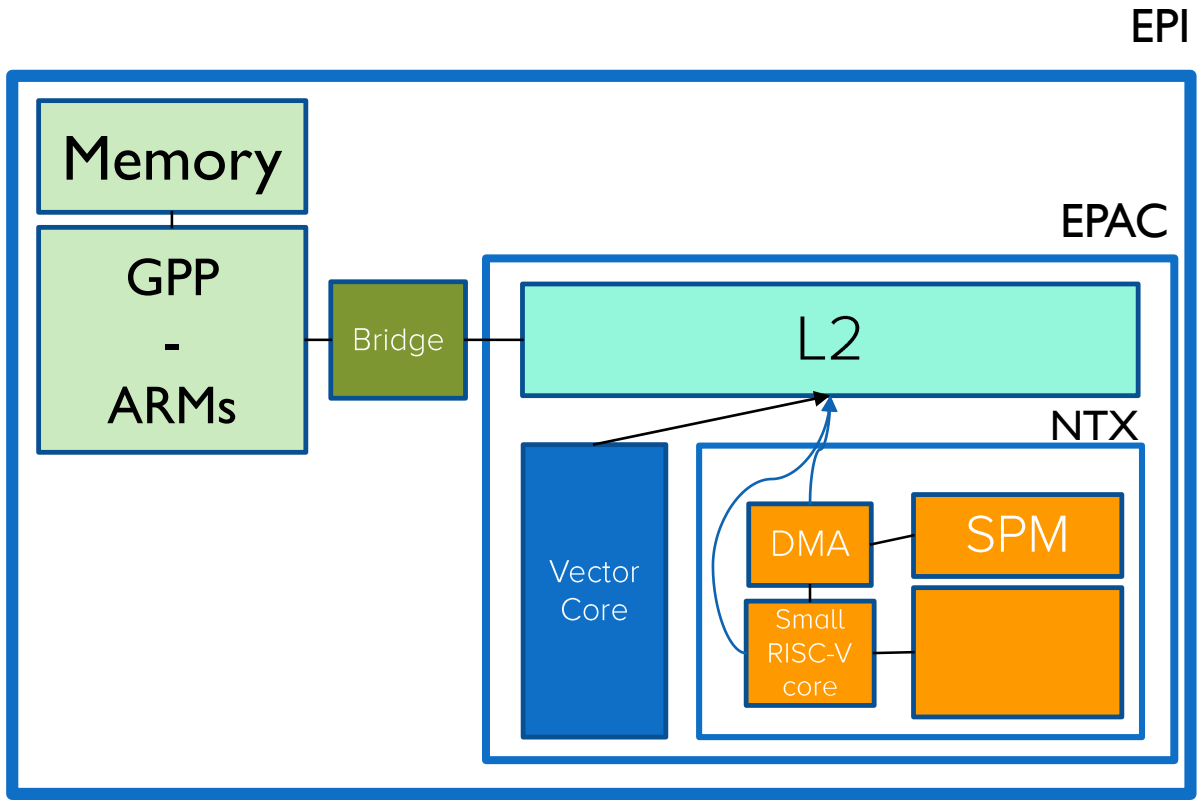
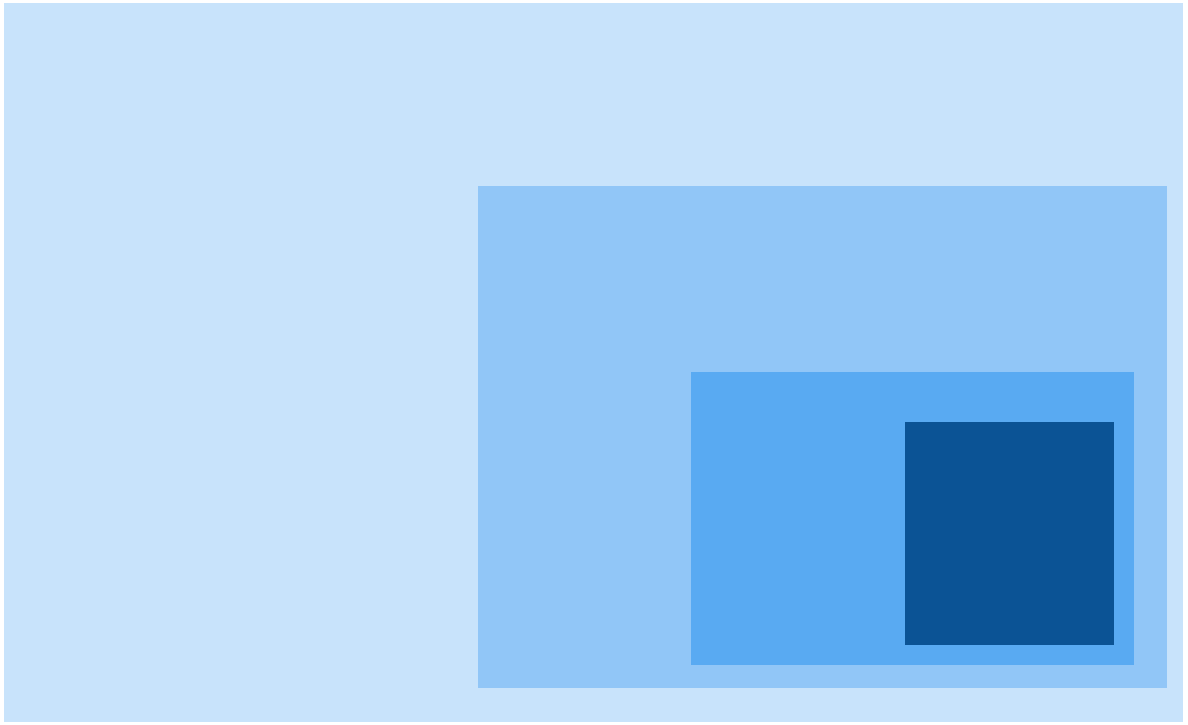
SOFTWARE

JESUS LABARTA

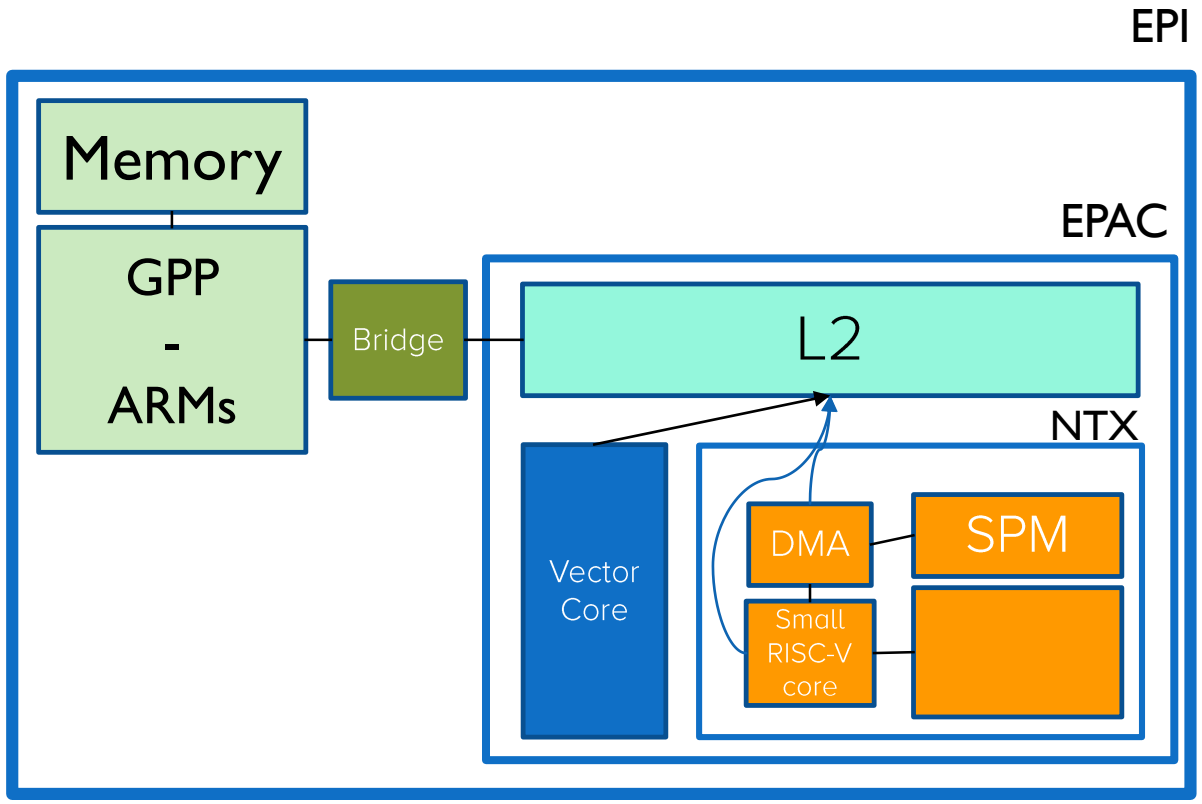
EPI: A HIGHLY HETEROGENEOUS/HIERARCHICAL SYSTEM



EPI: A HIGHLY HETEROGENEOUS/HIERARCHICAL SYSTEM



EPI: A HIGHLY HETEROGENEOUS/HIERARCHICAL SYSTEM



PROGRAMMING FOR LARGE SCALE HPC

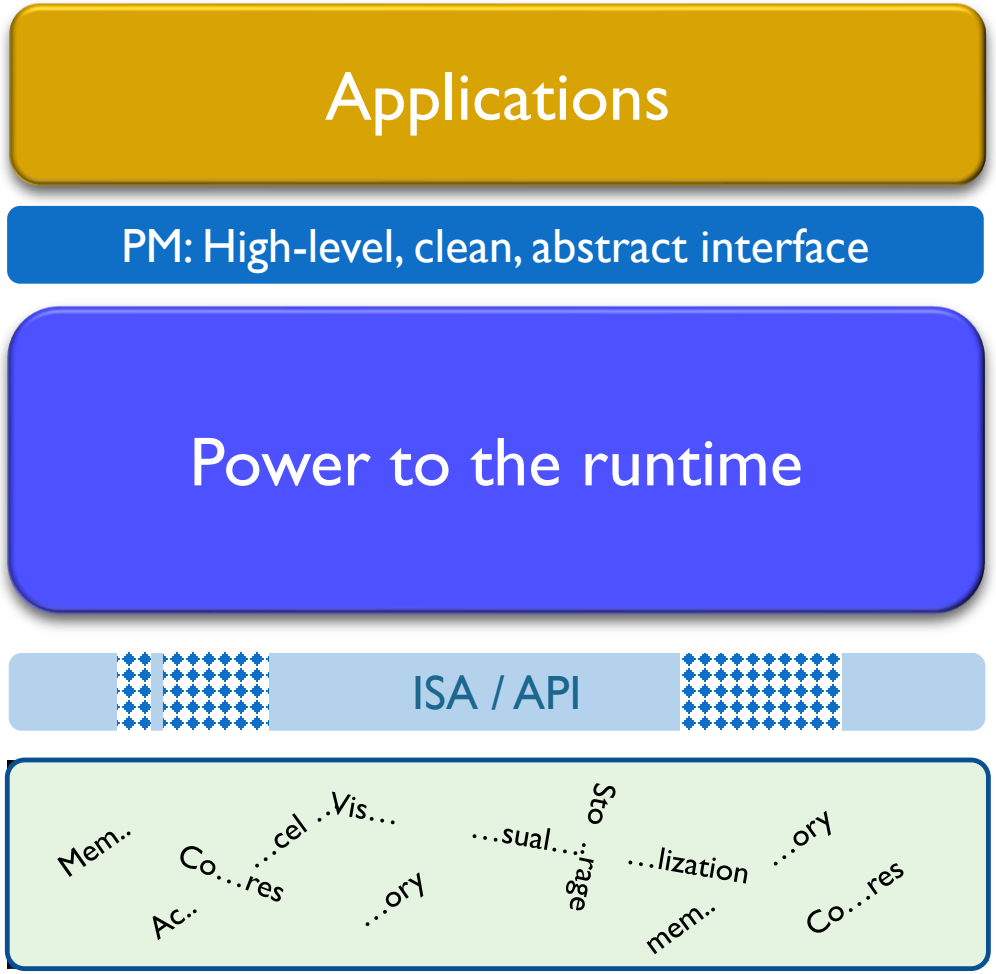
VISION

- The multicore and memory revolution
  - ISA leak …
  - Plethora of architectures
    - Heterogeneity
    - Memory hierarchies
- Complexity + variability = Divergence
  - Between our mental models and actual system behavior

The power wall made us go multicore
and the ISA interface to leak
→ our world is shaking

What programmers need ? HOPE !!!

VISION IN THE PROGRAMMING REVOLUTION



General purpose

Task & data based

Forget about resources

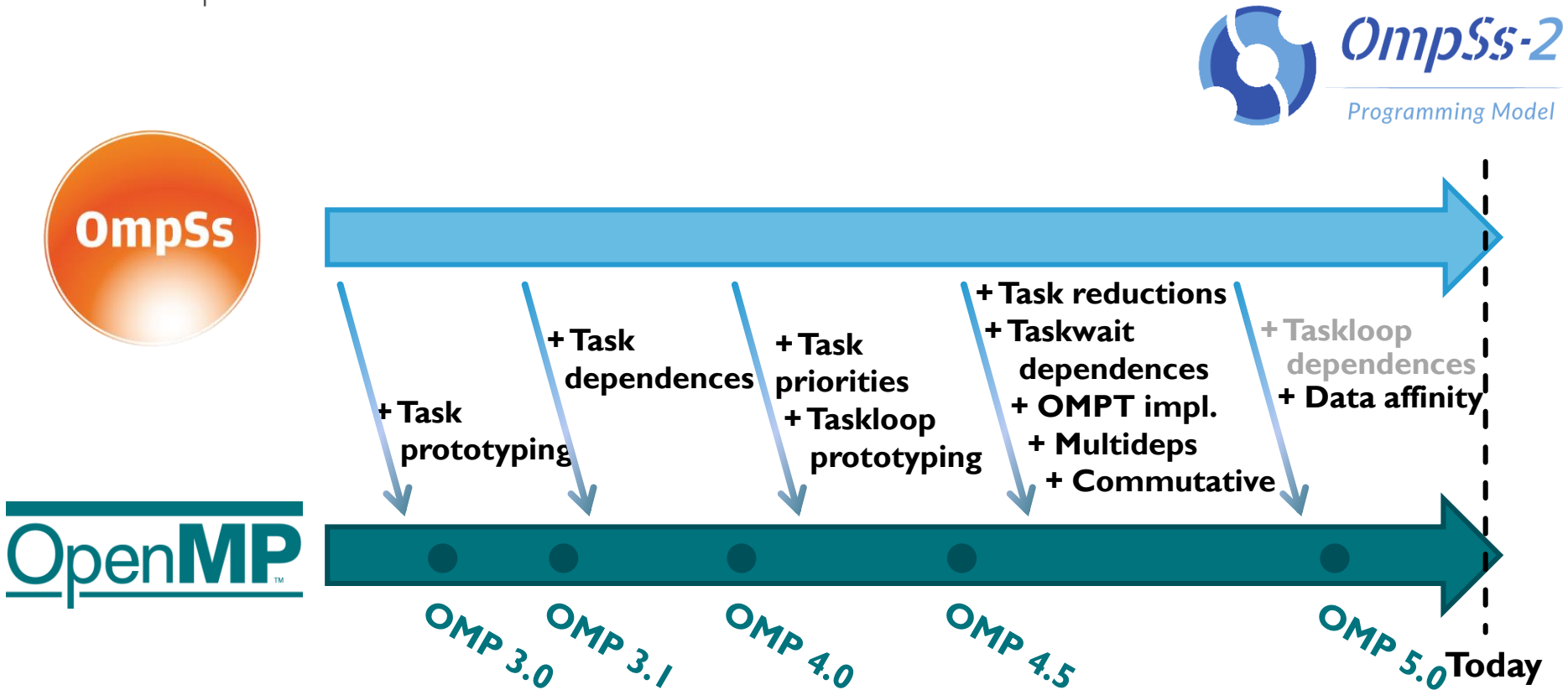
Decouple:
Minimal & sufficient permeability?

Intelligence
&
Resource management

"Reuse & expand" old architectural ideas
under new constraints

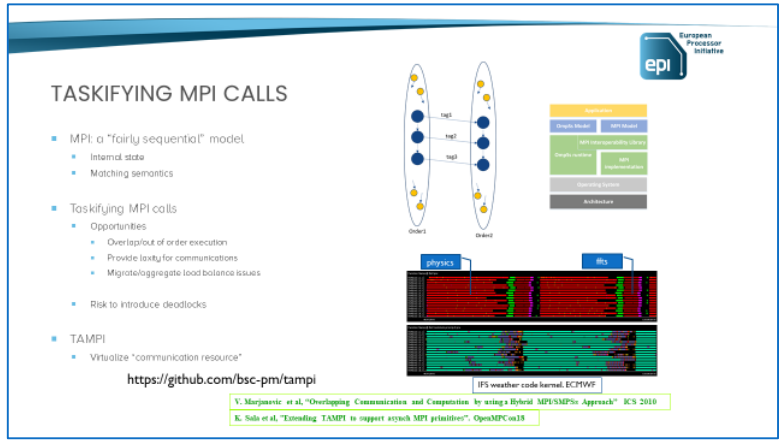
OMPSS AND OPENMP

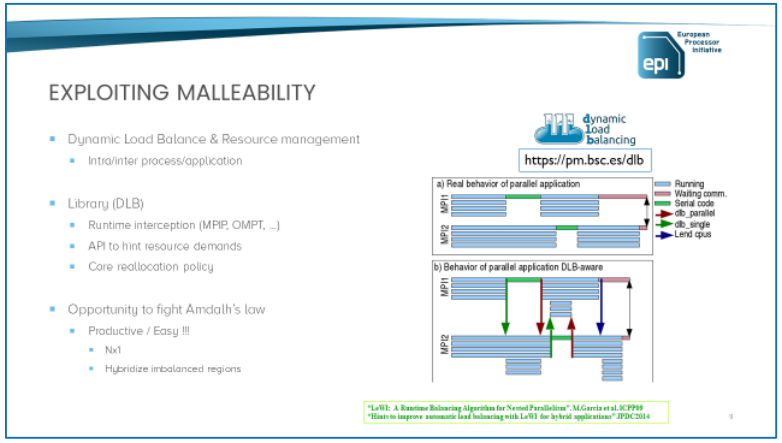
- A forerunner for OpenMP



IMPORTANT TOPICS/PRACTICES

- Regions
- Nesting
- Taskloops + dependences
- Taskify communications: MPI Interoperability
- Malleability
- Homogenize Heterogeneity
- Hierarchical “acceleration” / Hierarchical overdecomposition
- The osmotic porosity (hints, overheads,...)
- Beware of reductions on large arrays
- Memory management & Locality
- Beyond the node?
- Don´t mask your symptoms





PROGRAMMING IN EPI

PROGRAMMING MODEL VISION

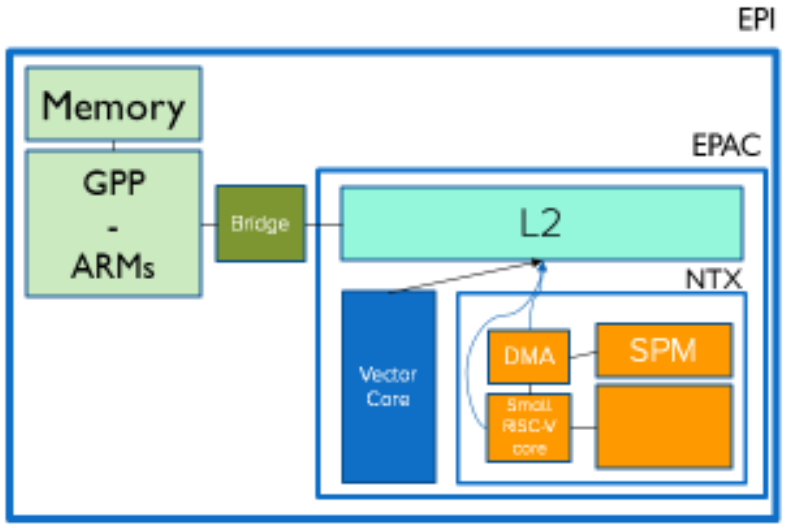
- MPI+OpenMP
  - Throughput oriented programming approach
    - Task based OpenMP
    - MPI – OpenMP interoperability (TAMPI)
    - Malleability in application + Dynamic resource (cores, power, BW) management
  - Intelligent runtimes
    - Handle overlap and locality (improve B/F)
    - Reduce overhead → hierarchical
    - Potential for architectural support for the runtime (Runtime Aware Architectures)

PROGRAMMING MODEL VISION

- **Offloading** to acceleration devices
  - General purpose OpenMP mechanism
  - Libraries for specialized acceleration devices
- (Large) **Vectors**
  - Auto vectorization
  - OpenMP SIMD clause

ACCELERATORS VISION @ EPI

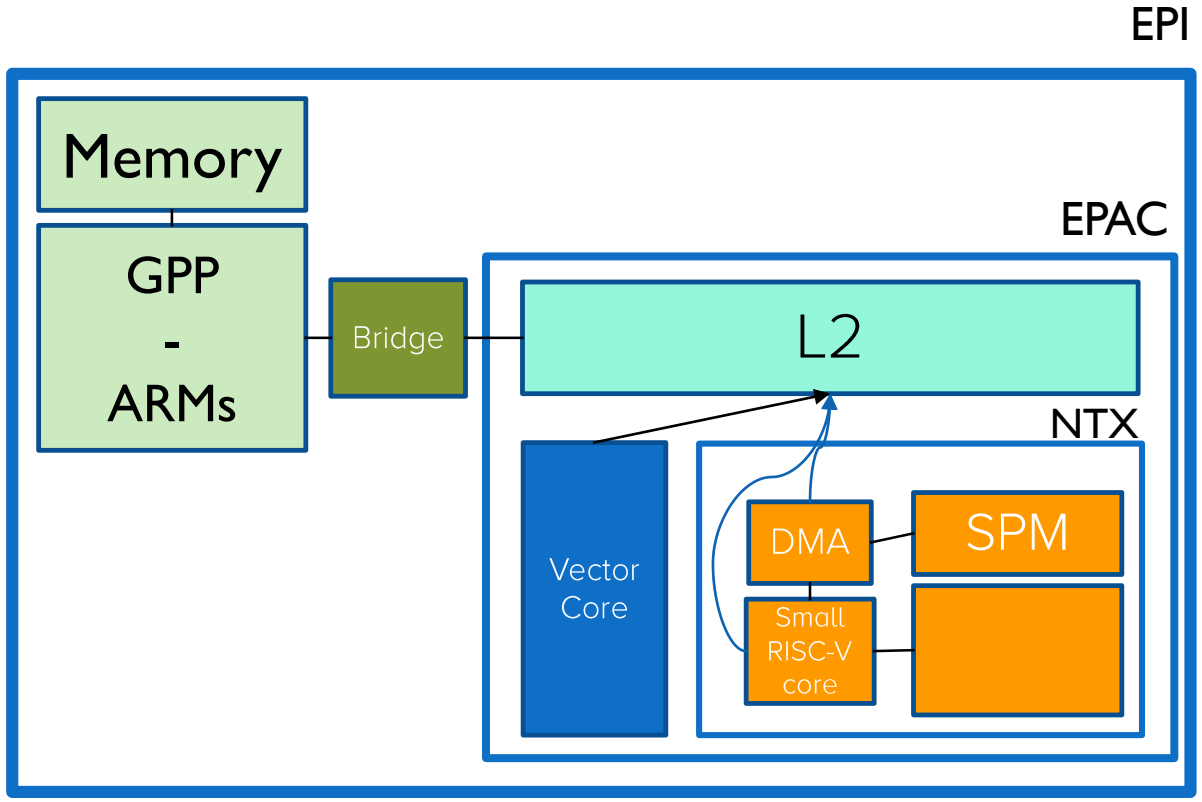
- High throughput devices
  - RISC-V VE
    - Long Vectors (>> # FUs)
      - Less instructions
      - Decouple FE-BE
      - Optimize memory throughput (High BW, B/F), Locality
    - ISA is important
    - "limited" number of control flows
  - Specific accelerators
- Hierarchical Acceleration
  - Nesting
  - Balanced hierarchy: number of levels, ratio
- Homogenized heterogeneity



VECTORS

EPI: A HIGHLY HETEROGENEOUS/HIERARCHICAL SYSTEM

- Some ARM - RISC-V commonality
  - towards (large?) vector
  - ARM:
    - Up to 32 DP
    - Predication driven.
  - RISCV:
    - Up to VLMAX (power of 2)
    - Explicit vector length



SIMD VS VECTOR

- Vector lengths
  - Dynamic instruction count
- Fixed vs. variable vector length
  - Static instruction count
    - Epilogues

ISA	MIPS-32 MSA	IA-32 AVX2	RV32V
Instructions (static)	22	29	13
Instructions per Main Loop	7	6	10
Bookkeeping Instructions	15	23	3
Results per Main Loop	2	4	64
Instructions (dynamic n=1000)	3511	1517	163

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0            # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0          # vl = t0 = min(mvl, n)
  c:  vld    v0, a1          # load vector x
 10:  slli   t1, t0, 3       # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2          # load vector y
 18:  add    a1, a1, t1      # increment pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1 # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0      # n -= vl (t0)
 24:  vst    v1, a2          # store Y
 28:  add    a2, a2, t1      # increment pointer to y by vl*8
 2c:  bnez   a0, loop        # repeat if n != 0
 30:  ret                    # return
```

RV32V

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
  0:  li      a1,-2
  4:  and     a1,a0,a1        # a1 = floor(n/2)*2 (mask bit 0)
  8:  sll     t0,a1,0x3       # t0 = byte address of a1
  c:  addu    v1,a3,t0        # v1 = &y[a1]
 10:  beq     a3,v1,38        # if y==&y[a1] goto Fringe
                              #    (t0==0 so n is 0 | 1)
 14:  move    v0,a2           # (delay slot) v0 = &x[0]
 18:  splati.d $w2,$w13[0]    # w2 = fill SIMD reg. with copies of a
Main Loop:
 1c:  ld.d    $w0,0(a3)       # w0 = 2 elements of y
 20:  addiu   a3,a3,16        # incr. pointer to y by 2 FP numbers
 24:  ld.d    $w1,0(v0)       # w1 = 2 elements of x
 28:  addiu   v0,v0,16        # incr. pointer to x by 2 FP numbers
 2c:  fmadd.d $w0,$w1,$w2     # w0 = w0 + w1 * w2
 30:  bne     v1,a3,1c        # if (end of y != ptr to y) go to Loop
 34:  st.d    $w0,-16(a3)     # (delay slot) store 2 elts of y
Fringe:
 38:  beq     a1,a0,50        # if (n is even) goto Done
 3c:  addu    a2,a2,t0        # (delay slot) a2 = &x[n-1]
 40:  ldc1    $f1,0(v1)       # f1 = y[n-1]
 44:  ldc1    $f0,0(a2)       # f0 = x[n-1]
 48:  madd.d  $f13,$f1,$f13,$f0# f13 = f1+f0*f13 (muladd if n is odd)
 4c:  sdc1    $f13,0(v1)      # y[n-1] = f13 (store odd result)
Done:
 50:  jr      ra              # return
 54:  nop                     # (delay slot)
```

MIPS32 MSA

```
# eax is i, n is esi, a is xmm1,
# pointer to x[0] is ebx, pointer to y[0] is ecx
 0: push   esi
 1: push   ebx
 2: mov    esi,[esp+0xc]   # esi = n
 6: mov    ebx,[esp+0x18]  # ebx = x
 a: vmovsd xmm1,[esp+0x10] # xmm1 = a
 10: mov   ecx,[esp+0x1c]  # ecx = y
 14: vmovddup xmm2,xmm1    # xmm2 = {a,a}
 18: mov   eax,esi
 1a: and   eax,0xfffffffc  # eax = floor(n/4)*4
 1d: vinsertf128 ymm2,ymm2,xmm2,0x1 # ymm2 = {a,a,a,a}
 23: je    3e              # if n < 4 goto Fringe
 25: xor   edx,edx         # edx = 0
Main Loop:
 27: vmovapd ymm0,[ebx+edx*8] # load 4 elements of x
 2c: vfmadd213pd ymm0,ymm2,[ecx+edx*8] # 4 mul adds
 32: vmovapd [ecx+edx*8],ymm0 # store into 4 elements of y
 37: add   edx,0x4
 3a: cmp   edx,eax         # compare to n
 3c: jb    27              # repeat loop if < n
Fringe:
 3e: cmp   esi,eax         # any fringe elements?
 40: jbe   59              # if (n mod 4) == 0 go to Done
Fringe Loop:
 42: vmovsd xmm0,[ebx+eax*8] # load element of x
 47: vfmadd213sd xmm0,xmm1,[ecx+eax*8] # 1 mul add
 4d: vmovsd [ecx+eax*8],xmm0 # store into element of y
 52: add   eax,0x1         # increment Fringe count
 55: cmp   esi,eax         # compare Loop and Fringe counts
 57: jne   42 <daxpy+0x42>  # repeat FringeLoop if != 0
Done:
 59: pop   ebx             # function epilogue
 5a: pop   esi
 5b: ret
```

IA-32 SSE and AVX2

https://www.sigarch.org/simd-instructions-considered-harmful/

ARM SVE

- Vector Length Agnostic (VLA)
  - Hardware implementation chooses vector size (within architectural maximum of 2048 bits in multiple of 128 bits)
- Scalable vector length
  - 32 vector registers
- Per lane predication
  - Instructions to create predicates
- Gather load & Scatter store
- Vector partitioning
- Fault tolerant speculative vectorization
  - Memory
- Horizontal vector operations
  - Reductions
- Serialized vector operations
  - Allowing the vectorization of loops wit pointer chasing accesses

RISC-V VECTOR EXTENSION

- Strip-mined loops – no remainder handling needed
- Masking on (almost) every vector instruction
- Strided loads and stores, scatters, gathers
- Reduction instructions (sum, min/max, and/or, …)
- Orthogonal set of vector operations, parity with scalar ISA
- Fault-only-first loads for loops with data dependent exits

https://github.com/riscv/riscv-v-spec/

AXPY @ ARM, RISC, OPENMP ACCELERATOR

DAXPY

- Scalar C code

```
1 void daxpy(double *x, double *y, double a, int n)
2 {
3     for (int i = 0; i < n; i++) {
4         y[i] = a*x[i] + y[i];
5     }
6 }
```

DAXPY @ ARM

- ARMv8-A scalar code

```
1     // x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
2     daxpy_:
3         ldrsw x3, [x3] // x3=*n
4         mov x4, #0 // x4=i=0
5         ldr d0, [x2] // d0=*a
6         b .latch
7     .loop:
8         ldr d1, [x0, x4, lsl #3] // d1=x[i]
9         ldr d2, [x1, x4, lsl #3] // d2=y[i]
10       fmadd d2, d1, d0, d2 // d2+=x[i]*a
11       str d2, [x1, x4, lsl #3] // y[i]=d2
12       add x4, x4, #1 // i+=1
13    .latch:
14       cmp x4, x3 // i < n
15       b.lt .loop // more to do?
16       ret
```

DAXPY @ ARM SVE

- ARMv8-A SVE code
  - Predication centric approach
    - Predication compute instructions
      - Architecture Implementation dependent !!!
      - The system decides
  - VL ~ FUs

```
1     // x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
2     daxpy_:
3        ldrsw x3, [x3] // x3=*n
4        mov x4, #0 // x4=i=0
5        whilelt p0.d, x4, x3 // p0=while(i++<n)
6        ld1rd z0.d, p0/z, [x2] // p0:z0=bcast(*a)
7     .loop:
8        ld1d z1.d, p0/z, [x0, x4, lsl #3] // p0:z1=x[i]
9        ld1d z2.d, p0/z, [x1, x4, lsl #3] // p0:z2=y[i]
10       fmla z2.d, p0/m, z1.d, z0.d // p0?z2+=x[i]*a
11       st1d z2.d, p0, [x1, x4, lsl #3] // p0?y[i]=z2
12       incd x4 // i+=(VL/64)
13    .latch:
14       whilelt p0.d, x4, x3 // p0=while(i++<n)
15       b.first .loop // more to do?
16       ret
```

Micro architecture decides

Predicated execution

DAXPY @ RISC-V VE

- RISC-V VE
  - Set up: architecture decides VL

```
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y

saxpy:
    vsetvli a4, a0, e32, m8
    vlw.v v0, (a1)
    sub a0, a0, a4
    slli a4, a4, 2
    add a1, a1, a4
    vlw.v v8, (a2)
    vfmacc.vf v8, fa0, v0
    vsw.v v8, (a2)
    add a2, a2, a4
    bnez a0, saxpy
    ret
```

Micro architecture decides

DAXPY @ OPENMP ACCELERATOR DIRECTIVES

- Accelerator directives

```
void axpy_ref          (double a, double *dx, double *dy, int n) {
    #pragma omp target teams distribute parallel for
    for(int i = 0; i<n; ++i)
        y[i] = a*x[i] + y[i];
}
```

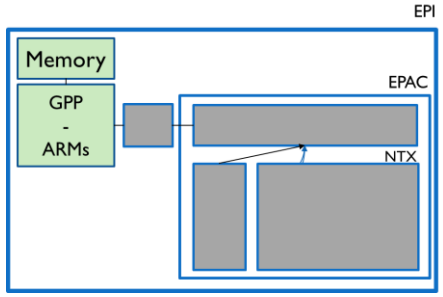
Accelerator specific resource/scheduling model

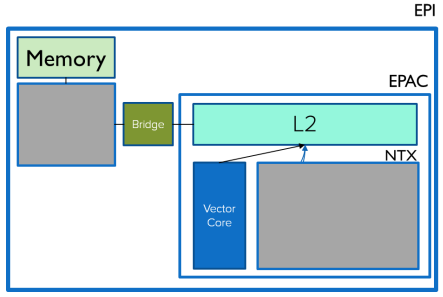
GPUish

AXPY @ EPI

DAXPY @ EPI

- Scalar C code



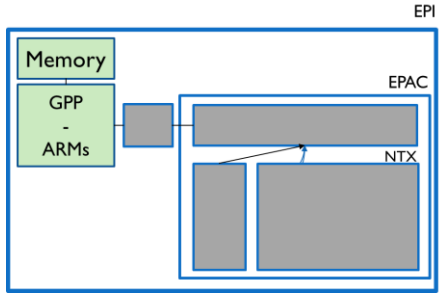


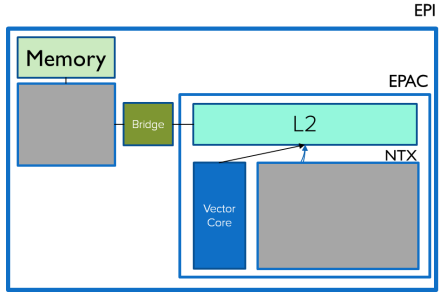
On ARM / On RISC-V

```
void axpy_ref          (double a, double *dx, double *dy, int n) {
   int i;



   for (i=0; i<n; i++) {
      dy[i] += a*dx[i];
   }
}
```

DAXPY @ EPI

- Vector OpenMP C code



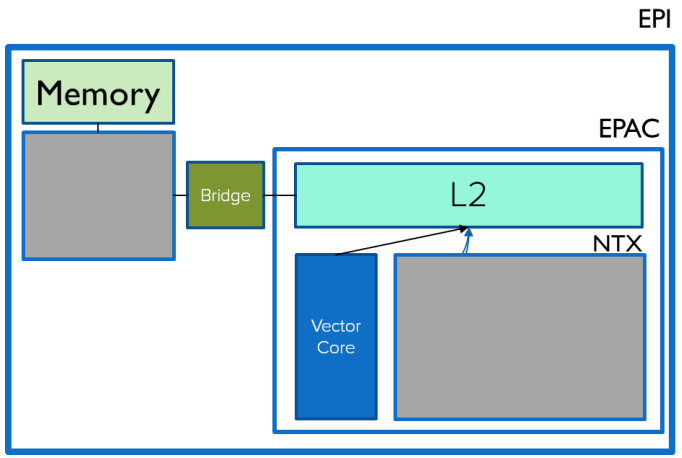


On ARM / On RISC-V

```
void axpy_SIMD        (double a, double *dx, double *dy, int n) {
   int i;


   #pragma omp simd
   for (i=0; i<n; i++) {
      dy[i] += a*dx[i];
   }
}
```

DAXPY @ EPI

- Vector intrinsics @ C code

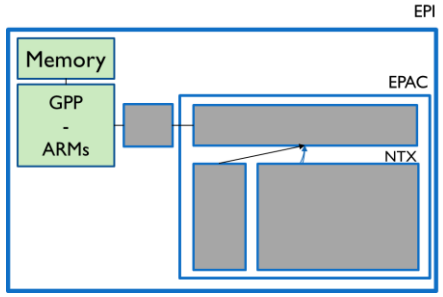


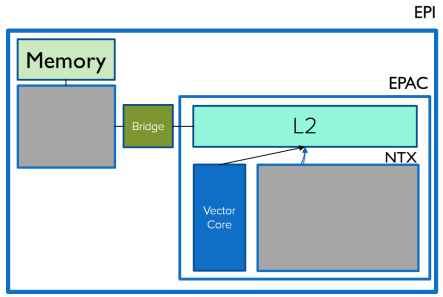
On RISC-V

```
void axpy_intrinsics  (double a, double *dx, double *dy, int n) {
   int i;
   int gvl = __builtin_epi_vsetvl(n, __epi_e64, __epi_m1);
    __epi_1xf64 v_a = __builtin_epi_vbroadcast_1xf64(a, gvl);

   for (i=0; i<n; ) {
       gvl = __builtin_epi_vsetvl(n - i, __epi_e64, __epi_m1);
       __epi_1xf64 v_dx = __builtin_epi_vload_1xf64(&dx[i], gvl);
       __epi_1xf64 v_dy = __builtin_epi_vload_1xf64(&dy[i], gvl);
       __epi_1xf64 v_res = __builtin_epi_vfmacc_1xf64(v_dy, v_a, v_dx, gvl);
       __builtin_epi_vstore_1xf64(&dy[i], v_res, gvl);
       i += gvl;
    }
}
```

DAXPY @ EPI

- OpenMP taskified





On ARM / On RISC-V

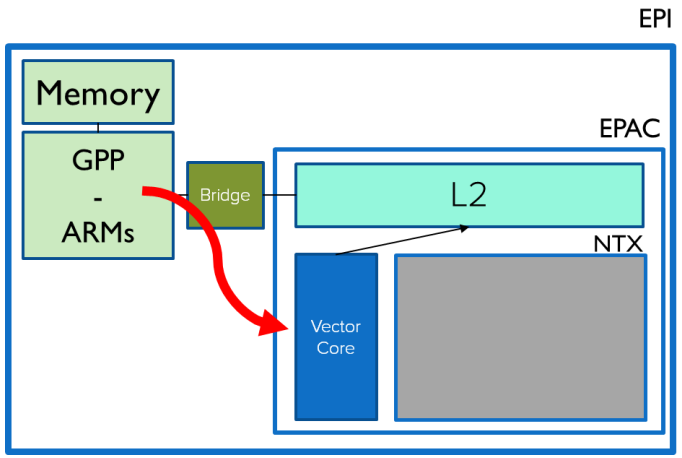
```
void axpy_omp          (double a, double *dx, double *dy, int n) {
   int I, chunk;


   #pragma omp taskloop
   for (i=0; i<n; i+=TS) {
      chunk= n>i+TS? TS : n-i;

      axpy_SIMD        (a, &dx[i], &dy[i], chunk);
   }
}
```

DAXPY @ EPI

- Offloading

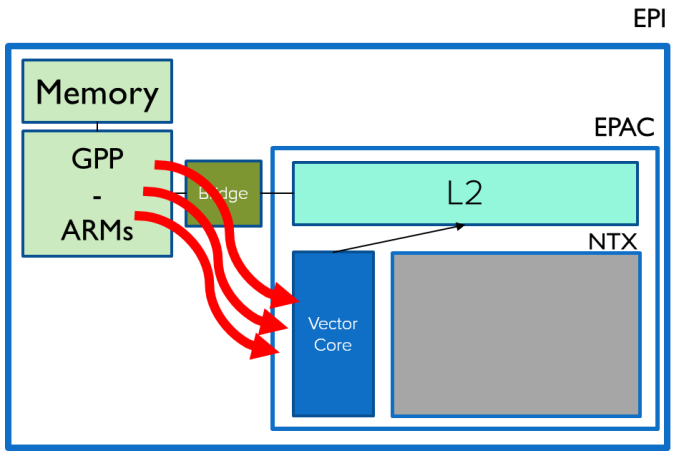
From ARM → RISC-V

```
void axpy_offload     (double a, double *dx, double *dy, int n) {
   int i;


   #pragma omp target map(to:dx[0:n-1], tofrom:dy[0:n-1])

   axpy_SIMD(a, dx, dy, n);       // axpy_intrinsics, ...

}
```



DAXPY @ EPI

- Offload from each ARM thread to a RISC-V core

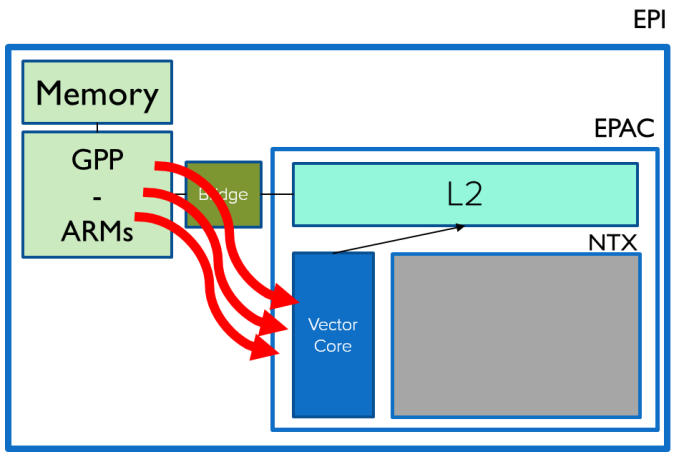


Parallel ARM → RISC-V

```
void axpy_omp_nest    (double a, double *dx, double *dy, int n) {
   int I, chunk;


   #pragma omp taskloop
   for (i=0; i<n; i+=TS) {
      chunk= n>i+TS? TS : n-i;
      #pragma omp target map(to:dx[i:i+chunk], tofrom:dy[i:i+chunk])
      axpy_SIMD       (a, &dx[i], &dy[i], chunk);
   }
}
```

DAXPY @ EPI

- Offload from each ARM thread to a RISC-V core

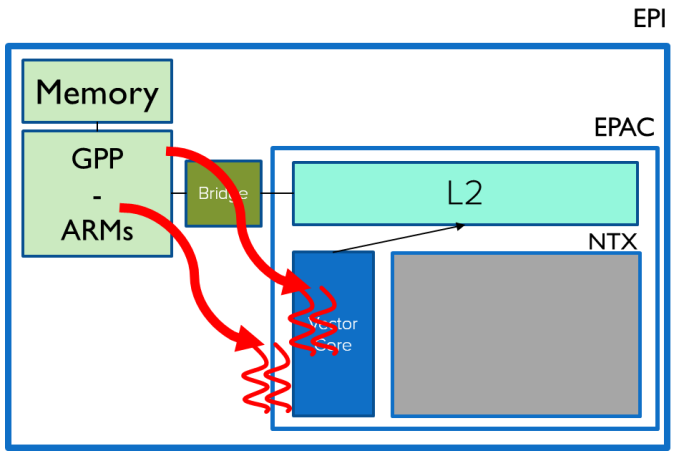


Parallel ARM → RISC-V

```
void axpy_omp_nest_2  (double a, double *dx, double *dy, int n) {
   int I, chunk;


   #pragma omp taskloop
   for (i=0; i<n; i+=TS) {
      chunk= n>i+TS? TS : n-i;
      #pragma omp target map(to:dx[i:i+chunk], tofrom:dy[i:i+chunk])
      axpy_intrinsics (a, &dx[i], &dy[i], chunk);
   }
}
```

DAXPY @ EPI

- Offload from several ARMs to several RISC-Vs

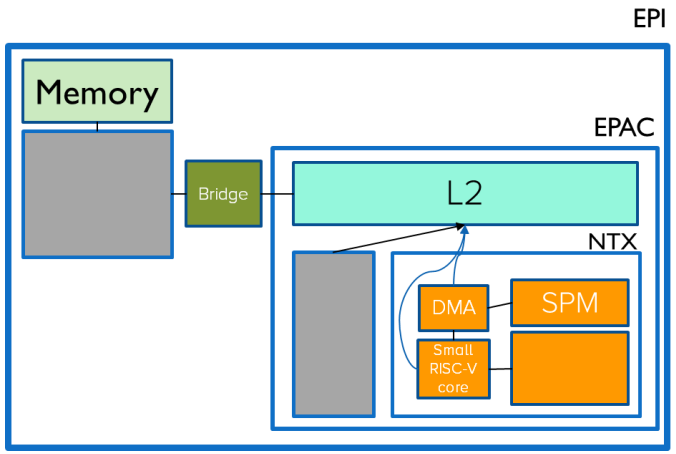


On ARM

```
void axpy_omp_nest_3  (double a, double *dx, double *dy, int n) {
   int i, chunk;


   #pragma omp taskloop
   for (i=0; i<n; i+=TS) {
      chunk= n>i+TS? TS : n-i;
      #pragma omp target map(to:dx[i:i+chunk], tofrom:dy[i:i+chunk])
      axpy_omp        (a, &dx[i], &dy[i], chunk);
   }
}
```

DAXPY @ EPI

- NTX driver for axpy
  - To be run on small RISC-V core in the NTX
  - Double buffering used to prefetch data and overlap with computation



On RISC-V

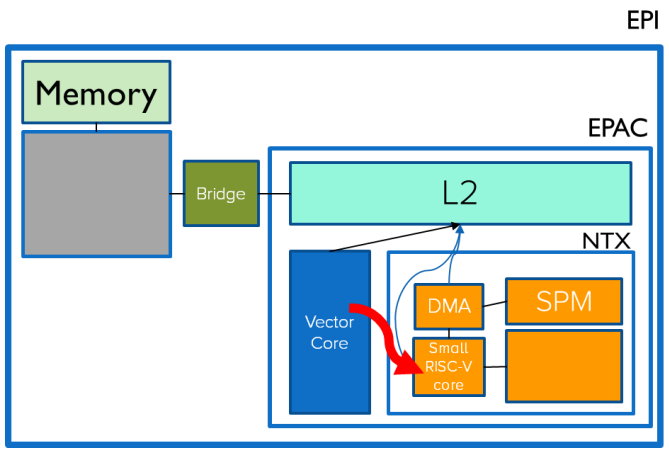
```
void axpy_ntx_drv(float a, float *dx, float *dy, int n) {
  DoubleBuffer bx(TS, dx, IN), by(TS, dy, INOUT);
  bx.prefetch(); by.prefetch();
  ntx_cfg({TS}, {a,0}, {bx,8}, {by,8});
  for (int i = 0; i < n; i += TS) {
    ntx_sync();
    bx.swap(); by.swap();
    ntx_fmac();
  }
  ntx_sync();
  by.flush();
  by.flush();
}
```

DAXPY @ EPI

- Offloading from vector core to NTX

From Vector core (RISC-V) → NTX (RISC-V)

```
void axpy_ntx             (double a, double *dx, double *dy, int n) {


   #pragma omp target

   axpy_ntx_drv (a, dx, dy, n);
}
```

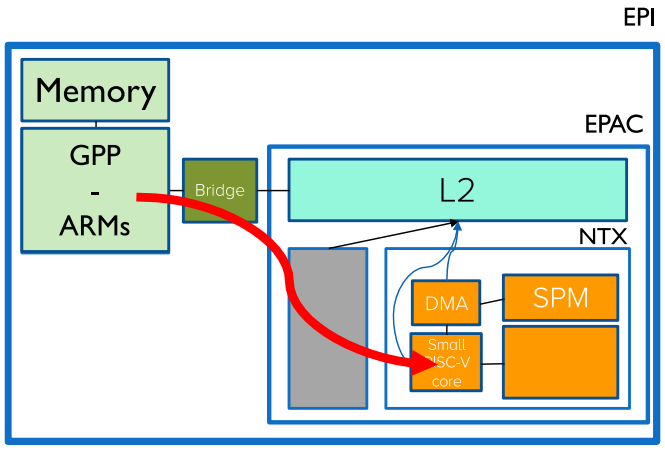


DAXPY @ EPI

- Offloading from ARM to NTX

From ARM core → NTX (RISC-V)

```
void axpy_a2ntx          (double a, double *dx, double *dy, int n) {


   #pragma omp target map(to:dx[0:n-1], tofrom:dy[0:n-1]) device(ntx)

   axpy_ntx_drv (a, dx, dy, n);
}
```

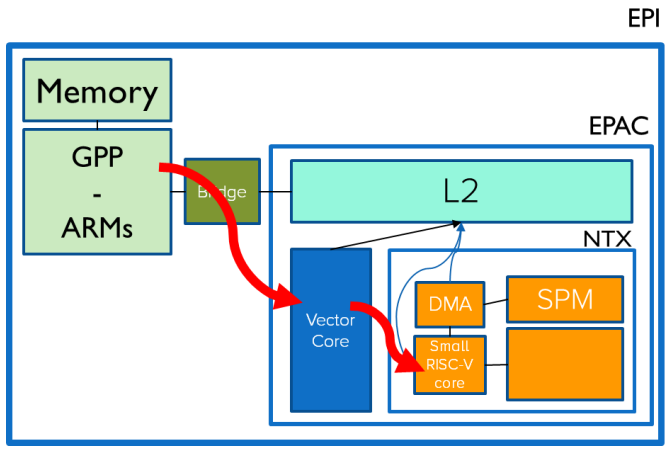


DAXPY @ EPI

- From ARM to RISC-V vector to NTX

From ARM → RISC-V → NTX

```
void axpy_2_nest      (double a, double *dx, double *dy, int n) {
   int i;


   #pragma omp target map(to:dx[0:n-1], tofrom:dy[0:n-1])

   axpy_ntx (a, dx, dy, n);
}
```

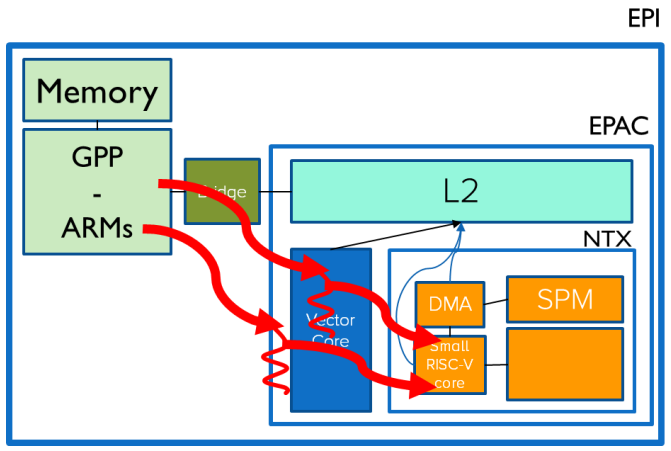


DAXPY @ EPI

- … and combinations !!

From ARM → RISC-V → NTX

```
void axpy_par_2_nest (double a, double *dx, double *dy, int n) {
   int I, chunk;


   #pragma omp taskloop
   for (i=0; i<n; i+=TS) {
      chunk= n>i+TS? TS : n-i;

      axpy_stx (a, i&dx[i], &dy[i], chunk);
   }
}
```



EMULATION

ARMIE

- Axpy @ ARM SVE

input n = 30*1024			
Exec	Total instructions	SVE instructions	Non-SVE instructions
scalar	519141	0	519141
SVE 128	380884	184328	196556
SVE 256	265684	92168	173516
SVE 512	208124	46088	162036
SVE 1024	179324	23048	156276
SVE 2048	164924	11528	153396

input n = 20000*1024			
Exec	Total instructions	SVE instructions	Non-SVE instructions
scalar	245911516	0	245911516
SVE 128	153751543	122880008	30871535
SVE 256	76951543	61440008	15511535
SVE 512	38551503	30720008	7831495
SVE 1024	19351503	15360008	3991495
SVE 2048	9751543	7680008	2071535

Scalar code:

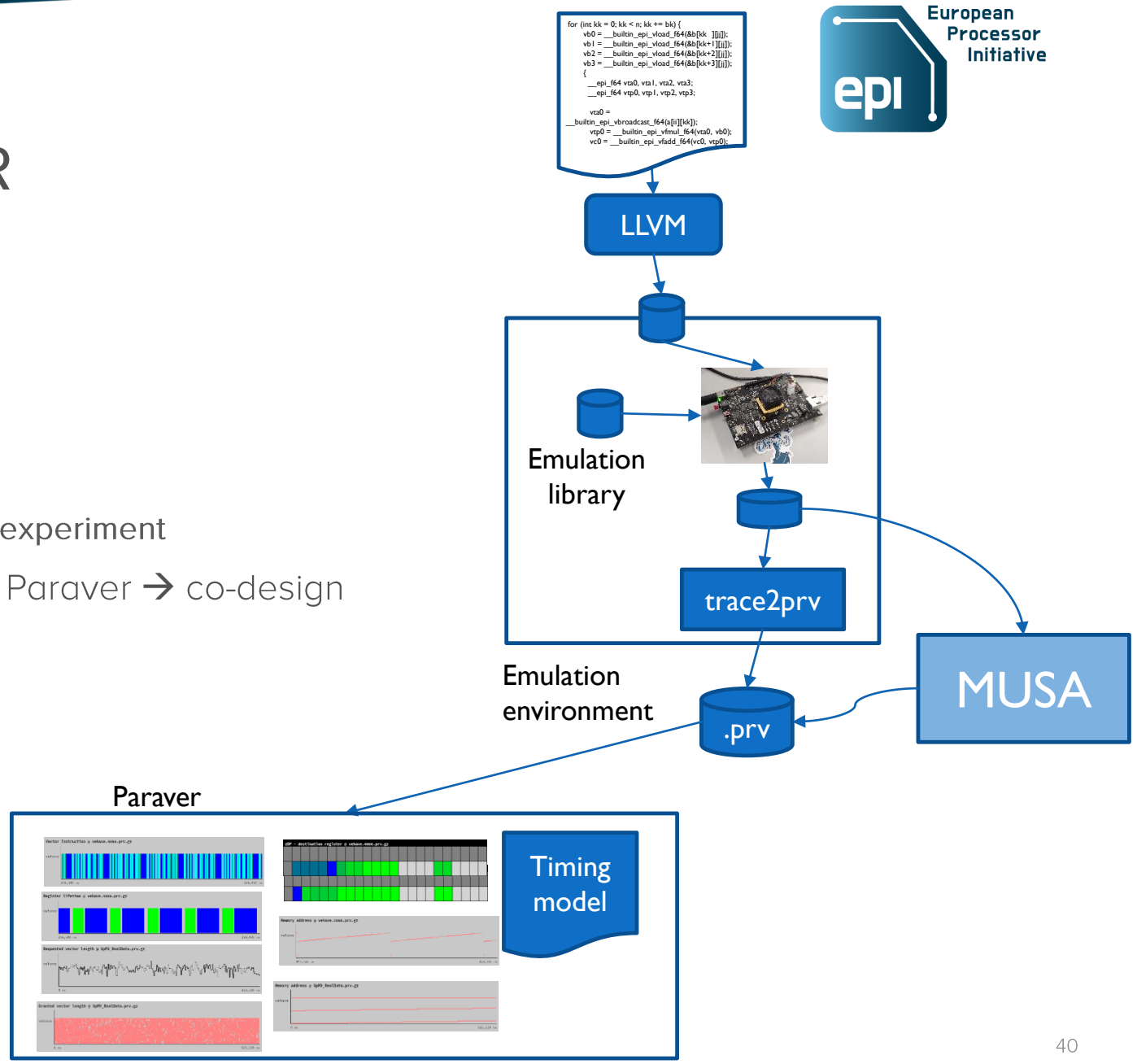
```
    and     x8, x21, #0xffffffff
    fmov    d0, #3.00000000
    mov     x9, x19
    mov     x10, x20
.LBB3_10:
    ldr     d1, [x9], #8
    ldr     d2, [x10]
    subs    x8, x8, #1
    fmadd d1, d1, d0, d2
    str     d1, [x10], #8
    b.ne    .LBB3_10
```

SVE code:

```
    mov     x9, xzr
    whilelo  p1.d, xzr, x8
    fmov     z0.d, #3.00000000
    ptrue    p0.d
.LBB3_17:
    ld1d     { z1.d }, p1/z, [x19, x9, lsl #3]
    ld1d     { z2.d }, p1/z, [x20, x9, lsl #3]
    fmad     z1.d, p0/m, z0.d, z2.d
    st1d     { z1.d }, p1, [x20, x9, lsl #3]
    incd     x9
    whilelo p1.d, x9, x8
    b.mi     .LBB3_17
```

RISC-V VECTOR EMULATOR

- Source
  - C + Intrinsics
- Microbenchmarks
  - MxM, SpMV, FE, FFT
  - As many as you think relevant/would like to experiment
- Timing models and very detailed analytics in Paraver → co-design
  - Vector length
  - Register use
  - Cache sizes and policies (I, D)
  - Latency we can tolerate?
  - B/F
  - …



COMPILER

```
void SpMV_vec(double *a, long *ia, long *ja, double *x, double *y, int nrows) {
   for (int row = 0; row < nrows; row++) {
       int nnz_row = ia[row + 1] - ia[row];
       int rvl, gvl;        // requested & granted vector lengths
       int idx = ia[row];
       y[row]=0.0;
       for(int colid=0; colid<nnz_row; colid +=gvl ) {    //blocking on MAXVL
           rvl = nnz_row - colid;
           gvl = __builtin_epi_vsetvl(rvl, __epi_e64, __epi_m1);
           __epi_1xf64 va = __builtin_epi_vload_1xf64(&a[idx+colid], gvl);
           __epi_1xi64 v_idx_row = __builtin_epi_vload_1xi64(&ja[idx+colid], gvl);
           __epi_1xi64 vthree = __builtin_epi_vbroadcast_1xi64(3, gvl);
           v_idx_row = __builtin_epi_vsll_1xi64(v_idx_row, vthree, gvl);
           __epi_1xf64 vx = __builtin_epi_vload_indexed_1xf64(x, v_idx_row, gvl);
           __epi_1xf64 vprod = __builtin_epi_vfmul_1xf64(va, vx, gvl);
           __epi_1xf64 partial_res = __builtin_epi_vbroadcast_1xf64(0.0, gvl);
           partial_res = __builtin_epi_vfredsum_1xf64(vprod, partial_res, gvl);
           y[row] += __builtin_epi_vgetfirst_1xf64(partial_res);
   }
```

Standard C/C++

Vector data types

Can be passed as arguments

Compiler does:
- Register allocation
- Instruction scheduling
- Spill/save/restore

Intrinsic operations

… C variables

… vector variables

… vector length

COMPILER

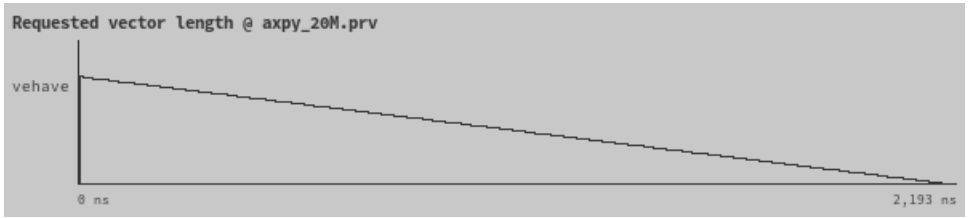
```
void axpy_intrinsics  (double a, double *dx, double *dy, int n) {
   int i;
   int gvl = __builtin_epi_vsetvl(n, __epi_e64, __epi_m1);
    __epi_1xf64 v_a = __builtin_epi_vbroadcast_1xf64(a, gvl);

   for (i=0; i<n; ) {
       gvl = __builtin_epi_vsetvl(n - i, __epi_e64, __epi_m1);
       __epi_1xf64 v_dx   = __builtin_epi_vload_1xf64(&dx[i], gvl);
       __epi_1xf64 v_dy   = __builtin_epi_vload_1xf64(&dy[i], gvl);
       __epi_1xf64 vtmp   = __builtin_epi_vfmul_1xf64(v_a, v_dx, gvl);
       __epi_1xf64 v_res  = __builtin_epi_vfadd_1xf64(vtmp, v_dy, gvl);
       __builtin_epi_vstore_1xf64(&dy[i], v_res, gvl);
       i += gvl;
    }
}
```

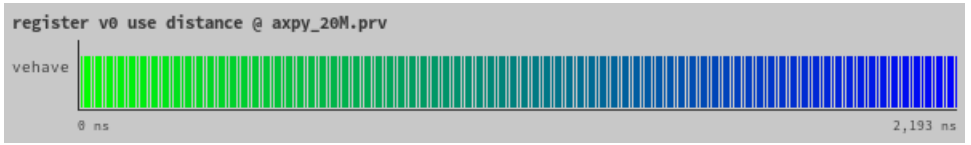
```
axpy_intrinsics:
        vsetvli a3, a2, e64, m1
        sext.w  a3, a3
        vsetvli a4, a3, e64, m1
        vfmv.v.f        v0, fa0
        addi    a3, zero, 1
        blt     a2, a3, .LBB2_3
        mv      a3, zero
.LBB2_2:
        sext.w  a4, a3
        slli    t0, a4, 3
        add     a6, a0, t0
        subw    a5, a2, a3
        vsetvli a7, a5, e64, m1
        sext.w  a5, a7
        vsetvli a4, a5, e64, m1
        vle.v   v1, (a6)
        vfmul.vv        v1, v0, v1
        add     a4, a1, t0
        vle.v   v2, (a4)
        vfadd.vv        v1, v1, v2
        vse.v   v1, (a4)
        addw    a3, a3, a7
        blt     a3, a2, .LBB2_2
.LBB2_3:
        ret
```

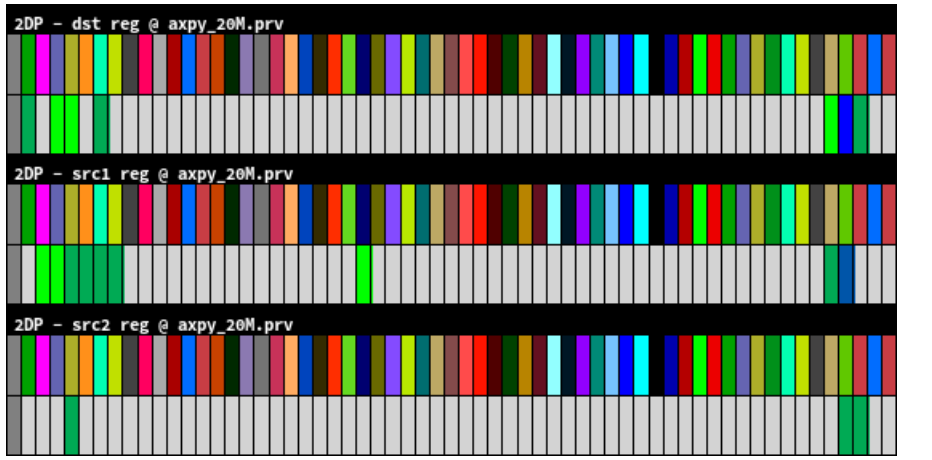
TRACE ANALYSIS

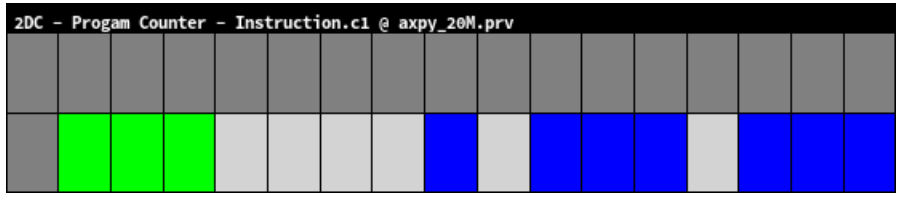
- Aggregated metrics
- Global pattern
- Timing models

vfadd	vsetvli	vfmul	vle	vse	vfmv
313	628	313	626	312	1







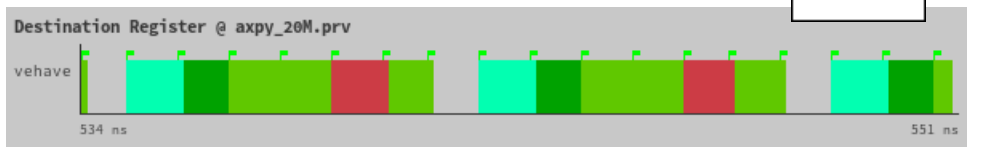


TRACE ANALYSIS

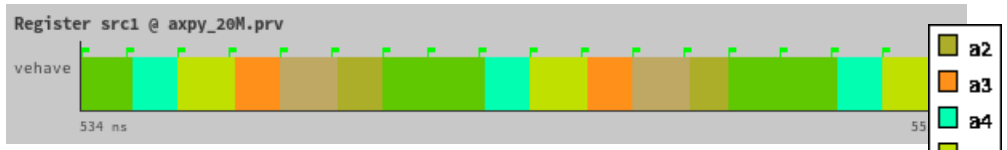
- Microscopic pattern

- Co-design insight
  - For architects
  - For compiler designers
  - For application developers

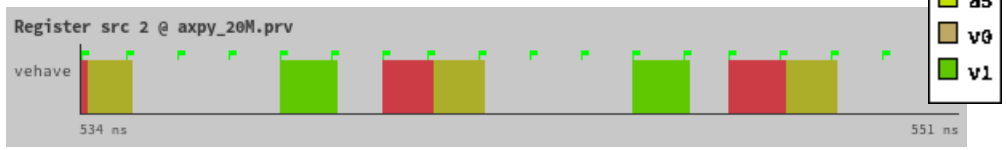
Dst reg

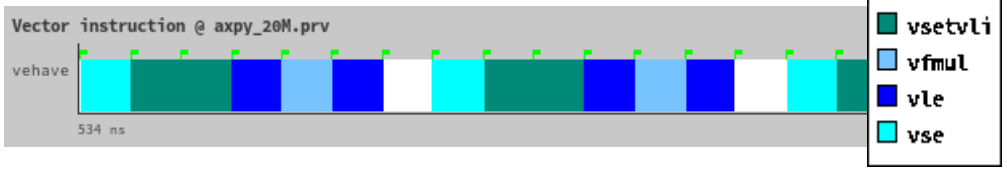


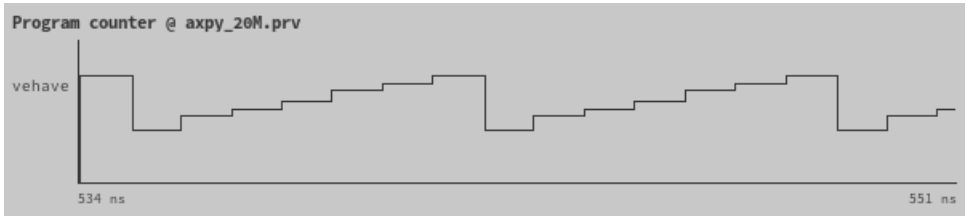
src1 reg



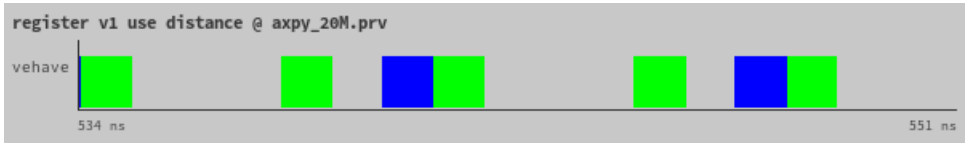
src2 reg







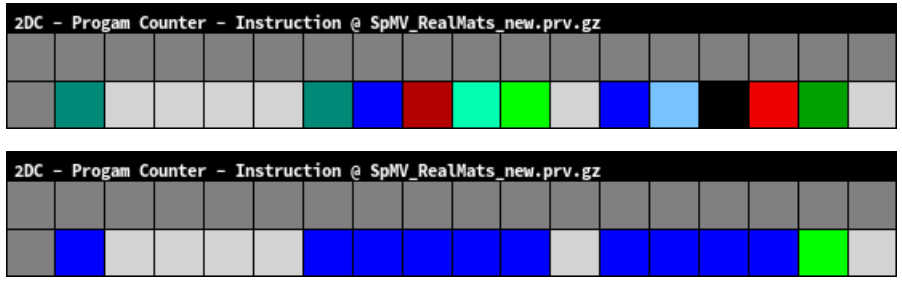
Reg v1 use distance

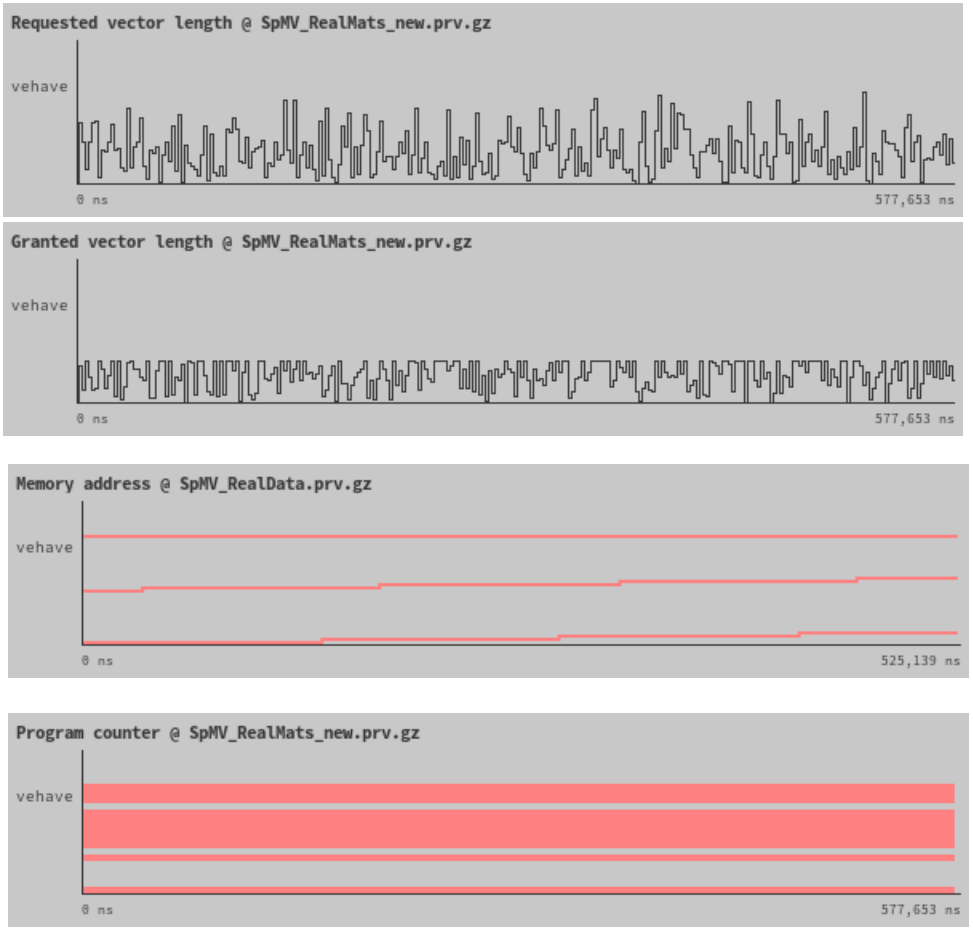


Reg v2 use distance

TRACE ANALYSIS

- SpMV on real matrix





CONCLUSION

JESUS LABARTA

CONCLUSION

- MPI + OpenMP (task based)
  - Interoperability (TAMPI)
  - Dynamic load balance (DLB)
- Hierarchical / Nesting
- Long vectors
- Specific acceleration devices
- Homogenizing heterogeneity
- Productivity
- Insight

- Developing architecture and system software

Good progress