SYNOPSYS®
Silicon to Software™

Tmatch, a flexible stereo image matching accelerator designed with ASIP Designer

University day

Erik Brockmeyer

17-Nov-2021

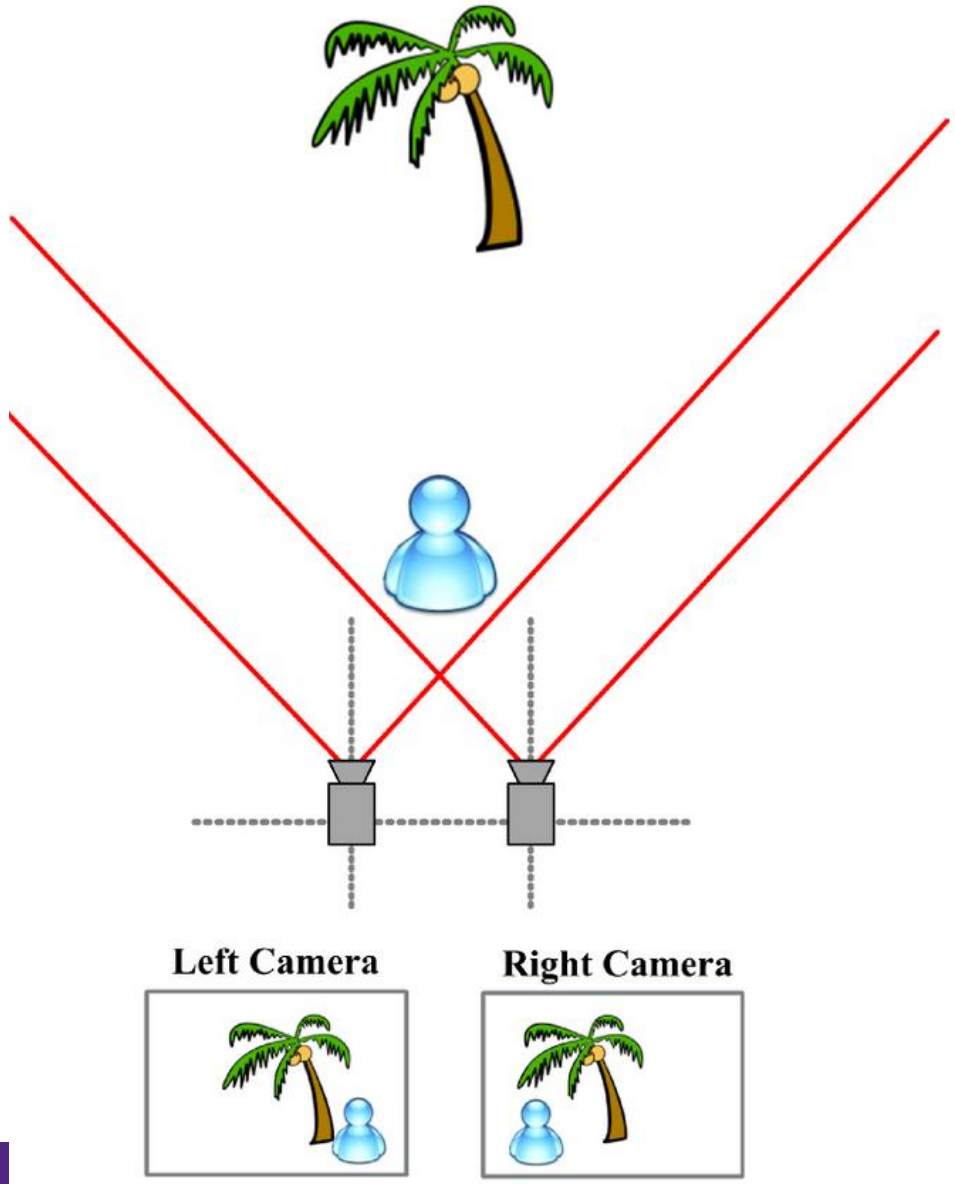
CONFIDENTIAL INFORMATION

The information contained in this presentation is the confidential and proprietary information of Synopsys. You are not permitted to disseminate or use any of the information provided to you in this presentation outside of Synopsys without prior written authorization.

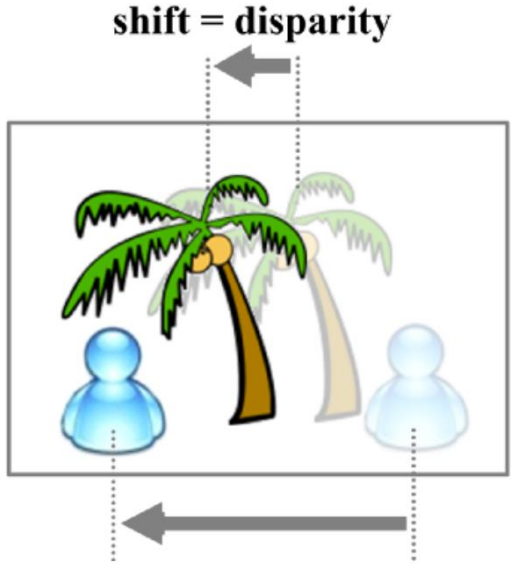
IMPORTANT NOTICE

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to update this presentation or develop the products with the features and functionality discussed in this presentation. Additionally, Synopsys' services and products may only be offered and purchased pursuant to an authorized quote and purchase order or a mutually agreed upon written contract with Synopsys.

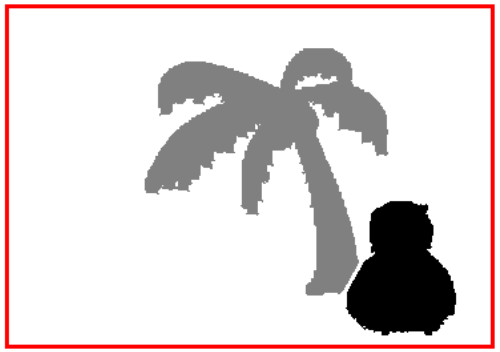
Disparity map of a stereo image



- Disparity is horizontal shift

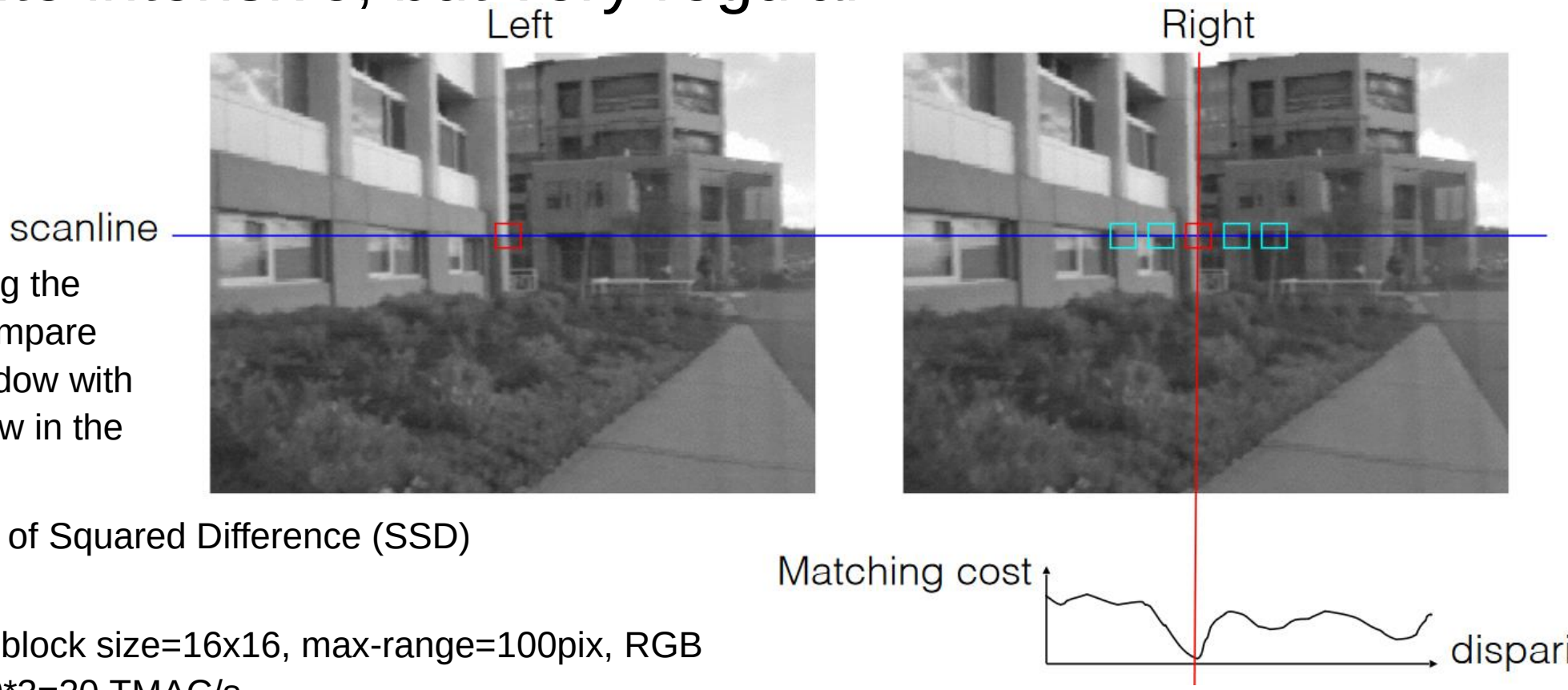


- Computed disparity map (for each pixel)



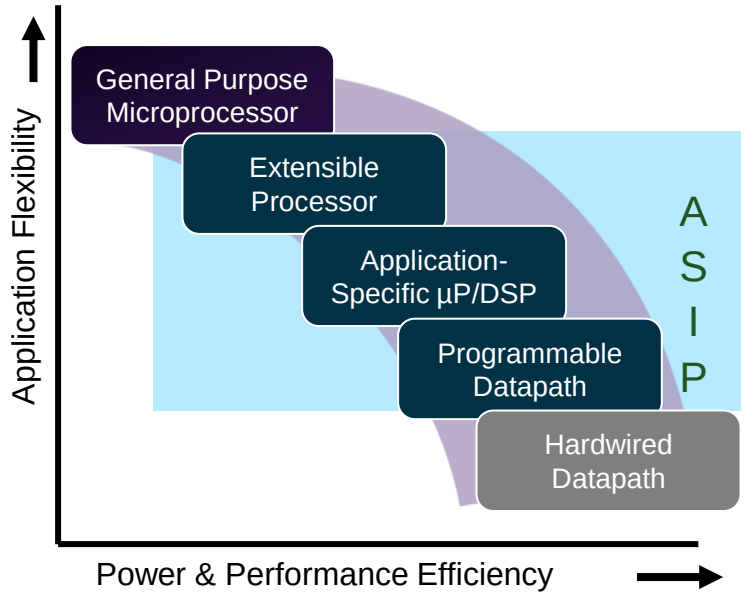
- Used to compute depth

Very compute intensive, but very regular



- Slide a window along the epipolar line and compare contents of that window with the reference window in the left image
- Matching cost: Sum of Squared Difference (SSD)
- 4K images, >30fps, block size=16x16, max-range=100pix, RGB
  →8M*30*16*16*100*3=20 TMAC/s
  →20K mults/cycle @1GHz
- Architecture optimization: Specialization, ILP, vectorization, multicore, memory?? → Design space exploration
- Application optimization: Reuse, algorithm? → Arch/appl co-optimization
- Regular algorithm

ASIP architecture exploration



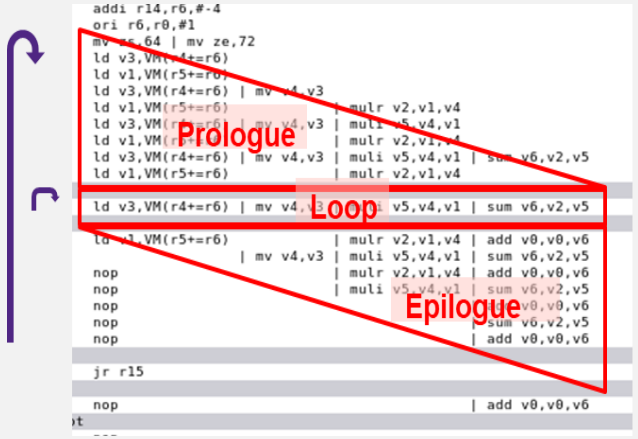
- ASIP: Application-Specific Instruction Set Processor
  - Anything between general-purpose µP and hardwired data-path
    - http://en.wikipedia.org/wiki/Application-specific_instruction-set_processor
  - Deploys classic hardware tricks (parallelism and customized data paths) while retaining programmability – Hardware efficiency with software programmability
- Common examples
  - DSPs, accelerators, filter engines, packet processors, in-house proprietary processors

ASIP Designer SDK

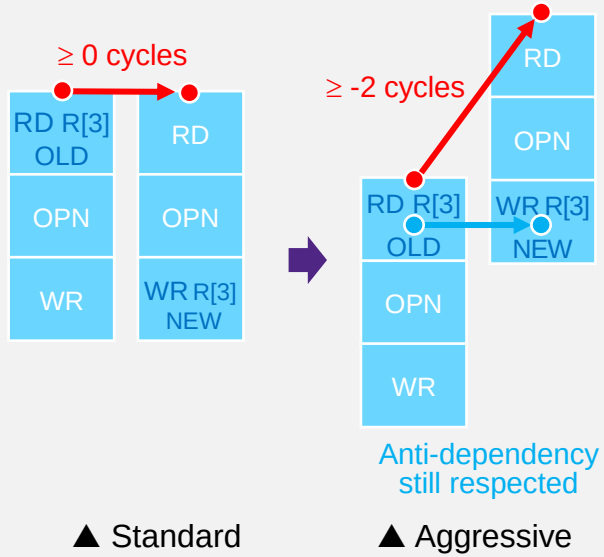
Compiler Back End and Parallel DSP Architectures

- Target architectures
  - ILP with multiple slots
    - E.g. 5 slots, typically heterogeneous
  - Wide SIMD
  - Central or distributed register file(s)
    - Limited capacity
  - Deep pipeline
    - Typically, ~10 stages, with 5 stages for the execute pipe

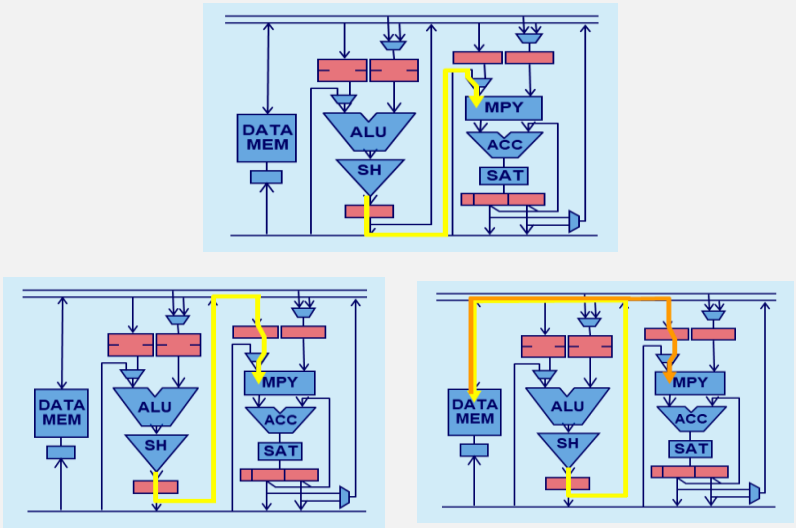
- Compiler optimizations
  - VLIW Scheduling including software pipelining for inner and higher-level loops



  - Aggressive scheduling



  - Register allocation for distributed register files



Out of the box compilation on a trv32p5x

Ideal multicore parallelization would require 166000 cores!

```
7388  4  c.nop
7390  5  mv      x30, x8
7394  5  li      x13, 128
7398  5  li      x8,  128
7402  5  li      x9,  128
7406  5  bltz    x26, 60
7410  5  bge     x26, x19, 56
7414  5  lw      x13, 0(x5)
7418  5  mv      x12, x26
7422  5  bgtz    x26, 6
7426  5  c.li    x12, 0
7428  5  blt     x12, x19, 6
7432  5  c.lwsp  x12, 68(x2)
7434  5  c.add   x12, x25
7436  5  mul     x12, x4,  x12
7440  5  addi    x9,  x12, 1
7444  5  add     x8,  x9,  x13
7448  5  c.addi  x9,  1
7450  5  c.add   x12, x13
7452  5  c.add   x13, x9
7454  5  lbu     x9,  0(x12)
7458  5  lbu     x8,  0(x8)
7462  5  lbu     x13, 0(x13)
7466  5  mv      x15, x29
7470  5  lbu     x12, -1(x15!)
7474  5  lbu     x14, 0(x15)
7478  5  c.and   x9,  x11
7480  5  c.sub   x14, x9
7482  5  c.and   x8,  x11
7484  5  c.sub   x12, x8
7486  5  lbu     x9,  2(x15)
7490  5  c.and   x13, x11
7492  5  mul     x8,  x14, x14
7496  5  c.sub   x9,  x13
7498  5  c.add   x8,  x30
7500  5  mul     x13, x12, x12
7504  5  mul     x30, x9,  x9
7508  5  c.add   x8,  x13
7510  5  c.add   x29, x4
7512  5  c.addi  x26, 1
7514  5  c.add   x8,  x30
7516  4  c.lwsp  x26, 128(x2)
```

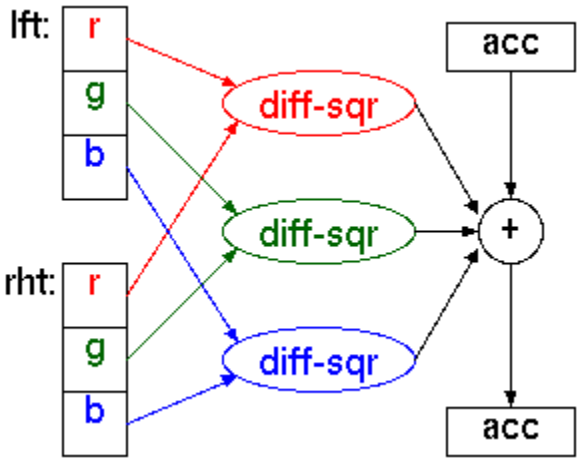
```
44  for (int i=0+half_block_size; i<left->h-half_block_size; i++)
45  {
46    for (int j=0+half_block_size; j<left->w-half_block_size; j++)
47    {
48      if (j+disparity_min-half_block_size<0 || j+disparity_max+half_block_size>COL -1) {
49        for (int range=disparity_min; range<=disparity_max; range++)
50        {
51          SSD = 0;
52
53          for (l_r = -half_block_size + i; l_r <= half_block_size + i; l_r++)
54          {
55            for (l_c = -half_block_size + j; l_c <= half_block_size + j; l_c++)
56            {
57              r_r = l_r;
58              r_c = l_c + range;
59
60              if ((r_c < 0) || (r_c > (COL -1))) {
61                pix[0] = 128;
62                pix[1] = 128;
63                pix[2] = 128;
64              }
65              else {
66                pix[0] = right->data[r_r*COL*CH+std::min(std::max(0, r_c), COL-1)*CH];
67                pix[1] = right->data[r_r*COL*CH+std::min(std::max(0, r_c), COL-1)*CH+1];
68                pix[2] = right->data[r_r*COL*CH+std::min(std::max(0, r_c), COL-1)*CH+2];
69              }
70
71              SSD += square(left->data[l_r*COL*CH+l_c*CH] - pix[0])
72                   + square(left->data[l_r*COL*CH+l_c*CH+1] - pix[1])
73                   + square(left->data[l_r*COL*CH+l_c*CH+2] - pix[2]);
74            }
75
76          }
77
78          if (SSD < SSD_value[i*COL+j] - THRESHOLD)
79          {
80            disp[i*COL+j] = range;
81            SSD_value[i*COL+j] = SSD;
82          }
83        }
84      }
    else {
```

26 cycles (avg)

- 166 Tcyc/s
- @1GHz → 166 Kcores
- Very unpractical

Straight forward specialization and RGB-vectorization

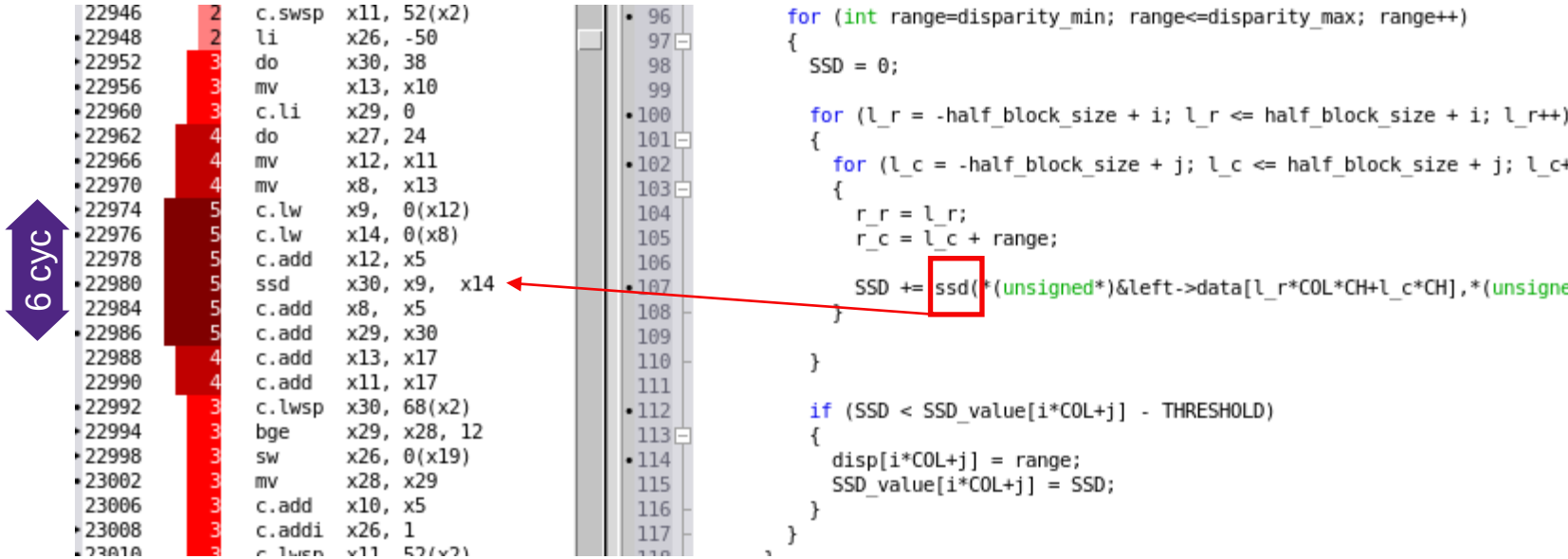
13x faster, though still far insufficient

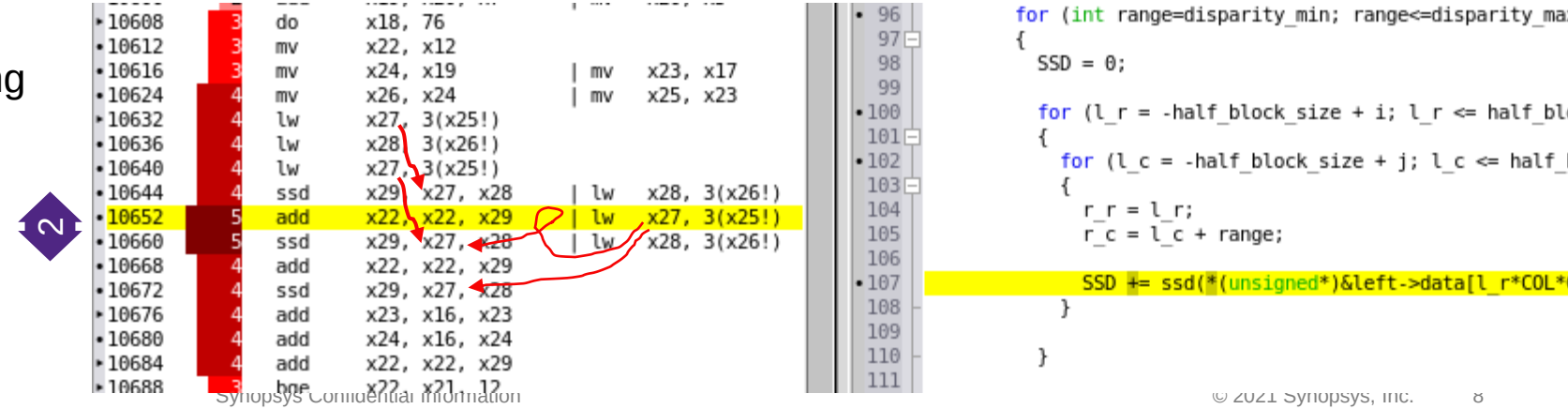


- SIMD3: RGB pixel in w32
- Specialized diff-sqr-acc
- 2 cycle innerloop feasible:
  - Zloop + aggressive software pipelining
  - ILP: Load || compute

→ 6.4 Titer/s → 12.8 Tcyc/s

- @1GHz → 12.8 Kcores
- Still very unpractical





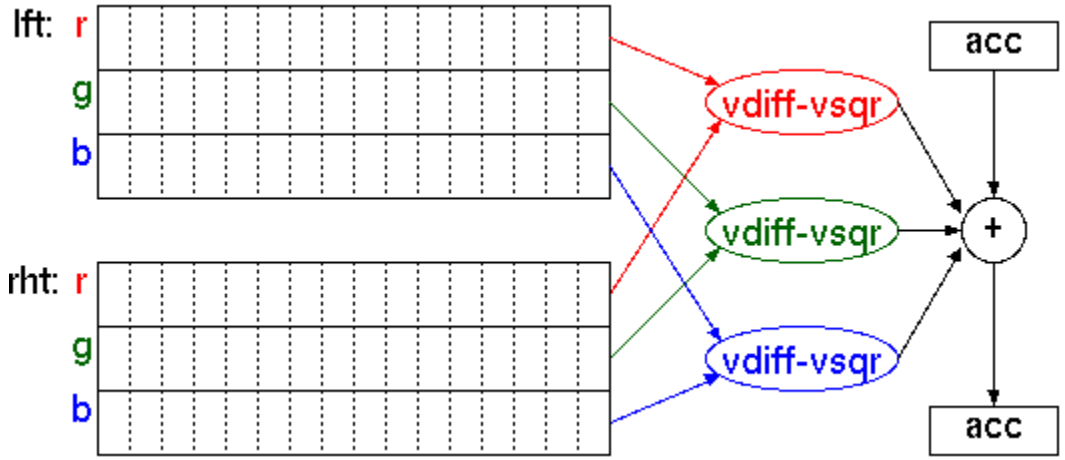
Further vectorization

RVV with custom vector instr alike solution
DP: 512b + 1 ldst unit + with optimal (dyn) scheduling → 2 cycle innerloop

16x faster, but still far insufficient

- SIMD3x16 → 24x16=384b vectors
- Specialized vector diff-sqr-acc
- 2 cycle innerloop feasible, assuming:
  - Zloop + software piplineing
  - ILP: Load || compute
- 16x faster: 400 Giter/s → 800 Gcyc/s
- @1GHz → 800 cores
- Still very unpractical

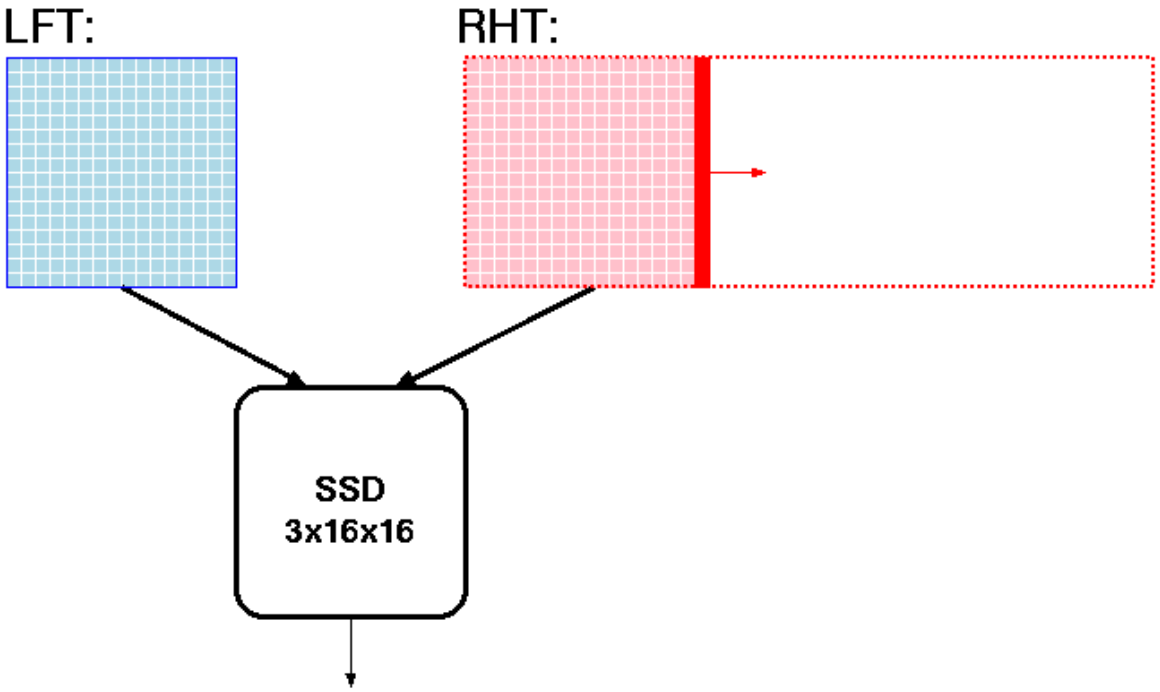
- How to proceed?
  - Outerloop SIMD over whole image?
    →4Kpix vectors?? Wide memory??




```
14952    2  nop                    | lw   x27, 0(x7)
14960    3  addi   x29, x5,  0     | nop
14968    3  addi   x30, x25, 0     | nop
14976    3  addi   x28, x16, 0     | nop
14984    3  do     x23, 32         | nop
14992    3  nop                    | lv   v1,  x14(x30!)
15000    3  nop                    | lv   v0,  x9(x29!)
15008    4  nop                    | lv   v0,  x9(x29!)
15016    4  ssd    x28+=v0,  v1    | lv   v1,  x14(x30!)
15024    3  addi   x0,  x0,  0     | nop
15032    3  bge    x28, x27, 24    | nop
15040    3  nop                    | sw   x26, 0(x6)
15048    3  addi   x27, x28, 0     | nop
15056    3  addi   x26, x26, 1     | nop
15064    3  addi   x25, x25, 1     | nop
15072    2  nop                    | sw   x27, 4(x7!)
15080    2  addi   x24, x24, 1     | nop
```

2

```
128         pix_t chess_storage(VM)* p_lpixel = &lpixels[(i-half_block_s
129         pix_t chess_storage(VM)* p_rpixel = &rpixels[(i-half_block_s
130
131         int row_inc = 2;
132
133         for (l_r = -half_block_size + i; l_r < half_block_size + i;
134         {
135           //Note: block size must be integer multiple of Vector leng
136           for (int s = 0; s < 2*half_block_size; s+=VSIZE) {
137             SSD = pix_ssdac(SSD,*(vpix_t chess_storage(VM)*)(p_lpixe
138           }
139
140           p_lpixel+=COL*row_inc; p_rpixel+=COL_exp*row_inc;
141
142         }
143
144
```



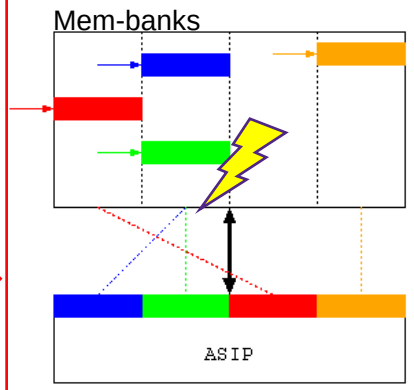
Even more specialization/SIMD!

16x16pix shift reg to avoid very wide loads

- Datapath: matrix of diff-sqr-sum; 768 MAC
- M-Regs: SIMD3x16x16 (LFT+RHT) + shift
- Vector load/store: SIMD3x16=384b
- Column reads? → Vector address

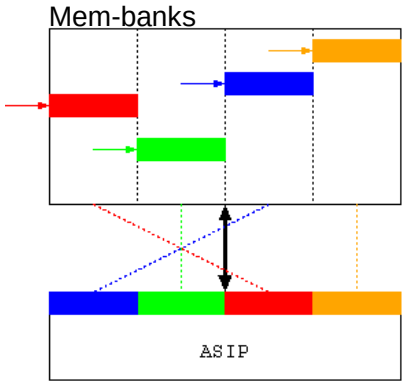
- Vector address has an arbitrary address per lane
- Read column

Vector address gather/scatter



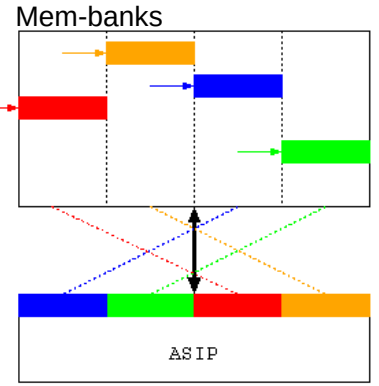
Expensive and indeterministic

Vector address conflict free



Large shuffle can be expensive and timing critical <16? <32?

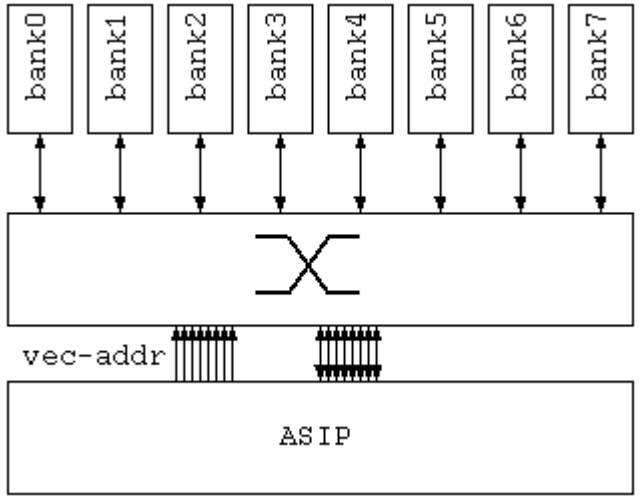
Vector address rotate only



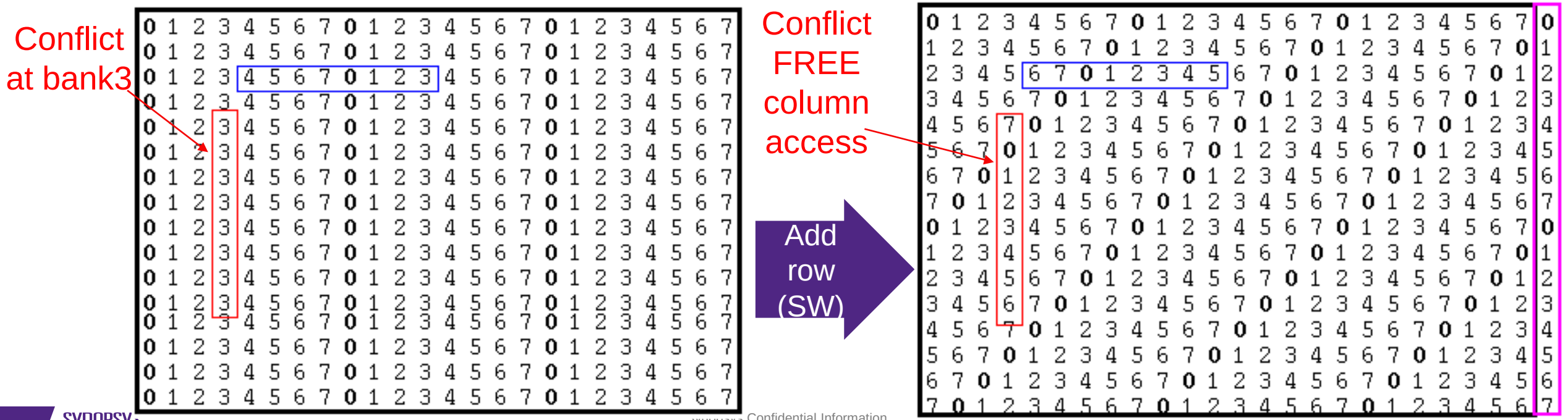
Can be too limited

Key architectural features

Row and column image access



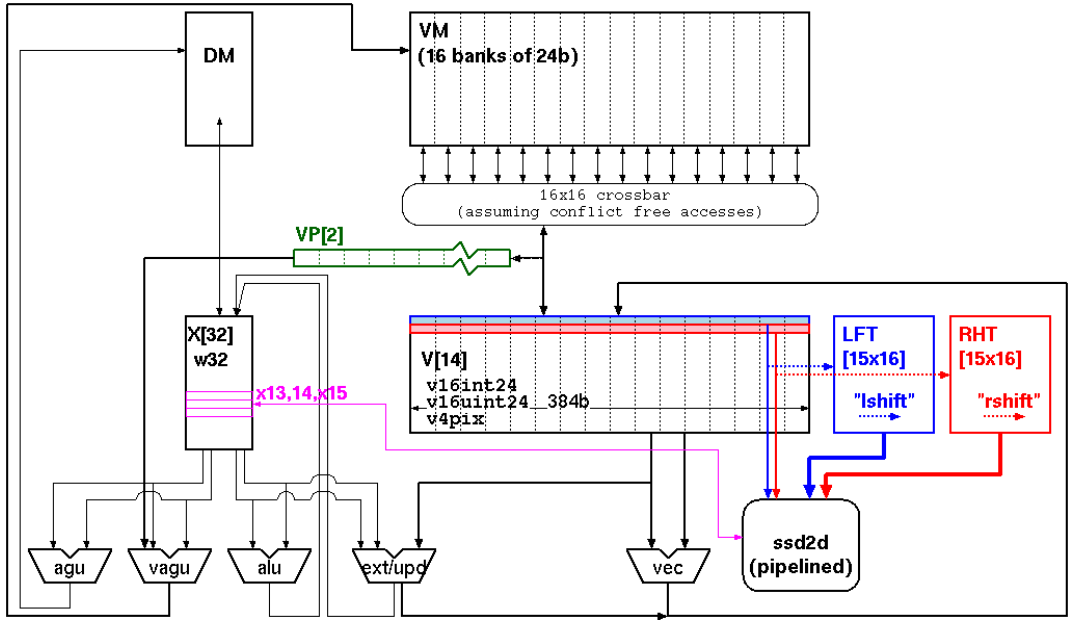
- Images are typically stored sequentially in memory by row
- “Scalar” addressing (1 address per vector) will allow row access only
- “Vector” addressing (1 address per vector element) will allow both row and column access
  - Memory layout of 1 pixel per memory address
  - But….regular layout will cause memory access conflict on column addresses
  - → Optimized data layout for conflict free row and column access



Single cycle innerloop computing a 16x16 SSD matrix

23x faster; without extra memory bandwidth

- Specialized regs/datapath/memIf/pipelined
- Column reads? → Vector address
- ILP: Load || shift || compute (1 cycle innerloop!)
- 35Gcyc/s ; @1GHz → 35 cores

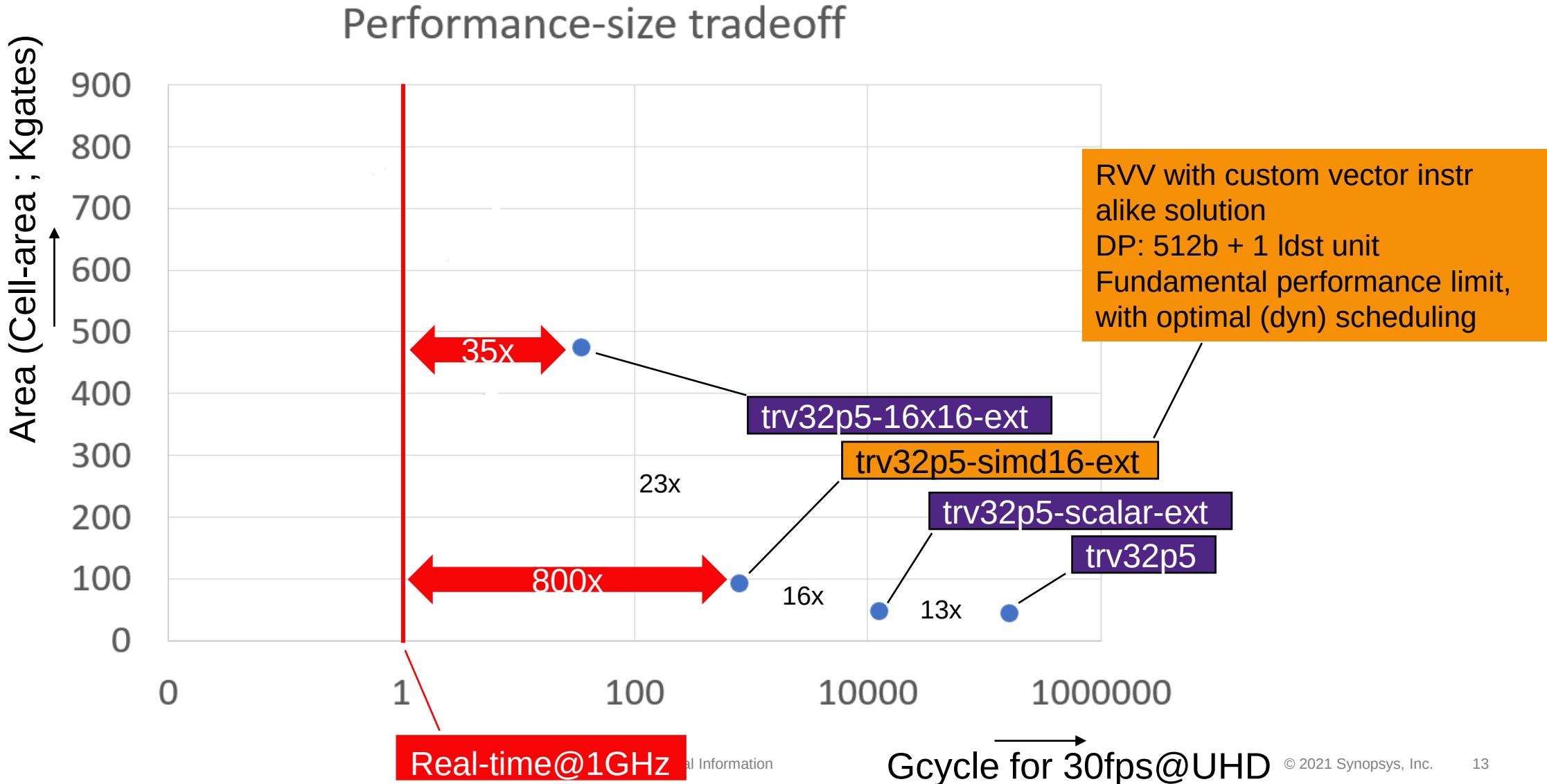



```
1988    2   rshift                  | lv   v0,  x1(vp1!)
1996    2   lshift                  | lv   v1,  x1(vp0!)
2004    2   rshift                  | lv   v0,  x1(vp1!)
2012    2   lshift                  | lv   v1,  x1(vp0!)
2020    2   rshift                  | lv   v0,  x1(vp1!)
2028    2   lshift                  | lv   v1,  x1(vp0!)
2036    2   rshift                  | lv   v0,  x1(vp1!)
2044    2   lshift                  | lv   v1,  x1(vp0!)
2052    2   rshift                  | lv   v1,  x1(vp0!)
2060    2   lshift                  | lv   v0,  x1(vp1!)
2068    2   rshift                  | lv   v0,  x0(vp1!)
2076    2   rshift
2080    2   lshift
2084    2   do      x8,  24         | lv   v1,  x1(vp0!)
2092    2   addi    x13, x0,  -25   | lv   v1,  x1(vp0!)
2100    2   lw      x14, 12(x2)     | lv   v1,  x1(vp0!)
2108    3   ssd2d   x14,x15,lft,rht | lv   v1,  x1(vp0!)
2116    2   addi    x19, x19, 1
2120    2   addi    x20, x20, 1
2124    2
2128    2   sb      x15, 1(x18!)
2132    1   andi    x14, x9,  7
2136    1   bne     x17, x14, 80
```

```
 55      vpix_t l7; mpix_t ldelay;
 56    • for (int i=0; i<VSIZE-1; i++) chess_unroll_loop(*) {   l7 = *vpl; vpl=vpl+bc(1); ldelay
 57    • l7 = *vpl;
 58
 59      // LOAD right block  at the start of the disparity range
 60    • chess_vector_ptr<pix_t chess_storage(VM)> vpr = vptr_col_init(&rpixels[(y)*rwidth + xc+
 61      vpix_t r7; mpix_t rdelay;
 62    • for (int i=0; i<VSIZE-1; i++) chess_unroll_loop(*) {   r7 = *vpr; vpr=vpr+bc(1); rdelay
 63
 64      // iterate over disparity range AND find the minimum
 65      int val = 0x7fffffff;
 66      int loc;
 67      int idx=DISPARITY_MIN;
 68      vint24_t* restrict pssd2 = SSD;
 69    • for (int range=0; range<DISPARITY_RANGE; range++) {
 70    •   vpix_t r7 = *vpr; vpr=vpr+bc(1);
 71    •   ssd2d(ldelay,l7,rdelay,r7,rdelay,idx,val,loc);
 72      }
 73
 74    •   disp[(y+HALF_BLOCK_SIZE)*width+xc+0+HALF_BLOCK_SIZE] = loc;
 75    }
 76
 77    • if ((y&7) == 0)
```

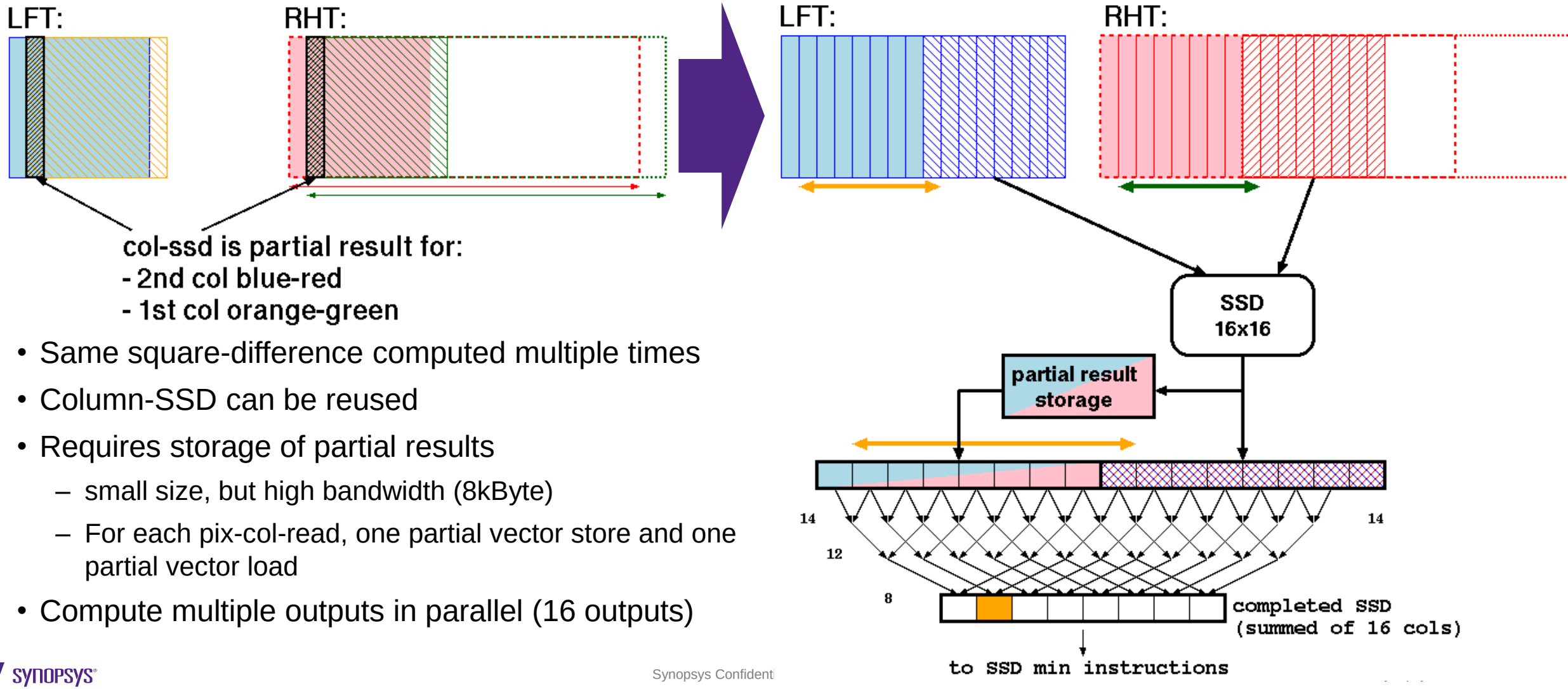
Explored architectures

Still a 35x away from real-time!



Reuse of partial results

Architecture support for efficient storage and processing of partial results



- Same square-difference computed multiple times
- Column-SSD can be reused
- Requires storage of partial results
  - small size, but high bandwidth (8kByte)
  - For each pix-col-read, one partial vector store and one partial vector load
- Compute multiple outputs in parallel (16 outputs)

Rewrite application code, compiler solves all details

Match computation memory performance to optimize area

→ Optimization of C code: Native code development / testing

→ Improved performance because partial result reuse (virtually 16x)

→ 3 cycle ssd2d instruction (256 MAC/cycle); mixed latency

→ overall 5x more performance and 3x area reduction

```
5424   126616  rshift                  | lv   v0,  x0(vp1!)
5432   126616  rshift                  | lv   v1,  x11(vp0!)
5440   126616  lshift                  | lv   v4,  0x10(x3)
5448   126616  do      x18, 0x28       | lv   v5,  0x10(x29!)
5456   126616  ssd2d   v6              | lv   v1,  x11(vp0!)
5464   126616  lw      x13, 0xc(x2)    | lv   v2,  0x0(x3)
5472   12788216 sum2d  v7,v6,v5        | lv   v5,  0x10(x29!)
5480   12788216 ssd2d  v6              | lv   v1,  x11(vp0!)
5488   12788216 min2d  v4,v7,v4        | sv   v6,  0x10(x30!)
5496   126616  addi    x23, x23, 0x10  |
5504   126616  ext_v4w8 x29,v2,x19     |
5512   126616  sw      x29, 0x10(x27!) |
5520   126616  ext_v4w8 x13,v2,x11     |
5528   126616  sw      x13, 0x10(x26!) |
5536   126616  ext_v4w8 x29,v2,x1      |
5544   126616  ext_v4w8 x13,v2,x21     |
```

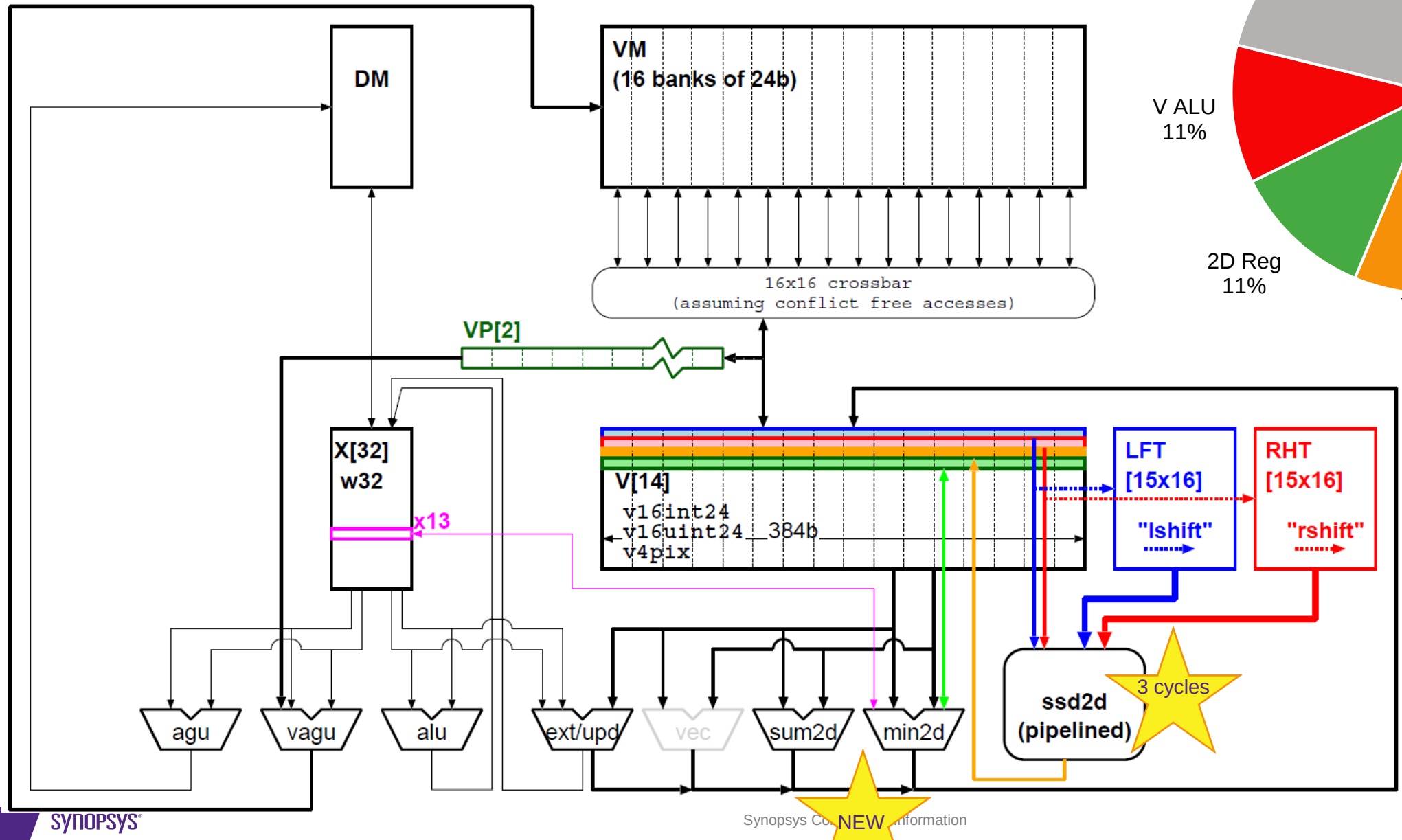
```
// iterate over disparity range AND find the minimum
vint24_t vSSDmin = bc(0xffffff);
vint24_t vDSPmin = bc(0);
int idx=-disparity;
vint24_t* restrict pssd2 = SSD;
for (int range=0; range<DISPARITY_RANGE; range++) chess_loop_range(5,)  {
  vpix_t r7 = *vpr; vpr=vpr+bc(1);
  vint24_t ssd_col = ssd2d(ldelay,l7,rdelay,r7,rdelay);
  vint24_t vSSDblc = sum2d(ssd_col,SSD[range]);
  vSSDmin = min2d(vSSDblc,vSSDmin,idx,vDSPmin);
  pssd2[range] = ssd_col; // SSDcol
}
*(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+0 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,0);
*(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+4 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,1);
*(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+8 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,2);
*(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+12+HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,3);
```

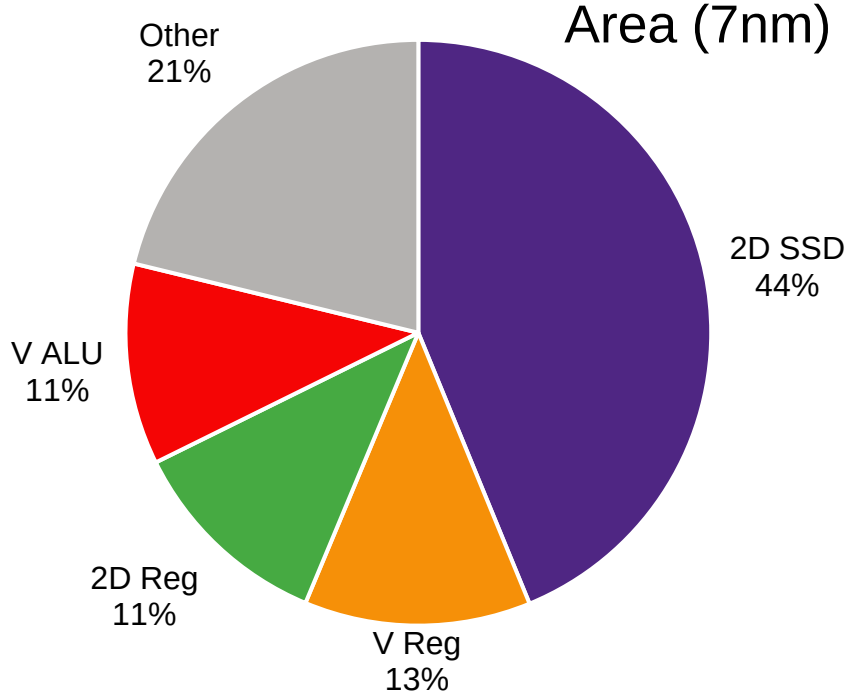
3 cycle loop
→3cyc ssd2d

2 load, 1 store
Pix and sum in VM

Tmatch v1.0 - datapath block diagram

20% smaller, 6.5x faster (5.5x more needed)





Tmatch v1.1 feature additions

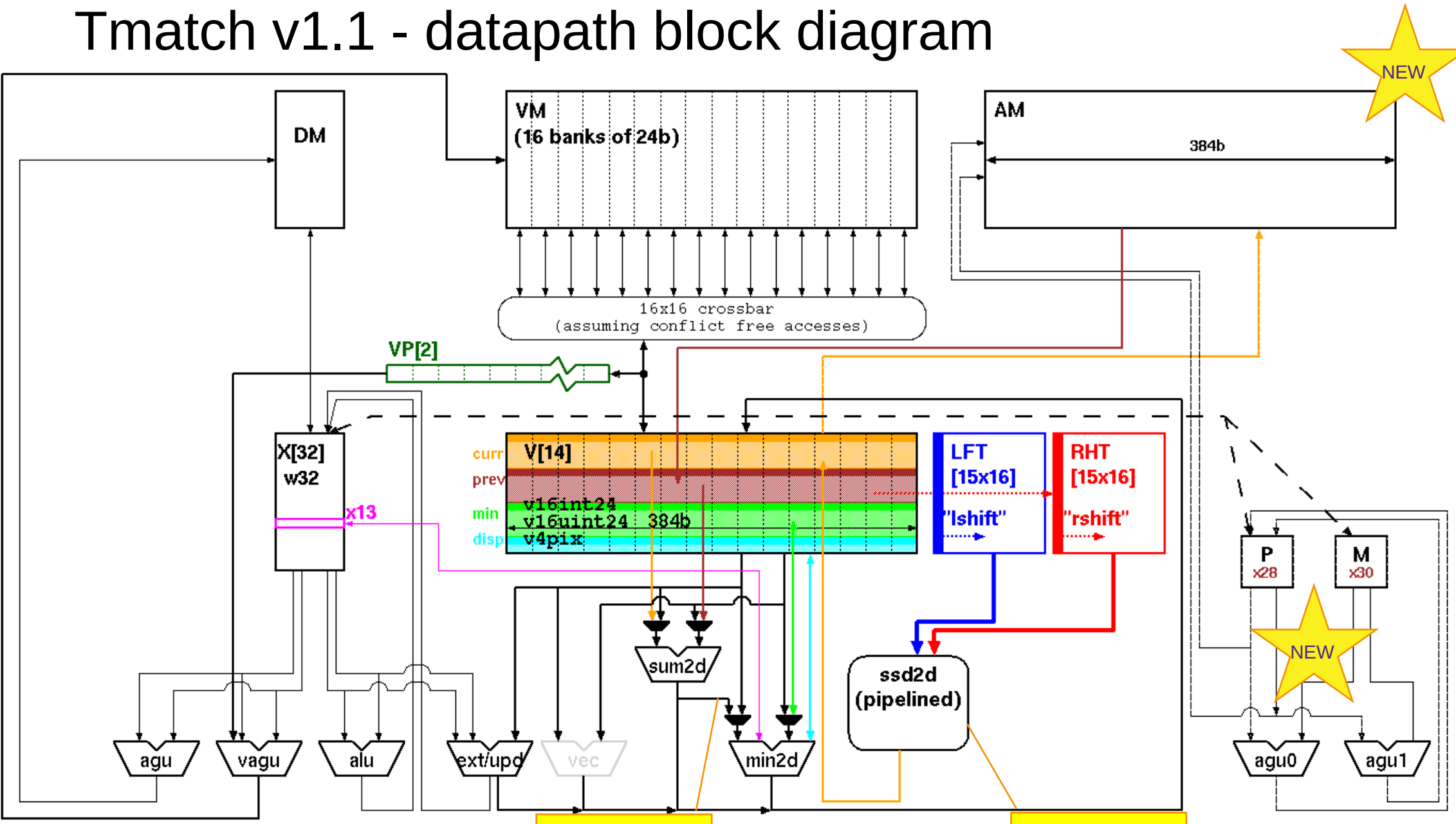
1.6x bigger, 2.4x faster: Enable single cycle kernel inner loops

- Separate Vector accumulator memory (AM)
  - 24b x 16 wide memory to accumulate SSD across pixel disparity range
  - Separate read and write ports
- Scalar based AGU for AM
  - Base + increment (post-modify) using dedicated scalar (X) registers
- 5 slot instruction parallel issue
  - Does not increase instruction word size (remains 64-bit)

```
3944    2  nop                    | l24  x28, 0(x3)
3952    2  nop                    | nop                  | lv   lft, x30(vp1!)   | nop                  | nop
3960    2  nop                    | nop                  | lv   rht, x30(vp0!)   | nop                  | nop
3968    2  addi    x29, x28, 0    | lv   v13, 1(x3)
3976    2  nop                    | lv   v9,  17(x3)
3984    2  nop                    | nop                  | lv   rht, x30(vp0!)   | nop                  | nop
3992    2  nop                    | nop                  | lv   rht, x30(vp0!)   | nop                  | nop
4000    2  nop                    | ld v4,AM{x28+=x30}   | lv   rht, x30(vp0!)   | nop                  | ssd2d v2,v0,v1
4008    2  nop                    | ld v4,AM{x28+=x30}   | lv   rht, x30(vp0!)   | nop                  | ssd2d v2,v0,v1
4016    3  st v2,AM{x29+=x30}     | ld v4,AM{x28+=x30}   | lv   rht, x30(vp0!)   | minsum2d v9,v13,v4   | ssd2d v2,v0,v1
4024    2  st v2,AM{x29+=x30}     | ld v4,AM{x28+=x30}   | nop                   | minsum2d v9,v13,v4   | ssd2d v2,v0,v1
4032    2  st v2,AM{x29+=x30}     | ld v4,AM{x28+=x30}   | nop                   | minsum2d v9,v13,v4   | ssd2d v2,v0,v1
4040    2  st v2,AM{x29+=x30}     | ld v4,AM{x28+=x30}   | nop                   | minsum2d v9,v13,v4   | ssd2d v2,v0,v1
4048    2  st v2,AM{x29+=x30}     | nop                  | nop                   | minsum2d v9,v13,v4   | nop
4056    2  st v2,AM{x29+=x30}     | nop                  | nop                   | minsum2d v9,v13,v4   | nop
4064    2  addi    x13, x12, 0    | add  vp0,vp0,v0
4072    2  ext_v4w8 x26,v13,x8    |
4080    2  ext_v4w8 x26,v13,x30   | sw   x26, 16(x22!)
4088    2  ext_v4w8 x26,v13,x1    | sw   x26, 16(x23!)
```

```
44      // iterate over disparity range AND find the minimum
45      vuint24_t vSSDmin = bcu(0xffffff);
46      vint24_t  vDSPmin = bc(0);
47      int idx=-disparity;
48      vuint24_t chess_storage(AM) * restrict pssd2 = SSD;
49      for (int range=0; range<disparity_range; range++) chess_loop_range(7,) chess_prepare_for_pipelining {
50        vpix_t r7 = *vpr; vpr=vpr+bc(1);
51        rdelay = rshift(rdelay,r7);
52        vuint24_t ssd_col = ssd2d(ldelay,rdelay);
53        vuint24_t vSSDblc = sum2d(ssd_col,SSD[range]);
54        vSSDmin = min2d(vSSDblc,vSSDmin,idx,vDSPmin);
55        pssd2[range] = ssd_col; // SSDcol
56      }
57      *(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+0 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,0);
58      *(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+4 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,1);
59      *(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+8 +HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,2);
60      *(int*)&disp[(y+HALF_BLOCK_SIZE)*width+xc+12+HALF_BLOCK_SIZE] = ext_v4i8(vDSPmin,3);
61    vpr=vpr+bc(1-disparity_range);
```

Tmatch v1.1 - datapath block diagram

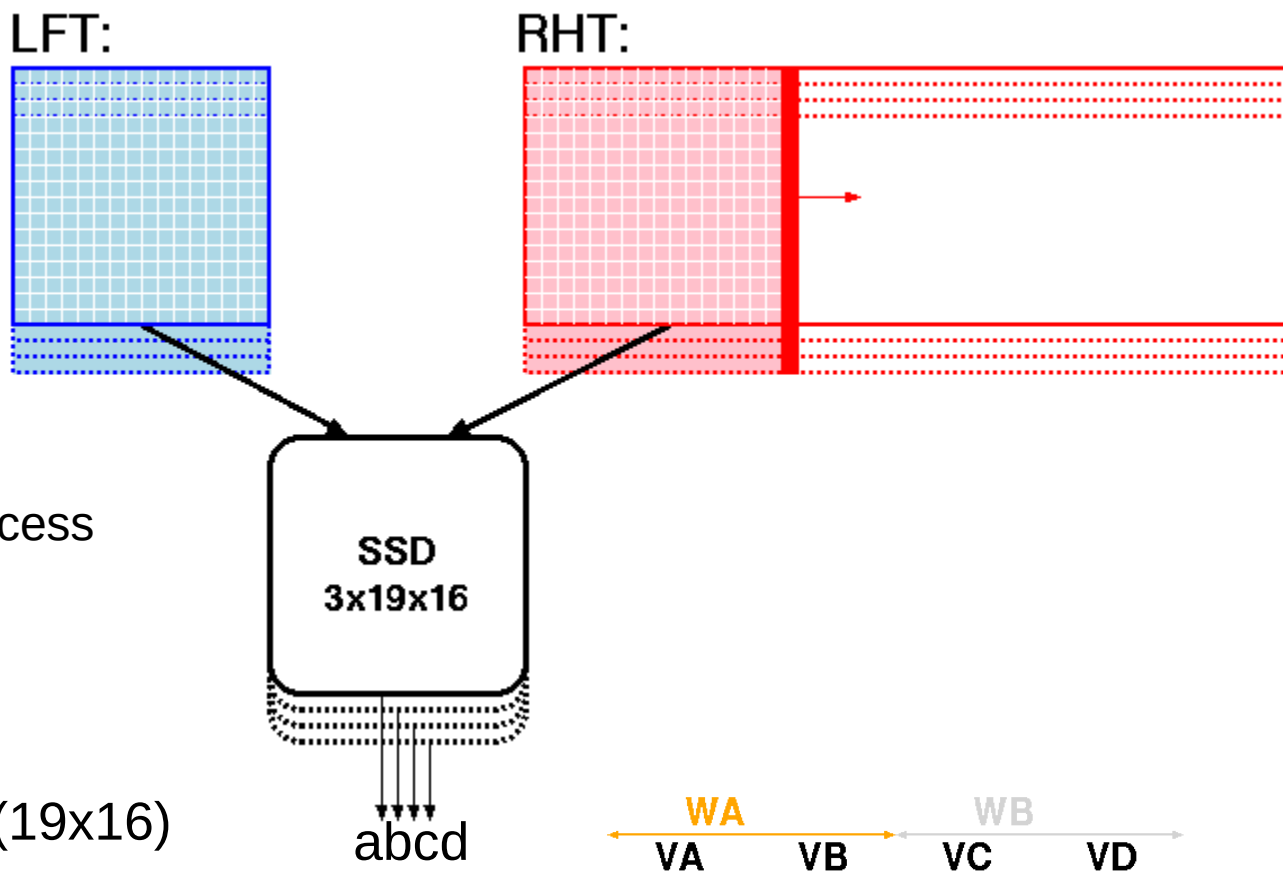


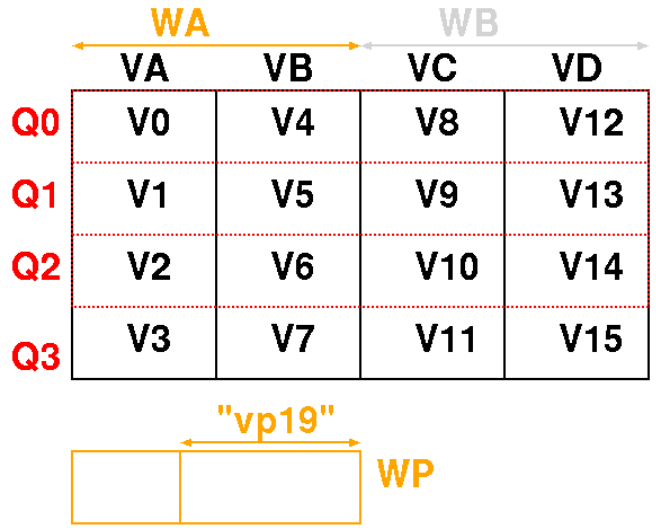
Tmatch V1.2 feature additions

Extend architecture to process 4 rows in parallel

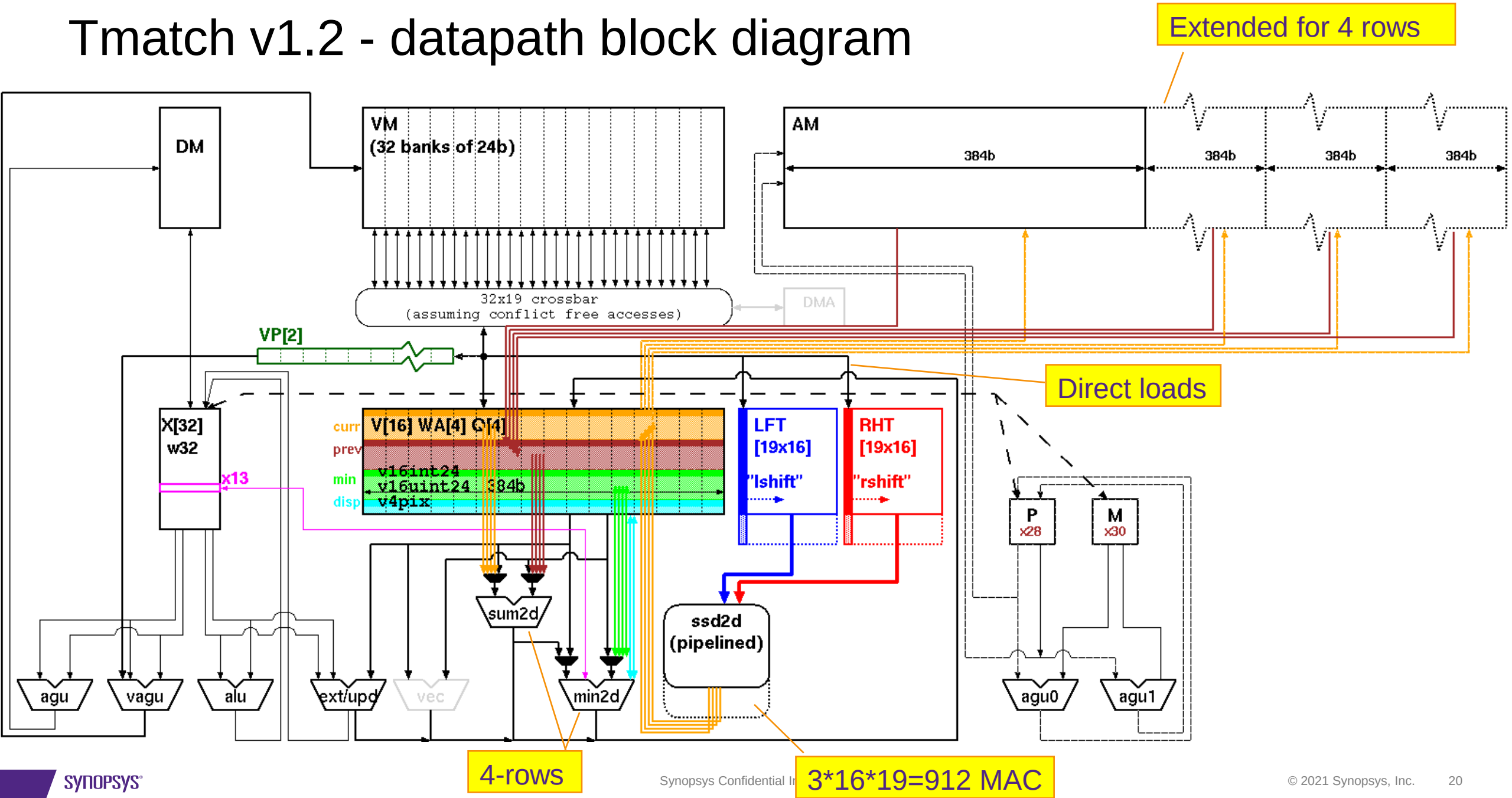
1.23x bigger, 3.62 faster:

- Expand VM to 32-banks
  - Enable parallel (awkward) 19-high column direct access (4 rows)
  - Provide bandwidth to support DMA interleaved access
    - 13 of 32-banks available every cycle
- Expand AM width by x4
  - Support accumulation of 4 row SSD
- Extend LFT, RHT, and ssd2d to support 4 rows (19x16)
  - 4x performance with <25% increase in size
- Restructure V to allow double (W) and quad (Q) access
  - Minimize register file size increase





Tmatch v1.2 - datapath block diagram



Massive parallelism by combining specialization, ILP, SIMD

Even higher performance if we account for the partial result reuse

- Count OPS in Tmatch inner loop

Hwdo:
2: cmp+add

64 loads + 1 AGU

64 loads + 1 AGU

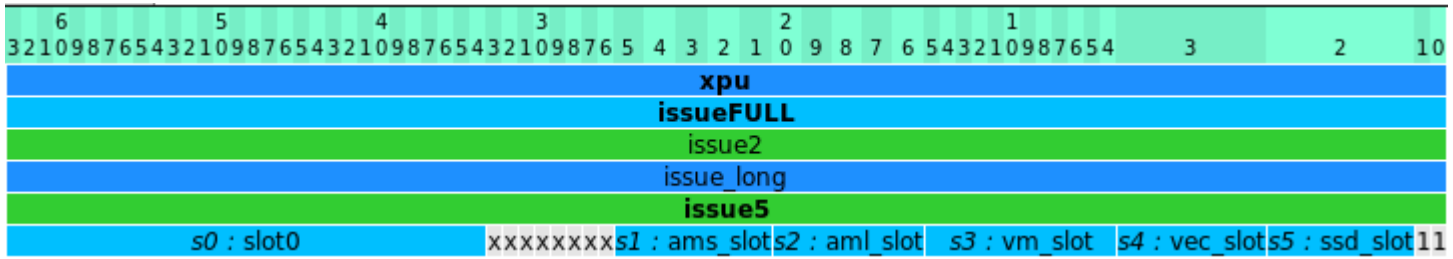
20 loads
20 AGU
2 boundary check + incr
20 boundary padding
256 “rshift”

4*98 add (sum)
4*16 min-val
4*16 min-idx
1 idx++

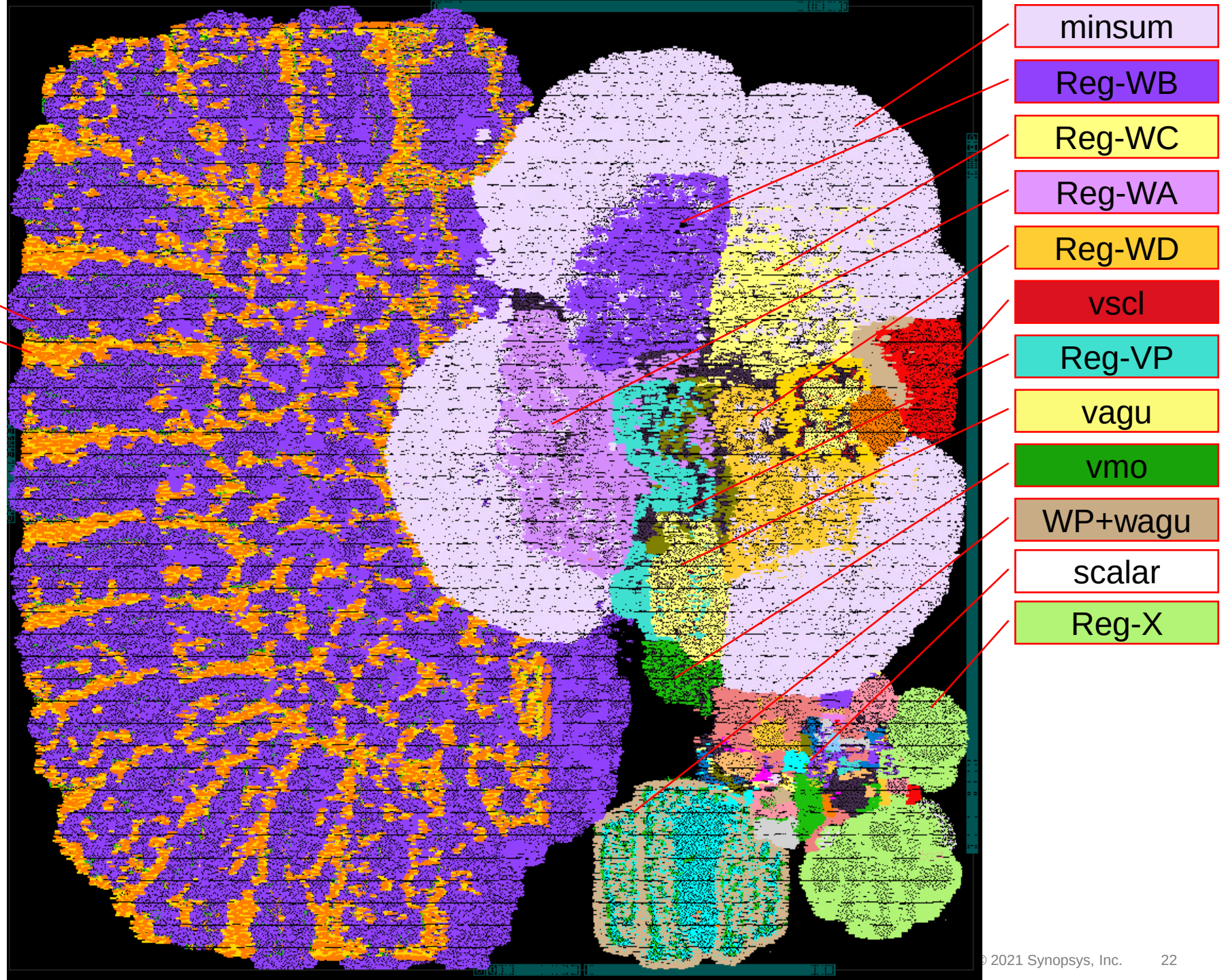
912 diff
912 mul
608 sum (RGB)
432 sum-col

```
928   2  nop                      | ld q1,AMq{x28+=x30}  | lv  rht,  VMw[wp1+=1%lin],x24 | nop                    | ssd2d q0,lft,rht
936   3  st q0,AMq[x29+=x30]     | ld q1,AMq{x28+=x30}  | lv  rht,  VMw[wp1+=1%lin],x24 | minsum2d q0,q1,q2,wa3  | ssd2d q0,lft,rht
944   2  st q0,AMq[x29+=x30]     | ld q1,AMq{x28+=x30}  | nop                           | minsum2d q0,q1,q2,wa3  | ssd2d q0,lft,rht
952   2  st q0,AMq[x29+=x30]     | ld q1,AMq{x28+=x30}  | nop                           | minsum2d q0,q1,q2,wa3  | ssd2d q0,lft,rht
```

- 3835 OPS/cycle in inner loop @ 1GHZ = 3.8 TOPS
- 3.8TOPS/1.5W = 2.5 TOPS/W

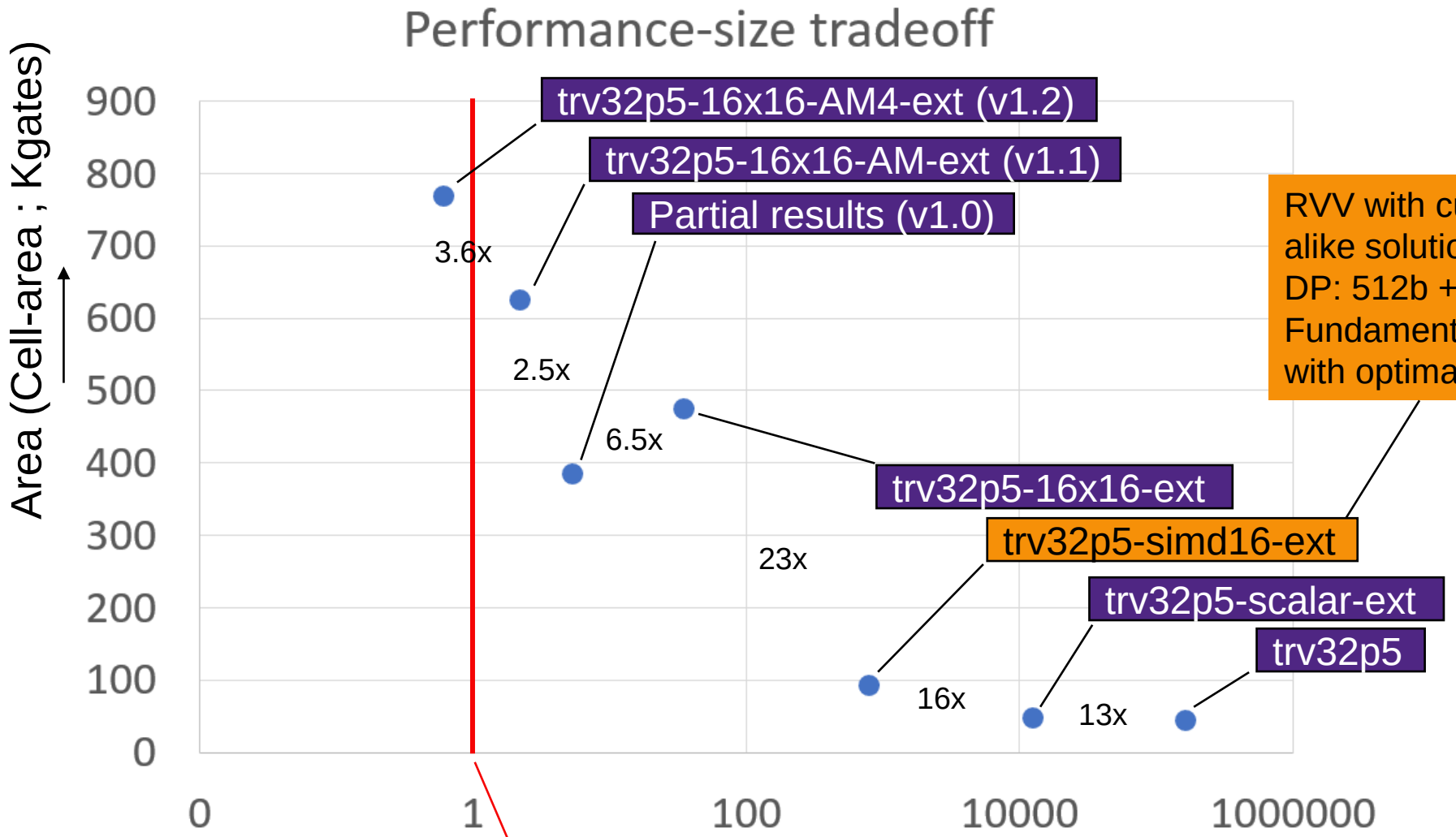


Synthesis 7nm



- 768kGates (cell area)
- Easily meeting 1GHz
  - Pipelined instructions
- No congestion problems
- Memory:
  - No memory timing problems (including shuffle)
  - Memory similar size as core (with cyclic line buffers)
  - Sufficient memory bandwidth to stream in/out data
  - PM is small

Explored architectures



Conclusion

- ASIP designer allows to easily explore many architecture alternatives (special functions/regs/connections, SIMD, ILP, pipeline, memory-IF, …)
- Compiler and synthesis in the loop give fast and accurate feedback
- High level application description (C code) enables easy optimization (Native verification)
- Compiler will solve all gritty details
- Efficient solutions ranging from very flexible to very dedicated architectures
- Programmability allows easier integration and wider applicability


SYNOPSYS®
Silicon to Software™