APPENDIX

A. Reconfigurability

Invoking a logic gate within the SpinPM array translates into pre-setting the output, connecting all cells participating in computation to *LL* by setting the corresponding *WL*s, and setting a voltage between *BSL*s of the inputs and output cells that equals to V_{gate}, which depends on the type of the logic gate. Therefore, modulo output pre-set, the complexity of reconfiguration is very similar to the complexity of addressing in the memory array. SpinPM is reconfigurable along two dimensions:

- Each cell can serve as an input or as an output for a logic gate depending on the computational demands of the workload within the course of execution.
- For a fixed input-output assignment, the logic function itself is reprogrammable. For example, we can reconfigure the gate from Fig.1(b)/(c) to implement another function than NOR by simply changing V_{gate}, to, e.g., V_{NAND} (and applying a different output pre-set, as need be).

By default, SpinPM acts as an MRAM array. A dedicated architecturally visible set of registers keep the configuration bits to program SpinPM cells as logic gate input/outputs. These configuration bits capture not only the physical location in the array, but also whether the cell represents an input or an output, the pre-set value for the output, and V_{gate}. A fixed or floating portion of the SpinPM array can keep these configuration bits as part of the machine state, as well.

B. Spatio-Temporal Scheduling

The goal of classic memory data layout optimizations is to perform as many computations as possible per unit data delivered from the memory to the processor, as the data communication between the processor and the memory represents the bottleneck. SpinPM, on the other hand, brings compute capability to the data to be processed. The goal becomes *minimizing the direct physical distance between the cells participating in computation*. Considering that an output cell can serve as an input cell in subsequent steps of computation, the physical location of the cells carrying the input data for subsequent steps can dynamically change as computation proceeds.

This optimization problem gives rise to two strongly correlated sub-problems: the layout of data to be processed in the memory array, and the spatio-temporal scheduling of computations within the array. In this regard, the optimization problem has many analogies to floor-planning and placement algorithms deployed in the computer aided design of digital systems, which aim to minimize the "distance" (in terms of wire length) between interconnected circuit blocks. In SpinPM context, "interconnected blocks" translate into interconnected cells (over *LL*) participating in computation (Section II-A). We will look closer into this effect in Section C.

SpinPM hence features a unique trade-off between data replication and parallelism: Due to the internal array structure, (unless replicated), the same cell can only participate in one computational step at a time, which may impair further opportunities for parallel execution. Data replication can unlock more parallelism in such cases, at the expense of a larger memory footprint.

As an example, Figure 7 illustrates a time lapse of logic operations on two datasets, each 3-bit in length, in a SpinPM array of four columns (rotated in horizontal direction for ease of illustration). Time *T0* captures the initial state. *T1-T4* is spent on preset operations of output cells in all four rows, conservatively through standard writes of one row at a time, before a sequence of bitwise ORs and MAJ3s take place (on each column in parallel). Time T5 shows the OR operation on the first bits of the datasets, in each column at the same time (bold bit values are involved in computation). *T6* and *T7* do the same for next two bits of the datasets. The last stage of computation, T8, performs MAJ3 on the three bit result of the previous sequence of ORs. Last step, *R*, highlights the final result bits in each column.

C. Practical Considerations

Array Size: The maximum column height (i.e., the maximum number of rows) per SpinPM array depends on the gate voltage V_{gate} (Section II-A), the interconnect material for LL and BSL (which connects the input and output cells together in forming a gate), as well as the technology node. We conduct the following experiment to determine the maximum column height: We consider a two-input, one output SpinPM gate which has the input cells and the output cell located in adjacent columns. In each experiment, we shift the output cell further away from the input cells, by one cell at a time. The process continues until we reach the terminating condition, which is when the current through the output cell falls below the required critical switching current for the most conservative input cell resistance states.

Assuming copper interconnect segments of 160nm for LL, for representative SpinPM gates used in pattern matching, this analysis renders approximately 2K cells per column at 22nm, where the latency overhead induced by this maximum distance computation barely reaches $< 1\%$ of the switching time of the SHE-MTJ as detailed in Section IV).

The feasibility of array dimensions is contingent upon the correct functionality of the array at subarray granularity. Our circuit-level analysis reveals a maximum subarray size of 512×512 bits, without sacrificing the reliability of array functionality.

Since SpinPM cells use two transistors, the area of each cell is dominated by the transistors. The area of each SpinPM cell, at 22nm technology node, is roughly $100F^2$ considering the current density requirement of MTJ devices which governs the size of the cell (MTJs are placed on top of transistors, and roughly consume $\sim 5\%$ of the transistor area).

Array Periphery: Peripheral overheads, mainly induced by addressing and control operations, can play a vital role in determining the pattern matching throughput. Accordingly, throughout the evaluation, we consider the time and energy overhead of peripheral circuitry including row and column decoders, multiplexers, and sense amplifiers. For memory read and write operations a SpinPM array is not quite different than a standard STT-MRAM array, hence we model periphery after the standard STT-MRAM. During computation, however, as all columns operate in parallel, column decoder overhead does not apply (which we conservatively keep). The periphery during computation rather becomes similar to the periphery of Pinatubo [34], an alternative spintronic PIM substrate (although SpinPM computation relies on a different mechanism, totally excluding sense amplifier involvement during computation contrary to Pinatubo). Even during computation where all columns are active, the current draw in an SpinPM array remains relatively modest. For example, using specifics for SHE-MTJ devices (as detailed in Section IV), a 128MB array would still consume considerably less current than a DDR3 SDRAM write operation [42].

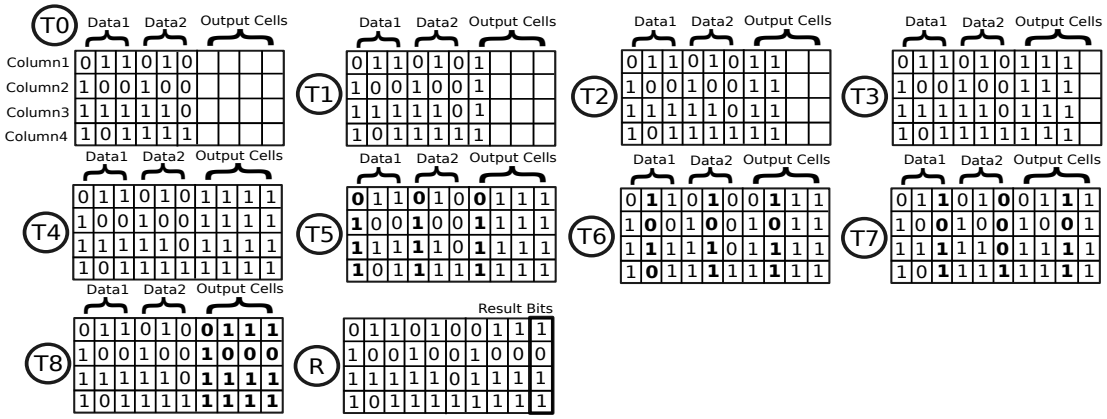


Fig. 7: Timing diagram of example logic execution.

Preset Overhead: Each logic operation requires the output to be (pre)set to a predefined value. Computation is column parallel, i.e., in all columns, the output cell resides in the very same row. Accordingly, before firing column-parallel computation, the corresponding row where the output cells reside should be preset. To this end, we can use a "gang" preset, which presets all cells in the output row(s) simultaneously. The alternative is relying on the standard write operation, which can preset (columns in) one row at a time. The gang preset is equivalent to a parallel COPY operation – where all columns compute in parallel and where the output cells are all in the respective row subject to gang preset. Hence, the discussion about the periphery overhead during column-parallel computation directly applies here, and the current draw remains to be modest. For standard write based preset as well, the current draw is similar: As one gate can be actively computing in a column at a time, only one cell needs to be preset per column, and all rows are preset one after another.

Read Disturbance: Read disturbance is an issue that arises when read current and write current become similar due to non-linear technology scaling of different electrical components of an array. SHE-MTJ devices have separate read and write paths through the device, eliminating read disturbance issue.

D. System Interface

SpinPM can serve as a stand-alone compute engine or a co-processor attached to a host processor. Following the near-memory processing taxonomy from [2], due to the reconfigurability (Section A), both SpinPM design points still fall into the "programmable" class. A classic system has to specify how to offload both *computation and data* to the co-processor, and how to get the results back from the co-processor. For a SpinPM co-processor, we do not need to communicate data values – instead, the SpinPM array requires (ranges of) data addresses to identify the data to process, and the specification for computation, i.e., which function to perform on the corresponding data.
We will next cover the SpinPM system stack to support in-memory execution semantics for pattern matching.

SpinPM Instructions: In addition to conventional memory read and write, SpinPM instructions cover computational building blocks for in-memory pattern matching. SpinPM instructions hence form two classes: data transfer (read, write) and computational (arithmetic/logic). By construction, computational SpinPM instructions are *block* instructions: two dimensional vector instructions, which operate *on all columns* and *on a subset of rows* of an SpinPM array at a time. Hence, key operands for any computational SpinPM instruction are the row numbers of the source(s) (i.e., input(s) to computation) and destination(s) (i.e., output(s) to computation). Depending on the size of the pattern matching problem, multiple SpinPM arrays may be deployed in parallel. Therefore, the computational subset of SpinPM instructions facilitates gang-execution on all SpinPM arrays, as well. In the following, we will generically use the term SpinPM *substrate* to refer to all arrays participating in computation. We also make the distinction between *macro-* and *micro*-instructions. The set of micro-instructions covers actual bit-level operations performed in the SpinPM substrate, while the set of macro-instructions forms the high-level programming interface.

Programming Interface: To match SpinPM's column-level parallelism, memory allocation and declaration of variables (which represent inputs and outputs to computation) happen at column granularity. Depending on the problem, a variable may cover the entire column or only a portion. The following code snippet provides an example, where an integer variable `x` gets written (assigned) to row `r` and column `c` in a SpinPM array (line 5):

```
1  int x = ...;
2  ...
3  int y;
4  preset(r, ncell, val);
5  int_pm x_pm = write_pm(x, r, c, sizeof(x));
6  y = readdir_pm(x_pm);
```

In this case, besides `x` and `y`, `ncell`, `val`, `c` and `r` represent (already defined) integer values. The SpinPM-specific (composite) data type $\texttt{int}_{\texttt{pm}}$ captures row and column coordinates for each variable stored in the array. $\texttt{x}_{\texttt{pm}}$ in line 5 keeps this information for variable `x`, after it gets written to column `c`, from row `r` onwards, by the $\texttt{write}_{\texttt{pm}}$ function. The subsequent read in line 6, conducted by the $\texttt{readdir}_{\texttt{pm}}$, directly assigns the value of `x` to `y`. SpinPM also features a read function, $\texttt{read}_{\texttt{pm}}$, which has a similar interface to $\texttt{write}_{\texttt{pm}}$ with explicit row and column specification. We consider each such function as a macro-instruction.

The `preset` function in line 4 presets `ncell` number of (consecutive) cells, starting from row `r`, each to value `val`. SpinPM features different variants of this function, including one to gang-preset the entire scratch area (Fig. 3), and another where `val` is interpreted as a bitmask (of `ncell` bits) rather than a single-bit preset value which applies over the entire range of the specification.

Each pattern matching problem to be mapped to SpinPM features three basic stages:

(i) Allocating and initializing the reference, pattern, and scratch regions in each array (Fig. 3);
(ii) Computation;
(iii) Collecting the pattern matching outcome.

Variants of `preset` and $\texttt{write}_{\texttt{pm}}$ functions cover stage (i); and variants of $\texttt{read(dir)}_{\texttt{pm}}$, stage (iii). Stage (ii) can take different forms depending on the encoding of pattern and reference characters, but generally primitives such as $\texttt{add}_{\texttt{pm}}\texttt{(int start, int end, int}_{\texttt{pm}}\texttt{ result)}$ apply, which sums all cell contents between rows `start` and `end`, on a per column basis, and writes the result back where `result` points. $\texttt{add}_{\texttt{pm}}$ macro-instruction can directly implement Phase-2 from Algorithm1 to calculate the bit-count on the match string (Section III-B).

To better facilitate flexibility of development, both gate-level and function-level macros are available to the developers. The following code excerpt implements DNA sequence pre-alignment application on SpinPM.

Step-1: For all reads, do:
$xpm[i] = writepm(spatterns[i], i, sizeof(spatterns[i]))$
//write the search patterns to SpinPM. *i* in *writepm* denotes column index.
Step-2: (Preset) For match string bits, do: $ms[i] = 0$ or 1 //dependent on logic function
Step-3: (Preset) For reduction adder output bits, do: $adder\_out[i] = 0$ or 1 //dependent on logic function
Step-4: (Preset) For scratch bits, do: $sb[i] = 0$ or 1 //dependent on logic function
Step-5: For all bits in reference and search patterns in this $row\_index = i$, do:
$xor(xpm[i], xref[i], sb[i])$ //XOR on first bits of base character, store output in scratch space
$xor(xpm[i+1], xref[i+1], sb[i+1])$ //XOR on second bits of base character, store output in scratch space
$nor(sb[i], sb[i+1], ms[i])$ // NOR operation on scratch output bits, store in match string
Step-5a: Update row_index
Step-6: $reduction(ms, adder\_out)$ //Reduction on match string bits (ms) to store the output in $adder\_out$
Step-6a: Preset scratch bits if required: $sb[i] = 0$ or 1 //dependent on preset algorithm
Step-7: $read(adder\_out)$
Step-7a: Go back to *Step-2* for next location search.

Code Generation: Code generation simply entails translating a sequence of macro-instructions to a sequence of micro-instructions for the SpinPM memory controller (SMC) to drive the in-place computation. Micro-instructions specify the type of operation and the rows to connect as inputs and outputs. For example, $\texttt{nand}(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j}, \texttt{r}_\texttt{k})$ specifies row $\texttt{r}_\texttt{i}$ as the output and row $\texttt{r}_\texttt{j}$ and $\texttt{r}_\texttt{k}$ as inputs to form a NAND gate in the SpinPM array. The macro-instruction $\texttt{nand}_{\texttt{pm}}$, on the other hand, performs the very same operation on multi-bit operands (of width `ncell`): $\texttt{nand}_{\texttt{pm}}(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j}, \texttt{r}_\texttt{k}, \texttt{ncell})$. In this case, $\texttt{r}_\texttt{i}$, $\texttt{r}_\texttt{j}$, and $\texttt{r}_\texttt{k}$ still demarcate the starting rows for the source and destination (`ncell` bit) operands. $\texttt{nand}_{\texttt{pm}}$ hence translates into a sequence of `ncell` number of `nand` micro-instructions. For $\texttt{add}_{\texttt{pm}}$ type of macro-instructions, on the other hand, a spatio-temporal scheduling pass (Section B) determines the corresponding composition of micro-instructions. The goal is to maximize the throughput performance for the given data layout. This usually translates into masking the overhead of presets or other types of writes (per row) by coalescing when possible. By construction, variants of `preset` macro-instruction trigger a sequence of memory writes (as many as the number of rows), as at most one row can be written at a time.

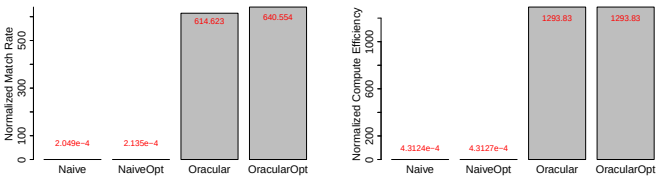
SpinPM Memory Controller (SMC): SMC orchestrates computation in the SpinPM substrate, and the communication with the host processor. SpinPM features an internal clock. During computation, SMC allocates each micro-instruction a specific number of cycles to finish depending on the operation and operand widths. This time window includes peripheral overheads and the scheduling overhead due to SMC, besides computation. After the allocated time elapses (and unless an exception is the case), SMC fetches the next set of micro-instructions. SMC features an instruction cache where micro-instructions reside until they are issued to the SpinPM substrate. Before issue, SMC decodes the micro-instructions using a look-up table to initiate preset, and subsequently, to set the appropriate voltage level on input and output BSL (as a function of the operation, as explained in Section II-B), before activating the corresponding rows in the specified arrays for computation. The look-up table keeps the voltage level and the preset value for each bit-level operation from Section II-B, which form a SpinPM micro-instruction. No look-up table access is necessary for read and write operations.

E. A Closer Look into Performance and Energy

We will next provide a detailed throughput performance and energy characterization, along with a sensitivity analysis, using *DNA* as a case study. SHE-MTJ is considered as the cell technology used in SpinPM. To characterize SpinPM based DNA sequence pre-alignment, we use a NVidia Tesla K20X GPU based implementation of the common Burrows-Wheeler Aligner (BWA) algorithm [43]. We deploy the very same reference and input pattern pool for the GPU and SpinPM. In order for the comparison to be fair, we only take the basic pattern matching portion of the GPU baseline into consideration (Section III). Specifically, we will consider two design points, which differ in how the patterns (from the input pattern pool) get assigned to columns for matching. In other words, how patterns are *scheduled* for computation in the SpinPM array: The first one is a *Naive* implementation, where we take one pattern and blindly copy it to every column of all arrays to perform similarity search. The second implementation, on the other hand, features *Oracular* pattern scheduling, which can avoid assigning a pattern to a column where a too dissimilar (reference) fragment resides. *Oracular* is straight-forward to implement by adding a search-space pruning step before full-fledged mapping takes place, as explained in Section III-B, by e.g., using hash-based filtering [44]. We will leave exploration of this rich design space to future work, but cover the overhead of a representative practical implementation in the following. Any practical SpinPM implementation would fall somewhere in the spectrum between these two extremes.

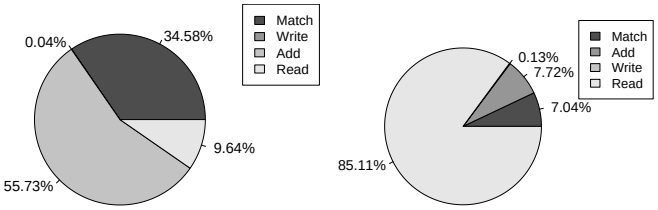
Naive Design ($Naive$): The caveat here is the very high overhead of redundant computation, due to processing one pattern at a time and mapping each such pattern naively to all reference fragments. As a single pattern is matched to the entire reference, across all arrays, at a time, the apparent serialization hurts the throughput, in terms of the number of patterns matched per second. per second as *match rate*.

Oracular Pattern Scheduling ($Oracular$): The oracular scheduler resides between the input pattern pool and SpinPM, and controls to which column in which array each pattern goes. $Oracular$ may still feed a given pattern to multiple columns, in multiple arrays, however, does not consider columns which



(a) Match Rate (patt/sec) (b) Compute Eff. (patt/sec/mW)

Fig. 8: Performance and Energy Characterization.



(a) Energy (b) Latency

Fig. 9: Breakdown of energy and latency in computation.

carry a too dissimilar (reference) fragment. In other words, $Oracular$ directs patterns to columns and arrays in a way such that achieving a high similarity score becomes more likely. While $Oracular$ bases its pattern scheduling decisions on perfect information, a practical implementation of this idea would incur the overhead of gathering this information, i.e., extracting a schedule to keep pattern matching confined to columns where a high similarity score is more likely. In any case such smart scheduling of patterns benefits the throughput performance by reducing redundant computation which eats from the energy budget.

However, since all columns in a SpinPM-SHE array perform pattern matching (in lock-step but) in parallel, before computation begins, we require that all columns have their patterns ready. Scheduling patterns takes time, which might further affect the throughput performance of SpinPM, if we let the array sit idly, waiting for scheduling decisions to take place. We can mask this overhead, as drawing pattern scheduling decisions for all the columns in an array takes less time than writing patterns in the columns of that array. This, in effect, would not introduce any timing overhead towards the system throughput, although there is an energy overhead.

Fig.8 shows the throughput performance and compute efficiency, normalized to GPU baseline, for $Naive$ and $Oracular$, when processing a pool of 3M patterns. We use match rate (in terms of number of patterns processed per second) for throughput; match rate per milliwatt, for compute efficiency. $Naive$ yields very low throughput – by mapping each pattern to every column of each array at a time, and thereby increasing the total execution time significantly. $Oracular$ pattern scheduling is very effective in eliminating this inefficiency: we observe that the throughput performance w.r.t. $Naive$ increases by orders of magnitude in this case.

To put these throughput values in context, we can look at the time required to process the pool of 3M patterns, which is over 38 hours, using 300 arrays under $Naive$. The fundamental limitation for $Naive$ is the redundancy in computation. Since at a time, $Naive$ feeds only one pattern into all SpinPM arrays, the total time required to process the entire pool of patterns is higher. The effective throughput is limited by the time taken to align one pattern in one column. On the other hand, $Oracular$ only takes about 0.05 seconds for the same pool of patterns. This drastic change in runtime is due to feeding multiple patterns into SpinPM arrays at the same time.

It is fundamental to the understanding of the performance and energy characterization to identify the individual contributions of actual computation stages – i.e., Stages(1)–(8) from Section IV. Fig.9 shows the distribution of energy and latency components. The preset overheads are 67.67% and 70.7% in energy and latency, respectively, where the bit-line (BL) driver energy and latency overheads are $< 1\%$. The breakdowns in Fig.9 do not contain preset and BL driver related overheads. Apart from these, we observe that the majority of the energy (Fig.9a) is consumed by the match operations and additions during similarity score computations. However, in case of latency (Fig.9b), the dominant components change to read-outs of similarity scores (i..e, Stage (8)) while the match and additions have similar shares. In case of both energy and latency, writes (i.e., Stage (1)) consume $< 1\%$ of the share.

This breakdown clearly identifies preset overhead as the essential bottleneck. Also, although the time required by the match and similarity score compute phases are not drastically different, the energy required by the similarity score compute phase is around $1.5\times$ of that of match phase. Accordingly, we next look into preset and similarity score computation operations for optimization opportunities.

Optimized Designs ($NaiveOpt$, $OracularOpt$): As the reduction tree for addition (Fig.4b), which is at the core of similarity score computations, already represents an efficient design, we focus on optimizations to reduce the preset overhead. Since presets are inevitable for logic operations, it is not possible to entirely get rid of them. However, we can still hide preset latency through careful scheduling of presets.

As presets do not correspond to actual computation, $Naive$ and $Oracular$ simply perform them in between computation. The challenge comes from successive steps in computation using the very same set of cells to implement logic functions. Instead of interrupting computation to preset these cells every time a few computation steps are completed, we can distribute such consecutive steps to different cells, using the scratch area from Fig.3, and preset them at once, before computation starts. We call the resulting designs $NaiveOpt$ and $OracularOpt$, respectively. The *NaiveOpt* and *OracularOpt* bars in Figure 8a and Figure 8b capture the resulting energy and throughput performance. We observe that, for each design option, energy consumption of the optimized case is unchanged. This is because the optimization only changes the scheduling of presets, where the total number of presets performed still remains the same. The throughput performance, on the other hand, skyrockets in both cases thanks to gang presets (Section C).

Practical Search Space Pruning: The throughput for $Oracular$ represents the theoretically achievable maximum. We next consider a practical implementation, as detailed in Section III-B. For the GRIM filter based implementation, we observe that the filtering overhead, as compared to the actual pattern matching overhead, is very insignificant and therefore has effectively no impact, even considering very high substring lengths (used for chunking the patterns and the reference in converting them to bit-vectors). However, the accuracy of the filtering still has an impact, as captured by Fig. 10. This figure shows, without loss of generality, how the throughput and compute efficiency (both normalized to NMP baseline) of DNA changes when the number of locations (column indices) in an array for possible matches increases (which would be the case under heavy aliasing during hashing). The decrease in

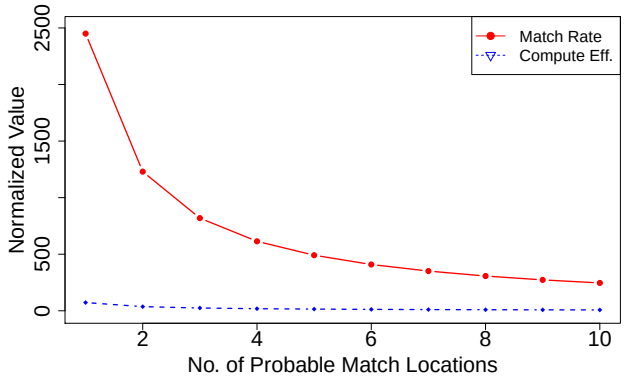
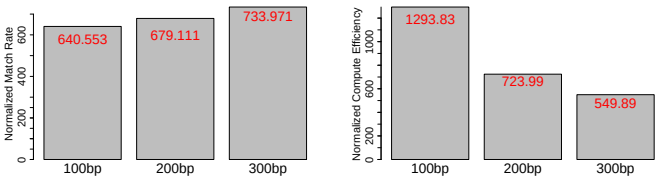


Fig. 10: Impact of filtering inaccuracy on throughput.

performance numbers is intuitive since more match locations refer to more iterations of the same set of search patterns through SpinPM arrays. Luckily, even for a high degree of filter inaccuracy, SpinPM-SHE can perform better than the baseline.

How close a practical implementation can come to *Oracular* strongly depends on the actual values of the patterns, as well, which may or may not ease scheduling decisions. Since each array keeps consecutive fragments of the reference, it is always possible that patterns directed into a particular array do not have any matches in any of the columns. We may not always be able to eliminate such ill-schedules, depending on the pattern values, where the incurred redundant computation would degrade performance. The feasibility of any pattern scheduler is contingent upon the distribution of the patterns, in terms of the columns in the arrays where the most similar fragments reside.

Sensitivity Analysis: Up until now, we have used a pattern length of 100 characters. We will next examine the impact of pattern length on energy and throughput characteristics. Without loss of generality, we confine the analysis to $OracularOpt$. For the purpose of design space exploration, we experiment with pattern lengths of 200 and 300 characters, which are representative values for the alignment of short DNA sequences [13]. We keep the array structure the same, while the reference length remains fixed by construction.



(a) Match rate (patt/sec) (b) Compute Eff. (patt/sec/mW)

Fig. 11: Sensitivity to pattern length for *OracularOpt*.

Fig.11 summarizes the outcome. Understandably, with the pattern length increasing, more computation becomes necessary to generate the similarity scores in each row. However, this effect does not directly translate into degraded performance: The throughput for increasing pattern lengths remains close to the baseline throughput for 100-character patterns. This is because the preset optimization is scalable. Increasing pattern length translates into more scratch bits for presets, which acts against throughput going down sharply. Irrespective of the application domain, the maximum pattern length is actually limited by technology constraints, since the required number of cells per column also increases with increasing pattern length. We further observe that the compute efficiency (i.e., the match rate per mW) decreases due to increases in computation per alignment, which is congruent with the intuition.