Li YW, Tan GM, Li XQ. FuHsi: Shifting base-calling closer to sequencer via in-cache acceleration. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 40(2): 482-499, Mar. 2025. DOI: 10.1007/s11390-025-3886-5, CSTR: 32374.14.s11390-025-3886-5

FuHsi: Shifting Base-Calling Closer to Sequencer via In-Cache Acceleration

Ye-Wen Li[1, 2] (李叶文), *Member, CCF*
Guang-Ming Tan[1, 2, *] (谭光明), *Senior Member, CCF, Member, ACM, IEEE*
and Xue-Qi Li[1, *] (李雪琦), *Member, CCF, ACM, IEEE*

[1] *State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: liyewen19s@ict.ac.cn; tgm@ict.ac.cn; lixueqi@ict.ac.cn

Received October 23, 2023; accepted January 14, 2025.

Abstract Base-calling is an essential step in the analysis of third-generation genome data. Many previous hardware efforts aimed at enhancing processing in the workflow. However, an order of magnitude throughput gap still exists. In this paper, we propose FuHsi to improve the end-to-end throughput of the base-calling process. FuHsi is an in-cache accelerator that only introduces three components to the traditional CPUs in the sequencer. We propose FuHsi Cache, which offloads the bottleneck operations to cache arithmetic. Specifically, we accelerate beam search, string conversion, and MAC (multiply-accumulate) using algorithm/hardware co-design. We also introduce FuHsi APIs and FuHsi Controller to provide coarse-grained control for FuHsi Cache. Experimental results show that FuHsi can achieve 45.7x, 113.1x, and 100x throughput per watt speedup compared with an NVIDIA Jetson baseline, an NVIDIA A100 GPU baseline, and the Helix accelerator, respectively. FuHsi can provide base-calling requests for up to 15 ONT sequencers simultaneously.

Keywords genome base-calling, in-cache accelerator, domain-specific architecture, genome analysis, Nanopore sequencing

1 Introduction

The emerging technology of Nanopore (third-generation) genome sequencing[1–3] is revolutionizing genome research, industry, and markets. This technology can generate highly extended DNA fragments (a.k.a. long reads), essential for identifying large structural variations and improving the quality of de-novo assembly. Additionally, portable real-time Nanopore sequencing devices, such as MinION and SmidgION, have proven effective in tracking Zika, Ebola, and COVID-19[4–6].

Base-calling is the first step in third-generation genome analysis, which is responsible for restoring the electrical signals to the original bases, i.e., A, C, G, T, and N. As shown in Fig.1, the base-calling step consists of two phases: encoding and decoding. The encoding phase converts the continuous electrical signals into a probability distribution matrix of base encoding, while the decoding phase further converts the probability distribution matrix into a continuous sequence of bases. Currently, mainstream base-calling tools (such as SACall[7], CATCaller[8], Causalcall[9], and so on[10, 11]) have achieved an accuracy of over 90%.

Despite the high accuracy of these efforts, there is an order-of-magnitude gap between base-calling throughput and sequencer throughput. For instance, the average throughput of CATCaller on a mainstream edge platform, the NVIDIA Jetson AGX Orin,

Regular Paper

The work was supported in part by the National Key Research and Development Program of China under Grant Nos. 2021YFB-0300202 and 2022YFB4500403, and the National Natural Science Foundation of China under Grant Nos. 62202454, 62032023, and T2125013.

*Corresponding Author (Xue-Qi Li is responsible for the overall design and guidance of the paper work, and architecture design; Guang-Ming Tan is the instructor of the work and is responsible for algorithm optimization.)

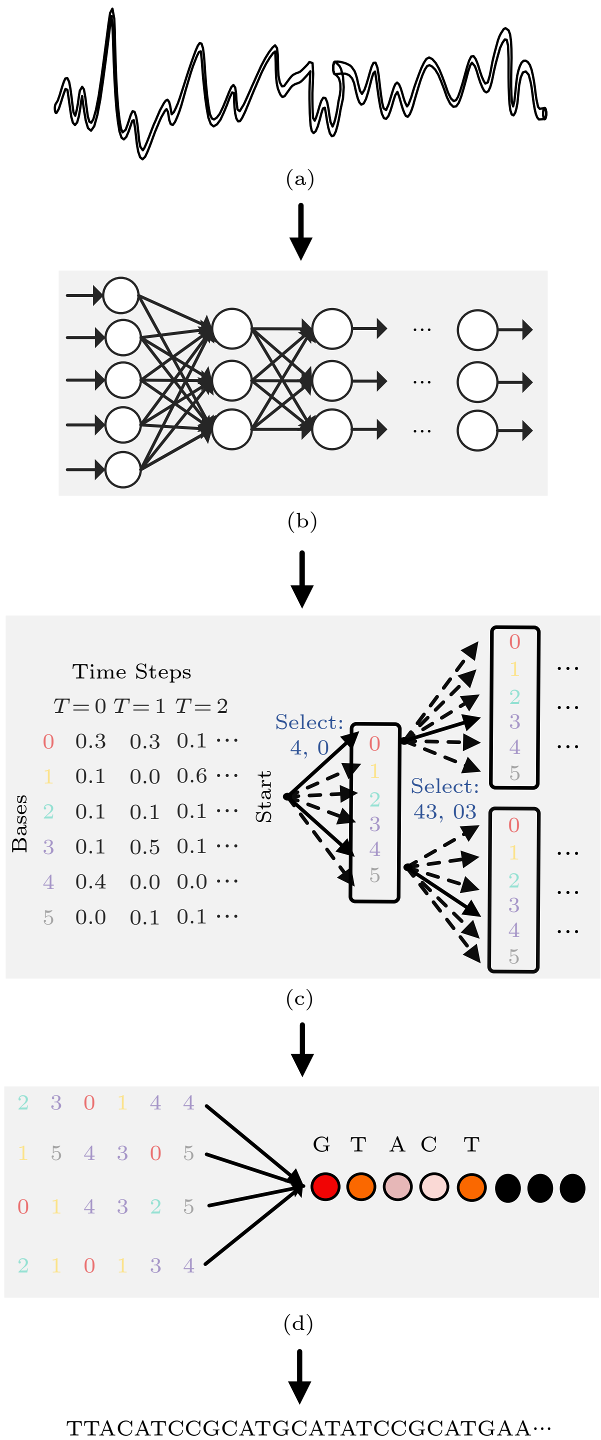


Fig.1. Nanopore base-calling pipeline including the encoding and decoding phases. (a) Input base-calling signal (fast5 format). (b) Neural network step. (c) Beam search step. (d) String conversion step.

is 519 000 samples/s, which is only 0.25x of the throughput of the MinION sequencer.

We argue that parallelism and data movement are bottlenecks in base-calling, and recent designs fail to solve those problems. First, the GPUs fail to solve the parallelism and data movement problems. The GPU Tensor Core can provide highly parallel acceleration for neural network computation of base-calling but still lacks support for other non-neural network operations (e.g., beam search and string conversion in Subsection 2.1), which does not fully utilize the high parallelism of GPUs. The data movement between the main memory and the processor (inputs, weights, and base-calling intermediate data) also limits base-calling performance. Second, previous accelerators (e.g., Helix[12], GenPIP[13], and Swordfish[14]) solve the parallelism problem but cannot cope with the frequent data movement of base-calling. They accelerate base-calling as a deep-learning task, and SOT-MRAM (Spin-Orbit Torque Magnetic Random-Access Memory) and ReRAM (Resistive Radom-Access Memory) provide substantial parallelism for neural network Dot-Product computations. However, other non-neural network operators in base-calling require frequent data movement, and the overhead of data-in and data-out (ADCs and DACs) greatly degrades the throughput of base-calling.

CPUs are often the fundamental components of many edge devices (e.g., ONT sequencers), and the cache is a core component of a CPU. Therefore, performing computations within the cache has been a primary approach in recent years to increase data processing parallelism and improve the efficiency in edge devices. The key idea of in-cache computing[15, 16] is to activate multiple cache rows simultaneously and use analog computation to achieve operations such as AND and NOR. The advantage of in-cache computing lies in both high parallelism and low energy consumption. For instance, duality cache[15] supports 1 146 880 parallel floating-point operations with a 3.5% processor die area overhead for the Xeon E5-2697 with a 35 MB cache. Moreover, since in-cache computing uses read-and-write operations inside the cache, the energy consumption is only 19% of traditional L3 cache read-and-write operations[16].

The goal of this paper is to systematically analyze the bottleneck operators of base-calling and enhance data processing efficiency by shifting base-calling operators closer to the sequencer via an enhanced cache. Thus, we introduce FuHsi①, which is an in-cache base-calling accelerator. The FuHsi system en-

①FuHsi is a goddess from Chinese mythology who was the first to summarize the I-sing, an ancient system of cosmology and philosophy.

hances the traditional CPU by introducing only three key components, i.e., FuHsi Cache, FuHsi Controller, and FuHsi API. FuHsi Cache enhances the traditional cache to accelerate all bottleneck operators in base-calling. From the software perspective, FuHsi API uses a Python style to define all the key operators coarse-grained, making it easy to integrate with existing machine-learning frameworks. FuHsi Controller supports fetching and decoding FuHsi APIs and provides fine-grained control for our APIs.

For the arithmetic aspect, we propose three key in-cache base-calling operators. First, we propose in-cache beam search, which offloads the beam search computations to the cache via algorithms/architecture co-design. Second, we propose an algorithm-specific mapping mechanism to eliminate data transfer overhead in string conversion and then introduce a De-Morgan-Laws based transformation strategy that can efficiently execute within the cache while providing flexibility. Third, we propose the alignment-free MAC structure for the dot-product operation, reducing the latency required for exponent alignment during the addition-and-reduction process.

Our specific contributions are listed as follows.

1) We obtain three key observations that clearly characterize base-calling applications, which motivate us to design an energy-efficient base-calling accelerator.

2) We propose the FuHsi in-cache acceleration system. Specifically, we propose the FuHsi Cache to enhance the existing CPU in ONT the sequencer. We propose the FuHsi API to coordinate the execution between the host CPU and cache, and propose the FuHsi Controller to provide fine-grained control for the cache.

3) We shift and accelerate the key arithmetic of base-calling by proposing the in-cache beam search, the in-situ string conversion, and the alignment-free MAC algorithms and structures.

4) Experimental results show FuHsi achieves an average throughput improvement of 8.34x and 2.13x, and a power consumption reduction of 5.49x and 24.48x, compared with the Jetson AGX Orin and NVIDIA A100 GPU, respectively. FuHsi also achieves a two-order-of-magnitude throughput/watt improvement over the Helix accelerator. Our FuHsi accelerator can serve up to 15 sequencers simultaneously.

2 Background

2.1 Nanopore Base-Calling

Base-calling is responsible for restoring the electrical signals to the original bases, i.e., A, C, G, T, and N. The input is the base-calling signals sequenced from the Nanopore sequencer (Fig.1(a)), and the output is the original base. The process has three steps.

1) *Neural Network*. This step converts the electrical signals into a base probability matrix (Fig.1(b)). Since an electrical signal represents 6 to 10 input bases around it, it can be modeled as a time series prediction problem. A significant amount of work is based on deep neural networks (e.g., CNN[9], LSTM[10], and Transformer[7, 8]) to improve accuracy.

2) *Beam Search*. The input for this step is a probability matrix that contains the probability of possible bases at each time step. The connectionist temporal classification (CTC)[17] and beam search are adopted in this step to obtain the global optimal base sequence. Fig.1(c) shows the search when the beam width is two, only expanding two bases with the most significant probabilities each time step.

3) *String Conversion*. Since bases are stored in compressed format in the front process, they need to be converted into natural bases (Fig.1(d)). According to a predefined decoding table, we convert the compressed sequences into natural bases and splice them in order.

2.2 Processing-in-Cache

Modern general-purpose processors and accelerators integrate large on-chip cache to take full advantage of locality. They not only occupy a large number of on-chip resources but also have costly data movement over the cache hierarchy. To address these shortcomings, recent work[15, 16, 18–20] refines and tunes the cache structure and converts it into a large number of data-parallel units.

Fig.2(a) shows the basic idea of in-cache computing. By activating two word-lines (WLs), WL_i and WL_j, simultaneously, the voltages of the two word-lines will interfere with each other and reflect the final result on the bit-line (BL). By giving a reference voltage, an AND operation is performed by sensing the bit-line, while an NOR operation can be performed by sensing the bit-line bar (BLB). Jelika *et al.*[21] showed that no data corruption occurs even when 64 word-lines are activated simultaneously. Furthermore, neural cache[18] and duality cache[15] use cache to accelerate integer and floating-point neural networks, respectively. Fig.2(b) shows the circuit diagram of the SRAM (Static Random-Access Memory) cell, using six transistors to represent 0 or 1.

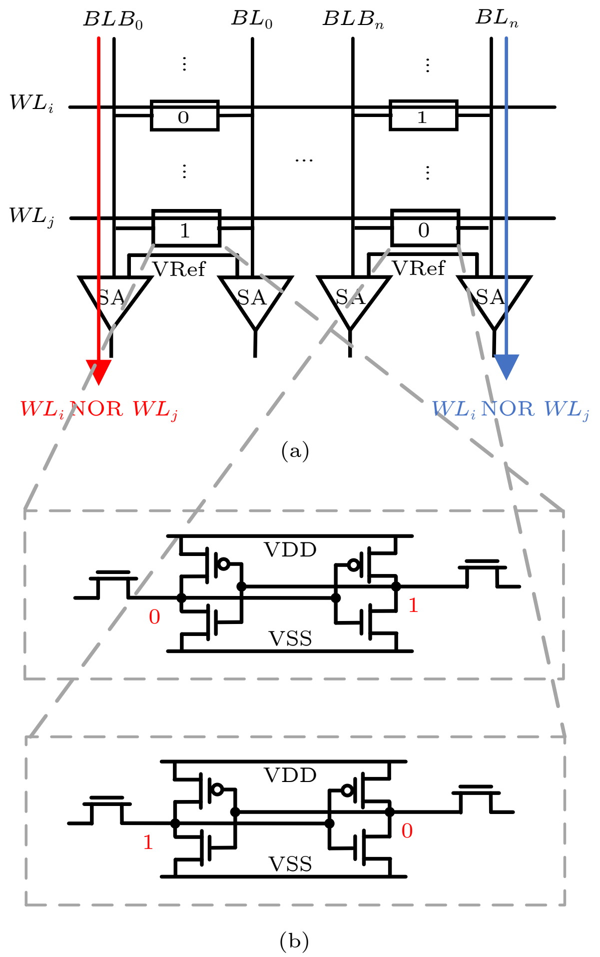


Fig.2. (a) SRAM circuit for in-place operations. Two SRAM rows (WL_i and WL_j) are activated simultaneously. (b) 6T-SRAM circuit for 0 and 1.

Comparison with Other PIMs. Considering the resources available on the sequencer, there are three processing-in-memory (PIM) approaches to accelerating base calling: in-DRAM, in-flash, and in-cache. In-DRAM and in-flash are more suitable for accelerating memory-bound tasks (e.g., FM-Index search[22], graph mining[23]). The limited logical resources and low clock frequency limit the computing ability. In contrast, in-cache computing can provide a large number of computing resources (1 474 560 ALUs for 45 MB cache[15]) and a high-speed internal operating frequency (8 GHz)②, making it suitable for accelerating base-calling.

3 Characterization

We analyze the base-calling workload and obtain the following three key observations.

Observation-1. There is a significant gap between sequencer throughput and base-calling throughput. Fig.3(a) compares the throughput of different base-calling tools with the throughput of the ONT sequencer. The sequencer has a sequencing throughput of 2 050 000 samples/s. Some base-calling tools can achieve this throughput if we focus on the through-

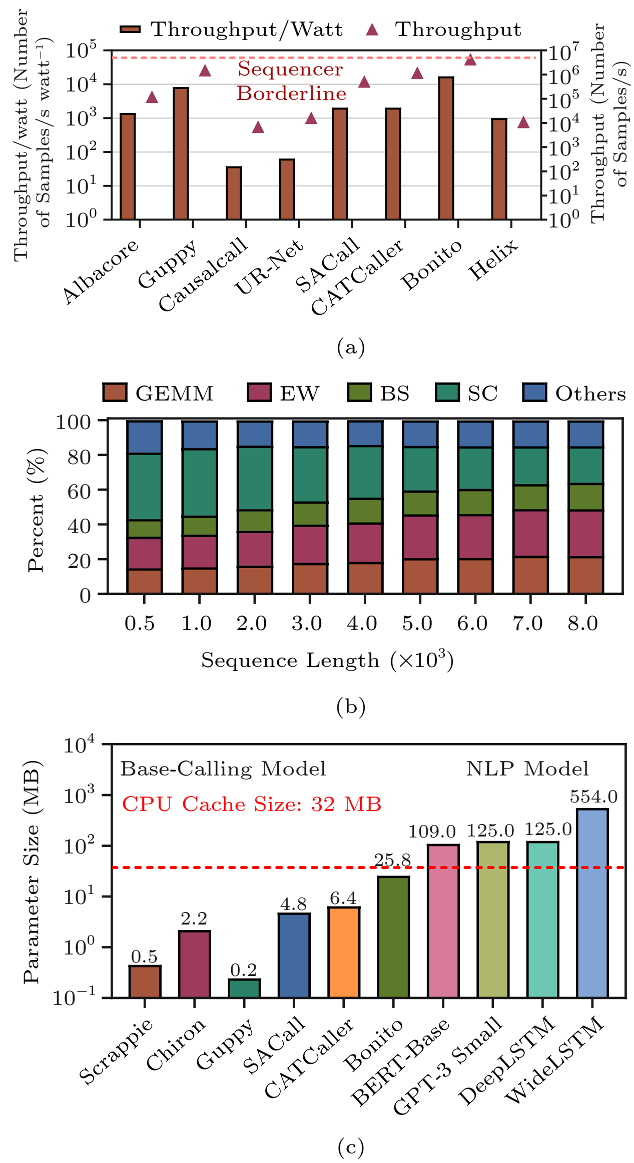


Fig.3. Base-calling characterization. (a) Throughput per watt comparison and throughput comparison of base-calling tools and sequencers. (b) Run-time breakdown of operations for base-calling as a function of input sequence length. The data loading time is not included here. EW: element-wise, BS: beam search, SC: string conversion. (c) Parameter size comparison of base-calling and NLP models.

②The external read/write frequency of the cache is 2 GHz. Using the sense-amplifier cycling technique proposed by Cache Automaton[24], the internal frequency of the cache can reach 8 GHz.

put. However, those tools are very computationally intensive. For example, Bonito requires a high-end NVIDIA A100 GPU to achieve a throughput of 4 300 000 samples/s. The current borderline throughput for ONT sequencers is 34 167 samples/s watt^{-1}. When considering the base-calling energy efficiency, there is a 1–3 order of magnitude gap between the base-calling tools and sequencers.

Observation-2. The encoding phase (e.g., neural network) and decoding phase (e.g., beam search and string conversion) are performance bottlenecks and need to be accelerated. Fig.3(b) shows the run-time breakdown of operations for base-calling as a function of input sequence length③. The bottleneck in the encoding phase is the general matrix to matrix multiplication (GEMM) operation and element-wise operation, which occupy 32%–48% of the execution time. The bottlenecks in the decoding phase are the beam search operation and the string conversion operation, which also occupy 36%–48% of the execution time. All other operations occupy 15%–19% of the execution time.

Observation-3. The base-calling neural network model is lightweight compared with other applications. Fig.3(c) compares the parameter sizes of the base-calling and natural language processing (NLP) model. The parameter sizes of the base-calling models are almost 0.2 MB–6.4 MB, while those of NLP models are all larger than 100 MB.

Considering all the above three observations, we conclude with three philosophies to design a base-calling accelerator. First, energy efficiency is crucial in designing a base-calling accelerator. Our base-calling accelerator needs to have not only a high throughput but also an advantage in energy consumption. Second, we must accelerate all the bottleneck operations of base-calling. Due to Amdahl's law, only accelerating matrix multiplication like neural network applications cannot improve the throughput of base-calling. Third, such a small model makes base-calling suitable to execute on edge devices. The thermal design power (TDP) of the sequencer is 60 watts, and base-calling computations can be executed directly within the sequencer.

4 FuHsi System

Based on our observation, in this section, we first introduce a novel sequencer data processing system, followed by a description of the computational and control flow of the data processing accelerator. Then, we will discuss integrating the new accelerator components into the sequencer system through APIs.

4.1 New Data Processing Component

Traditional Sequencer. Fig.4(a) shows a traditional base-calling pipeline. First, the CPU core controls the flow cell to start sequencing using MinKNOW software. Second, the flow cell sequences the input genome into electrical signals (fast5 format) and writes them to DRAM. Third, the CPU transmits the electrical signals to the remote server (e.g., NVIDIA A100 GPU) and completes the base-calling process. This scheme is less energy efficient and still far from edge genome analysis.

New Sequencer with In-Cache Accelerator. Since CPUs are widely available computational resources in sequencers, base-calling models are generally lightweight. Our key idea is to achieve computational acceleration of base-calling by enhancing the CPU cache in the sequencer. Fig.4(b) demonstrates our proposed sequencer. Both sequencing and base-calling can be done inside the sequencer using our architecture. In the followings, we will show 1) the in-cache base-calling architecture by enhancing traditional CPUs; and 2) how to use in-cache operations to accelerate the bottleneck arithmetic under the energy requirements of the sequencers.

4.2 Architecture Overview

Modified Components. Fig.5(a) illustrates the architecture overview of FuHsi. Compared with typical CPUs, FuHsi introduces two new components: the FuHsi Controller and the FuHsi Cache. The FuHsi Controller receives specific APIs from CPUs and controls the execution of these APIs. The FuHsi Cache is modified from a typical CPU's last-level cache (LLC) and can offload the base-calling operations.

Cache Slices Partition. Fig.5(b) demonstrates the cache slice organization. Since FuHsi only supports the acceleration of Nanopore base-calling, it is a specialized genome analysis application. To reduce the design complexity of the FuHsi Controller, we use profiling to determine the run-time of each operation

③The profiling model is CATCaller[8], and the hardware platform is NVIDIA A100 GPU. We adopt the input sequence length from 500 to 8 000.

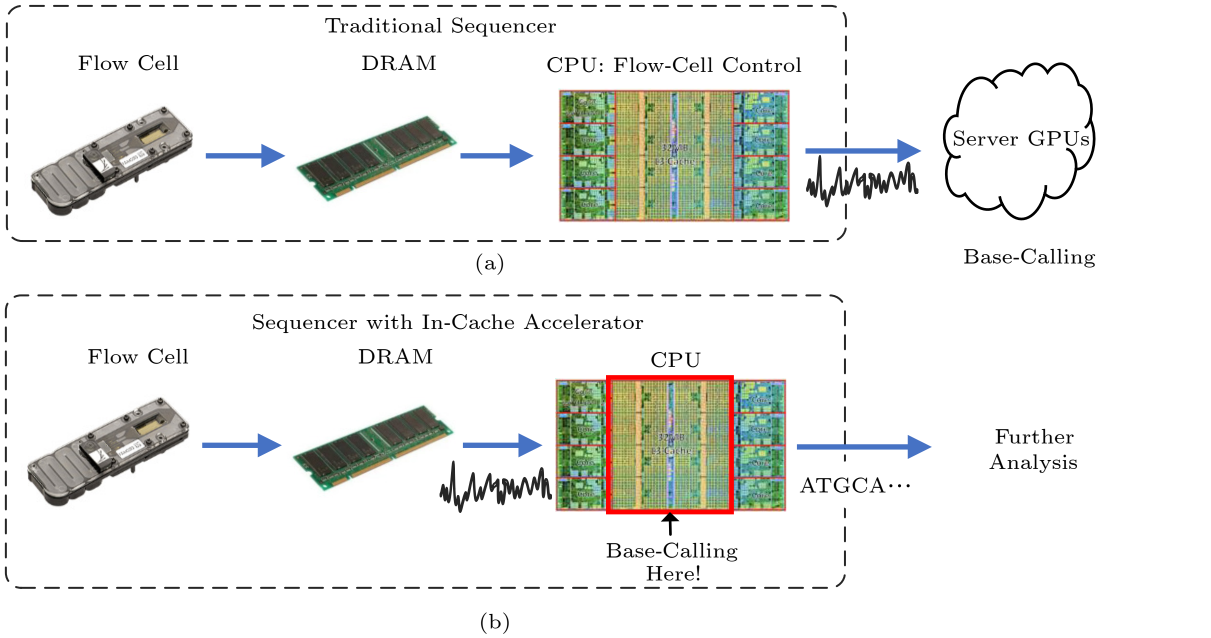


Fig.4. (a) Traditional Nanopore sequencer. The CPU is used for the flow-cell control, and the base-calling analysis is offloaded to the remote server. (b) New sequencer with in-cache accelerator. The CPU is used for both flow-cell control and base-calling analysis.

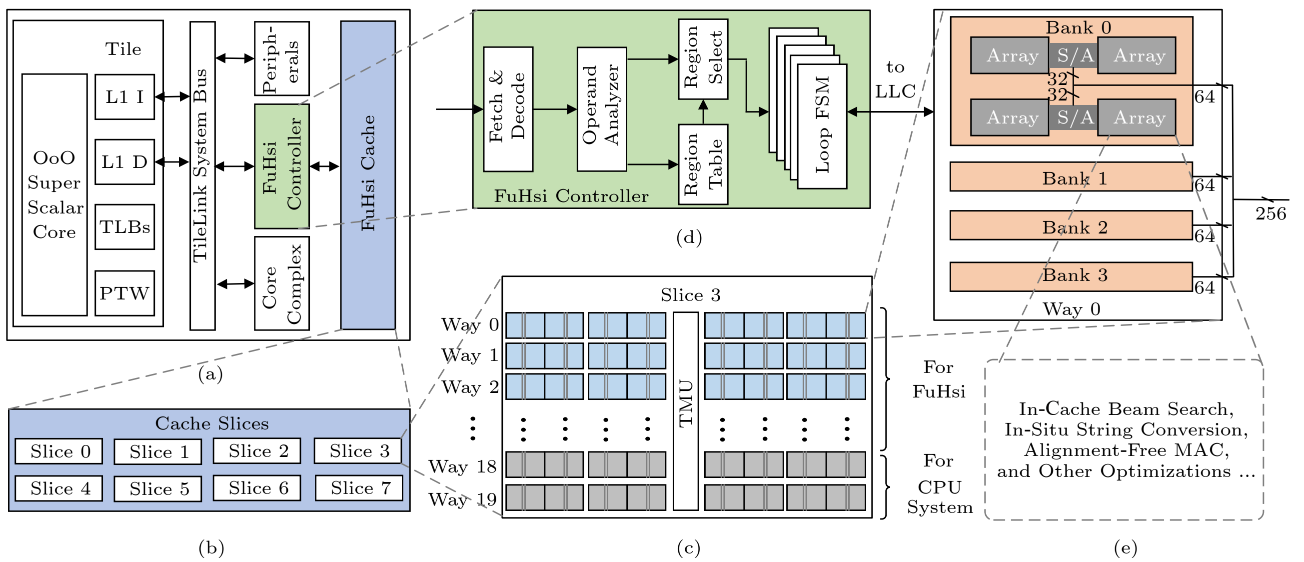


Fig.5. FuHsi architecture overview. (a) Whole CPU architecture of FuHsi, which adds two components to traditional CPUs, i.e., FuHsi Controller and FuHsi Cache. (b) Cache slice composition of FuHsi. It is the same as that of traditional CPUs. (c) Single cache slice contains 20 ways, 18 ways for FuHsi, and two ways for the CPU system. (d) Architecture of FuHsi Controller. It supports fetching and decoding FuHsi APIs and provides fine-grained control for our APIs. (e) Each way contains four banks and each bank contains four SRAM arrays. The FuHsi arithmetics are offloaded into the SRAM array.

and divide the executable region of each operation in advance. For example, slice 0 to slice 3 execute encoding (neural network) instructions, slice 4 and slice 5 execute beam search instructions, and slice 6 and slice 7 execute string conversion instructions. This partitioning is determined based on the execution time proportion of key operators through pre-profiling, and the partitioning message is stored in the region table shown in Fig.5(c). Since our cache arrays support all operations and only account for 0.13% of the chip area, changing the partition strategy does not affect the execution of operations.

Fig.5(c) illustrates the internal partitioning within a slice. In the accelerator mode, way 0 to way 17 are used for in-cache computation, while way 18 and way 19 are allocated for normal CPU operations. It should be clarified that the 2-way cache for the CPU is only used to control the code, while the 18-way cache for FuHsi is not only used for computing but also for storing the data (weight, beams, and bases). This is the reason why in-cache computing reduces data movement. Two cache ways for CPU normal operations will increase the run-time of the scalar part of the system. Thanks to the end-to-end analysis of the base-calling process, we provide sufficient speedup for the vector and tensor operations. The remaining scalar operations are mainly used for base-calling program control and scalar operations. We evaluate the base-calling execution latency breakdown in Subsection 6.3. All ways from 0 to 19 are used for regular CPU operations in the CPU mode.

Control Flow. Fig.5(d) presents our controller, divided into the pre-execution and execution control flow. The pre-execution control flow activates the accelerator mode of cache, pre-places weights, and writes the pre-profiling message to the region table. After parsing the instructions, the execution control flow allocates computing requests to available cache arrays based on the partitioning message. We implement a loop FSM for each bank to simplify programming and reduce computation latency, allowing control over micro-operations within the cache. All banks' loop FSMs have the same implementation logic and support control over all operations.

4.3 Software Integration

We design a software library to coordinate execution between the host CPU and the FuHsi Cache. As shown in Table 1, FuHsi offers Python-style APIs, which can be flexibly integrated with existing machine-learning frameworks. The APIs are categorized into three types, i.e., preparation, data transmission, and computation. The preparation APIs are used to select FuHsi's operating mode, deploy weight data into the cache, and configure the region table. The transmission APIs handle sending input data and receiving the base-calling output. The computation APIs are used to offload key operators in base-calling, which are invoked multiple times. We use the neural cache[18] to implement the in-cache element-wise operations. It is manual to write base-calling programs using our APIs, and we will support automated use of our APIs in the future.

Table 1. FuHsi APIs

Type	API	Description
Preparation	set_mode()	Working mode choice
	weight_deploy()	Deploy the weight data
	set_region()	Configure region table
Transmission	send_input()	Send input data to cache
	get_result()	Get the output base from cache
Computation	mac()	Compute MAC
	top_k()	Get the top-bw elements
	string_convert()	Convert the input tensor to the output base
	mask_filled()	Compute masked_filled
	elem_add()	Compute element-wise addition operation
	elem_mul()	Compute element-wise multiplication operation

5 FuHsi Arithmetic

5.1 In-Cache Beam Search

Bottleneck Analysis. Base-calling uses beam search to obtain the best candidate, which can overcome the local optimization problem for long sequences. Algorithm 1 shows the original beam search algorithm process. At each time step, each of the bw beams is extended by C characters, $bw \times C$ beams have to be sorted and account for $O(bw \times C \times \log(bw \times C))$. As this sorting operation happens for each of the T time steps, the overall time complexity is $O(T \times bw \times C \times \log(bw \times C))$. The two inner loops only account for $O(bw \times C)$. Therefore, the performance bottleneck of beam search is the sorting operation.

Algorithm 1. Beam Search Algorithm

Input: probability matrix PM, beam width bw.
Output: most probable labeling $Score$.

$Beams \leftarrow \varnothing$; $Score(\varnothing, 0) \leftarrow -1$;
// Go over all time steps
for $t = 1$ to T **do**
    $Beams \leftarrow \{\}$;
    $\hat{B} \leftarrow Sort(Beams)[0 : bw]$;
    // Go over candidate beams
    for $y \in \hat{B}$ **do**
        Calculate $Score(y, t)$ using PM;
        Add y to $Beams$;
        // Extend current beam labeling
        for $c \in C$ **do**
            $y' \leftarrow y + c$;
            Calculate $Score(y', t)$ using PM;
            Add y' to $Beams$;
        end
    end
end
Normalize $Score(y, T)$ according to $|y|$;
return Highest $Score(y, T)$;

Optimization Opportunity. It is inefficient to perform the sorting operation in-cache directly since in-cache computing has limited operations (e.g., AND and NOR)[15, 16, 18, 19, 24]. For instance, the widely used compare-and-merge implementation[25, 26] would result in high latency using cache-supported operations. Thanks to the fact that Algorithm 1 first sorts all the beams and then takes the top-bw beams. We can merge those two operations into a top-k operation, which can be implemented efficiently using the cache operation.

Multi-Layer Fuzzy Search Strategy. The floating point numbers have a unique storage format. For example, FP16's format is $(-1)^{sign} \times 2^{exponent-31} \times 1.mantissa$. Here, $sign$, $exponent$, and $mantissa$ represent the sign bit, exponent bits, and mantissa bits of a floating-point number, respectively. Since we only need to get the top-k beams and do not require an explicit order, we can transform the floating point comparison into the exponent comparison. For instance, finding the top-k negative floats can be converted into a sequential search for the smallest exponent operation④.

We propose a multi-layer mask to search the minimized exponent. Let the total number of bits of the search space be N, and to ensure the search space is continuous, we define the interval of fuzzy strings as $(P \times 2^n,\ P \times 2^n + 2^n - 1)$, where $\log_2 P + n = N$. Fig.6 shows an example: the search space N, the first-class fuzzy mask n, and P are 5, 3, and 4, respectively. We can generate four fuzzy matching strings, i.e., "00XXX", "01XXX", "10XXX", and "11XXX".

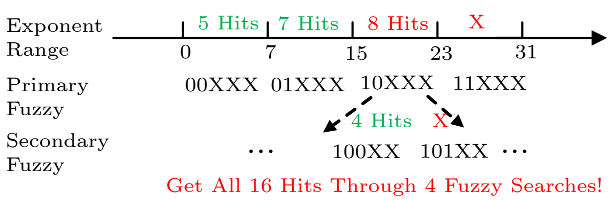


Fig.6. Three-layer mask example searching 16 hits from the exponent range of 0 to 31. Sixteen hits can be searched by using secondary fuzzy and searching three intervals.

Compared with the original algorithm, our strategy has three improvements.

1) *Partial Sorting.* The original algorithm finds all the beams before performing subsequent operations. Our algorithm only gets the corresponding beams in each mask, which saves the intermediate storage overhead.

2) *Multi-Layer Fuzzy Matching.* As shown in Fig.6, the total number of beams we need is 16, and the range 16–23 has eight matches. We can search for the second-layer fuzzy mask ("100XX") to get the remaining four beams. Since exact matches are difficult to get, we set a parameter σ to tolerate getting more beams. Since the correct beams are a subset of tolerated beams, this strategy does not affect the accuracy but only increases the computation time. We evaluate the amount of extra computation produced by this strategy in Subsection 6.3.

3) *Early Exit.* We usually do not need to traverse all the first-level fuzzy masks. We terminate the computation on the second-level mask "101XX" (16–23) in Fig.6.

Architecture Design. The key cache operation required for fuzzy matching is CAM search[15]. Fig.7(a)

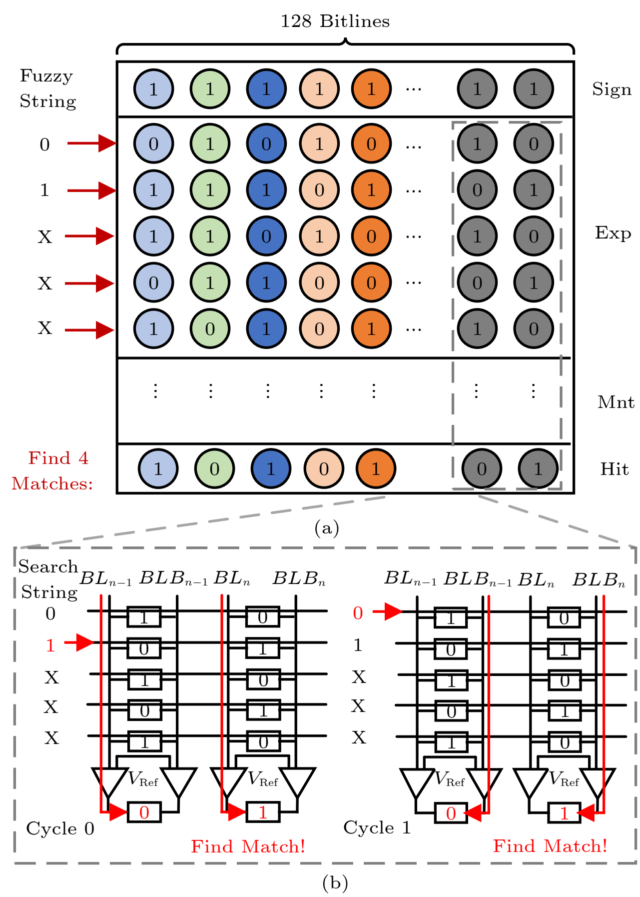


Fig.7. (a) In-cache operations searching the "01XXX" string in 128 columns. Four strings are found in the SRAM array. (b) Two-cycle in-cache CAM search searching the "01XXX" string. Cycle 0 searches all "1" of the input string from all SRAM columns, and cycle 1 searches all "01XXX" from all SRAM columns. The columns that hit cycle 0 and cycle 1 are considered the exact string as the inputs.

④Finding top-k integers and finding top-k positive floats are subsets of finding top-k negative numbers. When the exponent bits are all the same, the same idea is used to compare the mantissa bits.

shows how to search for a particular string in the cache. Suppose that the strings "10101" and "01010" exist in the cache and the string to be matched is "01010". At cycle 0, we activate the cache cells corresponding to "1" in the search string and simultaneously read the result from BL. If all the cells are "1", the result will be "1". The columns with the result "1" mean there is the same "1" pattern as the search string. Similarly, we can activate the cells corresponding to "0" in the search string and read the result from the BLB. Fig.7(b) illustrates how to query the first level mask "01XXX". We can find out in two cycles that four beams in the 128 columns of data are in the range that satisfies the exponent of 7–15.

Memory Optimization. The output sequence is sequentially increased during the execution of the beam search. The output sequence is 1 k–2 M[27], and the memory overhead is up to 128 MB, far exceeding the edge CPU cache size. Since the expansion is just on top of the original sequence each time step, we store the extended sequence in DRAM and the pointer in the cache. The sequence with maximum probability is reloaded from DRAM when the complete sequence is searched.

5.2 In-Situ String Conversion

Bottleneck Analysis. String conversion is the last step of base-calling, which converts the output of beam search into final bases[28]. As shown in Algorithm 2, the time-consuming process is lines 12–16. It takes the input sequence (out, the output of beam search), converts it according to a pre-defined conversion table (int_to_char), and concatenates the base in order ($results$).

Acceleration Choices. Using multiple CPU threads to perform multiple base conversions simultaneously is a common way to accelerate this problem. However, this solution is limited by the number of CPU threads and requires serial processing of base concatenation, which limits the efficiency of the parallelism. Hardware custom can provide sufficient throughput but still suffers from the need for serial base concatenation. In addition, the hardware design can only support a fixed conversion table, which conflicts with the diversity of base-calling tools. This paper proposes algorithm-specific data mapping to solve the base concatenation problem and implement a flexible and efficient base conversion mechanism using the cache array.

Algorithm 2. String Conversion Algorithm

Input: beam search output *out*, read length *seq_len*.
Output: base-calling results *results*.

```
1  int_to_char ← ["-", "A", "T", "C", "G", " "];
2  results ← [];
3  // Go over all batches
4  for b, batch in enumerate(out) do
5      utterances ← [];
6      // Go over all sequences
7      for p, utt in enumerate(batch) do
8          size ← seq_len[b][p];
9          if size > 0 then
10             transcript_lst ← [];
11             for utt_idx in size do
12                 // String convert core
13                 char_key ← utt[utt_idx].item();
14                 char ← int_to_char[char_key];
15                 transcript_lst.append(char);
16             end
17             transcript ='' .join(transcript_lst);
18         end
19         else
20             transcript ='';
21         end
22         utterances.append(transcript);
23     end
24     results.append(utterances);
25 end
26 return results;
```

Algorithm-Specific Mapping. Fig.8 shows our proposed algorithm-specific mapping for string conversion. It has two key features. First, As shown in Fig.8(a), the cache array is divided horizontally into two parts, one for the output of the beam searches (out, each has o_b bits) and one for the converted bases ($results$, each has r_b bits). Assuming the row number of the SRAM array is R. If the starting row of a particular beam search is b_i, then the starting row of the conversion output is $o_b\lfloor R/(o_b+r_b)\rfloor + r_b b_i/o_b$. This is because different columns of the SRAM array compute simultaneously, and different rows compute iteratively. For a traditional 256×256 cache array, our conversion engine can convert 256 bases simultaneously, and the SRAM array can store 47 662 conversion bases. Second, since the different conversion tasks are independent, we assign out into a zigzag layout. As shown in Fig.8(b), we use the advanced TMU to transform the zigzag layout into a sequential one. Fig.8(c) shows the schematic of the TMU cell. Adding two NMOS can support the transposed direction of data reading and writing[18].

De-Morgan-Laws Based Conversion. We propose a transformation strategy based on De-Morgan's laws

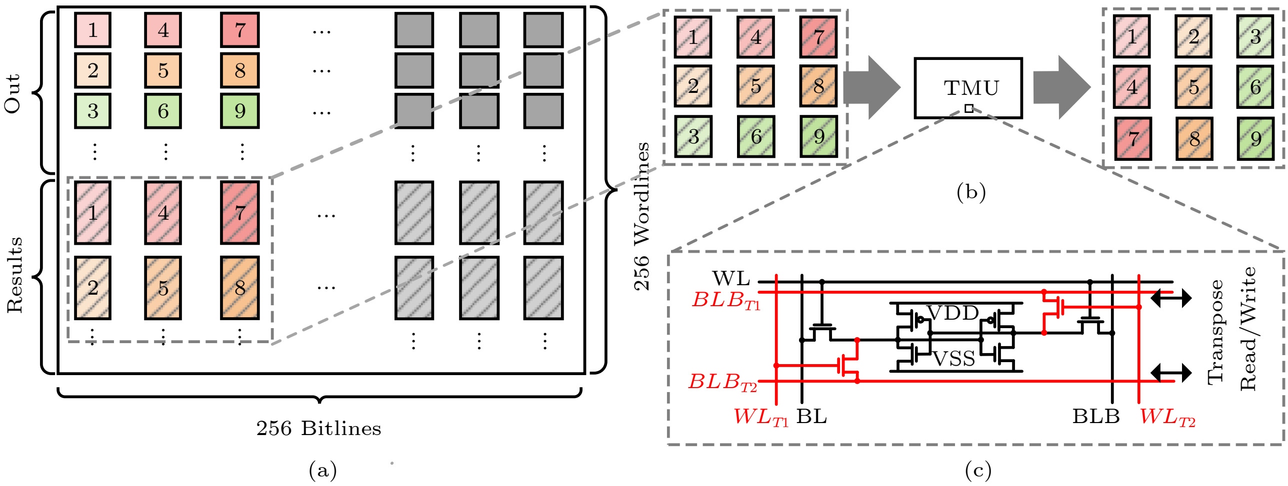


Fig.8. (a) Algorithm-specific mapping for string conversion. The input and output are both stored in row-first to support parallel conversions. (b) The row-first layout and the column-first layout. (c) The TMU cell design. It adds two transistors to the traditional 6T SRAM cell.

to use only in-cache operations and support flexible conversion. Suppose our conversion rule is {0 : "-", 1 : "A", 2 : "T", 3 : "C", 4 : "G", 5 : " "}, and assume that the input is $o_i (i \in [0, 3])$ and the output is $r_i (i \in [0, 7])$. Our proposed conversion strategy is as follows:

$$
\begin{aligned}
r_7 &= 0,\\
r_6 &= \neg o_2 o_0 \vee (\overline{o_2 o_1 \vee \neg o_2 \neg o_1}) \neg o_0,\\
r_5 &= \neg o_2 \neg o_1 \neg o_0 \vee o_2 \neg o_1 o_0,\\
r_4 &= \neg o_2 o_1 \neg o_0,\\
r_3 &= \neg o_2 \neg o_1 \neg o_0,\\
r_2 &= \neg o_2 \neg o_1 \neg o_0 \vee (\overline{o_2 o_1 \vee \neg o_2 \neg o_1}) \neg o_0,\\
r_1 &= \neg o_2 o_1 o_0 \vee o_2 \neg o_1 \neg o_0,\\
r_0 &= \neg o_2 \neg o_1 \vee \neg o_2 o_1 o_0 \vee o_2 \neg o_1 \neg o_0.
\end{aligned}
$$

The conversion can be done in a few cycles by converting r_i into cache-friendly operations (e.g., o_0, $\neg o_0$, $o_1 o_0$, $\neg o_1 \neg o_0$, $\overline{o_1 o_0 \vee \neg o_1 \neg o_0}$). For instance, r_3 is only one cycle, r_4 is only two, and r_2 is only four. The total cycles are 24 and can be further reduced to six cycles using the sense amplifier cycling techniques[24]. These operations can be supported by modified SRAM array peripherals[18].

5.3 Alignment-Free MAC

5.3.1 *Problem Definition*

As shown in Section 3, GEMM is the bottleneck operator for the encoding phase. The basic operation of GEMM is multiply-accumulate (MAC). Consider a MAC operation $Sum = \sum_{i=1}^{N} A_i B_i$, where $A_i B_i$ is element-wise multiplication. Let the result of $A_i B_i$ be P_i, and thus $Sum = \sum_{i=1}^{N} P_i$ is the reduction operation of addition. Since adding floating-point numbers requires ensuring that the exponent bits are the same, this severely limits parallelism. This problem can be defined as an optimization problem where we need to find an exp_{best} that guarantees the minimum consumption of shifts, i.e., $\min \sum_{i=1}^{N} |P_i.exp - exp_{\text{best}}|$. As a traditional algorithmic problem, we can solve this problem by sorting to find the median, and these operations are not easily implemented in the cache. In this paper, we propose alignment-free MAC to solve this problem, which previous work has not addressed[15, 18].

5.3.2 *Alignment-Free MAC*

Fig.9 shows the procedure of our alignment-free MAC, and it has three steps.

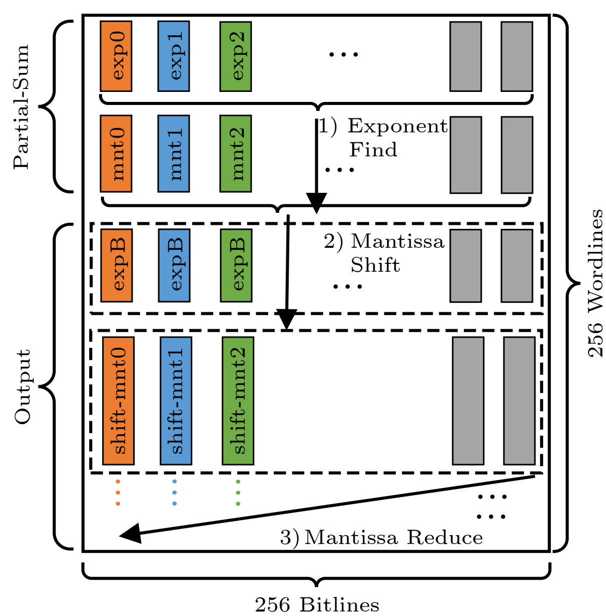


Fig.9. Process of alignment-free MAC architecture.

1) *Finding Exponent*. We count the exponent of the partial product generated in the MAC in the base-calling encoding phase. Fig.10 shows that 95% of the best exponential bits in the FP16 case are zero, and the remaining best exponential bits are also small. There are two reasons for this phenomenon: the first is the normalization operation, which makes the result far less than one. The second is that each partial product is obtained by multiplying it by two smaller numbers. Based on this observation, we set exp_{best} to 0.

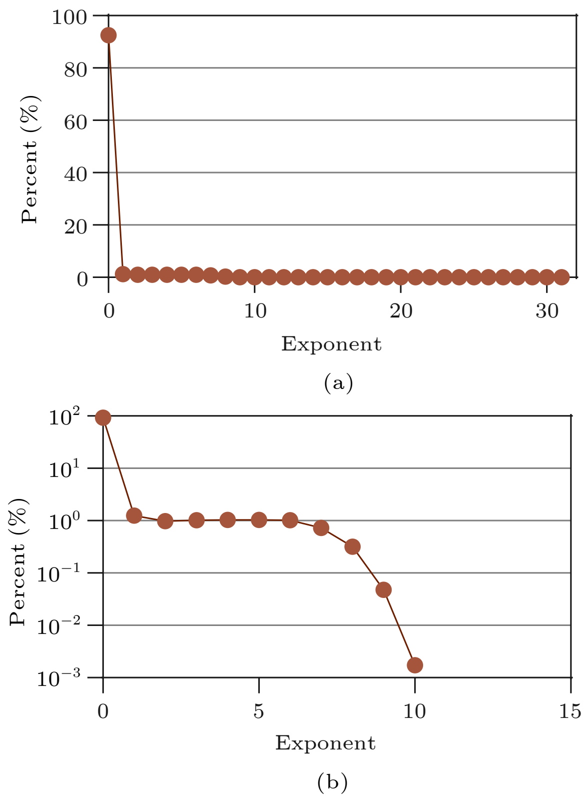


Fig.10. Distribution of exponent bits of the result of $A_i \times B_i$. (a) Exponent distribution ranges from 0 to 31. (b) Exponent distribution ranges from 0 to 15.

2) *Shift Mantissa*. Since exp_{best} is 0, the number of mantissa shifts is the value of the input exponent. To shift all inputs with the same exponent simultaneously, we use the CAM search structure (Subsection 5.1) to search all the inputs with the same exponent. To further reduce the time spent on CAM search, according to Fig.10, we determine the interval of the search for the exponent bit to be 30% of the total range of exponents. We also search for exponent bits larger than this range to guarantee precision and align them again. We use four masks, "00XXX", "01XXX", "10XXX", and "11XXX", to search the four exponent ranges and shift the mantissa of the hitting items. We evaluate the impact of different ranges on throughput in Subsection 6.3.

3) *Reduce Mantissa*. With the first two steps, the floating-point reduction problem converts to the reduction of mantissa. We use the reduced operation of the neural cache[18] to do the rest of the reduction process.

6 Evaluation Results

6.1 Methodology

Performance Simulator. We develop a cycle-accurate and functional simulator using Python to model the micro-architectural behaviors and measure execution time in the number of cycles. The base-calling models are implemented in PyTorch. The Aten library calls are fed into the simulator to produce cycle counts.

CAD Tools. We use ARM Neoverse N1 CPU as our base processor[29] and then use Cacti 7.0[30] to extend the last level cache to 32 MB⑤. We implement additional modules in Chisel3[31] and then generate Verilog. We synthesize each module using Synopsys Design Compiler with the SIMC 14 nm standard VT library to measure module area and critical path delay, and estimate the power using Synopsys PrimeTime PX. We set the processor's frequency as 3.0 GHz when considering the critical path delay. All results are scaled to 7 nm to fit the ARM Neoverse N1 CPU fabrication.

Benchmark Datasets. We use a set of fast5s from A. pittii (Ap), H. haemolyticus (Hh), S. marcescens (Smar), S. aureus (Sa), S. maltophilia (Smal), and K. pneumoniae (Ksb2, Nuh29, Inf032, Inf042) as datasets[32]. Those datasets are sequenced using MinION R9.4 flowcells.

Baselines and Platforms. Our baseline base-calling tools include SACall[7] and CATCaller[8]. The encoding phase of SACall is a neural network with a self-attention mechanism, and the decoding phase uses the original beam search and string conversion algorithms mentioned in Section 5. CATCaller reduces the computing cost of the self-attention mechanism by introducing dynamic convolutional techniques.

⑤We notice that the official LLC limit for the Arm Neoverse N1 core is only 4 MB. This paper uses the Arm Neoverse N1 core as the base core since it provides detailed power consumption and area[29]. The latest Cortex-A720 supports 32 MB of cache, proving our design's practicability at the edge.

We evaluate those baselines using the edge class platform (equipped with 12-core Arm Cortex-A78AE and NVIDIA Jetson AGX Orin) and the server class platform (equipped with 36-core Intel Xeon Gold 6354 CPU and NVIDIA A100 GPU). Table 2 shows the system configurations for the above platforms. The power consumption evaluation tools are Jetson-stats and nvidia-smi for the edge and server platforms, respectively.

6.2 Overall Results

Fig.11 compares the throughput and power consumption of Jetson GPUs, A100 GPUs, and FuHsi. To guarantee the resource utilization of the GPU can reach more than 95%, the batch size on the Jetson GPU and A100 GPU is set to 128 and 32, respectively.

6.2.1 *Throughput*

Figs.11(a) and (c) show the throughput comparison. For the SACall model, FuHsi FP16 achieves an average of 11.87x, 8.82x, 3.42x, 3.07x, and 2.12x speedup when compared with Jetson FP32, Jetson FP16, A100 FP32, A100 FP16, and FuHsi FP32, respectively. For the CATCaller model, FuHsi FP16 achieves an average of 12.68x, 8.34x, 4.73x, 4.62x, and 2.13x speedup when compared with Jetson FP32, Jetson FP16, A100 FP32, A100 FP16, and FuHsi FP32, respectively.

FuHsi also demonstrates a significant energy efficiency improvement when considering the power reduction in Figs.11(b) and (d). For the CATCaller model, FuHsi achieves 167 400 samples/s watt^{-1} for FP32 and 534 300 samples/s watt^{-1} for FP16. For the

Table 2. Architectural Specifications of Evaluated Platforms

Platform	Model	Core	Clock (GHz)	Cache (MB)	Memory
Edge GPU	Jetson AGX	2 048 Ampere	1.30	4	204.8 GB/s LPDDR5
Server GPU	NVIDIA A100	6 912 Ampere	1.41	40	1 555 GB/s HBM v2.0
FuHsi	ARM v8.2	2 Neoverse N1	3.00	32	204.8 GB/s LPDDR5

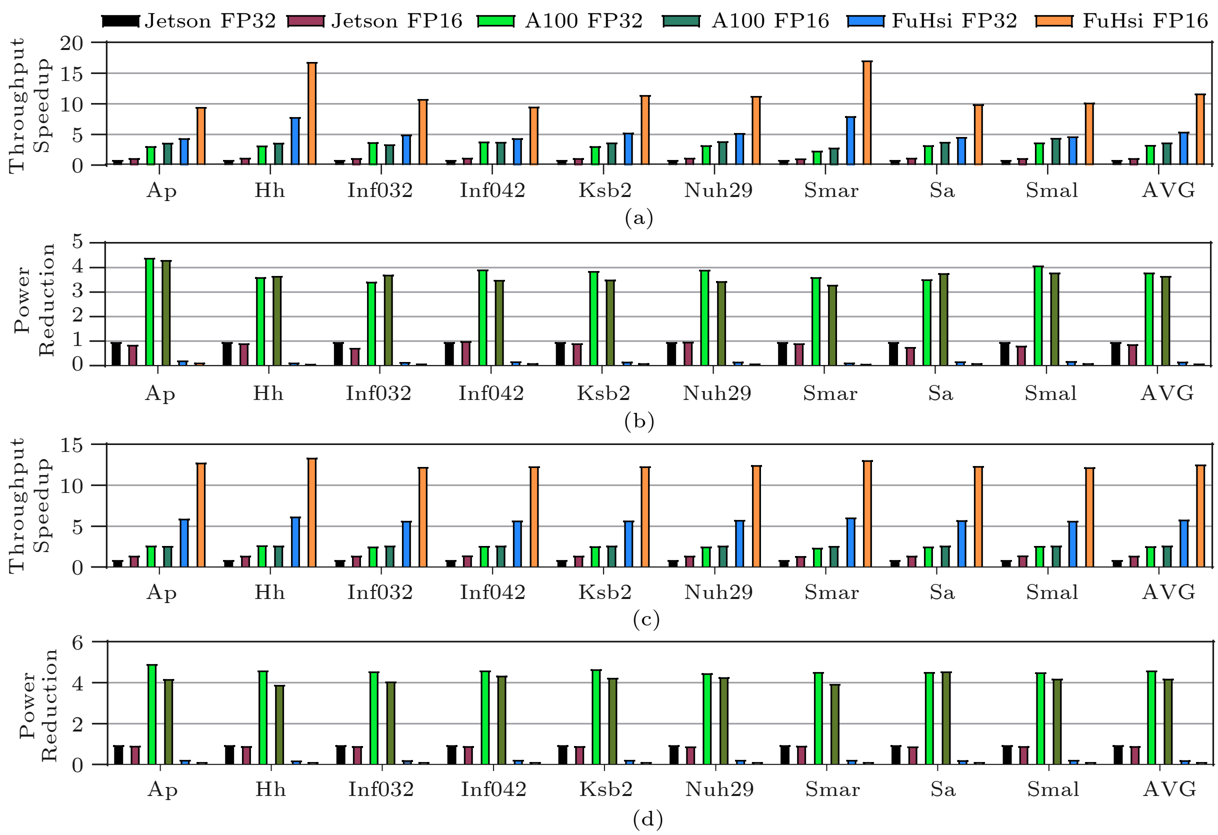


Fig.11. Performance comparison between FuHsi and benchmark platforms, i.e., NVIDIA Jetson AGX Orin GPU and NVIDIA A100 GPU. The evaluated model data types include float 32 (FP32) and float 16 (FP16). The evaluated models include CATCaller[8] and SACall[7]. All results are scaled to Jetson FP32. Higher throughput is better, and lower power reduction is better. (a) Throughput comparison of CATCaller. (b) Power consumption comparison of CATCaller. (c) Throughput comparison of SACall. (d) Power consumption comparison of SACall.

SACall model, FuHsi reaches 156 210 samples/s watt^{-1} for FP32 and 518 000 samples/s watt^{-1} for FP16. Considering a single ONT sequencer is 340 000 samples/s watt^{-1}, our design can provide base-calling requests for up to 15 ONT sequencers.

We discuss three key factors contributing to the FuHsi throughput. 1) FuHsi customizes in-cache strategies for all the time-consuming operators in base-calling, i.e., beam search, string conversion, GEMM, etc. 2) FuHsi provides massively parallel execution for compute-intensive operations. The 32 MB last-level cache provides 1 048 576 bit-serial ALUs operating at 3.0 GHz. 3) FuHsi reduces the overhead for unnecessary direct copy generated in PyTorch, which occupies 10% of the execution time in base-calling.

6.2.2 Power Consumption

Table 3 shows the power consumption and area of FuHsi. The total area of FuHsi is 171.165 mm^2, and the total power consumption is 9.598 W. The core takes up 58.9% of the area overhead, and the LLC takes up 40.7%. We add the TMU, the SRAM peripheral, and the FuHsi Controller to support base-calling operators and flow control, and the added components take up only 0.66%. Such small changes ensure the practicability of our design. Figs.11(b) and (d) compare the power consumption of FuHsi with CPUs and GPUs. For the SACall model, FuHsi FP16 achieves an average of 7.32x, 6.70x, 28.1x, 27.04x, and 1.50x reduction when compared with the Jetson FP32, the Jetson FP16, the A100 FP32, the A100 FP16, and the FuHsi FP32, respectively. For the CATCaller model, FuHsi FP16 achieves an average of 5.78x, 5.49x, 26.81x, 24.48x, and 1.55x reduction when compared with Jetson FP32, Jetson FP16, A100 FP32, A100 FP16, and FuHsi FP32, respectively.

Table 3. FuHsi Area and Power Breakdown

Component	Area (mm^2)	Power (W)	Area (%)
Core	100.825	4.914	58.900
LLC	69.600	4.240	40.700
Peripheral	0.150	0.390	0.087
TMU	0.250	0.007	0.146
Controller	0.340	0.047	0.199
Total	171.165	9.598	100.000

We also discuss two key factors that contribute to the FuHsi power reduction. 1) FuHsi uses the native CPU cache for base-calling computation, in contrast to traditional computing platforms that follow a host-device model to offload a large number of computing tasks to the device side. Taking NVIDIA Jetson as an example, CPU power consumption is only 10.8 W, while the power consumption of GPU is 32.9 W. 2) FuHsi has customized all bottleneck operators for base-calling; in contrast, GPU has customized acceleration for only a small number of base-calling operators. For instance, the usage of GPU Tensor Core in SACaller is only 24.9%.

6.2.3 Execution Latency Breakdown

Fig.12 shows the execution latency breakdown of A100 and FuHsi. 1) The A100 has a lower execution time in the scalar and weight load parts. This is because the A100 platform is equipped with a high-performance CPU (36-core Intel Xeon Gold 6354), and all cache ways are used for these operations. Specifically, the latencies of the scalar and weight load parts of A100 are 1.21x and 1.48x lower than FuHsi, respectively. 2) The A100's performance bottleneck is computing operations. In contrast, FuHsi achieves significant speedups in GEMM, element-wise, beam search, and string conversion operations.

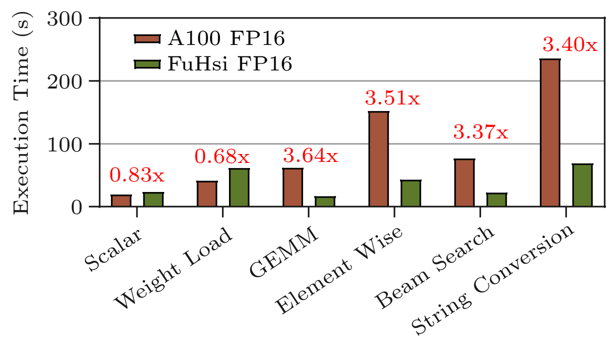


Fig.12. Execution latency breakdown of A100 and FuHsi. The batch size is 32, the model is CATCaller, the precision is FP16, and the dataset is Ap. The red text shows the speedup of FuHsi versus A100.

6.2.4 Compared with SOTA Accelerator

Helix[12] uses in-NVM (i.e., SOT-MRAM) to accelerate base-calling. We use the throughput in the original paper. Helix can achieve about 1 000 samples/s watt^{-1}. In contrast, FuHsi achieves about 100 000 samples/s watt^{-1}, which is two orders of magnitude higher than Helix's.

We discuss three reasons why in-cache is more suitable for base-calling than in-NVM (i.e., Helix). 1) NVM is computing energy efficient but has a high overhead of data replacement on the crossbar, making it suitable for computations with stationary data, such as matrix multiplication (weights stationary) and pattern matching (reference text stationary). The NVM can deliver speedups for the dot-product operation. However, beam search and string conversion are unsuitable for NVM as they continuously generate dynamic data (beams and intermediate strings) during computation. 2) In-cache makes a good trade-off between data replacement and computing efficiency[20]. The alignment-free MAC exploits the parallelism of dot-product. The in-cache beam search and in-situ string conversion exploit the high parallelism of bit-serial, and we design TMUs to support efficient data replacement for the cache. 3) As EVE[33] noted, the SOT-MRAM technology is currently immature, while the in-cache technology is easier to implement and integrate with CPUs. CPUs are also a demanded computing resource for genome sequencers and are more suitable as an offload engine for base-calling.

6.3 Optimization Analysis

6.3.1 Overall Performance Analysis

Figs.13 (a) and (b) show the incremental and ablation experiments of FuHsi, respectively.

As shown in Fig.13(a), we compare the throughput of Jetson, A100, Base, and Base plus optimizations proposed in this paper. We obtain two observations from this experiment. 1) The throughput of Base is 2.37x higher than that of Jetson but 1.20x lower than that of A100. The naive in-cache approach cannot be directly applied to the base-calling task. 2) The alignment-free MAC (noted as w/AFM), in-cache beam search (noted as w/BS), in-situ string conversion (noted as w/SC), and other optimizations (noted as w/Others) improve 1.50x, 1.10x, 1.44x, and 1.28x throughput speedup, respectively. The optimizations proposed in this paper can significantly improve the performance of the in-cache acceleration approach in base-calling tasks.

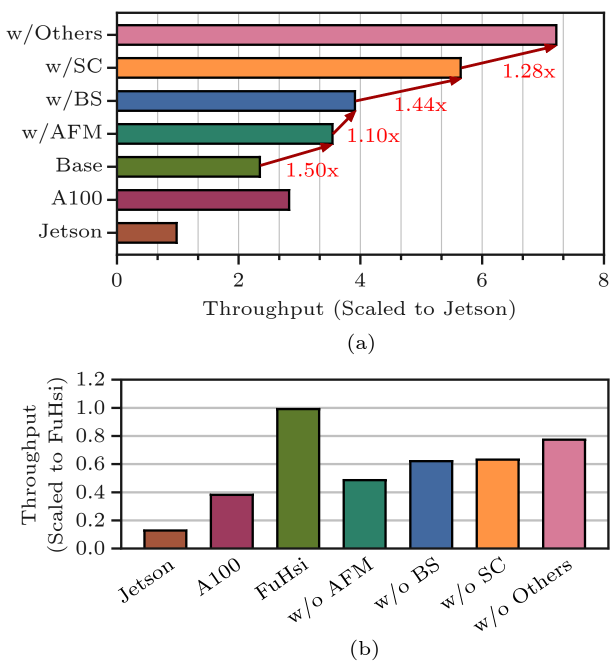


Fig.13. Performance analysis of FuHsi, where the model is CATCaller, the precision is FP16, and the dataset is Ap. The naive in-cache approach, the alignment-free MAC strategy, the in-cache beam search strategy, the in-situ string conversion strategy, and other optimizations are noted as Base, AFM, BS, SC, and Others, respectively. (a) Throughput speedups when optimizations are added individually. (b) Throughput speedups when individual optimizations are removed.

Fig.13(b) shows the system's performance without AFM, BS, SC, and Others. The throughput has 0.49x, 0.637x, 0.640x, and 0.78x reduction when compared with FuHsi, respectively. We obtain three observations from this experiment. 1) Our proposed strategies provide essential performance gain for the system, with the minor optimization contributing one-third of the system's throughput and the most significant optimization contributing more than half of the throughput. 2) Removing AFM has the most significant impact on performance since it is the first key operator of base-calling and the producer of the whole system. Even if the subsequent operators are accelerated, the system will be blocked, affecting the system throughput. 3) With just one optimization removed, the throughput of our system is higher than that of both A100 and Jetson. This demonstrates that each of our optimizations provides a considerable performance gain.

6.3.2 In-Cache Beam Search Analysis

Fig.14(a) compares the in-cache beam search used by FuHsi with the fast-ctc-decode (GPU baseline) and the work of Lu *et al.*[34] (FPGA baseline). Compared with the GPU and FPGA baseline, our design can achieve 11.63x and 3.60x throughput speedup, respectively. We also observe that the FPGA baseline and FuHsi speedup tend to decrease as the sequence length grows. When the input sequence is longer than the number of cells in the sorting block and the number of rows in the cache block, this introduces multiple rounds of iteration and results in longer latency.

Fig.14(b) also demonstrates the effect of the tolerance factor on the in-cache beam search performance. The optimal performance is achieved when the tolerance factor is 1. When the tolerance factor is set to 0, the comparison of sub-level exponent bits is generated to obtain the exact number of beams, and the per-

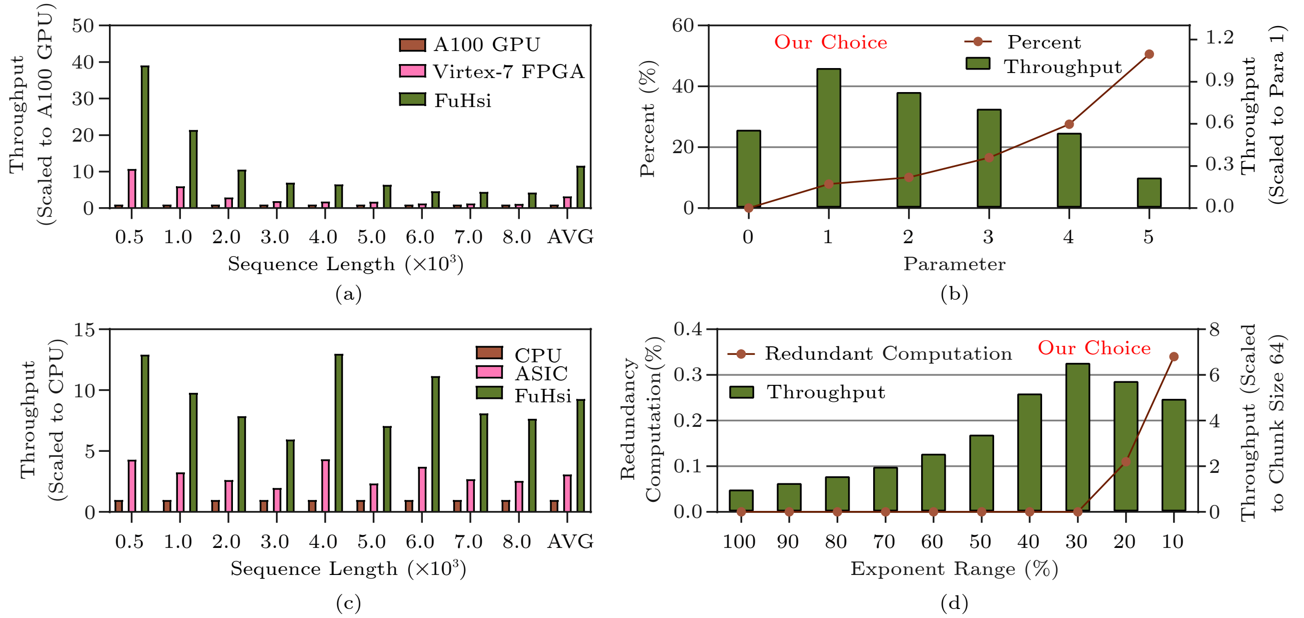


Fig.14. (a) Performance comparison of in-cache beam search with A100 GPU and FPGA baseline. (b) Impact of tolerance parameters of in-cache beam search's performance. (c) Performance comparison of in-situ string conversion with CPU and ASIC baseline. (d) Impact of exponent range on alignment-free MAC's performance. For these four experiments, the model is CATCaller, the precision is FP16, and the dataset is Ap.

formance is only 56% of the optimal performance. When the tolerance factor is too large (e.g., 5), a large number of irrelevant beams are generated that need to be computed, and the system performance is only 22% of the optimal performance.

6.3.3 *In-Situ String Conversion Analysis*

Fig.14(c) compares the in-situ string conversion with CATCaller (CPU baseline) and the ASIC implementation[⑥]. Compared with the 72-thread CPU and the ASIC baseline, our design can achieve 9.29x and 3.01x throughput speedup, respectively. Our method significantly improves the throughput of string conversion thanks to the in-cache, which provides much higher parallelism and no data movement than CPU and ASIC. In addition, since the computation of different bases of string conversion is uncorrelated, our design has stable throughput over different input lengths.

6.3.4 *Alignment-Free MAC Analysis*

Fig.14(d) analyzes the impact of different ranges of exponent of MAC, with a naive in-cache approach at 100% and 30% for the configuration used by FuHsi. The throughput of MAC increases from 1.0x to 6.27x when decreasing the exponent range, proving our strategy's effectiveness. The system's performance gradually increases as the exponent range decreases from 100% to 30%. The optimal performance is achieved with an exponent range of 30%. A range of exponent below 30% leads to a slight performance degradation. To guarantee accuracy, a search range that is too low will incur a large amount of search time for missing exponent bits.

6.4 Design Space Exploration

Sequence Length. Fig.15(a) evaluates the throughput with different input sequence lengths, from which we derive two observations. 1) Comparing the throughput when the sequence length is 64, the throughput with different input lengths ranges from 1.0x to 1.72x. This demonstrates that FuHsi has stable performance over a wide range of sequence lengths. 2) FuHsi achieves the highest throughput at an input sequence of 1 024, and then the throughput decreases as the input sequence becomes longer (1.43x at sequence length 8 192). This is because the performance of in-cache beam search decreases as the input sequence is longer.

[⑥]The CPU platform is Intel Xeon® Gold 6354 CPU @ 3.0 GHz, and we use 72 threads to run the CPU baseline. The ASIC implementation contains an input buffer, an output buffer, and a string conversion array. For a fair comparison, the area of the string conversion array is set to match the additional area of FuHsi, and the clock frequency is 3.0 GHz.

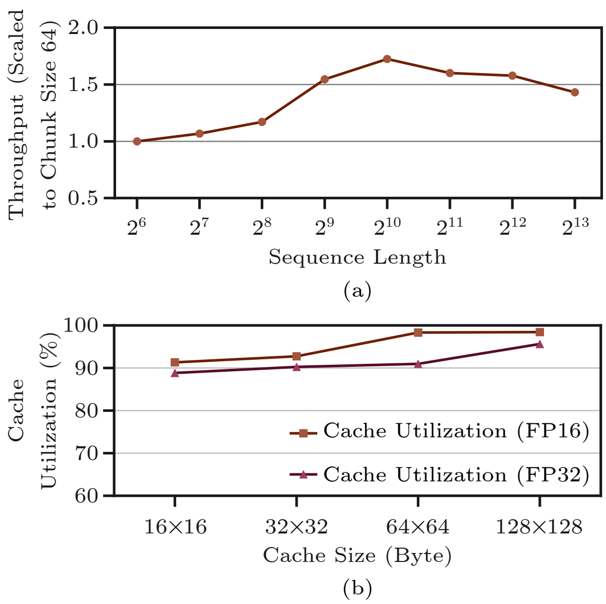


Fig.15. Design space exploration of FuHsi. (a) Different sequence lengths. (b) Cache sizes. The model is CATCaller, the precision is FP16, and the dataset is Ap.

Cache Size. Fig.15(b) demonstrates the effect of different cache block sizes on cache utilization, and we obtain three observations. 1) The average cache utilization of FuHsi is 95.22% and 91.45% for FP16 and FP32, respectively. For cache rows, the algorithm-specific mapping proposed by FuHsi allows the data to be regularly placed in the cache for in-situ computation. For the cache column, the large-scale matrix computations (128×128 and 256×256) dominate base-calling, constituting 90% or more of the operations. 2) Along with the increase in cache size, the cache utilization increases from 91.34% to 98.45% for FP16 and 88.86% to 95.66% for FP32. A large cache size produces fewer fragments that cannot be allocated to further tasks. 3) The cache utilization of FP16 is 4% higher than that of FP32 since FP32 requires a larger cache row for a single task, which generates more cache fragments with a fixed cache size.

7 Related Work

Base-Calling Accelerators. In recent years, there has also been some acceleration work for Nanopore base-calling, such as Helix[12], GenPIP[13], and SquiggleFilter[35]. Helix and GenPIP are based on SOT-MRAM, mainly for dot-product and CTC decoding. Since NVM has a high data replacement overhead, they are unsuitable for the acceleration of operators in base-calling that requires a lot of data replacement (i.e., beam search and string conversion). Their performance is two orders of magnitude lower than the sequencer throughput. SquiggleFilter accelerates the ONT Read-Until process using a modified sub-sequence Dynamic Time Warping architecture, which can handle up to 114 sequencer requests. However, it only applies to viral sequences and does not directly accelerate base-calling, making it a bottleneck in genome analysis. In this paper, FuHsi has comprehensively accelerated base-calling, which can afford the sequencing throughput of 15 sequencers.

AI Accelerators. Plenty of work has been proposed to accelerate image processing, natural language processing, and other areas[36–38]. ProSE[38] is an accelerator engine for accelerating protein language models. Since proteins have the property of long tokens, the workload characteristics are similar to base-calling. ProSE solves the problem of long tokens by introducing a heterogeneous GEMM systolic array. However, since ProSE is purely a neural network acceleration engine and does not accelerate the other base-calling bottleneck operators (e.g., beam search and string conversion), it cannot be used to accelerate the base-calling process. In this paper, FuHsi optimizes GEMM under in-cache architecture and accelerates the particular operator for the base-calling task.

8 Conclusions

In this paper, we proposed FuHsi to improve the end-to-end throughput of the base-calling process. We introduced the FuHsi Cache, which offloads bottleneck operations to cache arithmetic. For the arithmetic aspect, we proposed in-cache beam search, string conversion, and MAC using algorithm/hardware co-design. FuHsi also includes APIs and controller design to control the FuHsi Cache. Experimental results showed that FuHsi can achieve an impressive speedup of 45.7x, 113.1x, and 100x throughput per watt compared with an NVIDIA Jetson baseline, an NVIDIA A100 GPU, and the Helix accelerator, respectively. FuHsi can simultaneously handle base-calling requests for up to 15 ONT sequencers.

Acknowledgements We thank Han Bao, Zhi-Heng Lin, Wan-Qi Liu, and Rui-Hao Gao from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, for their valuable discussions.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Schadt E E, Turner S, Kasarskis A. A window into third-

generation sequencing. *Human Molecular Genetics*, 2010, 19(R2): R227–R240. DOI: 10.1093/hmg/ddq416.

[2] Jain M, Olsen H E, Paten B, Akeson M. The Oxford Nanopore MinION: Delivery of Nanopore sequencing to the genomics community. *Genome Biology*, 2016, 17(1): Article No. 239. DOI: 10.1186/s13059-016-1103-0.

[3] Deamer D, Akeson M, Branton D. Three decades of nanopore sequencing. *Nature Biotechnology*, 2016, 34(5): 518–524. DOI: 10.1038/nbt.3423.

[4] Hoenen T, Groseth A, Rosenke K *et al*. Nanopore sequencing as a rapidly deployable Ebola outbreak tool. *Emerging Infectious Diseases*, 2016, 22(2): 331–334. DOI: 10.3201/eid2202.151796.

[5] Faria N R, Sabino E C, Nunes M R T, Alcantara L C J, Loman N J, Pybus O G. Mobile real-time surveillance of Zika virus in Brazil. *Genome Medicine*, 2016, 8(1): Article No. 97. DOI: 10.1186/s13073-016-0356-2.

[6] Lu R, Zhao X, Li J *et al*. Genomic characterisation and epidemiology of 2019 novel coronavirus: Implications for virus origins and receptor binding. *The Lancet*, 2020, 395(10224): 565–574. DOI: 10.1016/S0140-6736(20)30251-8.

[7] Huang N, Nie F, Ni P, Luo F, Wang J. An attention-based neural network basecaller for Oxford Nanopore sequencing data. In *Proc. the 2019 IEEE International Conference on Bioinformatics and Biomedicine*, Nov. 2019, pp.390–394. DOI: 10.1109/BIBM47256.2019.8983231.

[8] Lv X, Chen Z, Lu Y, Yang Y. An end-to-end Oxford Nanopore basecaller using convolution-augmented transformer. In *Proc. the 2020 IEEE International Conference on Bioinformatics and Biomedicine*, Dec. 2020, pp.337–342. DOI: 10.1109/BIBM49941.2020.9313290.

[9] Zeng J, Cai H, Peng H, Wang H, Zhang Y, Akutsu T. Causalcall: Nanopore basecalling using a temporal convolutional network. *Frontiers in Genetics*, 2020, 10:1332. DOI: 10.3389/fgene.2019.01332.

[10] Xu Z, Mai Y, Liu D *et al*. Fast-bonito: A faster deep learning based basecaller for Nanopore sequencing. *Artificial Intelligence in the Life Sciences*, 2021, 1:100011. DOI: 10.1016/j.ailsci.2021.100011.

[11] Silvestre-Ryan J, Holmes I. Pair consensus decoding improves accuracy of neural network basecallers for Nanopore sequencing. *Genome Biology*, 2021, 22(1): Article No. 38. DOI: 10.1186/s13059-020-02255-1.

[12] Lou Q, Janga S C, Jiang L. Helix: Algorithm/architecture co-design for accelerating Nanopore genome basecalling. In *Proc. the ACM International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2020, pp.293–304. DOI: 10.1145/3410463.3414626.

[13] Mao H, Alser M, Sadrosadati M, Firtina C, Baranwal A, Cali D S, Manglik A, Alserr N A, Mutlu O. GenPIP: In-memory acceleration of genome analysis via tight integration of basecalling and read mapping. In *Proc. the 55th IEEE/ACM International Symposium on Microarchitecture*, Oct. 2022, pp.710–726. DOI: 10.1109/MICRO56248.2022.00056.

[14] Shahroodi T, Singh G, Zahedi M, Mao H, Lindegger J, Firtina C, Wong S, Mutlu O, Hamdioui S. Swordfish: A framework for evaluating deep neural network-based basecalling using computation-in-memory with non-ideal memristors. In *Proc. the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 28–Nov. 1, 2023, pp.1437–1452. DOI: 10.1145/3613424.3614252.

[15] Fujiki D, Mahlke S, Das R. Duality cache for data parallel acceleration. In *Proc. the 46th International Symposium on Computer Architecture*, Jun. 2019, pp.397–410. DOI: 10.1145/3307650.3322257.

[16] Aga S, Jeloka S, Subramaniyan A, Narayanasamy S, Blaauw D, Das R. Compute caches. In *Proc. the 2017 IEEE International Symposium on High Performance Computer Architecture*, Feb. 2017, pp.481–492. DOI: 10.1109/HPCA.2017.21.

[17] Hwang K, Sung W. Character-level incremental speech recognition with recurrent neural networks. In *Proc. the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing*, Mar. 2016, pp.5335–5339. DOI: 10.1109/ICASSP.2016.7472696.

[18] Eckert C, Wang X, Wang J, Subramaniyan A, Iyer R, Sylvester D, Blaaauw D, Das R. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proc. the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, Jun. 2018, pp.383–396. DOI: 10.1109/ISCA.2018.00040.

[19] Wang X, Yu J, Augustine C, Iyer R, Das R. Bit prudent in-cache acceleration of deep convolutional neural networks. In *Proc. the 2019 IEEE International Symposium on High Performance Computer Architecture*, Feb. 2019, pp.81–93. DOI: 10.1109/HPCA.2019.00029.

[20] Fujiki D, Khadem A, Mahlke S, Das R. Multi-layer in-memory processing. In *Proc. the 55th IEEE/ACM International Symposium on Microarchitecture*, Oct. 2022, pp.920–936. DOI: 10.1109/MICRO56248.2022.00068.

[21] Jeloka S, Akesh N B, Sylvester D, Blaauw D. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 2016, 51(4): 1009–1021. DOI: 10.1109/JSSC.2016.2515510.

[22] Huangfu W, Li X, Li S, Hu X, Gu P, Xie Y. MEDAL: Scalable DIMM based near data processing accelerator for DNA seeding algorithm. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.587–599. DOI: 10.1145/3352460.3358329.

[23] Talati N, Ye H, Yang Y, Belayneh L, Chen K Y, Blaauw D, Mudge T, Dreslinski R. NDMiner: Accelerating graph pattern mining using near data processing. In *Proc. the 49th Annual International Symposium on Computer Architecture*, Jun. 2022, pp.146–159. DOI: 10.1145/3470496.3527437.

[24] Subramaniyan A, Wang J, Balasubramanian E R M, Blaauw D, Sylvester D, Das R. Cache automaton. In *Proc. the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017, pp.259–272. DOI: 10.1145/3123939.3123986.

[25] Salamat S, Haj Aboutalebi A, Khaleghi B, Lee J H, Ki Y S, Rosing T. NASCENT: Near-storage acceleration of database sort on SmartSSD. In *Proc. the 2021 ACM/SIG-

DA International Symposium on Field-Programmable Gate Arrays, Feb. 2021, pp.262–272. DOI: 10.1145/3431920.3439298.

[26] Qiao W, Oh J, Guo L, Chang M C F, Cong J. FANS: FPGA-accelerated near-storage sorting. In *Proc. the 29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2021, 106–114. DOI: 10.1109/FCCM51124.2021.00020.

[27] Jain M, Koren S, Miga K H *et al*. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 2018, 36(4): 338–345. DOI: 10.1038/nbt.4060.

[28] Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R, 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 2009, 25(16): 2078–2079. DOI: 10.1093/bioinformatics/btp352.

[29] Christy R, Riches S, Kottekkat S, Gopinath P, Sawant K, Kona A, Harrison R. 8.3 a 3ghz Arm neoverse N1 CPU in 7nm FinFET for infrastructure applications. In *Proc. the 2020 IEEE International Solid-State Circuits Conference*, Feb. 2020, pp.148–150. DOI: 10.1109/ISSCC19947.2020.9062889.

[30] Balasubramonian R, Kahng A B, Muralimanohar N, Shafiee A, Srinivas V. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Architecture and Code Optimization (TACO)*, 2017, 14(2): Article No. 14. DOI: 10.1145/3085572.

[31] Bachrach J, Vo H, Richards B, Lee Y, Waterman A, Avižienis R, Wawrzynek J, Asanović K. Chisel: Constructing hardware in a Scala embedded language. In *Proc. the 49th Annual Design Automation Conference*, Jun. 2012, pp.1216–1225. DOI: 10.1145/2228360.2228584.

[32] Wick R R, Judd L M, Holt K E. Performance of neural network basecalling tools for Oxford Nanopore sequencing. *Genome Biology*, 2019, 20(1): Article No. 129. DOI: 10.1186/s13059-019-1727-y.

[33] Al-Hawaj K, Ta T, Cebry N, Agwa S, Afuye O, Hall E, Golden C, Apsel A B, Batten C. EVE: Ephemeral vector engines. In *Proc. the 2023 IEEE International Symposium on High-Performance Computer Architecture*, Feb. 25–Mar. 1, 2023, pp.691–704. DOI: 10.1109/HPCA56546.2023.10071074.

[34] Lu S, Lu J, Lin J, Wang Z, Du L. A low-latency and low-complexity hardware architecture for ctc beam search decoding. In *Proc. the 2019 IEEE International Workshop on Signal Processing Systems*, 2019, pp.352–357. DOI: 10.1109/SiPS47522.2019.9020324.

[35] Dunn T, Sadasivan H, Wadden J, Goliya K, Chen K Y, Blaauw D, Das R, Narayanasamy S. SquiggleFilter: An accelerator for portable virus detection. In *Proc. the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2021, pp.535–549. DOI: 10.1145/3466752.3480117.

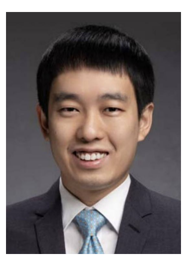
[36] Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, Temam O. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb. 2014, pp.269–284. DOI: 10.1145/2541940.2541967.

[37] Chen Y, Luo T, Liu S, Zhang S, He L, Wang J, Li L, Chen T, Xu Z, Sun N, Temam O. DaDianNao: A machine-learning supercomputer. In *Proc. the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp.609–622. DOI: 10.1109/MICRO.2014.58.

[38] Robson E, Xu C, Wills L W. ProSE: The architecture and design of a protein discovery engine. In *Proc. the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb. 2022, pp.655–668. DOI: 10.1145/3503222.3507722.

Ye-Wen Li received his B.S. degree in computer science from Minzu University of China, Beijing, in 2019. He is currently a Ph.D. candidate at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests encompass computer architecture and computational biology, with a particular focus on developing algorithms and domain-specific accelerators that enable faster and cheaper progress in biology and medicine.

Guang-Ming Tan received his B.S. degree in mathematics from Xiangtan University, Xiangtan, in 2002, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2008. He is a professor at ICT, CAS, Beijing. His research interests include parallel computing, domain-specific architecture, and big data.



Xue-Qi Li received his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, 2021. He is currently an associate professor at ICT, CAS, Beijing. His research interests include domain-specific architecture, processing in/near memory architecture, and algorithm-architecture-technology cross-layer optimization.