The Libre-SOC Hybrid 3D CPU

Draft SVP64 in-place Matrix Multiply and FFT / DCT for the Power ISA

OpenPOWER Summit 2021

Sponsored by NLnet Grants and NGI POINTER Grants

28th Oct 2021

Overview of Libre-SOC goals

- To create power-efficient mass-volume products
- To leverage the OpenPOWER ecosystem to do so
- To be entirely transparent for Security reasons
- To empower businesses to bring Secure transparent mass-volume products to market
- Mass-volume end-user products need 3D, Video, Audio **therefore we require small-size Matrices (3x3 but not with 75% utilisation, and 4x4) and the core strategic parts of A/V CODECs and that means DCT and FFT.** Anything else is a bonus (NTT with Galois Field bitmanip)

Overview of SVP64 goals

- High performance and high performance/watt
- Reduced code density (reduced I-Cache usage) https://arxiv.org/abs/2002.10143 - 3.5x power reduction
- Remain accessible for assembler writers and compilers alike
- Introduce true Vectorisation to the Power ISA (VSX is Packed SIMD)
- Be adopted via the external OPF ISA WG RFC process (not: be a non-official custom extension. proprietary custom extensions conflict with mass-volume adoption)

Reminder of Simple-V

```
https://libre-soc.org/openpower/sv/overview/
Greatly simplified (like x86 "REP" instruction):

 for (i = 0; i < VL; i++)
     GPR[RT+i] <= GPR[RA+i] + GPR[RB+i];

function op_add(RT, RA, RB, predr) # add not VADD!
 int i, id=0, irs1=0, irs2=0;
 for (i = 0; i < VL; i++)
   if (GPR[predr] & 1<<i) # predication
     GPR[RT+id] <= GPR[RA+irs1] + GPR[RB+irs2];
    if (reg_is_vectorised[RT]) { id += 1; }
    if (reg_is_vectorised[RA]) { irs1 += 1; }
    if (reg_is_vectorised[RB]) { irs2 += 1; }
```

SVP64 REMAP system

```
Register offsets are "REMAP"ed through a Hardware FSM
https://libre-soc.org/openpower/sv/remap/
remarkably similar to ZOLC
https://www.researchgate.net/publication/224647569

function op_add(RT, RA, rs2, predr) # add not VADD!
 int i, id=0, irs1=0, irs2=0;
 for (i = 0; i < VL; i++)
   if (GPR[predr] & 1<<i) # predication
     GPR[RT+REMAP(id)] <= GPR[RA+REMAP(irs1)] +
                          GPR[rs2+REMAP(irs2)];
    if (reg_is_vectorised[RT]) { id += 1; }
    if (reg_is_vectorised[RA]) { irs1 += 1; }
    if (reg_is_vectorised[s2]) { irs2 += 1; }
```

Matrix Multiply: Basics

```
(a00 a01 a02  x (b00 b01   =   (c00 c01
 a10 a11 a12)    b10 b11        c10 c11)  = ...
                 b20 b21)

(a00*b00 + a01*b10 + a02*b20 a00*b01 + a01*b11 + a02*b21
 a10*b00 + a11*b10 + a12*b20 a10*b01 + a11*b11 + a12*b21)

 (b00 b01    x (a00 a01 a02  =   (c00 c01 c02
  b10 b11       a10 a11 a12)      c10 c11 c12
  b20 b21)                        c20 c21 c22)  = ...

(b00*a00 + b01*a10  b00*a01 + b01*a11  b00*a02 + b01*a12
 b10*a00 + b11*a10  b10*a01 + b11*a11  b10*a02 + b11*a12
 b20*a00 + b21*a10  b20*a01 + b21*a11  b20*a02 + b21*a12)
```

Matrix Multiply: naive, with python for-loops

```
result = [] # final result
for i in range(len(A)):

  row = [] # the new row in new matrix
  for j in range(len(B[0])):

    product = 0 # the new element in the new row
    for v in range(len(A[i])):
        product += A[i][v] * B[v][j]
    row.append(product) # add sum of product to new row

  result.append(row) # add new row into final result
```

Matrix Multiply: suitable for Hardware scheduling

```
Unsuitable: creates massive Read-After-Write chains

for i in range(len(A)):
  for j in range(len(B[0])):
    for v in range(len(A[i])):
      product[i][j] += A[i][v] * B[v][j]

Suitable: can be parallelised / pipelined. RaW avoided

for i in range(len(A)):
  for v in range(len(A[i])):     # swapped
    for j in range(len(B[0])):   # with this
      product[i][j] += A[i][v] * B[v][j]
```

Matrix Multiply: Generalise but Specialise

- Why not make a general-purpose nested "Loop" system?
  - Other uses (algorithms) beyond Matrix Multiplication
  - Allow any arbitrary-sized loops
  - Allow any permutation of nesting
  - Allow reversing per-dimension
- Specialise by making Matrix Multiply "setup" quick/easy
  - two 32-bit instructions to set up A, B, C sizes
  - one 64-bit SVP64 FMAC instruction (hot-loop).
  - Nothing else needed. Saves on I-Cache
- Hardware turns out to be near-identical to ZOLC https://opencores.org/projects/hwlu https://libre-soc.org/openpower/sv/remap/
- Concept is actually borrowed from Aspex Array-String Processor 1D/2D/3D Memory DMA "reordering" Engine (except applied to the register file)

Matrix Multiply: unit test / example

```
def test_sv_remap2(self):
    lst = ["svshape 5, 4, 3, 0, 0",
             "svremap 0b11111, 1, 2, 3, 0, 0, 0, 0",
             "sv.fmadds 0.v, 8.v, 16.v, 0.v"
            ]
              REMAP fmadds FRT, FRA, FRC, FRB
```

```
svshape 5, 4, 3, 0, 0 => A: 3x5 B: 3x4
                      => C: 3x3
svremap (enable) (F)RS, (F)RT, (F)RA, (F)RB, (F)RC
sv.fmadds: uses fp0 as accumulator
     product[i][j] += A[i][v] * B[v][j]
```

Matrix Multiply: Ehm that's all Folks

- Really is that straightforward: no actual Vector ops
  - Does not dictate or limit micro-architectural detail
  - Issues Scalar FMACs into existing back-end hardware
  - Can use any 4-operand instruction (GF, INT, Bitmanip)
  - No Power-2 limits. Any operand width (8/16/32/64)
- Limited to 127 scalar ops and in-place registers. Future?
  - https://arxiv.org/abs/2002.10143 CISC-like load-and-inc
  - Auto-load/store (tagged) registers, keeps RISC ISA
  - Extend to memory-based arbitrary NxN matrix sizes
  - Still power-efficient: no I-cache usage during FMAC issue
- Future can be investigated as part of EUR 22.6m EU Grant https://libre-soc.org/SEP-210803722-Libre-SOC-8-core/

DCT / FFT / DFT / NTT: what if we could REMAP?

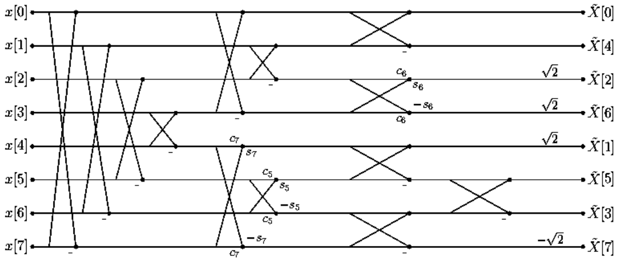
- Can we create a REMAP Schedule for FFT (etc)? YES
  - More complicated than Matrix Schedules but same principle
  - Again: issues Scalar instructions into back-end micro-arch
  - Requires 5-operand (3-in, 2-out) new Scalar Instructions
  - Any operand width (8/16/32/64)
- Limited to in-place registers and Power-of-Two. Future?
  - Again: CISC-like auto-load/store-and-increment
  - https://arxiv.org/abs/2002.10143
  - Again: still power-efficient (no I-Cache usage in loops)
- Again: can be investigated as part of EUR 22.6m EU Grant https://libre-soc.org/SEP-210803722-Libre-SOC-8-core/

DCT / FFT / DFT / NTT: other implementations?

- Texas Instruments TMS320 and C6xx DSPs (VLIW)
  - 14 u-Ops per VLIW (including Zero-Overhead Looping)
  - Performs Odd/Even FP32 looping single-instruction FFT
  - Cannot do anything other than FP32
  - Otherwise absolutely brilliant and elegant (20+ years)
- Qualcom Hexagon DSP
  - Again: VLIW (29 RISC-like u-Ops in 1 cycle)
  - Seriously power-efficient and effective
  - Has ZOLC and Complex-number Multiply
  - Only seems to handle the inner loop of FFT though
- SVP64
  - not limited to inner loop (handles entire triple-loop)
  - like Hexagon, not limted to type of operation inside loop
  - Complex-number ops: a bit too heavy-duty for now (later?)

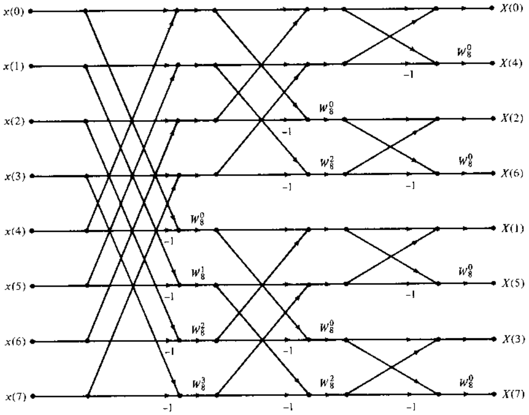
Discrete Cosine Transform (DCT): Basics

- Standard DCT Schedule (messy, impossible for SIMD)
- Output is in bit-reversed order
  0b000 = 0b000 (in: 0 out: 0)
  0b001 = 0b100 (in: 1 out: 4) ...
  0b110 = 0b011 (in: 6 out: 3)
  0b111 = 0b111 (in: 7 out: 7)



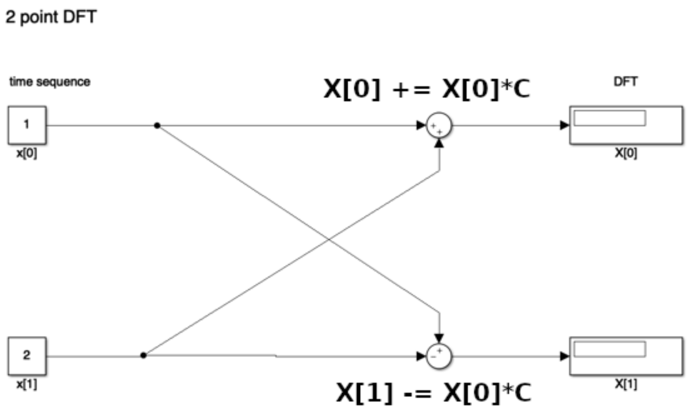
Fast Fourier Transform (FFT/DFT): Butterfly Basics

- Standard Butterfly Schedule (again: messy, but less so)
- Output, again, is in bit-reversed order



FFT: 3-in, 2-out butterfly

- One multiply (by coefficient), one add, one subtract
- inputs: X[0] X[1] C(oeff) outputs: X[0] X[1]



DFT: Project Nayuki Radix-2 Cooley-Tukey DIT (1)

```
coef = (2 if inverse else -2) * cmath.pi / n
exptable = [cmath.rect(1, i*coef) for i in range(n // 2)]
vec = [vec[reverse_bits(i, levels)] for i in range(n)]
size = 2
while size <= n:
    halfsize, tablestep = size // 2, n // size
    for i in range(0, n, size):
        k = 0
        for j in range(i, i + halfsize):
            temp = vec[j + halfsize] * exptable[k]
            vec[j + halfsize] = vec[j] - temp
            vec[j] += temp
            k += tablestep
    size *= 2
```

DFT: Project Nayuki Radix-2 Cooley-Tukey DIT (2)

```
coef = (2 if inverse else -2) * cmath.pi / n
exptable = [cmath.rect(1, i*coef) for i in range(n // 2)]
vec = [vec[reverse_bits(i, levels)] for i in range(n)]
size = 2
while size <= n:
    hs, tablestep = size // 2, n // size
    for i in range(0, n, size):
        k = 0
        for j in range(i, i+hs):
            # Twin-Butterfly 3-in 2-out: one instruction
            C = exptable[k]
            vec[j+hs], vec[j] = 2B(vec[j+hs], vec[j], C)
            k += tablestep
    size *= 2
```

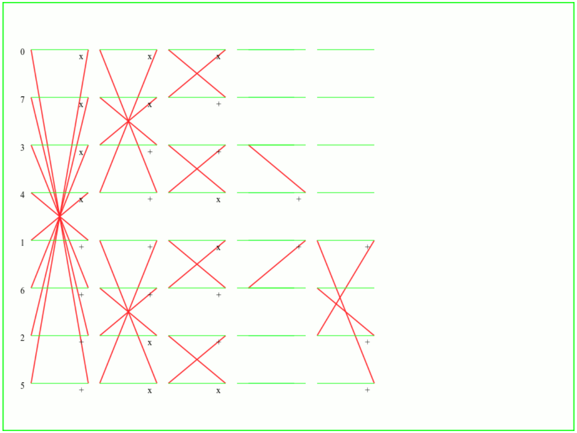
DFT: Project Nayuki Radix-2 Cooley-Tukey DIT (3)

- What if the Triple Loop could be done with REMAP?
- Register Offsets j, j+hs, k created automatically?
- Only one actual inner loop instruction (Twin-butterfly)
- 3-in (X0/X1/C) 2-out (X0/X1) allows for in-place FFT
- Hardware FSM (like ZOLC) creates offset triplet
  - Technically not that hard to implement (for Radix-2)
  - Exact same principle as Triple-loop for Matrices

```
for j,k,hs in REMAP_TRIPLE_LOOP_GENERATOR():
            # Twin-Butterfly 3-in 2-out: one instruction
            C = exptable[k]
            vec[j+hs], vec[j] = 2B(vec[j+hs], vec[j], C)
```

DCT: pre-arrange (pre-load) data

- Arrange input data such that output falls into place
- (another) Twin 3-in 2-out Mul-Add in-place instruction



FFT (Complex numbers) and DCT coefficients?

- Problem (DCT): DCT Cosine Coefficients change ($\cos + 0.5$) depending on the layer. Cannot do as single instruction
- Problem (FFT): Complex number butterfly multiplication involves 4 multiplies. Cannot do in-place as single instruction
- Solution: "Vertical-First" Vectors (Mitch Alsup 66000 ISA)
- Understanding of SVP64 "Vertical-First" 30min video https://youtube.com/watch?v=fn2KJvWyBKg
- Basically involves stepping "vertically" through instructions then ("stepping") to the next offset (REMAP), loop with bc
- Horizontal-first: run through the entire REMAP schedule on a single instruction before repeating looping on the next

Summary

- Goal is to create a mass-volume low-power embedded SoC suitable for use in netbooks, chromebooks, tablets, smartphones, IoT SBCs.
- This means a computational focus on 3D and Audio/Video.
  - Critical not to waste 75% of Power-2 SIMD Lanes on 3x3
- Reducing core work to a one-instruction hot-loop inherently reduces power consumption because the I-Cache is 100% idle.
- REMAP system completely independent from the instructions it REMAPs. Applies to future scalar ops (GF, Bitmanip)
- Future versions involve proper Zero-Overhead Loop-Control and hidden "tags" to automatically perform CISC-like auto-load/store-and-inc (for much larger data sets)
- Please help contribute: it's your Open Power ISA too.

The end

Thank you

Questions?

- Discussion: Libre-SOC-ISA mailing list
  http://lists.libre-soc.org/mailman/listinfo/libre-soc-isa
- Libera IRC #libre-soc
- http://libre-soc.org/
- http://nlnet.nl/PET
  https://www.ngi.eu/ngi-projects/ngi-pointer/