# Supplementary materials - Efficient end-to-end long-read sequence mapping using minimap2-fpga integrated with hardware accelerated chaining

**Kisaru Liyanage[1,2,3], Hiruna Samarakoon [1,2,3], Sri Parameswaran[4], Hasindu Gamaarachchi[1,2,3,*]**

[1] School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia.
[2] Genomics Pillar, Garvan Institute of Medical Research, Sydney, NSW, Australia.
[3] Centre for Population Genomics, Garvan Institute of Medical Research and Murdoch Children's Research Institute, Australia.
[4] School of Electrical and Information Engineering, University of Sydney, Sydney, NSW, Australia.
[*] h.gamaarachchi@garvan.org.au

## List of Tables



## List of Figures



## Supplementary Notes

**Table S1. Data points of the plots in Figures 1e and 1f that compare accuracy with ONT simulated reads**

| | *minimap2* | | *minimap2-fpga* | | | |
|---|---|---|---|---|---|---|
| | | | Intel FPGA-based System | | Xilinx FPGA-based System | |
| | Error rate of mapped reads | Fraction of mapped reads | Error rate of mapped reads | Fraction of mapped reads | Error rate of mapped reads | Fraction of mapped reads |
| w/ base-level alignment | 0.00E+00 | 0.960462 | 0.00E+00 | 0.96047 | 0.00E+00 | 0.960464 |
| | 1.03E-06 | 0.970672 | 1.03E-06 | 0.970683 | 1.03E-06 | 0.970673 |
| | 2.02E-06 | 0.991377 | 2.02E-06 | 0.991377 | 2.02E-06 | 0.991377 |
| | 3.02E-06 | 0.992854 | 3.02E-06 | 0.992854 | 3.02E-06 | 0.992854 |
| | 5.03E-06 | 0.993346 | 5.03E-06 | 0.993346 | 5.03E-06 | 0.993346 |
| | 6.04E-06 | 0.993838 | 6.04E-06 | 0.993838 | 6.04E-06 | 0.993838 |
| | 1.11E-05 | 0.994384 | 1.11E-05 | 0.994382 | 1.11E-05 | 0.994382 |
| | 1.31E-05 | 0.994952 | 1.31E-05 | 0.994952 | 1.31E-05 | 0.994952 |
| | 4.70E-04 | 0.997682 | 4.70E-04 | 0.997682 | 4.70E-04 | 0.997682 |
| | 1.71E-03 | 0.99988 | 1.71E-03 | 0.99988 | 1.71E-03 | 0.99988 |
| w/o base-level alignment | 0.00E+00 | 0.970514 | 0.00E+00 | 0.970515 | 0.00E+00 | 0.970516 |
| | 1.01E-06 | 0.987495 | 1.01E-06 | 0.987496 | 1.01E-06 | 0.987495 |
| | 2.02E-06 | 0.988688 | 2.02E-06 | 0.98869 | 2.02E-06 | 0.98869 |
| | 5.05E-06 | 0.989864 | 5.05E-06 | 0.989867 | 5.05E-06 | 0.989867 |
| | 6.05E-06 | 0.990971 | 6.05E-06 | 0.990973 | 6.05E-06 | 0.990971 |
| | 1.51E-05 | 0.992102 | 1.51E-05 | 0.992103 | 1.41E-05 | 0.992101 |
| | 2.52E-05 | 0.993325 | 2.52E-05 | 0.993325 | 2.42E-05 | 0.993323 |
| | 4.72E-05 | 0.99485 | 4.72E-05 | 0.99485 | 4.62E-05 | 0.994848 |
| | 2.03E-03 | 0.99988 | 2.03E-03 | 0.99988 | 2.03E-03 | 0.99988 |

Based on data in Table 1, error rate of mapped reads (accumulative number of wrong mappings / accumulative number of mapped reads) and the fraction of mapped reads (accumulative number of mapped reads / total number of mapped reads) are calculated. The data points in this table are used for plotting the accuracy graphs in Figures 1e and 1f.

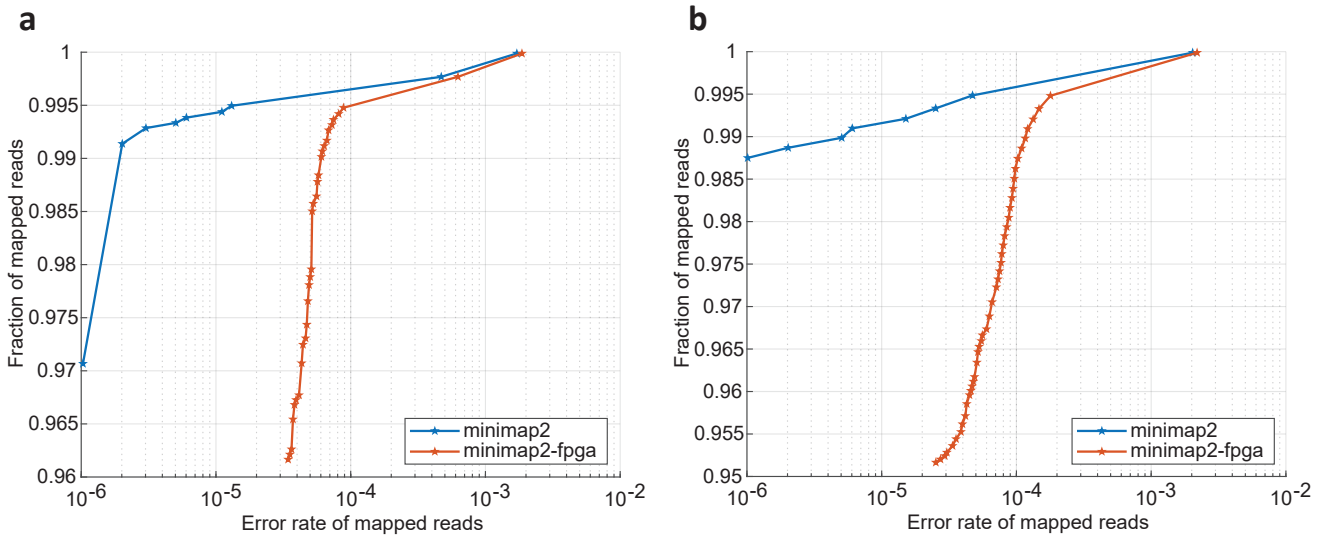


**Figure S1. Accuracy of *minimap2-fpga* when combined with previous hardware accelerator [6] on the Intel FPGA-based system**

(**a**) Accuracy comparison performed with simulated ONT reads for original *minimap2* vs. *minimap2-fpga* with base-level alignment. (**b**) Accuracy comparison performed with simulated ONT reads for original *minimap2* vs. *minimap2-fpga* without base-level alignment.

## 1 FPGA-based Chaining Step Accelerator

The FPGA-based hardware accelerator architecture in our previous work [6] was extended as shown in Figure S2. With the modifications to the hardware accelerator, it was able to achieve better accuracy (compare Supplementary Figure S1 vs. Figure 1e, 1f) in the chaining output while keeping the FPGA resource usage low (so that the accelerator could fit in the limited FPGA area available to use). Furthermore, we successfully maintained the desired speed-up factor throughout the optimization process. Algorithm 1 describes the HLS algorithm used to generate the modified hardware architecture in Figure S2.

*minimap2*'s software chaining algorithm (Algorithm 2) computes $H$ (at most) chaining scores for each anchor, using previous $H$ (at most) anchors (*for* loop in lines 17-23 of Algorithm 2). In our previous hardware accelerator implementation [6], we parallelized this computation by utilizing $H$ score computation units to perform the computations in parallel. Considering the FPGA resource limitations and *minimap2*'s original algorithm definitions, $H$ was set to 64. However, to achieve a mapping accuracy similar to software implementation of *minimap2*, $H$ needed to be increased. Increasing the number of parallel score computation units ($H$) directly to achieve better accuracy, resulted in a design that couldn't fit in the FPGA being used. To resolve this resource utilization issue, the hardware architecture is modified so that the $H$ score computations corresponding to a single anchor are performed in $M$ sub-parts (see Figure S2 and Algorithm 1). Score computations in a sub-part are performed in parallel with only $P$ score computation units (note that previous accelerator [6] needed $H$ score computation units). As most recently used $H$ anchors ($A[i]$) and the maximum score values corresponding to them ($F[i]$), are needed for subsequent score computations, $H = M \times P$ anchors and their maximum score values are stored in first-in, first-out (FIFO) buffers implemented with hardware shift registers.

The modified hardware accelerator with additional control logic was optimized to have an Initiation Interval (II) of 3, making it able to process a sub-part every 3 clock cycles.

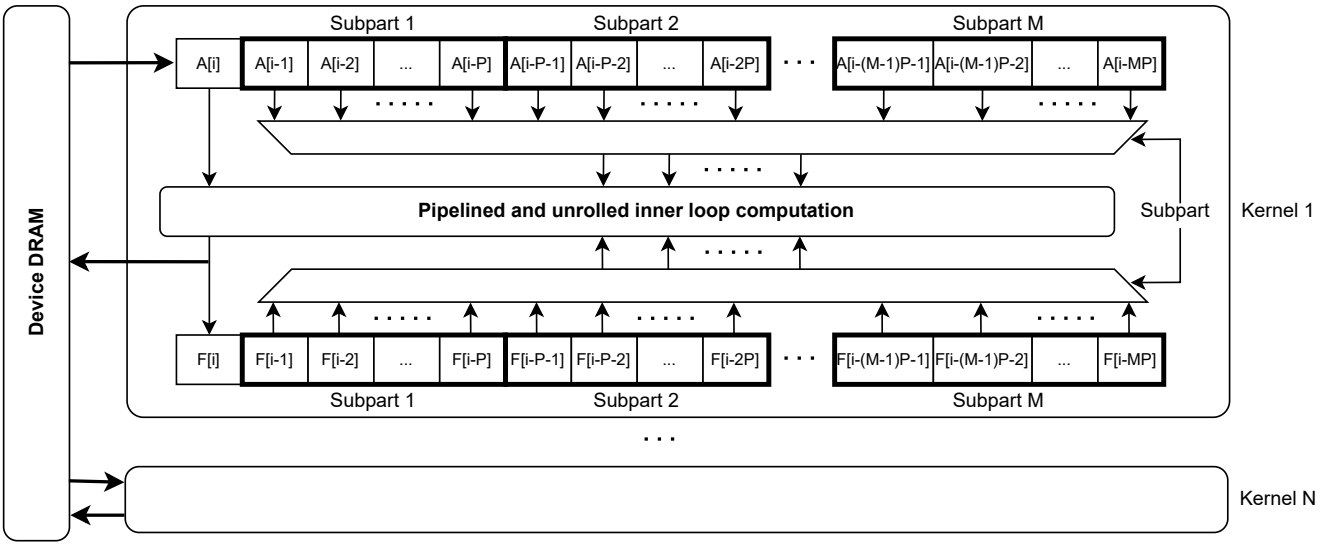


**Figure S2. The architecture of the FPGA-based hardware accelerator designed for *minimap2*'s chaining step acceleration**

The hardware architecture of our previous hardware accelerator [6] was modified to achieve better accuracy while preserving the speed-up when compared to original *minimap2* software.

**Algorithm 1** Hardware Accelerator's Behavior (HLS)

**Inputs:**
*a[]*: Array containing the anchors to be chained (stored in Device DRAM)
*num_subparts[]*: Array containing no. of sub-parts corresponding to each anchor's score computations (stored in Device DRAM)
*total_subparts*: Total no. of sub-parts for all the anchors
*q_span*: Seed length of anchors
*avg_qspan_scaled*: Average seed length of anchors scaled by a factor of 0.01
*max_dist_x*: Maximum reference sequence gap for anchor chaining
*max_dist_y*: Maximum query sequence gap for anchor chaining
*bw*: Bandwidth for anchor chaining
**Outputs:**
*f[]*: Array containing chaining score values (stored in Device DRAM)
*p[]*: Array containing best predecessors (stored in Device DRAM)

▷ Initialize FIFO buffers for storing most recently used anchors (*A[]*) and most recently computed chaining scores (*F[]*) locally
*A[M * P + 1] = {0}*
*F[M * P + 1] = {0}*
*i = 0* ▷ Initialize the counter used to denote the current anchor being processed
*subparts_processed = 0* ▷ Initialize a counter to keep track of already processed sub-parts for current anchor
*subparts_to_process = 0* ▷ Initialize a variable to store the total number of sub-parts for current anchor
**for** *g = 0; g < total_subparts; g++* **do** ▷ Loop over all sub-parts of the chaining task
    ▷ If all the sub-parts for previous anchor are processed, load next anchor (into FIFO *A[]*) and its sub-part count
    **if** *subparts_processed == 0* **then**
        *A[0] =* __prefetching_load*(&a[i])*
        *subparts_to_process =* __prefetching_load*(&num_subparts[i])*
    **end if**
    ▷ Initialize best score (*max_f*) and predecessor (*max_j*) for the current sub-part being processed
    *max_f = q_span*
    *max_j = -1*
    ▷ Compute *P* scores between anchor *i* (which is stored in *A[0]*) and current sub-part of anchors from *A[]* with *P* parallel score computing units, and update *max_f*, *max_j*
    #pragma unroll
    **for** *j = P; j > 0; j--* **do**
        *buffer_offset = subparts_processed * P* ▷ Compute buffer offset based on sub-parts processed so far for the anchor
        *score = compute_score(A[0], A[buffer_offset + j], F[buffer_offset + j], max_dist_x, max_dist_y, bw, q_span, avg_qspan_scaled)* ▷ Compute score between *A[0]* and *A[buffer_offset + j]*
        ▷ Update *max_f, max_j* if computed score is greater than current *max_f* and is not equal to *q_span*
        **if** *score >= max_f and score != q_span* **then**
            *max_f = score*
            *max_j = i - j - buffer_offset*
        **end if**
    **end for**
    ▷ Update best score and predecessor for the anchor *i* if *max_f* found from the current sub-part is greater than the best score found so far for the anchor (which is stored in *F[0]*)
    **if** *max_f > F[0]* **then**
        *F[0] = max_f*
        *f[i] = max_f*
        *p[i] = max_j*
    **end if**
    *subparts_processed++* ▷ Increment the processed sub-parts counter
    ▷ Shift locally stored anchors and chaining scores in the FIFO buffers if all the sub-parts for the current anchor are processed
    #pragma unroll
    **for** *reg = M * P; reg > 0; reg--* **do**
        **if** *subparts_processed == subparts_to_process* **then**
            *A[reg] = A[reg - 1]*
            *F[reg] = F[reg - 1]*
        **end if**
    **end for**
    ▷ If all the sub-parts for the current anchor are processed, increment anchor counter *i*, initialize score for the new anchor (i.e. *F[0]*) and reset *subparts_processed*
    **if** *subparts_processed == subparts_to_process* **then**
        *i++*
        *F[0] = 0*
        *subparts_processed = 0*
    **end if**
**end for**

## 2 Hardware-Software Split

A novel hardware-software split mechanism was introduced to process the chaining tasks either on hardware accelerator or the CPU (as software) based on their computation times predicted with theoretical models. Equation 1 and Equation 2 give the models derived to predict the time taken for a given chaining task to run on hardware accelerator and on CPU (as software) respectively.

The time taken to process a given chaining task on hardware ($T_{hardware}$, given in Equation 1) is the sum of the times taken for data transfer ($T_{data\_transfer}$ = input transfer from host to the hardware accelerator + output results transfer from hardware accelerator to the host) and the execution time on the hardware accelerator ($T_{execution}$). The data transfer time ($T_{data\_transfer}$) is proportional to the number of anchors being transferred ($n$). As the output for a single subpart is generated by the accelerator every $II$ clock cycles, the execution time in terms of clock cycles is $II \times total\_subparts$. Multiplying the clock cycles by the clock period ($T_{clock}$) gives the execution time in seconds ($T_{execution}$). The constant $C_1$ accounts for the initial pipeline delay and the overhead time taken for OpenCL API calls by the host.

To derive the relationship between the time taken to process a given chaining task on the CPU as software ($T_{software}$, as given in Equation 2), it is necessary to refer to the original software chaining algorithm. This algorithm is presented in Algorithm 2 (also presented in our previous work[6]). The number of computational steps can be estimated by the total number of iterations (i.e. total trip count) taken by the *for* loop in Lines 17-23 of Algorithm 2 and is represented by $\sum_{i=1}^{n} trip\_count_i$ in Equation 2. Computing the total trip count corresponding to a single anchor can be done prior to the real execution of the *for* loop by using the *while* loop in Lines 13-15 of Algorithm 2 (this loop computes the starting index ($st$) used in the *for* loop in Lines 17-23 and $min(i-H, st)$ gives the trip count corresponding to the anchor). Summing up the trip count values corresponding to all the anchors gives $\sum_{i=1}^{n} trip\_count_i$ in Equation 2. $C_2$ in Equation 2 accounts for the overhead setup time for the *for* loop in Lines 17-23 of Algorithm 2.

Based on a one-time experiment done with a representative dataset, the values for the coefficients in Equation 1 and 2 (i.e. $K_1$, $K_2$, $C_1$, $C_2$) are found. After, $T_{hardware}$ and $T_{software}$ are calculated for each chaining task at the run time. If $T_{hardware} < T_{software}$, the chaining task is decided to be processed on hardware (see Supplementary Note 3 for hardware scheduling details), otherwise, the chaining task is processed on the host CPU itself as software.

**Algorithm 2** *minimap2*'s Chaining Algorithm

**Inputs:**
*a[]*: Array containing the anchors to be chained
*n*: Total number of anchors
*max_dist_x*: Maximum reference sequence gap for anchor chaining
*max_dist_y*: Maximum query sequence gap for anchor chaining
*bw*: Bandwidth for anchor chaining
**Outputs:**
*f[]*: Array containing chaining score values
*p[]*: Array containing best predecessors

```
▷ Calculate average seed length (avg_qspan)
sum_qspan = 0
for i = 0; i < n; i++ do
    sum_qspan += a[i].y >> 32 & 0xff
end for
avg_qspan = sum_qspan / n
st = 0                                                          ▷ Starting index of inner loop
for i = 0; i < n; i++ do
    ▷ Initialize best score (max_f) and predecessor (max_j)
    max_f = q_span = a[i].y >> 32 & 0xff
    max_j = -1
    ▷ Find the starting index (st) of inner loop
    while st < i and a[i].x > a[st].x + max_dist_x do
        st++
    end while
    ▷ Compute scores between anchor i and previous H (at most) anchors and find the maximum score (max_f)
    for j = i - 1; j >= st and j >= i - H; j-- do
        score = compute_score(a[i], a[j], f[j], avg_qspan,
                                    max_dist_x, max_dist_y, bw)
        if score > max_f then
            max_f = score
            max_j = j
        end if
    end for
    ▷ Store maximum score and best predecessor for anchor i
    f[i] = max_f
    p[i] = max_j
end for
```

## 3 Hardware-Software Integration

The FPGA-based hardware accelerator is carefully integrated to the *minimap2* original software while minimally affecting the multi-threaded behavior of original software so that an efficient reduction in tool's end-to-end runtime can be achieved. OpenCL API functions are used for data transfer and communication between the host CPU (software) and the FPGA-based accelerator (hardware).

*minimap2* processes DNA reads on software with multiple threads by using a fork-join thread model. This allows *minimap2* to parallelize the computations being performed while maximally utilizing the multiple cores available in the CPU. When a chaining request is issued by any of the software threads of hardware-software integrated version of *minimap2* (i.e. *mm_chain_dp* function is called by a software thread), it is decided whether the requested chaining task should be processed on hardware or software with the hardware-software split mechanism discussed in Supplementary Note 2. If the chaining task is decided to be processed on software, the associated software thread continues to execute the chaining task on the CPU itself.

Since the FPGA devices are resource-constrained, the number of hardware kernels (*N* in Fig. S2) that can be configured on the device is usually lower than the number of software threads available on a typical high performance computing (HPC) system. Therefore, when the chaining task is decided to be processed on hardware, each software thread in the host CPU tries to schedule the chaining computation on one of the *N* hardware kernels configured on the FPGA device as given in Algorithm 3.

In lines 1-18 of Algorithm 3, given the predicted hardware and software execution times, the algorithm tries to find a hardware kernel on which the total time to finish processing the chaining task (*total_hw_time_pred*, which is the sum of the predicted hardware processing time (*hw_time_pred*) and the waiting time to access that kernel (*wait_time*)), is smaller than the predicted software execution time (*sw_time_pred*). If *total_hw_time_pred* is greater than or equal to *sw_time_pred*, the associated software thread executes the chaining task on the CPU itself. If *total_hw_time_pred* is smaller than *sw_time_pred*, the hardware kernel which satisfied the condition is recorded (*kernel_id*) and the chaining task is inserted into a first-in, first-out (FIFO) queue corresponding to the hardware kernel (*hw_queue[kernel_id]*), so that the CPU can wait and execute the task on the hardware kernel when the chaining task comes first in the queue. Lines 20-33 of Algorithm 3 correspond to the section where it takes the chaining task out from the queue and processes it on the previously recorded hardware kernel (*kernel_id*) when it is ready to do so. As this algorithm is performed on multiple software threads running in parallel and the *N* hardware kernels and the *N* hardware queues are shared among the multiple threads, necessary synchronization mechanisms (highlighted in blue color in Algorithm 3) are implemented with two mutex locks for each hardware kernel - one for accessing/modifying the queue corresponding to the kernel (called "queue lock") and one for accessing the kernel for chaining task execution (called "execution lock").

**Algorithm 3** Hardware Scheduling Algorithm

**Inputs:**
*hw_time_pred*: Predicted hardware execution time of chaining task (by Equation 1)
*sw_time_pred*: Predicted software execution time of chaining task (by Equation 2)

```
1:  should_wait = false
2:  for kernel_id = 0; kernel_id < NUM_HW_KERNELS; kernel_id++ do
3:      Acquire kernel's queue lock
4:      wait_time = hw_end_times[kernel_id] - current_time
5:      total_hw_time_pred = wait_time + hw_time_pred
6:      if total_hw_time_pred < sw_time_pred then
7:          hw_queue[kernel_id].push(tid)
8:          hw_end_times[kernel_id] = current_time + total_hw_time_pred
9:          should_wait = true
10:     end if
11:     Release kernel's queue lock
12:     if should_wait == true then
13:         break
14:     end if
15: end for
16: if should_wait == false then
17:     "process the chaining task on software"
18: end if
19:
20: got_in = false
21: while true do
22:     Acquire kernel's execution lock
23:     Acquire kernel's queue lock
24:     if hw_queue[kernel_id].front() == tid then
25:         hw_queue[kernel_id].pop()
26:         got_in = true
27:     end if
28:     Release kernel's queue lock
29:     if got_in == true then
30:         break
31:     end if
32:     Release kernel's execution lock
33: end while
34: "process the chaining task on hardware"
35: Release kernel's execution lock
```

# 4 Experiments and Benchmarks

## 4.1 Performance Benchmarking

The commands used for the performance benchmarking detailed under "Performance Comparison" are given below. The same commands were used both on Intel FPGA-based system and the Xilinx FPGA-based system.

For performance benchmarking with ONT data (w/o alignment and w/ alignment respectively):

```
$ minimap2_fpga -x map-ont -t 8 hg38noAlt.idx ont_reads.fastq > alignment.paf
$ minimap2_fpga -ax map-ont -t 8 hg38noAlt.idx ont_reads.fastq > alignment.sam
```

For performance benchmarking with PacBio CCS data (w/o alignment and w/ alignment respectively):

```
$ minimap2_fpga -x asm20 -t 8 hg38noAlt.idx pbccs_reads.fastq > alignment.paf
$ minimap2_fpga -ax asm20 -t 8 hg38noAlt.idx pbccs_reads.fastq > alignment.sam
```

## 4.2 Accuracy Benchmarking

The commands used for generating the simulated reads used in "Accuracy Evaluation" are given below.

For ONT simulated reads generation:

```
# generate raw signal data
$ squigulator hg38noAlt.fa -x dna-r9-prom -o ont_sim_reads.blow5 -n 1000000 -t 8
# basecall the raw signal data
$ buttery-eel  -i  ont_sim_reads.blow5  --guppy_bin /path/to/ont-guppy-6.3.7/bin/
 --config dna_r9.4.1_450bps_hac_prom.cfg -x cuda:all -o ont_sim_reads.fastq
 --port 5555 --use_tcp
```

For PacBio simulated reads generation:

```
# generate reads
$ pbsim --data-type CCS --length-max 20000 --depth 6 --model_qc data/model_qc_ccs
--length-mean 15000 hg38noAlt.fa
# convert reads to a mapeval compatible format
$ k8-Linux paftools.js pbsim2fq hg38noAlt.fa.fai <reads>.maf > pbccs_sim_reads.fa
```

## 5 Energy-Delay Product Estimation

**Table S2. Energy-delay product comparison on Xilinx FPGA-based system**

| Dataset | Mode | CPU core utilisation | | Average compute power (W) | | | Runtime (s) | | EDP (kJs) | | EDP reduction % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | mm2 | mm2_fpga | mm2 (CPU) | mm2_fpga | | mm2 | mm2_fpga | mm2 | mm2_fpga | |
| | | | | | CPU | FPGA | | | | | |
| ONT | w/o align. | 7.94 | 6.58 | 63.96 | 53.01 | 9 | 23,414.40 | 13,632.72 | 3.51E+07 | 1.15E+07 | **67.13** |
| | w/ align. | 7.96 | 7.25 | 64.12 | 58.40 | 9 | 37,525.25 | 26,759.62 | 9.03E+07 | 4.83E+07 | **46.55** |
| PacBio | w/o align. | 7.94 | 5.04 | 63.96 | 40.60 | 9 | 12,424.76 | 9,480.35 | 9.87E+06 | 4.46E+06 | **54.85** |
| | w/ align. | 7.96 | 7.55 | 64.12 | 60.82 | 9 | 71,630.86 | 66,718.01 | 3.29E+08 | 3.11E+08 | **5.53** |

**mm2**: *minimap2* , **mm2_fpga**: *minimap2_fpga*

The Intel Xeon E5-2686 v4 CPU in the Xilinx FPGA-based system is an 18-core CPU, but only 8 vCPUs are available for use on the AWS F1 instance (*f1.2xlarge*). The 18-core CPU has a thermal design power (TDP) of 145W. CPU power consumption was estimated by scaling the TDP value by CPU core utilisation ($Power_{CPU} = \frac{TDP}{Total\ no.\ of\ CPU\ cores} \times CPU\ core\ utilisation$).

For *minimap2_fpga*, total power was calculated as $Power_{Total} = Power_{CPU} + Power_{FPGA}$, where $Power_{FPGA}$ is the average FPGA power consumption measured by the dedicated tool. For *minimap2*, total power equated to CPU power (i.e. $Power_{Total} = Power_{CPU}$). Energy consumption was computed as $Energy = Power_{Total} \times Runtime$, and the energy-delay product (EDP) was determined as $EDP = Energy \times Runtime$. The estimated energy-delay product results for *minimap2* and *minimap2_fpga* are in Table S2.