Acceleration of Control Intensive Applications on Coarse-Grained Reconfigurable Arrays for Embedded Systems - Appendix

Benoît W. Denkinger, Miguel Peón-Quirós, Mario Konijnenburg, David Atienza, *Fellow, IEEE*, Francky Catthoor, *Fellow, IEEE*

APPENDIX
SORTING KERNEL MAPPING ASSEMBLY CODE ANALYSIS

This appendix details the execution of the sorting algorithm discussed in Section 4 at the instruction level for the ARM Cortex-M4 processor, the Ibex processor, and the VWR2A. Figure 1 shows the original C code of the sorting algorithm and the assembly instructions for the three implementations.

The ARM Cortex-M4 (Figure 1b) and the Ibex (Figure 1c) assembly codes are very similar. The main difference is the single-cycle compare and branch instruction feature of the Ibex platform (i.e. *–bge a6,a2,.L4* in this case–, while the ARM processor first compares the values, *cmp r6,r5*, and then branches depending on the output, *itt lt*. This explains the smaller inner loop instruction size of the Ibex compared to the ARM Cortex-M4 processor, with 9 and 10 instructions executed when the if-condition is true, respectively.

For the VWR2A, Figure 1d shows the assembly for all the different units (i.e., LCU, LSU, MXCU, and the PEs). The inner loop takes two instructions per unit, ten instructions in total (without counting the *nop* instructions). As all the units execute in parallel, the inner loop takes only two clock cycles, which is $5\times$ better than the processors.

Here, a detailed cycle-level walk-through of the VWR2A mapping is given to help the reader understand the mapping better. The mapping over the PEs is adapted to the architecture constraints and parallelization. The original data array is divided into two sub-arrays that are sorted by PE0-PE3, and PE1-PE2. The following line numbers correspond to Figure 1d. Lines 00 to 02 are initialization instructions where, for example, the data are loaded into VWR 0 (LSU - line 02: *ld.vwr #0, [r7]*). At line 03, PE0 and PE1 load the current data (data[i] in Figure 1a) from VWR 0 into r0 (PE0/1: *movs r0, vwr #0*). At the same time, PE3 stores in r1 the value coming from PE2 (PE3 - *movs r1, pet*, where pe[t/b] is for processing the top / bottom element). At line 04, PE0 and PE1 store a backup of data[i] to r1, while PE2 and PE3 initialize r0 with the array index of the current data (*i-th* element index). The MXCU increments by one the address of the VWRs (MXCU - line 04: *adds r0, r1, #1)*). At line 05, the inner loop starts with PE0 and PE1 comparing the current minimum value —data[i] stored in r0— with the next entry of the array —data[j] from VWR 0—, and PE3 increments the next entry address by one to keep track of the j index locally. Based on the sign flag resulting from the previous comparison, PE0 and PE1 update their local minimum value store in r0 (PE0 - line 06: *movs.sf r0, vwr #0, r0, self*). PE2 and PE3 do the same, but keep track of the minimum value index by using the sign flag (movs.sf) of pet (i.e., PE1) and peb (i.e., PE0), respectively. At the same time, the MXCU updates the address of the VWRs for the next iteration comparison, and the LCU decides whether or not to branch for another iteration of the inner loop or exit it (LCU - line 06: *bgepd r1, #1, .L1*).

The actual element swapping inside the array takes place only in the outer loop for VWR2A, while this is done every time a new minimum value is found for the ARM Cortex-M4 and the RISC-V Ibex (as shown in the C source code in Figure 1a). The VWR2A mapping only swaps the elements and their index in the PEs' local register file (i.e., r0). Once the inner loop is finished, PE0 and PE1 first store their local minimum value back to VWR 0 (PE0/1 - line 08: *movs vwr #0, r0*). Then, their local backup, in r1, of the value previously stored at this location is stored at the old index of the new minimum value. As PE0 and PE1 process their half of the input array, two, often different, old indexes are held by PE3 and PE2, respectively. These indexes are passed to PE0, which stores them, one after the other, to the SRF (only PE0 can write into it). The MXCU updates the VWRs address with these two values (MXCU - line 10-11), one after the other, and PE0 and PE1 store their local backup of the minimum value back to the VWR 0 (PE0/1 - line 11/12: *vwr #0, r1*).

Finally, the MXCU increments the array address by one at line 08 and updates the VWRs address with it at line 12 for the next iteration. This corresponds to incrementing the *i* variable by one to access the next i-th element of the input data array (outer loop in Figure 1a). PE2 also keeps a local track of this address. It updates it on line 12 and shares it with PE3 on line 03. PE0 and PE1 load the current data (data[i] in Figure 1a) VWR 0 into r0 (PE0/1 - line 03: *movs r0, vwr #0*) and another inner loop can start. Once the outer loop reaches its limit, the LSU stores the sorted array back

```
void sort(int16_t *data, int32_t size){
 int i, j, temp;
for(i=0; i<size-1; i++) {
  for(j=i+1; j<size; j++) {
   if(data[j] < data[i]) {
    temp = data[i];
    data[i] = data[j];
    data[j] = temp;
   }
  }
 }
}
```

(a)

```
void sort(int16_t *data, int32_t size)
    push    {...}
    movs    r2, #0
    mov     r4, r0
    subs    r7, r1, #1
.L0:cmp     r7, r2
    bgt.n   .L1
    pop     {...}
.L1:adds    r2, #1
    mov     r3, r2
.L2:cmp     r3, r1
    bne.n   .L3
    adds    r4, #2
    b.n     .L0
.L3:ldrsh.w r6, [r0, r3, lsl #1]
    ldrsh.w r5, [r4]
    cmp     r6, r5
    itt     lt
    strhlt  r6, [r4, #0]
    strhlt.w r5, [r0, r3, lsl #1]
    adds    r3, #1
    b.n     .L2
```

(b)

```
void sort(int16_t *data, int32_t size)
    li      a5,0
    addi    a7,a1,-1
.L0:blt     a5,a7,.L1
    ret
.L1:addi    a5,a5,1
    mv      a4,a0
    mv      a3,a5
.L2:bne     a3,a1,.L3
    addi    a0,a0,2
    j       .L0
.L3:lh      a6,2(a4)
    lh      a2,0(a0)
    bge     a6,a2,.L4
    sh      a6,0(a0)
    sh      a2,2(a4)
.L4:addi    a3,a3,1
    addi    a4,a4,2
    j       .L2
```

(c)

void sort(int16_t *data, int32_t size)

LCU:	LSU:	MXCU:	PE0:	PE1:	PE2:	PE3:
00: movs r0, #16	00: ld.srf #3, [r7]	00: movs r0, #0	00: nop	00: nop	00: nop	00: nop
01: subs r0, #1	01: movs r7, SRF[2]	01: movs r1, #0	01: nop	01: nop	01: nop	01: nop
02: nop	02: ld.vwr #0, [r7]	02: movs r2, #0	02: nop	02: nop	02: movs r1, #0	02: nop
03: L0:movs r1, r0	03:.L0:nop	03:.L0:nop	03:.L0:movs r0, vwr #0	03:.L0:movs r0, vwr #0	03:.L0:nop	03:.L0:movs r1, pet
04: nop	04: nop	04: adds r0, r1, #1	04: movs r1, vwr #0	04: movs r1, vwr #0	04: movs r0, peb	04: movs r0, r1
05:.L1:nop	05:.L1:nop	05:.L1:nop	05:.L1:cmp vwr #0, r0	05:.L1:cmp vwr #0, r0	05:.L1:nop	05:.L1:adds r1, r1, #1
06: bgepd r1, #1, .L1	06: nop	06: adds r0, r0, #1	06: movs.sf r0, vwr #0, r0	06: movs.sf r0, vwr #0, r0	06: movs.sf r0, peb, r0, pet	06: movs.sf r0, r1, r0, peb
07: nop	07: nop	07: movs r0, r1	07: nop	07: nop	07: nop	07: nop
08: nop	08: nop	08: adds r1, r1, #1	08: movs vwr #0, r0	08: movs vwr #0, r0	08: adds r0, #0	08: adds r0, #0
09: nop	09: nop	09: nop	09: movs SRF[5], pet	09: adds rout, peb	09: nop	09: nop
10: nop	10: nop	10: movs r0, SRF[5]	10: movs SRF[5], peb	10: nop	10: nop	10: nop
11: nop	11: nop	11: movs r0, SRF[5]	11: movs vwr #0, r1	11: nop	11: nop	11: nop
12: bgepd r0, #1, .L0	12: nop	12: movs r0, r1	12: nop	12: movs vwr #0, r1	12: adds r1, r1, #1	12: nop
13: ret	13: str.vwr #0, [r7]	13: nop	13: nop	13: nop	13: nop	13: nop

(d)

Fig. 1: Sorting algorithm assembly code. Red dashed (**- - -**) boxes highlight outer loop instructions, and green dashed-doted (**-·-·-**) boxes inner loop instructions. (a) Original C code. (b) ARM Cortex-M4 assembly code. (c) RISC-V Ibex assembly code. (d) VWR2A assembly code for one column.

into the SPM at line 13.

Dividing the original array into two sub-arrays allows the parallelization of the sorting algorithm on the PEs. However, it requires an extra step to recover the complete sorted array. In this specific case, the sorting algorithm is used to compute a median; therefore, only half of the complete sorted array is necessary to obtain the median. This limits the recovery step on the VWR2A. This step is not shown in Figure 1d for conciseness.

The outer loop size for the VWR2A is larger compared to the ARM Cortex-M4 and the Ibex processors because the PEs have to pass the index of the smallest values to the MXCU through the single-port SRF. However, the impact of the outer loop on the overall performance is limited. Moreover, the number of outer loop iterations is divided by two, compared to the processors, thanks to the parallelization of the algorithm.