1xf64 vr = __builtin_epi_vfmacc_1xf64(vy, va, vx, gvl);
        __builtin_epi_vstore_1xf64(&Y[i], vr, gvl);
        i+=gvl;
    }
}
```

Intrinsics: Use case

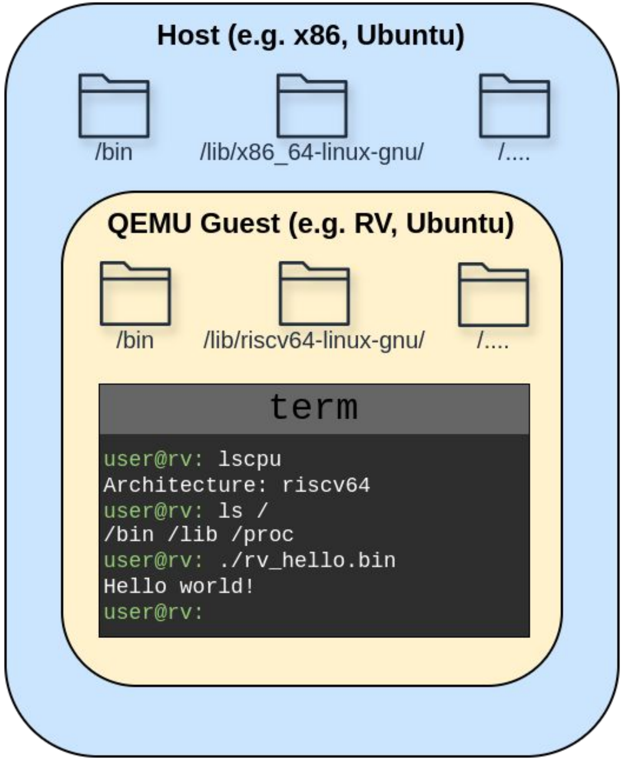
- Intrinsics can be useful to enable loop unrolling:

https://repo.hca.bsc.es/epic/z/a_VvQK

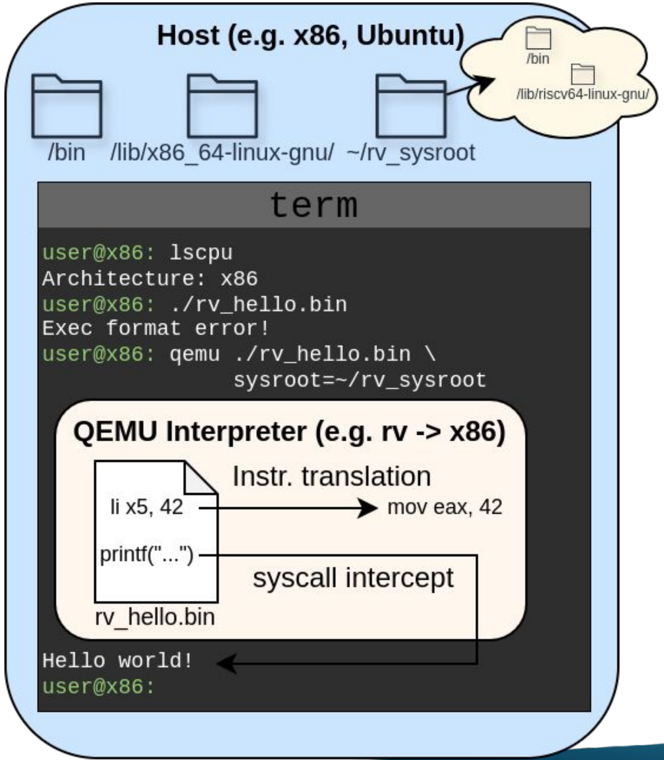
```c
void axpy_unroll(int N, double * X, double * Y, double alpha){
    long maxvl = __builtin_epi_vsetvl(N, __epi_e64, __epi_m1);
    int i;
    for(i=0; i<=N-maxvl*4;){
        __epi_1xf64 vx1 = __builtin_epi_vload_1xf64(&X[i], maxvl);
        __epi_1xf64 vy1 = __builtin_epi_vload_1xf64(&Y[i], maxvl);
        __epi_1xf64 va = __builtin_epi_vfmv_v_f_1xf64(alpha, maxvl);
        __epi_1xf64 vr1 = __builtin_epi_vfmacc_1xf64(vy1, va, vx1, maxvl);

        __epi_1xf64 vx2 = __builtin_epi_vload_1xf64(&X[i+maxvl], maxvl);
        __epi_1xf64 vy2 = __builtin_epi_vload_1xf64(&Y[i+maxvl], maxvl);
        __epi_1xf64 vr2 = __builtin_epi_vfmacc_1xf64(vy2, va, vx2, maxvl);

        __epi_1xf64 vx3 = __builtin_epi_vload_1xf64(&X[i+maxvl*2], maxvl);
        __epi_1xf64 vy3 = __builtin_epi_vload_1xf64(&Y[i+maxvl*2], maxvl);
        __epi_1xf64 vr3 = __builtin_epi_vfmacc_1xf64(vy3, va, vx3, maxvl);

        __epi_1xf64 vx4 = __builtin_epi_vload_1xf64(&X[i+maxvl*3], maxvl);
        __epi_1xf64 vy4 = __builtin_epi_vload_1xf64(&Y[i+maxvl*3], maxvl);
        __epi_1xf64 vr4 = __builtin_epi_vfmacc_1xf64(vy4, va, vx4, maxvl);

        __builtin_epi_vstore_1xf64(&Y[i], vr1, maxvl);
        __builtin_epi_vstore_1xf64(&Y[i+maxvl*2], vr2, maxvl);
        __builtin_epi_vstore_1xf64(&Y[i+maxvl*3], vr3, maxvl);
        __builtin_epi_vstore_1xf64(&Y[i+maxvl*4], vr4, maxvl);

        i += maxvl*4;
    }
    for(; i<N; ++i){ //Unroll tail
        Y[i] += X[i]*alpha;
    }
}
```

What is QEMU? a software emulator

System-level emulation

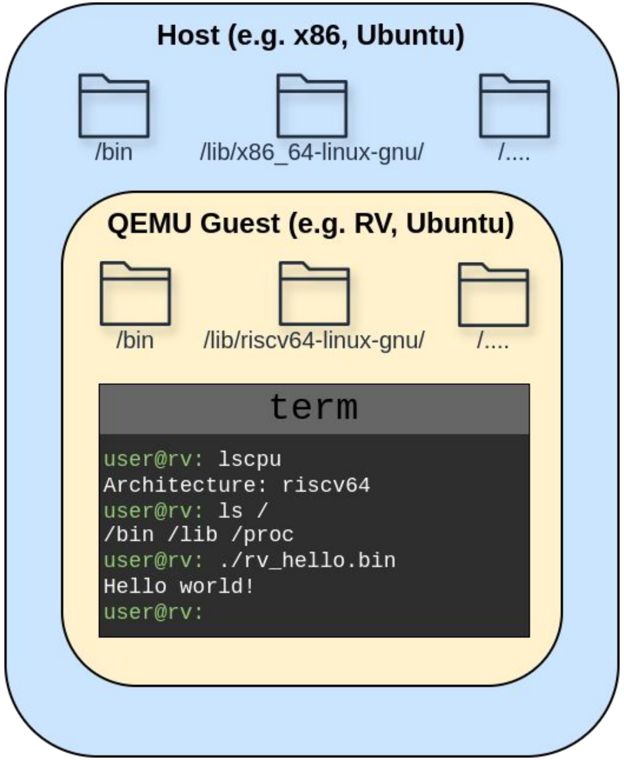


User-level emulation

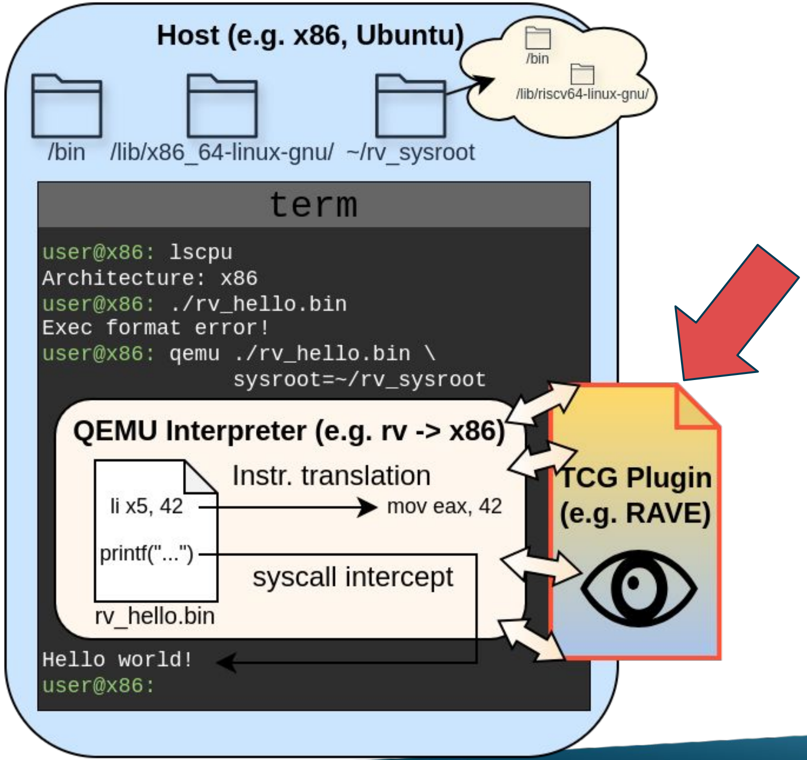


What is RAVE? an analysis/profiling plugin

System-level emulation



User-level emulation



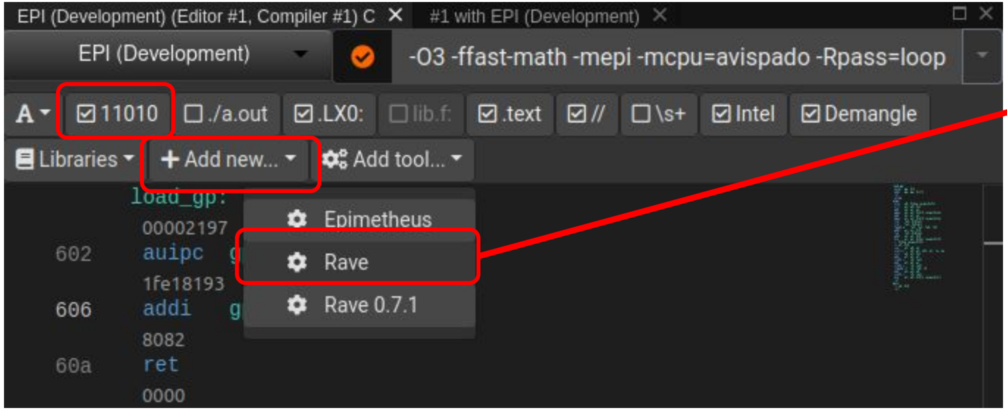
What is RAVE useful for?

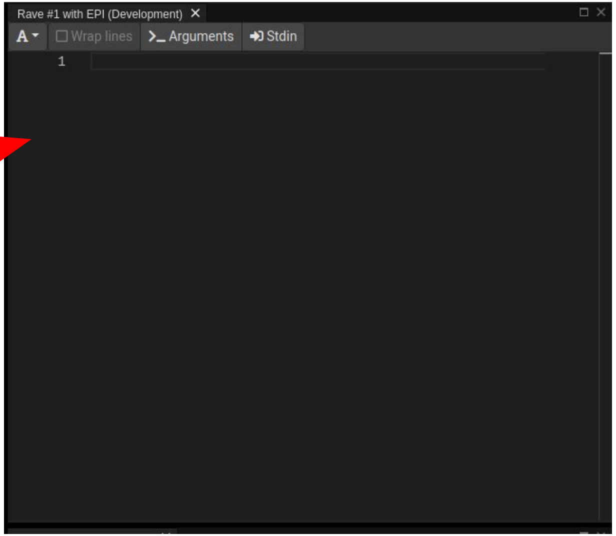
- **RAVE monitors** and **counts** metrics such as:
  - Number of emulated scalar and vector **instructions** *(you can compute Vec.Mix)*
    - Divided by type (Memory, Arithmetic, Mask, stride type, SEW, …)
  - Average Vector Length **(VL)**
  - Number of **bytes load/stored** with scalar/vector instructions
  - Program Counter **(PC)**

- RAVE provides:
  - **API** called for user application to instrument regions of interest
  - Generation of **reports/logs** at the end of the emulation
  - Generation of **Paraver traces** (BSC's format for traces)

Using RAVE in the compiler explorer (I)

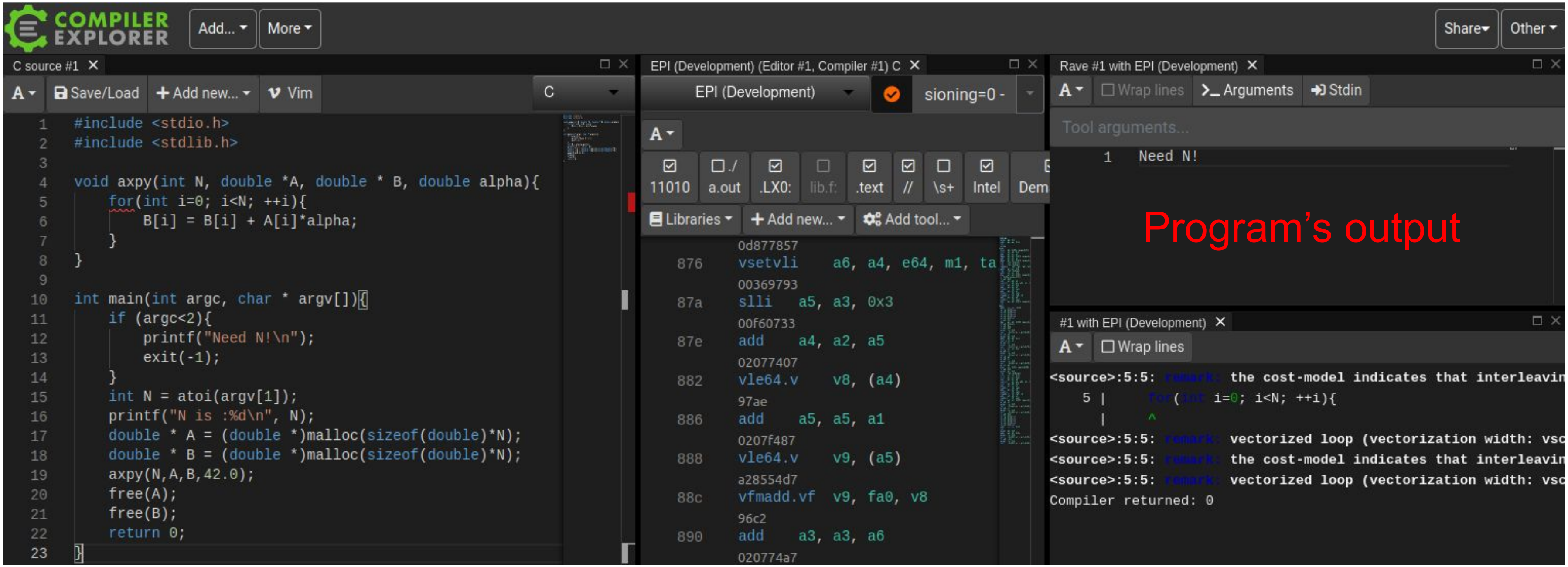
- You need to clock **Add tool** → **Rave** (or Rave 0.7.1 for RVV0.7)
- And click the **“11010”** checkbox to generate a binary
- We suggest adding the flag *“-mllvm -riscv-uleb128-reloc=0”* on EPI 1.0





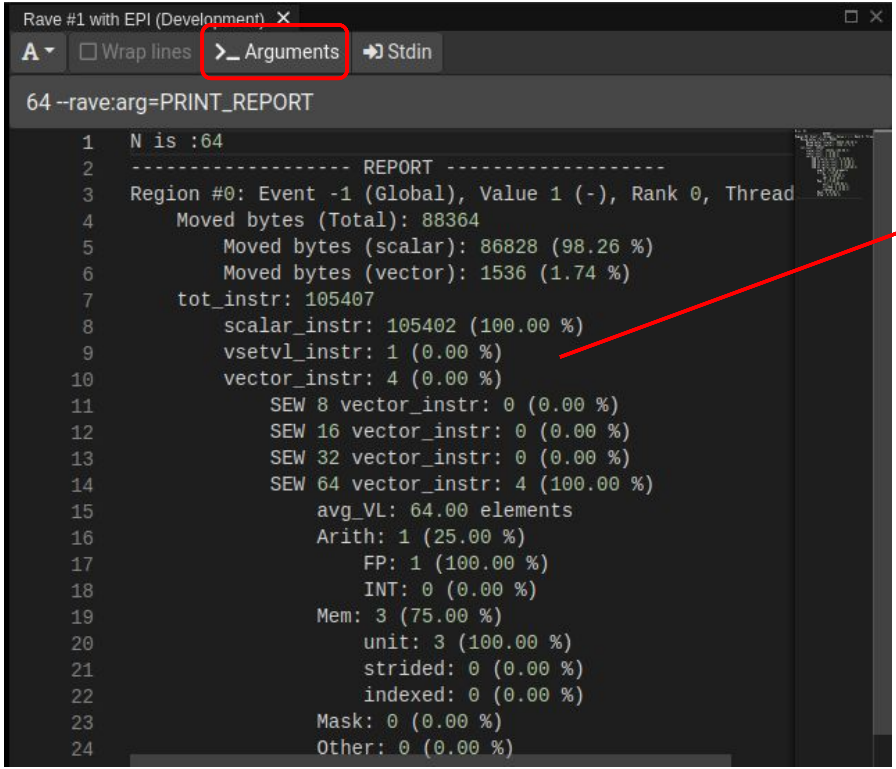
Using RAVE in the compiler explorer (II)

- Your program needs to have a “main” function in order to run:



Using RAVE in the compiler explorer (III)

- You can pass arguments to your program and to RAVE clicking on **“>_Arguments”**



Virtually all instructions are scalar (N is small, and we count **everything**)

Better to count only regions of interest!

Test it here: https://repo.hca.bsc.es/epic/z/oEJt_P

Using RAVE in the compiler explorer (IV)

- To use instrumentation, include the “rave_user_functions.h” header.
- Add the flag “-I/apps/qemu-rave/interfaces” for C and C++ compilation
- For Fortran, add “-I/apps/qemu-rave/interfaces /apps/qemu-rave/interfaces/rave_user_events_f.o”

```c
#include “rave_user_functions.h”
int main(){
  rave_name_event(1000,"Code Region")
  rave_name_value(1000, 1, "Ini")
  rave_name_value(1000, 2, "Compute")

  double array1[256], array2[256], array3[256];

  rave_event_and_value(1000, 1)
  ini_vectors(array1, array2, array3);
  rave_event_and_value(1000, 0)

  rave_event_and_value(1000, 2)
  for(int i=0; i<256; ++i)
     array3[i] += array1[i] + array2[i];
  rave_event_and_value(1000,0)
};
```

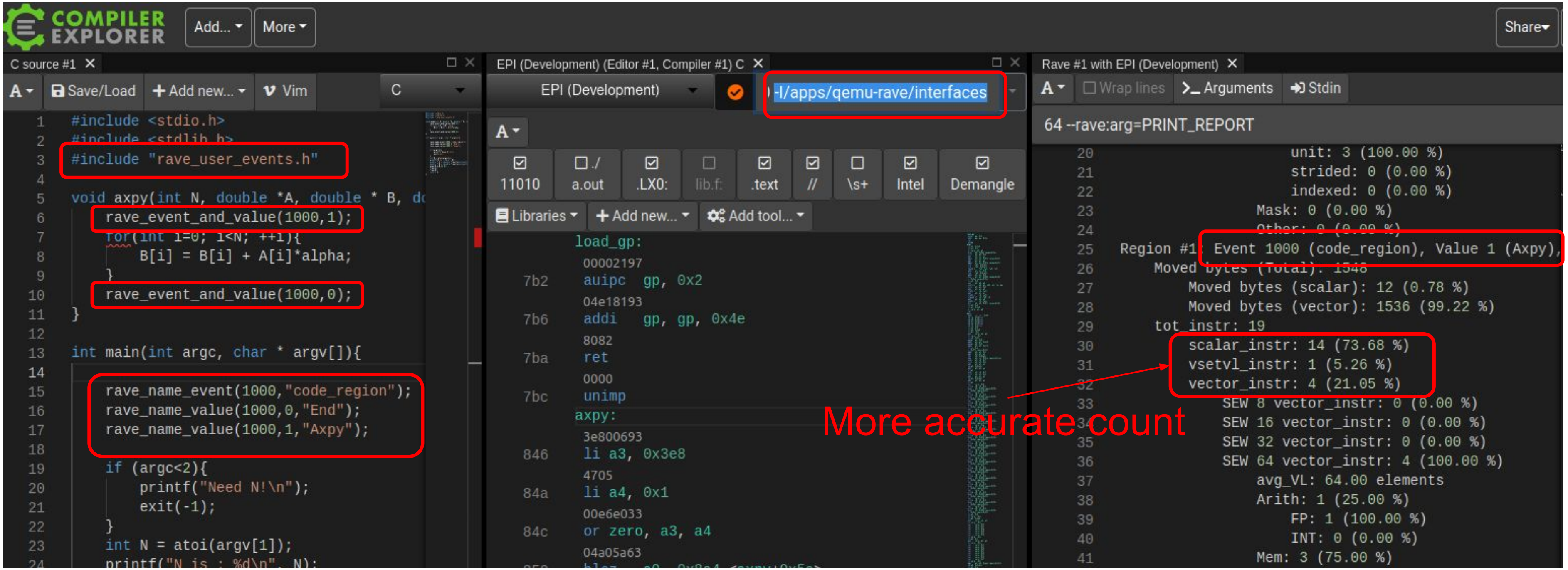
Define event 1000 = “Code Region”
Value 1 = “Ini”
Value 2 = “Compute”

Enclose first region with value 1 (“Ini”)

Enclose second region with value 2 (“Compute”)

Using RAVE in the compiler explorer (IV)

- Try it on https://repo.hca.bsc.es/epic/z/gqeLds



Using RAVE to improve the Vector Mix

- Sometimes the vector mix is lower than expected:

```
rave_event_and_value(1000,1);
for(int s=0; s<timesteps; ++s){
    for(int block_i=1; block_i<N-BY; block_i+=BY){
        for(int block_j=1; block_j<M-BX; block_j+=BX){
            for(int i=block_i; i<block_i+BY; ++i){
                for(int j=block_j; j<block_j+BX; ++j){
                    new_T[i*M + j] = 0.25*(T[M*i + j + 1]
                                         + T[M*(i+1) + j]
                                         + T[M*i + (j-1)]
                                         + T[M*(i-1) + j]);
                }
            }
        }
    }
    double * tmp = T;
    T = new_T;
    new_T = tmp;
}
rave_event_and_value(1000,0);
```

```
Region #1: Event 1000 (code_region), Value 1 (Simulation),
    Moved bytes (Total): 205482209
        Moved bytes (scalar): 682209 (0.33 %)
        Moved bytes (vector): 204800000 (99.67 %)
    tot_instr: 20006449
        scalar_instr: 18406449 (92.00 %)
        vsetvl_instr: 160000 (0.80 %)
        vector_instr: 1440000 (7.20 %)
```

Using RAVE to improve the Vector Mix

https://repo.hca.bsc.es/epic/z/ChTspc

- Using a pragma might help

```
for(int block_i=1; block_i<N-BY; block_i+=BY){
    for(int block_j=1; block_j<M-BX; block_j+=BX){
        for(int i=block_i; i<block_i+BY; ++i){
            #pragma clang loop vectorize(assume_safety)
            for(int j=block_j; j<block_j+BX; ++j){
                new_T[i*M + j] = 0.25*(T[M*i + j + 1]
                                     + T[M*(i+1) + j]
                                     + T[M*i + (j-1)]
                                     + T[M*(i-1) + j]);
            }
        }
    }
}
```

```
Region #1: Event 1000 (code_region), Value 1 (Simulati
  Moved bytes (Total): 205472209
    Moved bytes (scalar): 672209 (0.33 %)
    Moved bytes (vector): 204800000 (99.67 %)
  tot_instr: 9079342
    scalar_instr: 7479342 (82.38 %)
    vsetvl_instr: 160000 (1.76 %)
    vector_instr: 1440000 (15.86 %)
      SEW 8 vector_instr: 0 (0.00 %)
      SEW 16 vector_instr: 0 (0.00 %)
      SEW 32 vector_instr: 0 (0.00 %)
```

- And a “long” induction variable instead of “int”

```
for(int block_i=1; block_i<N-BY; block_i+=BY){
    for(int block_j=1; block_j<M-BX; block_j+=BX){
        for(int i=block_i; i<block_i+BY; ++i){
            #pragma clang loop vectorize(assume_safety)
            for(long j=block_j; j<block_j+BX; ++j){
                new_T[i*M + j] = 0.25*(T[M*i + j + 1]
                                     + T[M*(i+1) + j]
                                     + T[M*i + (j-1)]
                                     + T[M*(i-1) + j]);
            }
        }
    }
}
```

```
Region #1: Event 1000 (code_region), Value 1 (Simulati
  Moved bytes (Total): 205304209
    Moved bytes (scalar): 504209 (0.25 %)
    Moved bytes (vector): 204800000 (99.75 %)
  tot_instr: 5829017
    scalar_instr: 4229017 (72.55 %)
    vsetvl_instr: 160000 (2.74 %)
    vector_instr: 1440000 (24.70 %)
      SEW 8 vector_instr: 0 (0.00 %)
      SEW 16 vector_instr: 0 (0.00 %)
      SEW 32 vector_instr: 0 (0.00 %)
```

Using RAVE to improve the Average VL

- Only the inner-most loop gets vectorized:

https://repo.hca.bsc.es/epic/z/yPhD16

```c
void Matrix_Add(int N, int M, double * A, double * B, double * C){
    rave_event_and_value(1000,1);
    for(int i=0; i<N; ++i){
        for(int j=0; j<M; ++j){
            C[i*M + j] = A[i*M + j] + B[i*M + j];
        }
    }
    rave_event_and_value(1000,0);
}
```

```
        tot_instr: 474
            scalar_instr: 345 (72.78 %)
            vsetvl_instr: 1 (0.21 %)
            vector_instr: 128 (27.00 %)
                SEW 8 vector_instr: 0 (0.00 %)
                SEW 16 vector_instr: 0 (0.00 %)
                SEW 32 vector_instr: 0 (0.00 %)
                SEW 64 vector_instr: 128 (100.00 %)
                    avg_VL: 32.00 elements
                    Arith: 32 (25.00 %)
```

- Loop collapsing is a common technique to increase the VL:

```c
void Matrix_Add_ij(int N, int M, double* A, double* B, double* C){
    rave_event_and_value(1000,2);
    for(int ij=0; ij<N*M; ++ij){
        C[ij] = A[ij] + B[ij];
    }
    rave_event_and_value(1000,0);
}
```

```
        tot_instr: 51
            scalar_instr: 31 (60.78 %)
            vsetvl_instr: 4 (7.84 %)
            vector_instr: 16 (31.37 %)
                SEW 8 vector_instr: 0 (0.00 %)
                SEW 16 vector_instr: 0 (0.00 %)
                SEW 32 vector_instr: 0 (0.00 %)
                SEW 64 vector_instr: 16 (100.00 %)
                    avg_VL: 256.00 elements
                    Arith: 4 (25.00 %)
```

Using RAVE to improve to study Memory Mix

- We can study the stride of the memory accesses and try to remove strides:

```
struct RGB{
    double R;
    double G;
    double B;
};
struct RGB pixels[1024];
void Compute_Brightness_AoS(double * brightness){
    rave_event_and_value(1000,1);
    for(int i=0; i<1024; ++i){
        brightness[i] = pixels[i].R * pixels[i].G * pixels[i].B;
    }
    rave_event_and_value(1000,0);
}
```

```
Mem: 16 (66.67 %)
    unit: 4 (25.00 %)
    strided: 12 (75.00 %)
        Avg. Stride (B): 24.00
    indexed: 0 (0.00 %)
```

```
struct RGB_arr{
    double R[1024];
    double G[1024];
    double B[1024];
};
struct RGB_arr pixels_arr;
void Compute_Brightness_SoA(double * brightness){
    rave_event_and_value(1000,2);
    for(int i=0; i<1024; ++i){
        brightness[i] = pixels_arr.R[i] * pixels_arr.G[i] * pixels_arr.B[i]
    }
    rave_event_and_value(1000,0);
}
```

```
Mem: 16 (66.67 %)
    unit: 16 (100.00 %)
    strided: 0 (0.00 %)
    indexed: 0 (0.00 %)
```

https://repo.hca.bsc.es/epic/z/5u3hQj

Using RAVE to detect inefficiencies

- Indexed operations often require mixing datatypes, which introduces “Other” instructions

```
void Shuffle(int N, int * IDX, double * VAL, double * OLD){
    rave_event_and_value(1000,1);
    #pragma clang loop vectorize(assume_safety)
    for(int i=0; i<N; ++i){
        VAL[i] = OLD[IDX[i]];
    }
    rave_event_and_value(1000,0);
}
```

```
SEW 32 vector_instr: 64 (28.57 %)
    avg_VL: 384.00 elements
    Arith: 32 (50.00 %)
        FP: 0 (0.00 %)
        INT: 32 (100.00 %)
    Mem: 16 (25.00 %)
        unit: 16 (100.00 %)
        strided: 0 (0.00 %)
        indexed: 0 (0.00 %)
    Mask: 0 (0.00 %)
    Other: 16 (25.00 %)
SEW 64 vector_instr: 160 (71.43 %)
    avg_VL: 256.00 elements
    Arith: 32 (20.00 %)
        FP: 0 (0.00 %)
        INT: 32 (100.00 %)
    Mem: 96 (60.00 %)
        unit: 64 (66.67 %)
        strided: 0 (0.00 %)
        indexed: 32 (33.33 %)
    Mask: 0 (0.00 %)
    Other: 32 (20.00 %)
```

Using RAVE to detect inefficiencies

- Setting everything to the same data size helps:

```
void Shuffle_64(int N, size_t * IDX, double * VAL, double * OLD){
    rave_event_and_value(1000,2);
    #pragma clang loop vectorize(assume_safety)
    for(int i=0; i<N; ++i){
        VAL[i] = OLD[IDX[i]];
    }
    rave_event_and_value(1000,0);
}
```

https://repo.hca.bsc.es/epic/z/6YEpR6

```
Moved bytes (Total): 196672
    Moved bytes (scalar): 64 (0.03 %)
    Moved bytes (vector): 196608 (99.97 %)
tot_instr: 386
    scalar_instr: 226 (58.55 %)
    vsetvl_instr: 32 (8.29 %)
    vector_instr: 128 (33.16 %)
        SEW 8 vector_instr: 0 (0.00 %)
        SEW 16 vector_instr: 0 (0.00 %)
        SEW 32 vector_instr: 0 (0.00 %)
        SEW 64 vector_instr: 128 (100.00 %)
            avg_VL: 256.00 elements
            Arith: 32 (25.00 %)
                FP: 0 (0.00 %)
                INT: 32 (100.00 %)
            Mem: 96 (75.00 %)
                unit: 64 (66.67 %)
                strided: 0 (0.00 %)
                indexed: 32 (33.33 %)
            Mask: 0 (0.00 %)
            Other: 0 (0.00 %)
--------------------------------------------------------
```

Using RAVE to remove conditionals

- As we know, conditionals cause an overhead on the vectorization

```
void conditional(int N, double * A, double * B){
    for(int i=0; i<N; ++i){
        double new_value = A[i] + B[i];
        if (i > N/2) new_value += A[i-N/2] * 0.5;
        B[i] = new_value;
    }
}
```

```
avg_VL: 256.00 elements
Arith: 96 (29.91 %)
    FP: 64 (66.67 %)
    INT: 32 (33.33 %)
Mem: 128 (39.88 %)
    unit: 128 (100.00 %)
    strided: 0 (0.00 %)
    indexed: 0 (0.00 %)
Mask: 64 (19.94 %)
Other: 33 (10.28 %)
```

- Which can sometimes be avoided

```
void conditional(int N, double * A, double * B){
    for(int i=0; i<N/2; ++i){
        B[i] = A[i] + B[i];
    }
    for(int i=N/2; i<N; ++i){
        B[i] = A[i] + B[i] + A[i-N/2]*0.5;
    }
}
```

```
avg_VL: 256.00 elements
Arith: 48 (30.00 %)
    FP: 48 (100.00 %)
    INT: 0 (0.00 %)
Mem: 112 (70.00 %)
    unit: 112 (100.00 %)
    strided: 0 (0.00 %)
    indexed: 0 (0.00 %)
Mask: 0 (0.00 %)
Other: 0 (0.00 %)
```

https://repo.hca.bsc.es/epic/z/Z8QVfw

Using RAVE to help the compiler

- Sometimes the compiler does not reuse data

```
void reuse(int N, double * A, double * B, double *C,
           double * OUT_1, double * OUT_2){
    rave_event_and_value(1000,1);
    for(int i=0; i<N; ++i){
        OUT_1[i] = A[i] + B[i];
        OUT_2[i] = A[i] + C[i];
    }
    rave_event_and_value(1000,0);
```

```
avg_VL: 256.00 elements
Arith: 2 (25.00 %)
    FP: 2 (100.00 %)
    INT: 0 (0.00 %)
Mem: 6 (75.00 %)
    unit: 6 (100.00 %)
    strided: 0 (0.00 %)
    indexed: 0 (0.00 %)
```

- But we can help it (with **restrict** keyword or aux variables)

```
void reuse_restrict(int N, double* restrict A, double* restrict B,
 double* restrict C, double* restrict OUT_1, double* restrict OUT_2){
```

```
for(int i=0; i<N; ++i){
    double tmp = A[i];
    OUT_1[i] = tmp + B[i];
    OUT_2[i] = tmp + C[i];
}
```

```
avg_VL: 256.00 elements
Arith: 2 (28.57 %)
    FP: 2 (100.00 %)
    INT: 0 (0.00 %)
Mem: 5 (71.43 %)
    unit: 5 (100.00 %)
    strided: 0 (0.00 %)
    indexed: 0 (0.00 %)
```

https://repo.hca.bsc.es/epic/z/TmxCOb

Using RAVE and intrinsics

- With autovectorization, only the inner-loop “j” is vectorized:

```
void reduction(int N, int M, double *Y, double *X){
    rave_event_and_value(1000,1);
    for(int i=0; i<N; ++i){
        for(int j=0; j<M; ++j){
            Y[i] += X[j];
        }
    }
    rave_event_and_value(1000,0);
}
```

```
830     vsetvli     a4, zero, e64, m1, ta, ma
        06a41557
834     vfredusum.vs    v10, v10, v8
        cd80f057
838     vsetivli    zero, 0x1, e64, m1, ta, ma
        02037527
83c     vse64.v     v10, (t1)
```

```
Region #1: Event 1000 (code_region), Value 1 (Reduction),
    Moved bytes (Total): 2108420
        Moved bytes (scalar): 7172 (0.34 %)
        Moved bytes (vector): 2101248 (99.66 %)
    tot_instr: 16912
        scalar_instr: 9229 (54.57 %)
        vsetvl_instr: 3585 (21.20 %)
        vector_instr: 4098 (24.23 %)
            SEW 8 vector_instr: 0 (0.00 %)
            SEW 16 vector_instr: 0 (0.00 %)
            SEW 32 vector_instr: 0 (0.00 %)
            SEW 64 vector_instr: 4098 (100.00 %)
                avg_VL: 160.42 elements
                Arith: 2048 (49.98 %)
                    FP: 1536 (75.00 %)
                    INT: 512 (25.00 %)
                Mem: 1536 (37.48 %)
                    unit: 1536 (100.00 %)
                    strided: 0 (0.00 %)
                    indexed: 0 (0.00 %)
                Mask: 0 (0.00 %)
                Other: 514 (12.54 %)
```

Using RAVE and intrinsics

- With intrinsics, we can vectorize outer loops:

```
void reduction_outer(int N, int M, double *Y, double *X){
    rave_event_and_value(1000,2);
    for(int i=0; i<N;){
        long gvl = __builtin_epi_vsetvl(N-i, __epi_e64, __epi_m1);
        __epi_1xf64 vec_Y = __builtin_epi_vload_1xf64(&Y[i], gvl);
        for(int j=0; j<=M; ++j){
            __epi_1xf64 vec_X = __builtin_epi_vfmv_v_f_1xf64(X[j], gvl);
            vec_Y = __builtin_epi_vfadd_1xf64(vec_Y, vec_X, gvl);
        }
        __builtin_epi_vstore_1xf64(&Y[i], vec_Y, gvl);
        i += gvl;
    }
    rave_event_and_value(1000,0);
}
```

```
        0287d457
89a     vfadd.vf    v8, v8, fa5
```

```
Region #2: Event 1000 (code_region), Value 2 (Reduction_
    Moved bytes (Total): 10249
        Moved bytes (scalar): 2057 (20.07 %)
        Moved bytes (vector): 8192 (79.93 %)
    tot_instr: 4132
        scalar_instr: 3100 (75.02 %)
        vsetvl_instr: 2 (0.05 %)
        vector_instr: 1030 (24.93 %)
            SEW 8 vector_instr: 0 (0.00 %)
            SEW 16 vector_instr: 0 (0.00 %)
            SEW 32 vector_instr: 0 (0.00 %)
            SEW 64 vector_instr: 1030 (100.00 %)
                avg_VL: 256.00 elements
                Arith: 1026 (99.61 %)
                    FP: 1026 (100.00 %)
                    INT: 0 (0.00 %)
                Mem: 4 (0.39 %)
                    unit: 4 (100.00 %)
                    strided: 0 (0.00 %)
                    indexed: 0 (0.00 %)
                Mask: 0 (0.00 %)
                Other: 0 (0.00 %)
```

https://repo.hca.bsc.es/epic/z/PoRzZw