Pitfalls of UPMEM kernel development

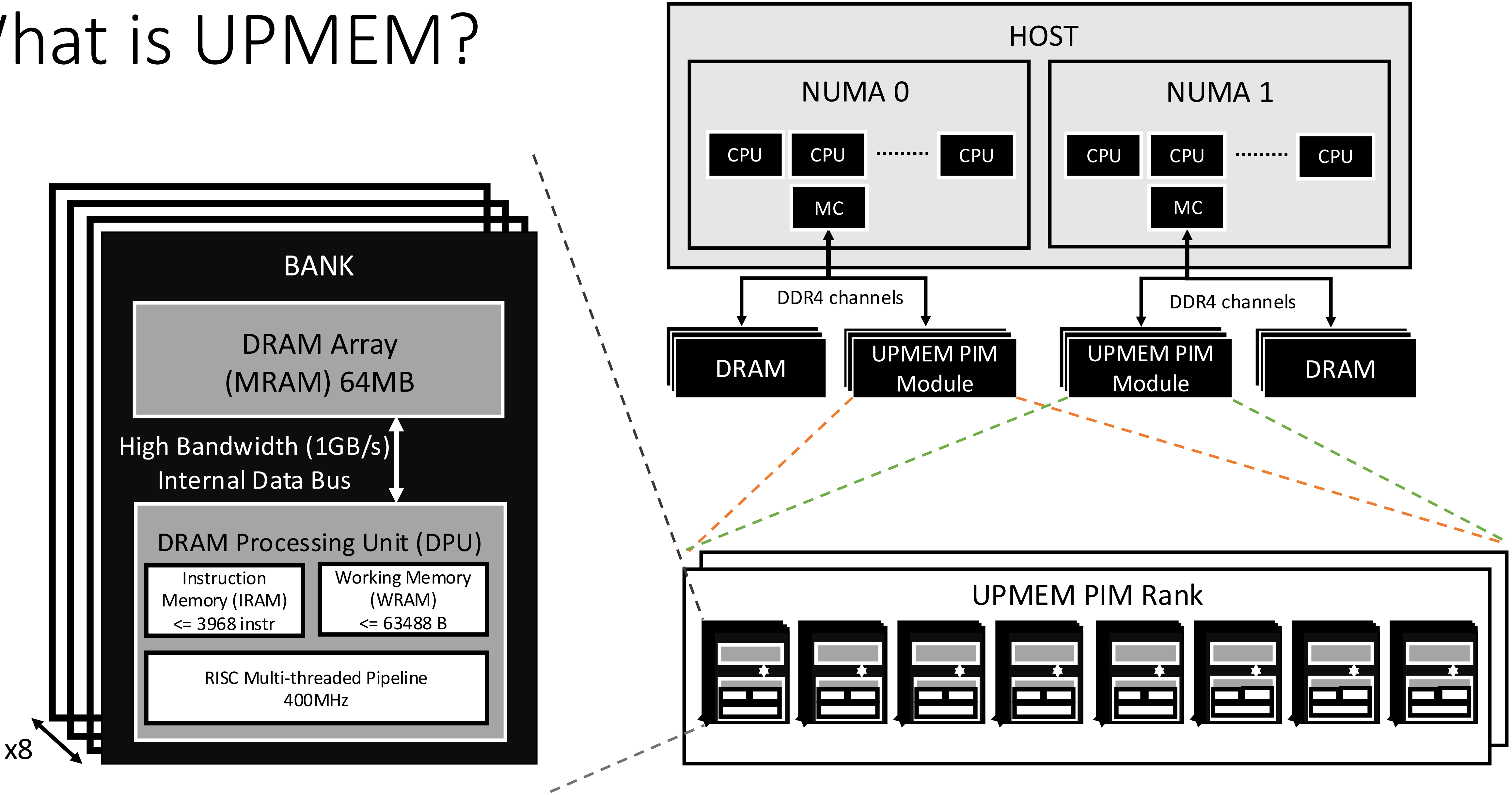
Heterogeneous Memory Software Lab, Poland

Krystian Chmielewski, Tadeusz Kobus, Paweł Piotrowicz, Jarosław Ławnicki, Uladzislau Lukyanau, Maciej Maciejewski, Zhang Hongjun

HUAWEI

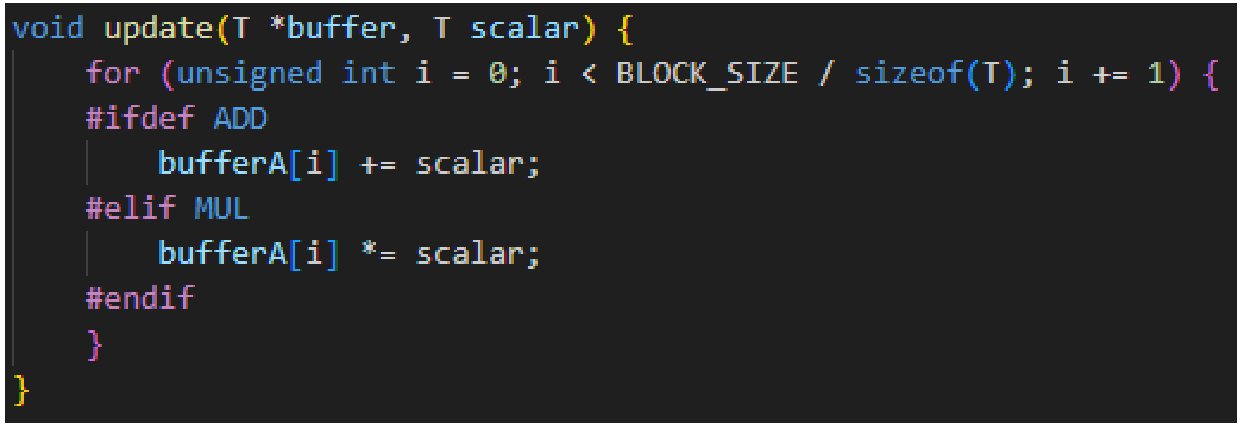
krystian.chmielewski@huawei.com

What is UPMEM?

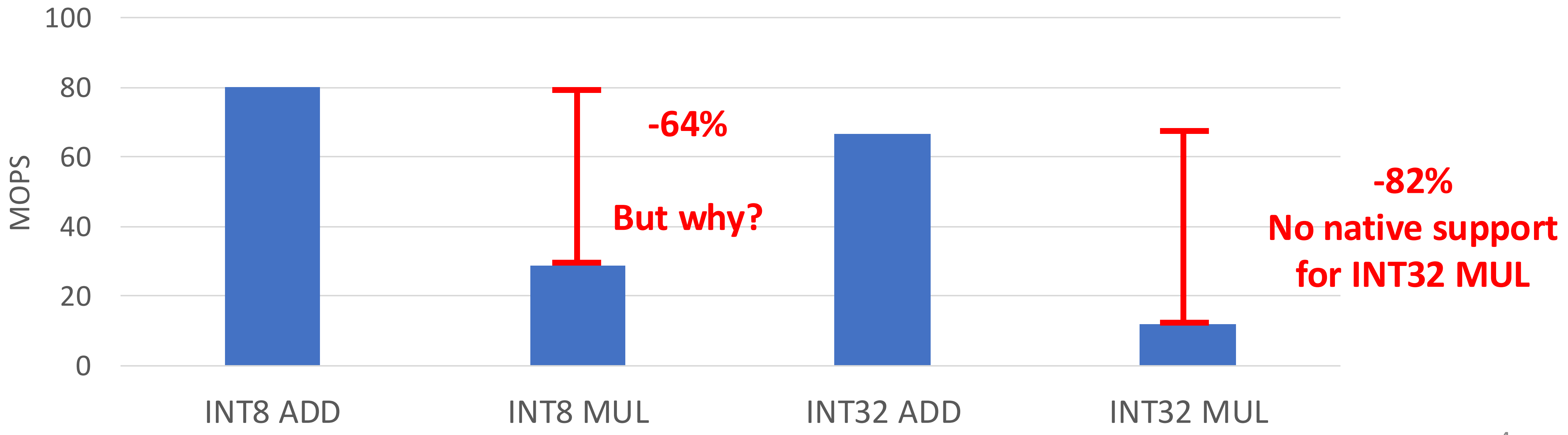

HOST
NUMA 0
CPU
CPU
CPU
MC
NUMA 1
CPU
CPU
CPU
MC
DDR4 channels
DDR4 channels
DRAM
UPMEM PIM Module
UPMEM PIM Module
DRAM
BANK
DRAM Array (MRAM) 64MB
High Bandwidth (1GB/s) Internal Data Bus
DRAM Processing Unit (DPU)
Instruction Memory (IRAM) <= 3968 instr
Working Memory (WRAM) <= 63488 B
RISC Multi-threaded Pipeline 400MHz
x8
UPMEM PIM Rank

ADD and MUL performance analysis

Basic operations on UPMEM



```
void update(T *buffer, T scalar) {
    for (unsigned int i = 0; i < BLOCK_SIZE / sizeof(T); i += 1) {
    #ifdef ADD
        bufferA[i] += scalar;
    #elif MUL
        bufferA[i] *= scalar;
    #endif
    }
}
```



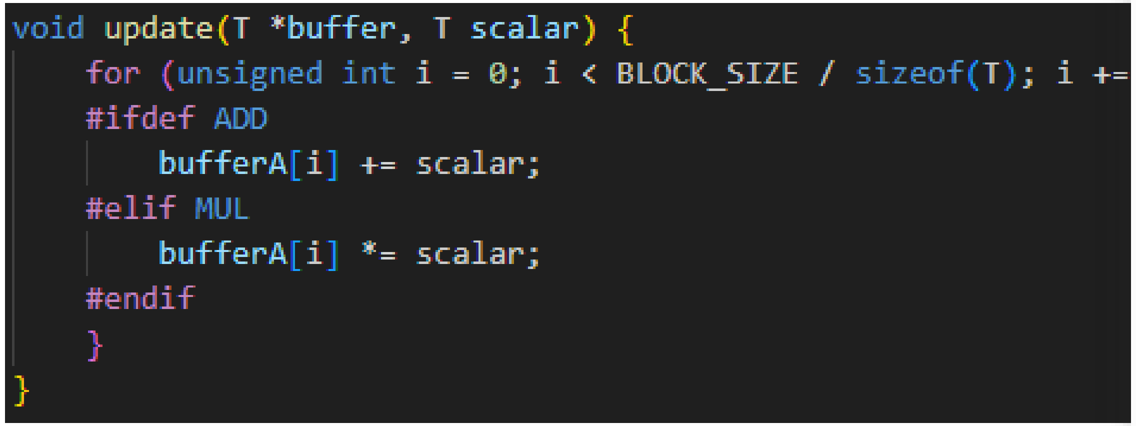
-64%

But why?

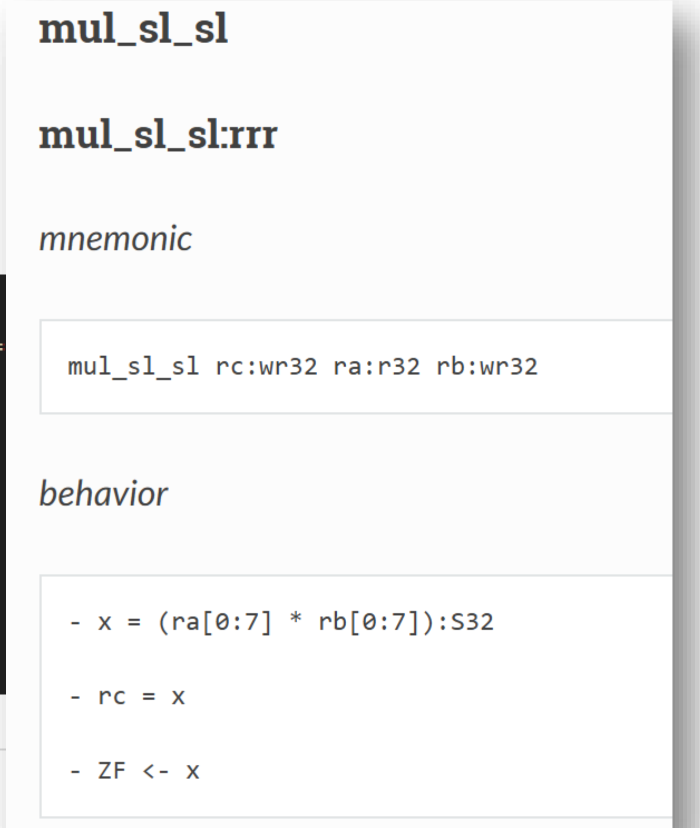
-82%

No native support for INT32 MUL

Basic operations on UPMEM



```
void update(T *buffer, T scalar) {
    for (unsigned int i = 0; i < BLOCK_SIZE / sizeof(T); i +=
    #ifdef ADD
        bufferA[i] += scalar;
    #elif MUL
        bufferA[i] *= scalar;
    #endif
    }
}
```



mul_sl_sl

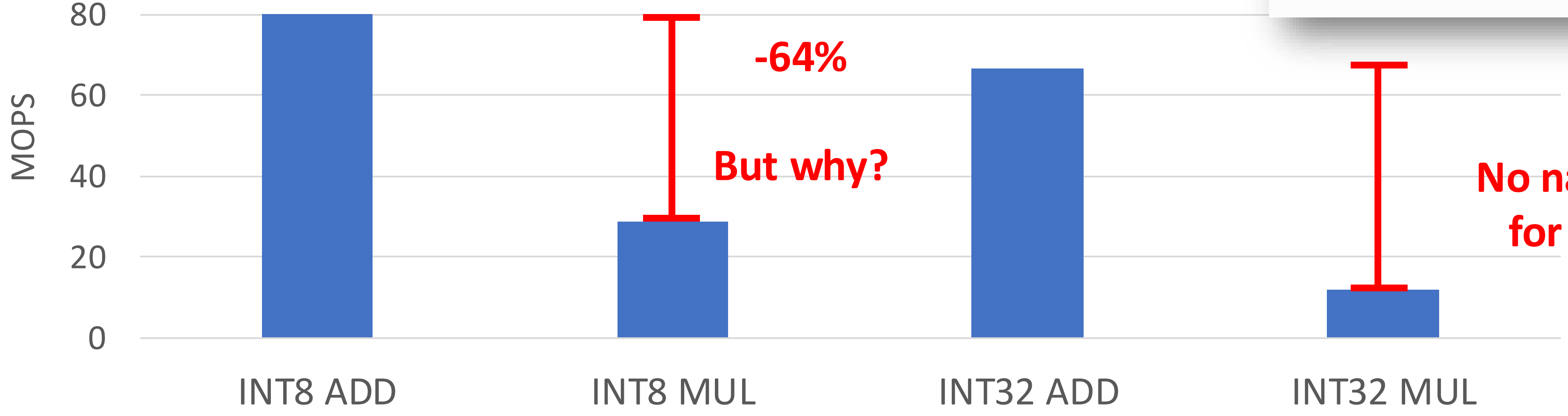
mul_sl_sl:rrr

mnemonic

```
mul_sl_sl rc:wr32 ra:r32 rb:wr32
```

behavior

```
- x = (ra[0:7] * rb[0:7]):S32

- rc = x

- ZF <- x
```



-64%

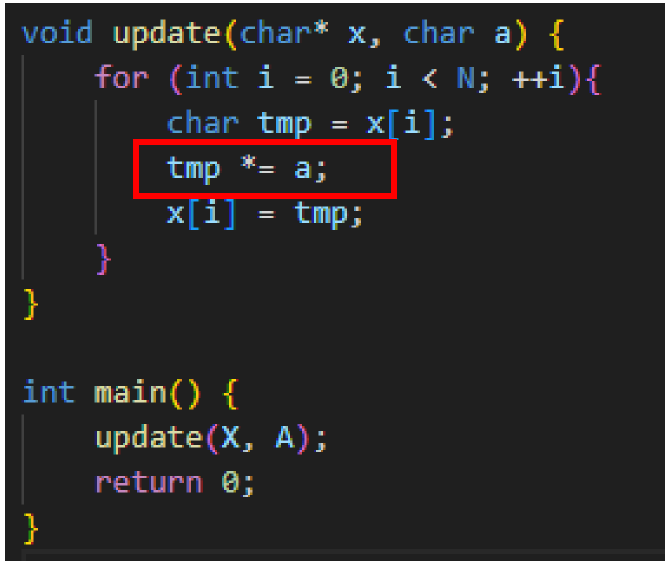
But why?

-82%

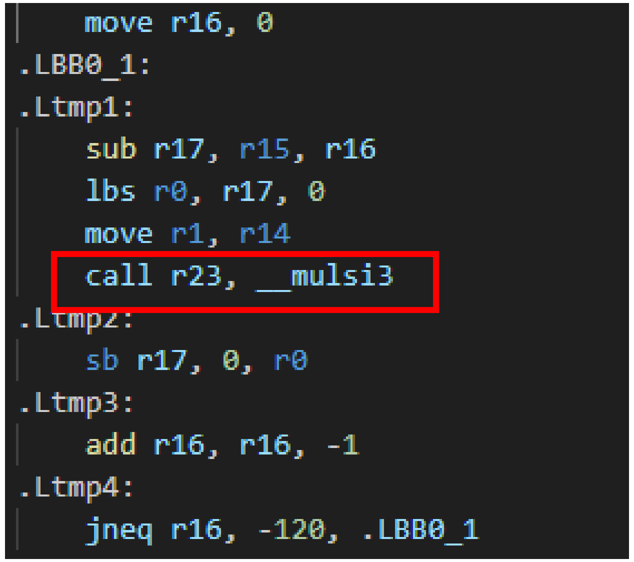
No native support for INT32 MUL

Issues with INT8 MUL

- Compiler sometimes uses
  INT32 MUL *__mulsi3* function
  (SHIFT&ADD, ~9 cycles)
  instead of
  INT8 MUL *mul_sl_sl* (1 cycle) instruction
- Often loops are not being unrolled at all
- Unrolling through *#pragma unroll* is often not possible – not enough IRAM

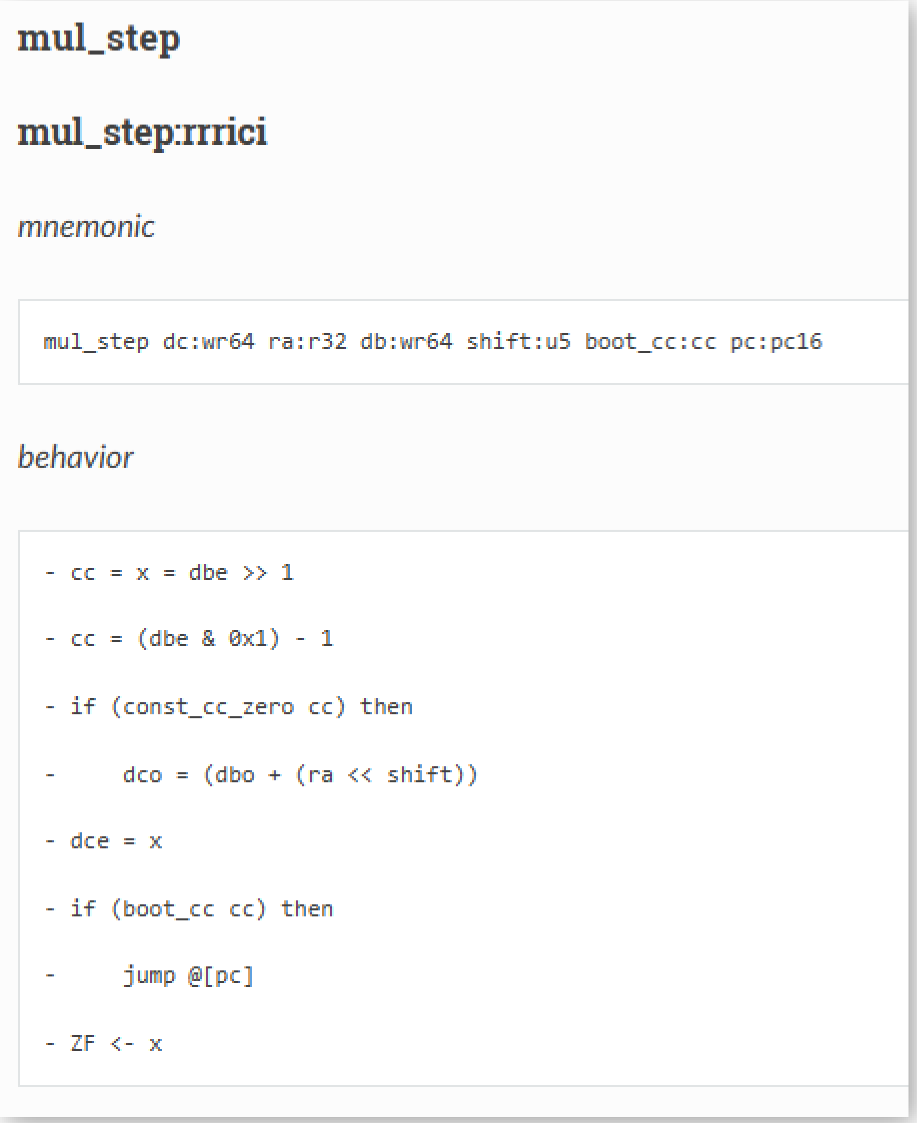


```
void update(char* x, char a) {
    for (int i = 0; i < N; ++i){
        char tmp = x[i];
        tmp *= a;
        x[i] = tmp;
    }
}

int main() {
    update(X, A);
    return 0;
}
```



```
    move r16, 0
.LBB0_1:
.Ltmp1:
    sub r17, r15, r16
    lbs r0, r17, 0
    move r1, r14
    call r23, __mulsi3
.Ltmp2:
    sb r17, 0, r0
.Ltmp3:
    add r16, r16, -1
.Ltmp4:
    jneq r16, -120, .LBB0_1
```

__mulsi3 – SHIFT & ADD



mul_step

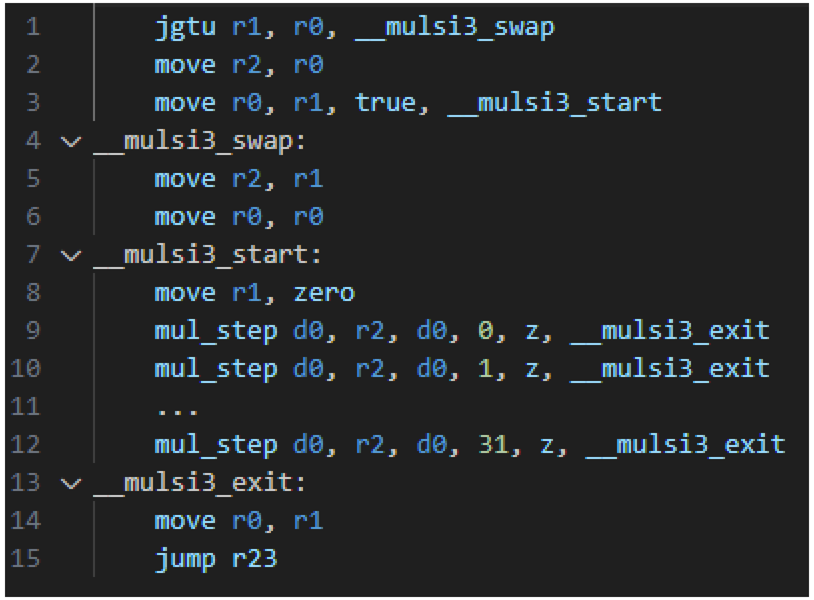
mul_step:rrrici

mnemonic

```
mul_step dc:wr64 ra:r32 db:wr64 shift:u5 boot_cc:cc pc:pc16
```

behavior

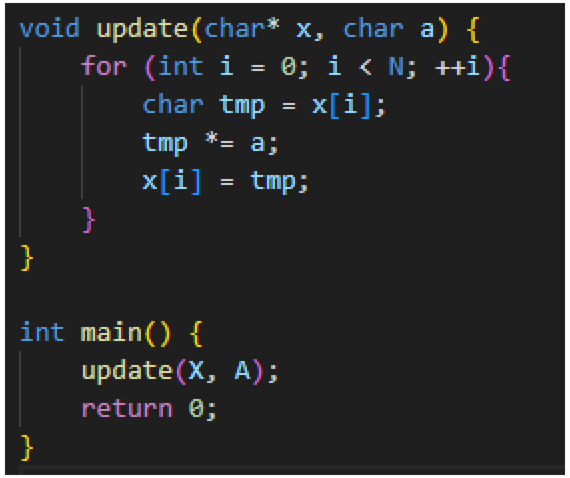
```
- cc = x = dbe >> 1

- cc = (dbe & 0x1) - 1

- if (const_cc_zero cc) then

-     dco = (dbo + (ra << shift))

- dce = x

- if (boot_cc cc) then

-     jump @[pc]

- ZF <- x
```



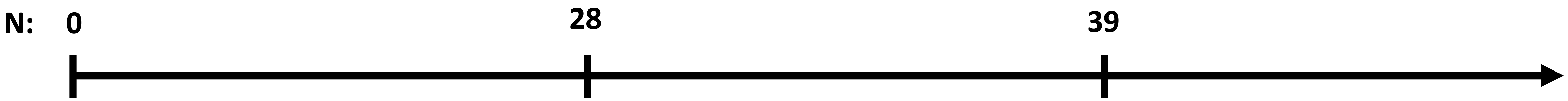
```
    jgtu r1, r0, __mulsi3_swap
    move r2, r0
    move r0, r1, true, __mulsi3_start
__mulsi3_swap:
    move r2, r1
    move r0, r0
__mulsi3_start:
    move r1, zero
    mul_step d0, r2, d0, 0, z, __mulsi3_exit
    mul_step d0, r2, d0, 1, z, __mulsi3_exit
    ...
    mul_step d0, r2, d0, 31, z, __mulsi3_exit
__mulsi3_exit:
    move r0, r1
    jump r23
```

Changing instructions on function inline

- Compilation with –O2 or –O3 will result in inlining based on N (if it’s compile time constant)
- Loop unrolling also depends on N
- Inside the main function, INT32 MUL (SHIFT&ADD) is used



```
void update(char* x, char a) {
    for (int i = 0; i < N; ++i){
        char tmp = x[i];
        tmp *= a;
        x[i] = tmp;
    }
}

int main() {
    update(X, A);
    return 0;
}
```



- Function is inlined
- Loop is fully unrolled
- INT8 MUL is used (1 cycle)

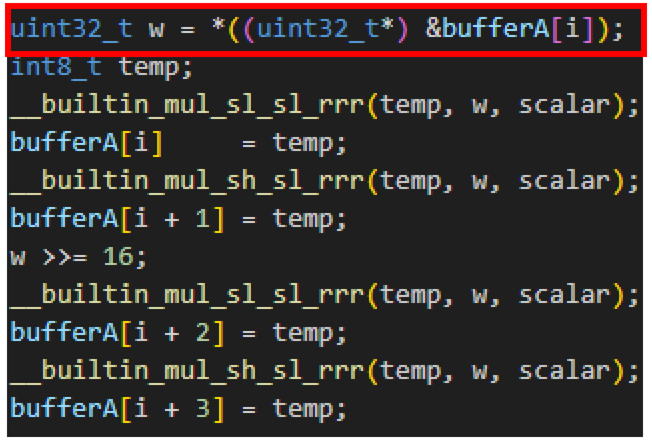
- Function is not inlined
- Loop is fully unrolled,
- INT8 MUL is used (1 cycle)

- Function is inlined
- Loop is not unrolled
- INT32 MUL (SHIFT&ADD) is used (~9 cycles)

INT8/INT32 MUL Optimization

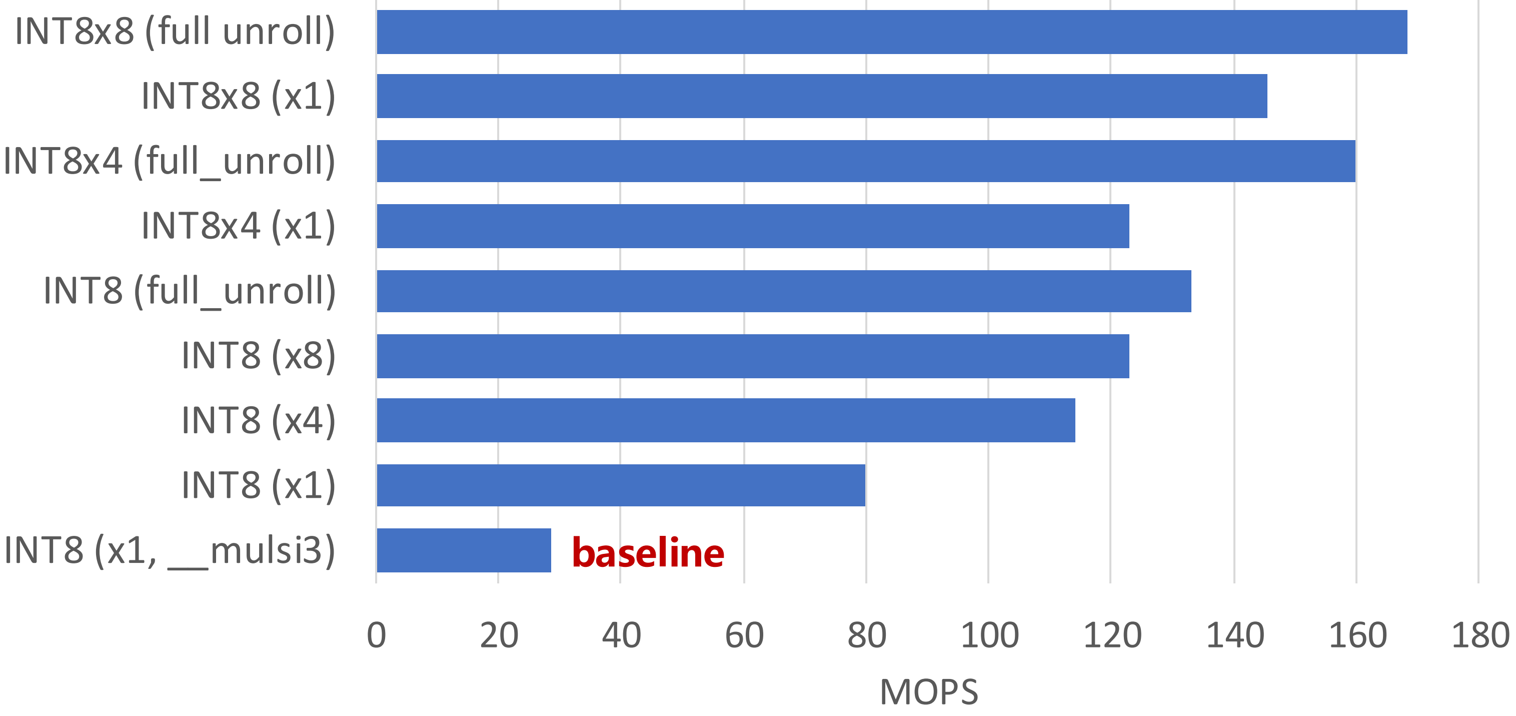
Optimized INT8 MUL

- Ensuring the use of the correct instruction for INT8
- Unrolling:
  - Full (compiler decides) *#pragma unroll*
  - Manual, e.g., x4 *#pragma unroll 4*
- Loading data by 4 or 8 bytes (INT8x4, INT8x8)



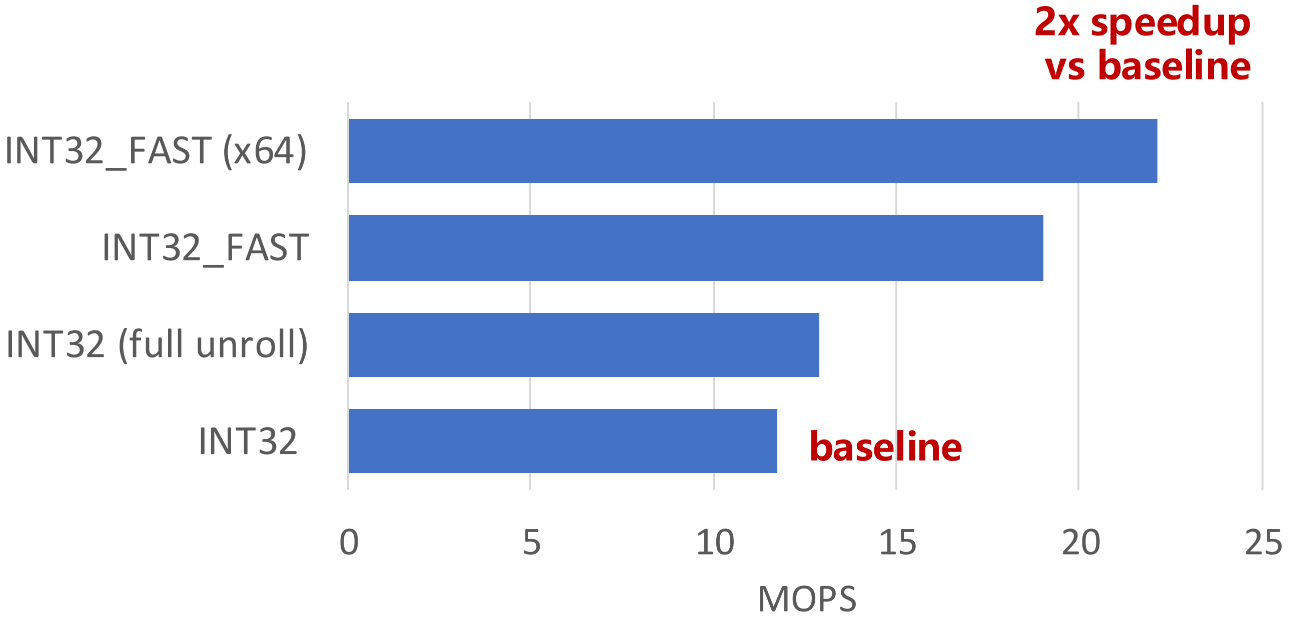
```
uint32_t w = *((uint32_t*) &bufferA[i]);
int8_t temp;
__builtin_mul_sl_sl_rrr(temp, w, scalar);
bufferA[i]     = temp;
__builtin_mul_sh_sl_rrr(temp, w, scalar);
bufferA[i + 1] = temp;
w >>= 16;
__builtin_mul_sl_sl_rrr(temp, w, scalar);
bufferA[i + 2] = temp;
__builtin_mul_sh_sl_rrr(temp, w, scalar);
bufferA[i + 3] = temp;
```

> 5x speedup vs baseline



Optimized INT32 MUL

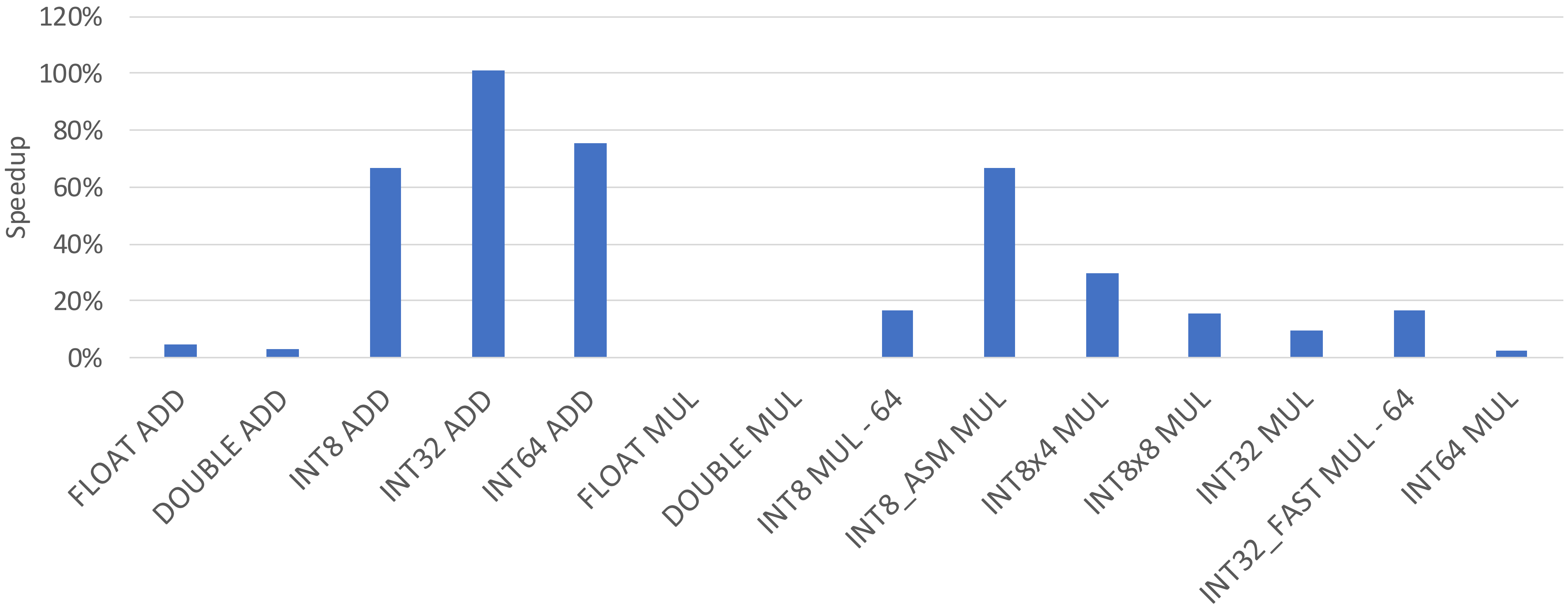
- Replacing *_mulsi3* (SHIFT&ADD, ~30 cycles) with a custom implementation that uses INT8 MUL
- Unrolling also helps



X = (x3, x2, x1, x0)
Y = (y3, y2, y1, y0)

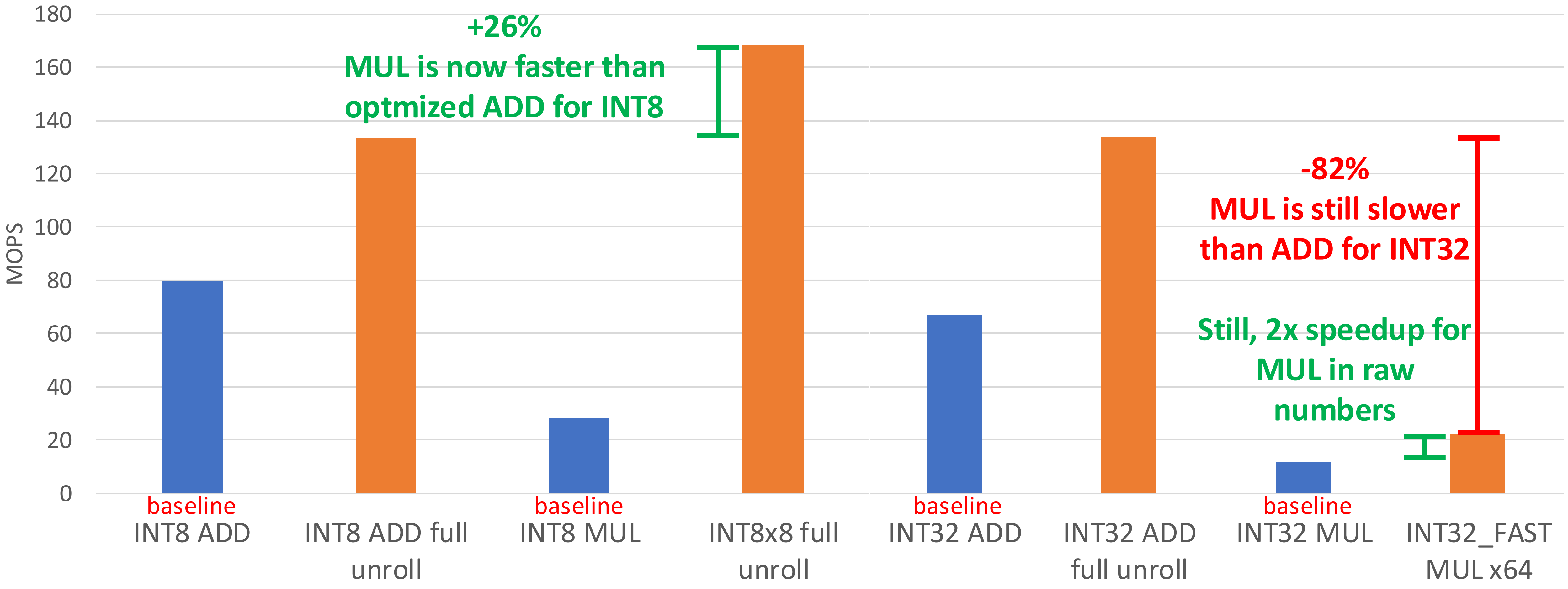
```
X * Y =    2^0   (x0 * y0)
         + 2^8   (x0 * y1 + x1 * y0)
         + 2^16 (x0 * y2 + x1 * y1 + x2 * y0)
         + 2^24 (x0 * y3 + x1 * y2 + x2 * y1 + x3 * y0)
```

Unrolling helps a lot

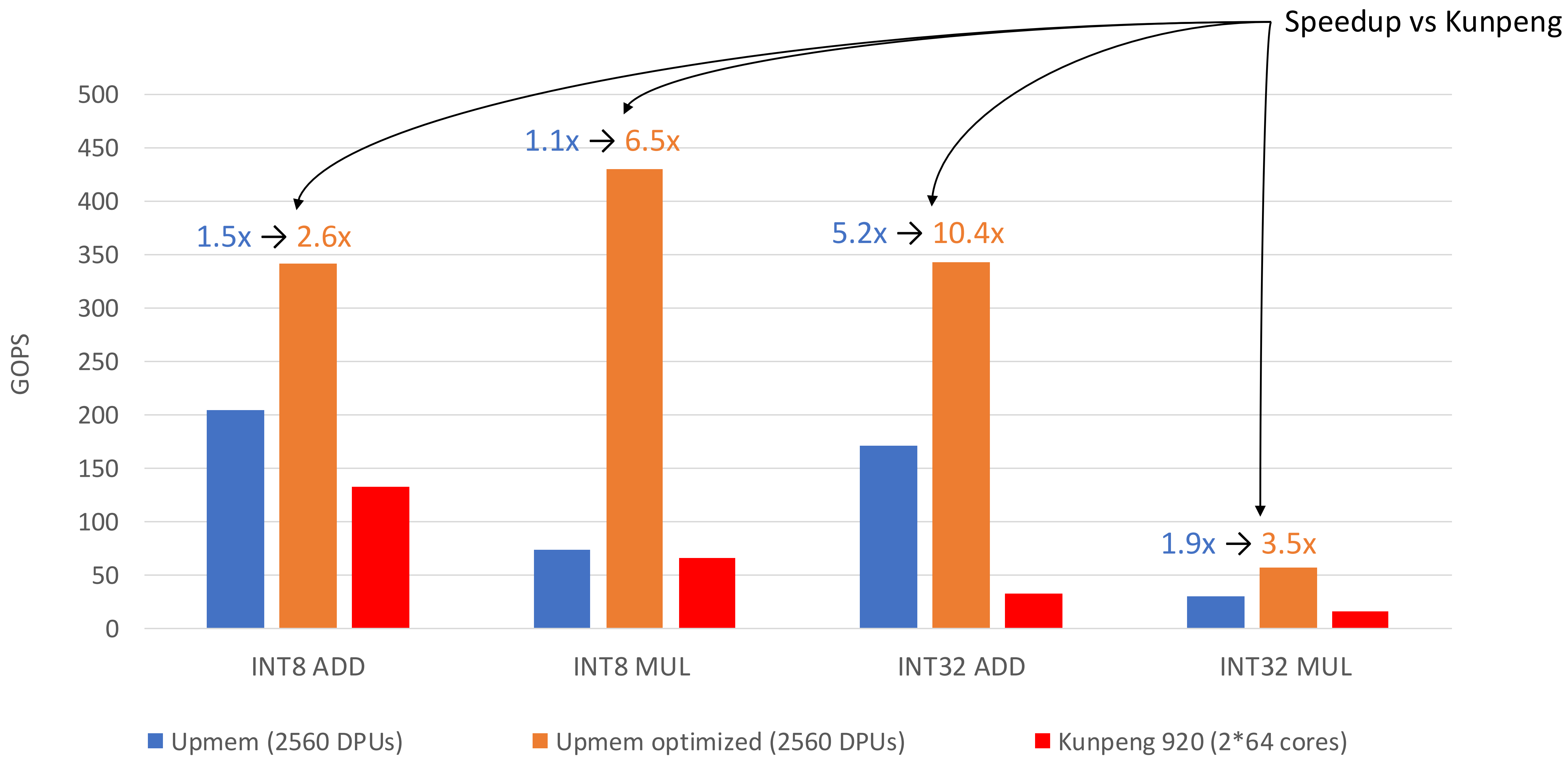


- Loops are pretty costly, when we are not spending a lot of cycles on single iteration
- Use *#pragma unroll* to force compiler to unroll loops

Basic operations on UPMEM revisited



UPMEM vs CPU



Speedup vs Kunpeng
1.5x → 2.6x
1.1x → 6.5x
5.2x → 10.4x
1.9x → 3.5x
GOPS
500
450
400
350
300
250
200
150
100
50
0
INT8 ADD
INT8 MUL
INT32 ADD
INT32 MUL
Upmem (2560 DPUs)
Upmem optimized (2560 DPUs)
Kunpeng 920 (2*64 cores)

Memory transfer optimizations

Memory transfers

Host → PIM (*dpu_prepare_xfer*, *dpu_push_xfer*):

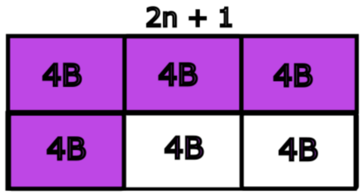
- Transfer size and symbol offset need to be a multiple of 8B → error otherwise
- Extra care is needed to handle the remainder, if any
- Host buffers need to be aligned up to 8B

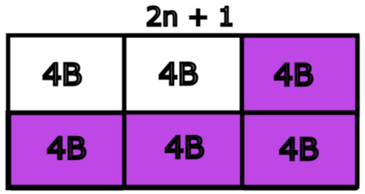
MRAM → WRAM (*mram_read, mram_write):*

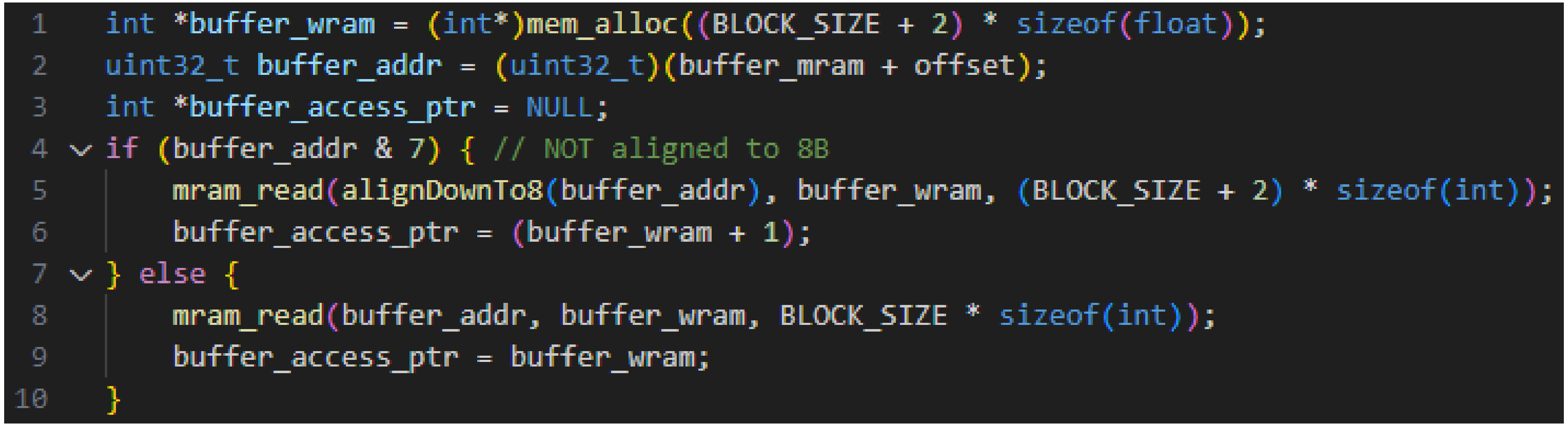
- Transfer size needs to be a multiple of 8B (max 2048B)
- Read/write operations silently align down the address to 8B → no error is raised
- *mram_read_unaligned* and *mram_write_unaligned* functions require ~300 more instructions
  → GEMV where the matrix is stored contiguously and row size is odd

Reading unaligned memory MRAM

- Address we read from/write to must be aligned to 8B
- Number of bytes read/written must be a multiple of 8B







```
int *buffer_wram = (int*)mem_alloc((BLOCK_SIZE + 2) * sizeof(float));
uint32_t buffer_addr = (uint32_t)(buffer_mram + offset);
int *buffer_access_ptr = NULL;
if (buffer_addr & 7) { // NOT aligned to 8B
    mram_read(alignDownTo8(buffer_addr), buffer_wram, (BLOCK_SIZE + 2) * sizeof(int));
    buffer_access_ptr = (buffer_wram + 1);
} else {
    mram_read(buffer_addr, buffer_wram, BLOCK_SIZE * sizeof(int));
    buffer_access_ptr = buffer_wram;
}
```

Optimized data transfers

UPMEM server:

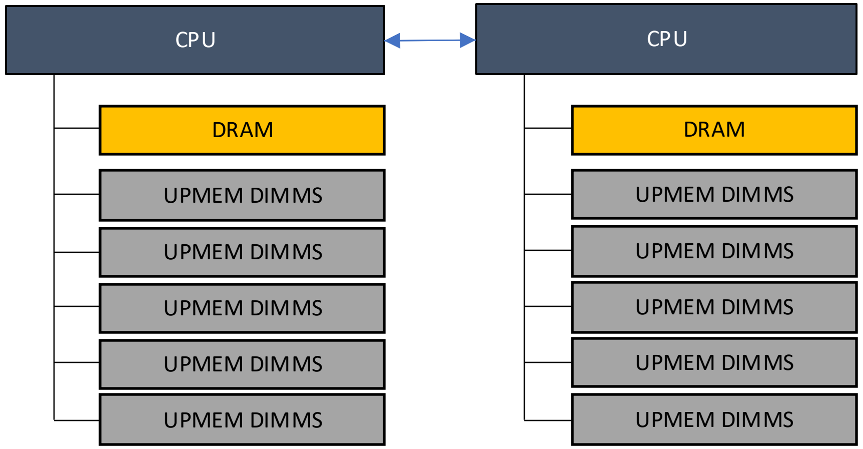
- DRAM can be the bottleneck (only 2 DDR4-3200 channels)
- Transposing data heavily engages the CPU

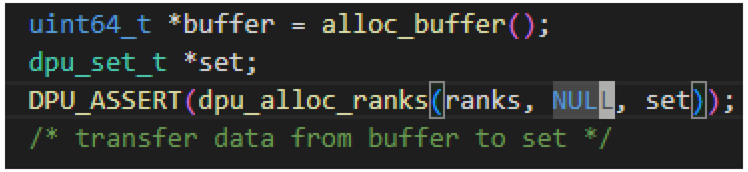
DPU allocation is not NUMA-aware:

- No way to allocate DPUs assigned to specific CPU
- A significant variance (~12%) in memory transfer speeds with default DPU allocation

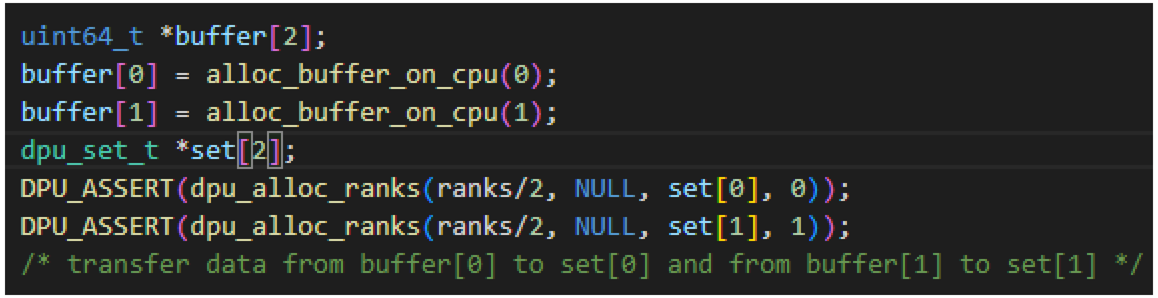
Simple API extension to allow DPU allocation on specific NUMA node:

- Changes only in the user-space library
- Simple filtering of available ranks during DPU allocations



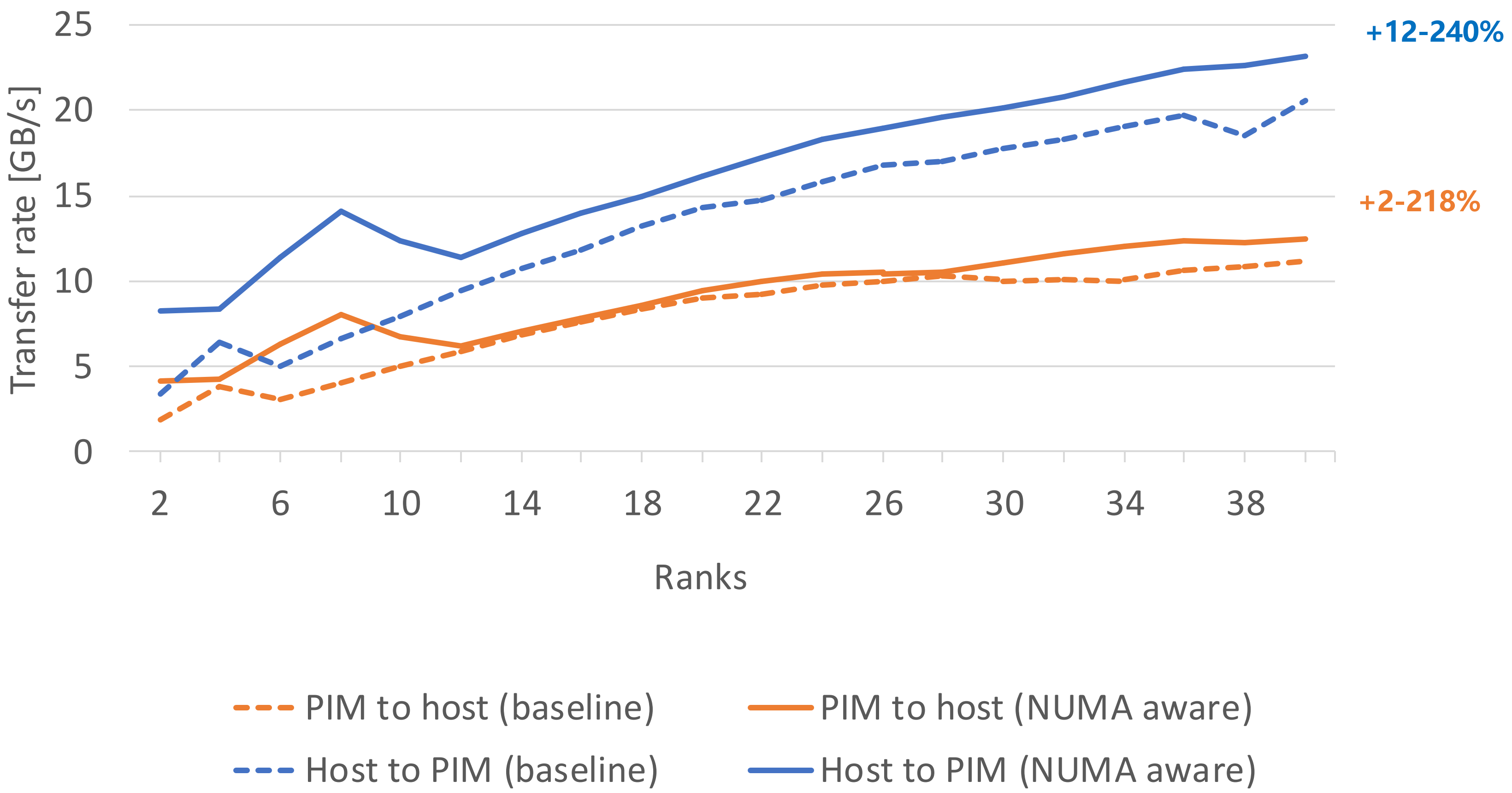


```
uint64_t *buffer = alloc_buffer();
dpu_set_t *set;
DPU_ASSERT(dpu_alloc_ranks(ranks, NULL, set));
/* transfer data from buffer to set */
```



```
uint64_t *buffer[2];
buffer[0] = alloc_buffer_on_cpu(0);
buffer[1] = alloc_buffer_on_cpu(1);
dpu_set_t *set[2];
DPU_ASSERT(dpu_alloc_ranks(ranks/2, NULL, set[0], 0));
DPU_ASSERT(dpu_alloc_ranks(ranks/2, NULL, set[1], 1));
/* transfer data from buffer[0] to set[0] and from buffer[1] to set[1] */
```

Optimized data transfers (parallel copy)



Summary

UPMEM Compiler:

- Generates inefficient INT8 MUL (__*mulsi3* SHIFT&ADD)
- Rarely performs unrolling

DPU Optimizations:

- A little extra care with **INT8 MUL** provides huge gains
- **INT32 MUL** performance is also improved
- The achieved performance of **INT8/INT32 ADD/MUL** makes UPMEM **viable** for testing AI workloads (albeit with quantization, FP remains unpractical on UPMEM)

Optimized data transfers:

- **NUMA-awareness** can go a long way

Contact me at: krystian.chmielewski@huawei.com

Thank you!