Exploiting Spatial Architectures for Edit Distance Algorithms

Jesmin Jahan Tithi
Stony Brook University
jtithi@cs.stonybrook.edu

Neal C. Crago
VSSAD, Intel Corporation
neal.c.crago@intel.com

Joel S. Emer
VSSAD, Intel Corporation / CSAIL, MIT
joel.emer@intel.com, emer@csail.mit.edu

***Abstract*—In this paper, we demonstrate the ability of spatial architectures to significantly improve both runtime performance and energy efficiency on *edit distance*, a broadly used dynamic programming algorithm. Spatial architectures are an emerging class of application accelerators that consist of a network of many small and efficient processing elements that can be exploited by a large domain of applications. In this paper, we utilize the dataflow characteristics and inherent pipeline parallelism within the edit distance algorithm to develop efficient and scalable implementations on a previously proposed spatial accelerator.**

We evaluate our edit distance implementations using a cycle-accurate performance and physical design model of a previously proposed triggered instruction-based spatial architecture in order to compare against real performance and power measurements on an x86 processor. We show that when chip area is normalized between the two platforms, it is possible to get more than a 50× runtime performance improvement and over 100× reduction in energy consumption compared to an optimized and vectorized x86 implementation. This dramatic improvement comes from leveraging the massive parallelism available in spatial architectures and from the dramatic reduction of expensive memory accesses through conversion to relatively inexpensive local communication.

I. Introduction

There is a continuing demand in many application domains for increased levels of performance and energy efficiency. While the number of transistors is expected to continue to scale with Moore's law for at least the next five years, the "power wall" has dramatically slowed single-core processor performance scaling. Recently, several accelerator architectures have emerged to further improve performance and energy efficiency over multi-core processors in specific application domains. These architectures are tailored using the properties inherently found in these domains, and can range in programmability. Perhaps the best-known examples are fixed-function accelerators, which are tailored to single algorithms such as video decoding, and GPUs, which are fully programmable and target data parallel and SIMT-amenable code.

Spatially programmed architectures are an emerging class of programmable accelerators that target application domains with workloads whose best-known implementation involves asynchronous actors performing different tasks, while frequently communicating with neighboring actors. Such architectures are typically made up of hundreds of small processing elements (PEs) connected together via an on-chip network. When an algorithm is mapped onto a spatial architecture, the algorithm's dataflow graph is broken into regions, which are connected by producer-consumer relationships. Input data is then streamed through this pipelined set of regions. The application domains that spatially-programmed architectures target spans a number of important areas such as signal processing, media codes, cryptography, compression, pattern matching, and sorting.

Edit distance is a broad class of algorithms that find use in many important applications, spanning domains such as bioinformatics, data mining, text and data processing, natural language processing, and speech recognition. The edit distance problem determines the minimum number of "non-match" data edits to convert a source string or data object S to a target string or data object T. The algorithm also keeps track of the specific data edits required to convert S to T, from a set of four possible data edits: insertion, deletion, match or substitution. For example, if S = "computer" and T = "commute", the minimum number of "non-match" edits to convert S to T is 2 and the edits are, $MMMSMMMD$ (where M = Match, S = Substitute, D = Delete).

The edit distance problem is ripe for acceleration, as the dynamic programming techniques typically used to solve the problem take $O(nm)$ time and space, where m and n are the lengths of the strings S and T respectively. However, the nature of data dependences within the algorithm makes vectorization and parallelization non-trivial on modern CPUs and GPUs. On the other hand, these same data dependences have very nice dataflow properties which are quite naturally mapped on spatial architectures using pipeline parallelism [1], [2]. Moreover, the exploitation of pipeline parallelism also enables the conversion of many memory references into much less expensive local communication which further improves efficiency.

In this paper, we study the potential of spatial architectures to solve the edit distance problem. Our main goal is to show that edit distance and similar algorithms naturally map to spatial architectures, improving performance and energy consumption substantially over general-purpose processors. We first build intuition on why spatial architectures are a good fit for edit distance and similar algorithms with local dependencies. We then describe three different algorithmic implementations of edit distance tailored to spatial architectures. We evaluate these implementations by conducting a detailed experimental analysis of performance and energy consumption on a triggered instruction-based spatial architecture [3]. Finally, we compare the performance and energy consumption of our implementations to highly optimized and vectorized x86 implementations.

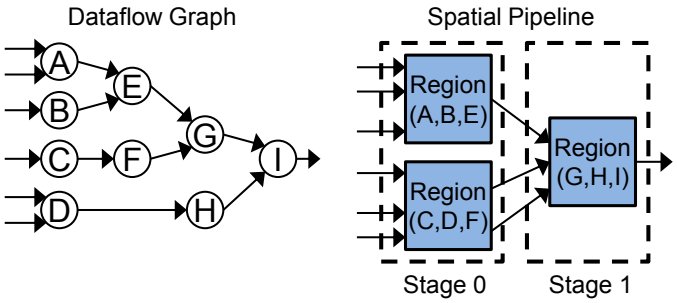

Fig. 1: Spatial programming example. Converting a dataflow graph to a spatial pipeline of regions.

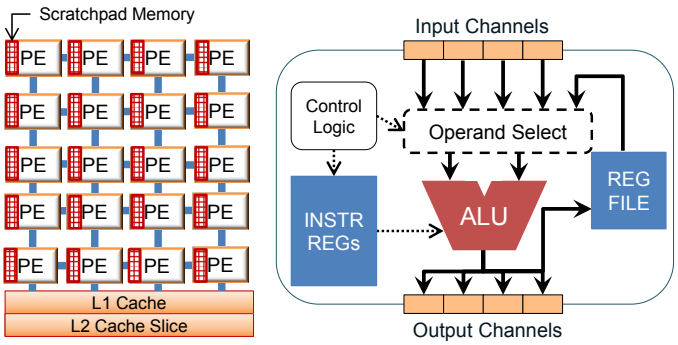

Fig. 2: Example spatial architecture. Network of PEs, scratchpad memory, and caches are shown alongside a PE diagram.

II. Background

A. Spatial Architectures

In the spatial programming paradigm, an algorithm's dataflow graph is broken up into regions so that it can be represented as a pipeline of computation. Sets of independent regions act as stages within the pipeline, with producer-consumer relationships between stages. Ideally, the number of operations in each region is kept small, as performance is usually determined by the *rate-limiting step*. Figure 1 presents an example dataflow graph and its corresponding representation in the spatial programming paradigm. In this example, each region is made up of three nodes from the original dataflow graph, and the total number of pipeline stages is two. Note that in Stage 0, two regions are independent and are executed in parallel. After the pipeline is generated, the input dataset can be streamed through the pipeline and the inherent pipeline parallelism can be exploited.

While the pipeline can in theory be mapped to general-purpose processors, executing the algorithm on the appropriate accelerator architecture can provide significant benefits. Accelerator architectures execute alongside general-purpose processors with the end goal of improving the performance and energy consumption of a select set of algorithms and application domains. Similar to how vector engines and GPUs are chosen to accelerate many vectorizable algorithms, spatial architectures are chosen to accelerate algorithms amenable to spatial programming. Spatial architectures are a computational fabric of hundreds or thousands of small processing elements (PEs) directly connected together with an on-chip network. The algorithm's pipeline is successfully mapped onto a spatial architecture by utilizing some number of PEs to implement each region of the dataflow graph, and then by connecting the regions using the on-chip network. As performance depends on minimizing the execution time of each pipeline stage, the regions are typically sized as small as possible, with the algorithm utilizing all of the available PEs.

Spatial architectures broadly fall into two categories, primarily based upon the basic unit of computation: logic-grained and coarse-grained. Field-programmable gate arrays (FPGAs) are among the most well known logic-grained spatial architectures and are most commonly used for ASIC prototyping and as stand-alone general logic accelerators. FPGAs are designed to emulate a broad range of logic circuits and use very fine-grained lookup tables (LUTs) as their primary unit of computation [4], [5]. More complex logical operations are constructed by connecting many LUTs together using the on-chip network. While the use of fine-grained LUTs results in a high-degree of generality, this generality results in much lower clock speeds for mapped algorithms when compared with ASIC implementations. In general, FPGAs and logic-grained spatial architectures sacrifice compute density for complete bit-level generality. It is also well known that the programming environment for FPGAs is particularly complex. FPGAs typically use a low-level programming model (e.g. VHDL or Verilog) due to being used primarily for ASIC prototyping. Additionally, the fine-grained nature of the LUTs creates a large solution search space for place and route algorithms, which can lead to unacceptably long compilation times.

However, a common observation is that many algorithms primarily utilize byte- or word-level primitive operations, which are not efficiently mapped to bit-level logic and logic-grained spatial architectures such as FPGAs. To partially address these inefficiencies, some FPGAs now provide digital-signal processing datapaths alongside the traditional LUTs. In contrast to FPGAs, coarse-grained spatial architectures are designed from the ground up to suit the properties of these algorithms. Coarse-grained spatial architectures optimize byte- and word-level primitive operations into hardened logic and through the utilization of ALUs within PE datapaths [6], [7], [8]. The hardened logic results in much higher compute density, which leads to faster clock speeds and reduces compilation times substantially compared to FPGAs. Generally speaking, these coarse-grained spatial architectures have a higher-level programming abstraction, which typically includes some notion of an instruction set architecture. In other words, PEs can be programmed by writing a sequence of software instructions, rather than requiring the hardware-level programming of an FPGA. The distinct advantages and large possible design space of coarse-grained spatial architectures has resulted in a significant amount of recent research. Specifically, there has been research into evaluating architectures, control schemes, and levels of integration with host processor cores [3], [9], [10], [11].

Given the clear benefits of coarse-grained compared to logic-grained, in this paper we focus on implementing edit distance on coarse-grained spatial architectures. Figure 2 presents the high-level architecture and PE-level architecture of the coarse-grained spatial architecture we consider. The architecture consists of a collection of PEs, scratchpad memory, a cache hierarchy, and an on-chip network. For our architecture, each PE has some control logic, an instruction memory, a register file, an ALU, and some number of input and output

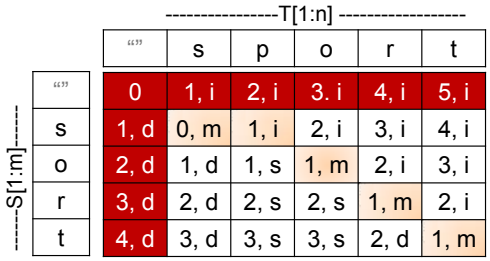

Fig. 3: Solving for edit distance using dynamic programming. The dark-shaded cells are the edits for solving the base cases, and the light-shaded cells are the required minimum edits.

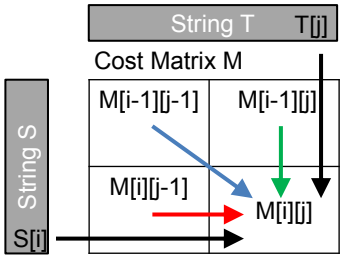

Fig. 4: Data flow dependences for calculating a cell in edit distance. Dependences are found along the row and column of the cost matrix, which limits the ability to vectorize.

connections to an on-chip network. To further support high compute density and provide efficiency, the instruction memory and register file within the PE are kept quite small, and the complexity of the ALU is kept low. PEs connect to each other, scratchpad memory, and the cache hierarchy using the on-chip network.

B. Edit Distance

In this section, we describe the edit distance problem and what makes it amenable to pipeline parallelism and spatial architectures. The edit distance problem is defined as finding the minimum edit cost to convert one string or data object into another string or data object. Solving the edit distance problem is interesting because of its prevalence in important application domains including bioinformatics, data mining, text and data processing, natural language processing, and speech recognition. In addition to those domains, achieving better performance and energy efficiency on edit distance through exploiting spatial architectures can also provide insight into how other applications with similar local dependencies might benefit when mapped to spatial architectures. Such domains include dynamic programming problems with local dependences (e.g., longest common subsequence, sequence alignment), virus scanners, security kernels, stencil computations, and financial engineering kernels.

$$ED(S[1:i], T[1:j]) =$$

$$\begin{cases} 0 \text{ if } i = j = 0, \\ CostOfInsert(T[1:j]) \text{ if } i = 0,\ 1 \le j \le n, \\ CostOfDelete(S[1:i]) \text{ if } j = 0,\ 1 \le i \le m, \\ \min \left\{ \begin{array}{c} MatchOrSub(S[i], T[j]) + ED(S[1:i-1], T[1:j-1]), \\ CostOfInsert(T[j]) + ED(S[1:i], T[1:j-1]), \\ CostOfDelete(S[i]) + ED(S[1:i-1], T[1:j]) \end{array} \right\} \\ \text{if } i, j > 0. \end{cases}$$

(Recurrence 1)

1) Overview of the Edit Distance Problem: Recurrence 1 solves the edit distance problem. The edit distance for converting S of length i to T of length j can be computed by taking the minimum of solutions to three smaller sub-problems. The first sub-problem is to *match or substitute* $S[i]$ with $T[j]$, and then recursively find the edit distance of converting S of length $i-1$ to T of length $j-1$. The second sub-problem is to *insert* the last character of $T(T[j])$ at the end of S, and then recursively find the edit distance of converting S of length i to T of length $j-1$. The third sub-problem is to *delete* the last character of $S(S[i])$, and then recursively find the edit distance of converting S of length $i-1$ to T of length j.

The costs of *match, substitute, delete* and *insert* are user-defined and can vary depending upon the application. In the most general edit distance problem, all costs are assumed to be the same (typically 1, except a cost of 0 for a match). The three base cases for the recurrence are enumerated in Recurrence 1:

- Converting a string S of length 0 to another string T of length 0 is a cost of 0.
- To convert a string S of length 0 to any string T of any length j, we *insert* all j characters of T which incurs a total cost of the sum of inserting each character from T of length j.
- To convert a string S of length i to a string T of length 0, we *delete* all characters from S which incurs a total cost of the sum of deleting each character from S of length i.

In the rest of the paper, we assume that the cost of a single insertion, deletion, and substitution is 1.

It is inefficient to use recursion to solve the edit distance problem due to the large number of sub-problem recomputation required. Therefore, dynamic programming principles are typically used to solve the problem in a bottom-up manner. The dynamic programming approach saves the result of each sub-problem in a table, enabling reuse rather than requiring recomputation. To find the required number of edits to convert a source string S to the target string T, a two-dimensional $m \times n$ cost matrix "M" is allocated, where m and n are the lengths of the two strings S and T, respectively. Each cell of the matrix $M(i,j)$ gives us the minimum number of edits to convert $S[1:i]$ to $T[1:j]$. We first populate the matrix with the base case solutions, and then compute the remaining cells row by row following the same recursive formula. Figure 3 shows the filled out cost matrix after executing the dynamic programming algorithm on $S =$ "*sort*" and $T =$ "*sport*" where cell $M[m][n]$ gives the final edit distance.

2) Exploitation of Pipeline Parallelism: In the dynamic programming approach to the edit distance problem, the matrix cells have local dependencies. Figure 4 presents the data dependences for calculating a single cell $M[i][j]$. Observe that the value of a cell $M[i][j]$ depends on its *top cell* $(M[i-1][j])$, *left cell* $(M[i][j-1])$ and *diagonal cell* $(M[i-1][j-1])$. Because of these dependencies, this computation is not vectorizable along the row or the column of the $2D$ cost matrix. While it is possible to vectorize along the diagonal by reshaping the cost matrix into a diamond, such an implementation

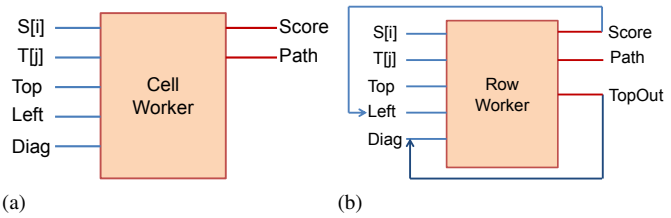

Fig. 5: a) A simple cell worker that computes a single cell of the cost matrix, b) An optimized row worker.

```
CalculateCell( row i, column j)
{
   insert_cost = M[ i ][ j-1 ] + CostOfInsert (T[ j ])
   delete_cost = M[ i-1 ][ j ] + CostOfDelete (S[ i ])
   match_substitute_cost = M[ i-1 ][ j-1 ] + MatchOrSub (S[ i ], T[ j ])
   [score, path] = Min (insert_cost, delete_cost, match_substitute_cost)
   M[ i ][ j ] = score;
   Path[ i ][ j ] = path;
}
```

Fig. 6: Pseudo-code for the calculation of a single cell.

requires dummy computations and frequent communication between cells must still be facilitated using expensive memory accesses. Conversely, the data dependencies found in edit distance naturally compose into pipeline parallelism: values produced by a worker responsible for computing a cell of the cost matrix can be consumed by workers computing adjacent cells. It is possible to get very efficient cell-level parallelism for edit distance in a spatial architecture because spatial architectures have the benefit of small efficient PEs and direct PE-to-PE communication capability. Note that cell-level parallelism is not feasible on multicore machines at all because of the prohibitive overhead of communication and scheduling.

III. Edit Distance on Spatial Architectures

In this section, we discuss how the edit distance problem maps down to spatial architectures. We start by describing the basic unit of computation, a worker, and utilize that worker to develop an initial naïve implementation. We analyze that naïve implementation and describe several possible optimizations. We also explain how a spatial implementation of edit distance makes more efficient use of memory.

A. Designing a Worker

To implement edit distance, we first create a core module that incorporates the data flow and state transitions needed to compute the value of a single cell of the cost matrix M. Part of this process is deciding which inputs and outputs are required for cell computation, and the relationship between the inputs and outputs of a cell with its neighboring cells. Figure 5(a) depicts an abstract worker which implements the core module and its inputs and outputs. As shown in Figure 4, the *score* of a cell $M[i][j]$ of the cost matrix depends primarily on its top ($M[i-1][j]$), $left$ ($M[i][j-1]$) and $diagonal$ ($M[i-1][j-1]$) cells, as well as the string characters $S[i]$ and $T[j]$. The score of the current cell $M[i][j]$ is determined by calculating the minimum of *insert cost* (*left* + CostOfInsert($T[j]$)), *delete cost* (*top* + CostOfDelete($S[i]$)) and *match/substitution cost* (*diagonal* + MatchOrSub($S[i]$, $T[j]$)), where $S[i]$ and $T[j]$ is the i^{th} character of S and j^{th} character of T, respectively. The path chosen for each cell, or the edit that resulted in the minimum score, can also be stored in a separate $path$ array, where $path[i][j]$ = (*delete, insert, match/substitute*). The data stored in the $path$ array can later be used to reconstruct the actual edits used to convert S to T in linear time. Therefore, in this initial approach we need 5 memory reads ($S[i]$, $T[j]$, Top, $Left$, $Diagonal$), 2 memory writes (score, path), 3 additions and 3 subtractions to compute the value for each cell.

Figure 6 shows simplified pseudo-code for a single cell computation of $M[i][j]$. First, the insert, delete, and match/substitute costs are computed using values from $M[][]$, $S[i]$, and $T[j]$. Next, the minimum between the three costs is chosen, returning both the score and the path values. Finally, the score and path values are written out to memory. In practice, we find that we can split this core module into two other smaller modules, each of which can be mapped onto a PE, in order to reduce the length of critical path. We define a *worker* as the unit of these two PEs that implements the core module. A collection of these workers is used to compute the score values of the entire cost matrix.

1) Optimization: One possible implementation of edit distance on a spatial architecture would be to use distinct workers to calculate the value of each cell $M[i][j]$. However, the scalability of such an approach to large problem sizes is quite limited considering that the requirement for $O(mn)$ workers would require at least $O(mn)$ PEs. Spatial architectures have finite physical resources by definition, and thus a scalable implementation needs to be able to map to those finite resources regardless of m and n. Therefore, we need to find an alternative solution to compute all the mn cells using only a limited number of workers, w. For example, if we consider assigning one worker to compute all the cells of a row of the cost matrix, we can observe the following patterns in the inputs and outputs of that *row worker* (see Figure 5(b)):

- Top ($M[i-1][j]$) at current cell position $M[i][j]$ becomes the $diagonal$ for the next cell $M[i][j+1]$.
- The current computed score $M[i][j]$ becomes $left$ for the next cell $M[i][j+1]$.
- $S[i]$ needs to be read only once from memory for the entire i^{th} row.
- $T[j]$ and top need to be read for each cell from memory.

Therefore, if we can reuse the values already read from memory and produced by the same row worker, that will save $O(3n)$ memory reads *($S[i]$, left, diagonal)* for each row (i.e., saves around $3mn$ reads out of $5mn$ reads in total).

Figure 7 presents a flow chart for the control path of the two row worker modules mapped to PEs. First, the delete and insert costs are computed and the minimum between the two is chosen, while in parallel the match-substitute cost is also computed and communicated. Then the minimum of the three costs is chosen, and the final score and path values are determined.

If we consider two consecutive row workers working on two consecutive rows of the score/cost matrix M, it is possible to observe some further memory access optimizations. A row worker working on the i^{th} row can send its current computed

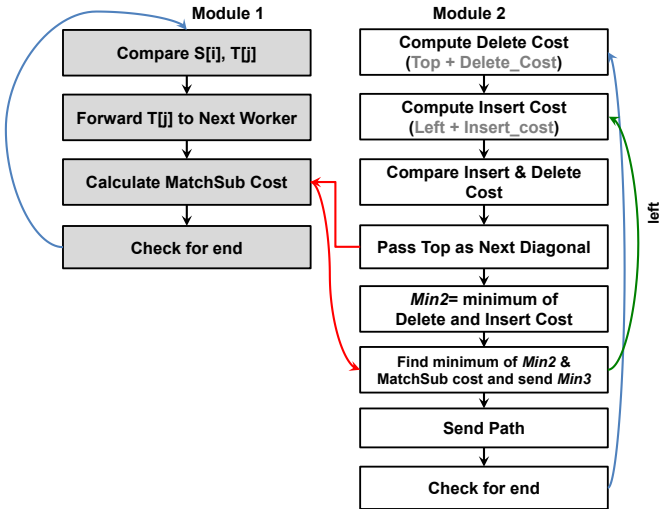

Fig. 7: Flow chart for the control flow paths of a single worker. Blue arrows show state transitions, green arrows show self-feedback, and red arrows show communication with other PEs.

score, $M[i][j]$ as the *top* value to the row worker working on the $(i+1)^{th}$ row. Similarly, $T[j]$ can also be reused by other row workers working on the j^{th} column of the score matrix by propagating $T[j]$ using local PE communication channels. Once a character $T[j]$ has been read from memory by row worker 1, row worker 1 can forward $T[j]$ to row worker 2, row worker 2 can forward it to row worker 3, and so on. Therefore, if we use w row workers to compute w consecutive rows of the cost matrix, any row worker k for $1 < k \leq w$, does not need to read $T[j]$ from memory. Similarly, each k^{th} row worker $(1 < k \leq w)$ receives its *top* value from row worker $k-1$'s computed scores. Hence, only the first row worker of a strip of w rows (*strip:* w consecutive rows of the cost matrix) needs to read the *top* and $T[j]$ values from memory. As a result, nearly all memory read operations remaining are removed and converted into less expensive local PE-to-PE communication, saving both memory bandwidth and energy consumption.

2) Mapping: Note that the row workers here proceed as a diagonal wave front. For example, when row worker k works on cell $M[i][j]$, row worker $k+1$ works on cell $M[i+1][j-1]$ and row worker $k+2$ works on cell $M[i+2][j-2]$ and so on, in a pipelined manner. Figure 8 shows the interconnection between two consecutive row workers and a possible layout of the row workers on a spatial architecture made with a serpentine grid of PEs where row worker W_k receives input from row worker W_{k-1} and sends output to row worker W_{k+1}.

Based on these observations and the resulting optimized row worker (Figure 8), we have designed three different algorithms to solve the edit distance problem, namely: naïve, strip mining, and tiling. For each of these algorithms, we primarily focus on computing the score, as prior work has shown that the edits can be reconstructed by recomputing the required subsections of the cost matrix while tracing in the backward direction [12], [13], [14], [15]. Hence, showing that our algorithms do better in computing the score should mean that they will also perform better in computing the edits. In each of these algorithms we use w row workers to compute the scores and paths of the first w consecutive rows from 1 to w, and then compute rows from $1+w$ to $1+2w$, and so

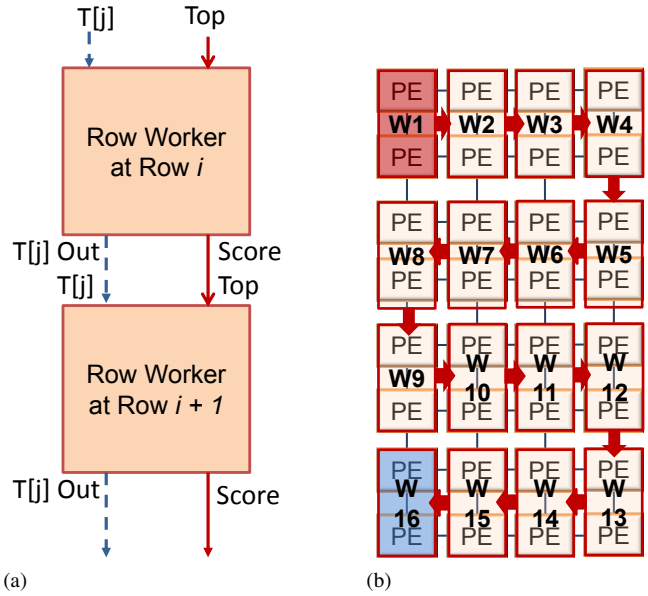

Fig. 8: a) Two row workers working on two consecutive rows and connected together using communication channels. b) A possible layout of the row workers on a grid of PEs.

on until the entire cost matrix has been computed. Note that string S is padded (and T for the tiled version), as needed to make the length divisible by w (or tile height).

B. Naïve Implementation

In the naïve approach, we use $O(mn)$ memory space to store the scores of the cost matrix (and path), and each row worker stores the score (and path) value to memory for each cell it computes. We connect a row worker's output to its own input and to other row workers as shown in Figures 5(b) and 8. These optimizations remove most of the memory reads otherwise required. However, $O(mn)$ memory writes are still required to store the $O(mn)$ score values for all cells in the resulting cost matrix.

C. Optimization: Use of Linear Memory Space

"Quadratic space kills before quadratic time". In the standard edit distance problem, the two-dimensional cost matrix M consumes $O(mn)$ memory space. However, this quadratic use of memory space becomes infeasible for large strings. Fortunately, it is possible to use linear memory space $(O(n))$ to store the cost matrix. Observe that for edit distance, the resulting output data is the final cost of converting S to T which can be found in cell $M[m,n]$. Therefore, we do not need to maintain data storage for other cells when they are no longer actively being used. To compute the score/cost for the i^{th} row, we only need the $(i-1)^{th}$ row as input. Rows before $(i-1)$ can be forgotten. Therefore, it is possible to use a cost matrix of linear size $n+1$, from which the first row worker (from the set of w row workers) reads its *top* inputs, and the last row worker writes its computed cost values which can be used as input for the next set of rows (computed on the same w row workers). We use this linear memory space optimization in our strip mining and tiling based algorithms.

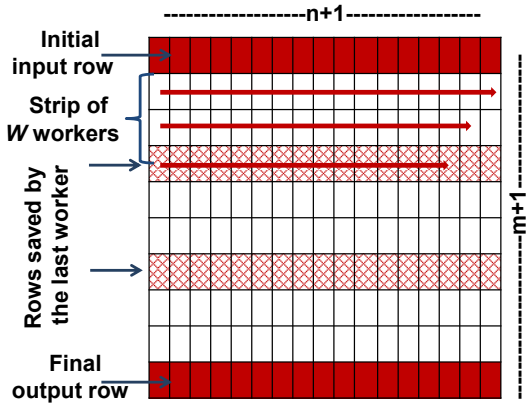

Fig. 9: Strip mining using memory and scratchpad memory. Initial and final rows are read from and written to memory, while the intermediate rows use either memory or scratchpad memory.

Although it is sufficient to use linear memory space ($O(n)$) to store the cost matrix, use of a two-dimensional cost matrix can aid in the reconstruction of actual edits/path in linear time. It is also possible use a separate two-dimensional path matrix while using linear memory space for the cost matrix, which allows linear time reconstruction of edits. Finally, it is also possible to use only linear memory space for the cost matrix, and reconstruct the edits in quadratic time without saving any path matrix [14], [15] which is discussed later in this section.

D. Strip Mining

The *strip mining* technique involves computing the two dimensional cost matrix in a strip-by-strip manner (*strips* of w consecutive rows from the cost matrix). In this approach we virtually divide the two-dimensional cost matrix into strips of size $w \times n$, where w denote the number of row workers and n denote the width of the cost matrix. In the strip mining approach we use a linear cost matrix array of size $n+1$, from which the first row worker reads its *top* inputs and the last row worker writes its computed cost values which are used as input for the next strip. As before, we use the optimized row worker for this algorithm. We use two different strategies for the strip mining algorithm as described below:

1) Strip Mining using Memory: In the strip mining using memory algorithm, only the first row worker of a strip of size w reads the score values from cost matrix and string T from memory, and only the last row worker stores the computed score values to memory (Figure 9). This organization reduces the total number of memory writes to the cost matrix to $O\left(\frac{m}{w}n\right)$ from $O(mn)$ in addition to the reduction in memory reads as described before. Furthermore, the number of memory reads of string T as well as the cost matrix reduces to $O\left(\frac{m}{w}n\right)$. To optimize this approach further, each row worker starts computing from the 0^{th} column instead of the 1^{st} column of the cost matrix. Note that in a typical edit distance algorithm, computation occurs starting from the 1^{st} column while using the 0^{th} column as input. If we had started computing from the 1^{st} column, we would need to read the $left$ and $diagonal$ cells from memory. However, if we start from the 0^{th} column, we can use the *top* value to compute the $left$ and $diagonal$ by adding 1 to the *top* value (which comes from memory for the first row worker and from the previous row worker for all other row workers). This approach reduces the number of memory reads by $2m$. Figure 9 shows how the strip mining algorithm works. The hash-patterned cells in the cost matrix are stored in memory by the last row worker and read by the first row worker $\frac{m}{w}$ times, while the white colored cells are not stored to memory and are instead communicated directly between row workers.

2) Strip Mining using PEs' scratchpad memory: Note that memory accesses can be expensive and involve various levels of the cache hierarchy and main memory. As an alternative, we can leverage the scratchpad memory of the PEs (Figure 2) to store intermediate cost matrix values. In the strip mining using scratchpad memory algorithm, the first row worker reads the cost matrix from memory only during the first iteration (i.e., only for the first strip of the algorithm). Similarly, the last row worker stores the scores to the linear space cost matrix M in memory only in the last iteration (last strip of the algorithm). In all other intermediate iterations, the first row worker reads the cost matrix values from scratchpad memory, where the last row worker has saved its computed cost matrix values in previous iteration. This reduces the number of memory reads and writes for the cost matrix to $O(n)$. The scratchpad memory based algorithm operates similarly to the memory based strip mining algorithm. The key difference is that only the initial and final rows are read from or written to memory. In Figure 9, the hash-patterned cells of the cost matrix are stored in internal PE scratchpad memory by the last row worker and read by the first row worker $\frac{m}{w}$ times.

One drawback of this approach is that the amount of scratchpad memory on spatial architectures is limited and therefore can limit the maximum length of T. Note that although use of scratchpad reduces the number of memory reads and writes from/to the linear cost matrix from $O\left(\frac{m}{w}n\right)$ to $O\left(n\right)$, $O\left(\frac{m}{w}n\right)$ reads of string T are required in both strip mining approaches. The tiling based computation as discussed in the next subsection provides a way to deal with this limited amount of storage, and can reduce the number of memory reads and writes even further if a proper tile size is chosen.

Memory Loads/Stores and Time Complexity:

For the strip mining algorithm using memory, the total number of memory reads and writes is $O\left((3(\frac{m}{w}n)+m\right)$. This cost comes from $O\left(\frac{m}{w}\right)$ memory reads of the cost matrix and string T of length n, and $\frac{m}{w}$ memory writes to cost matrix of the same size. String S of size m must also be read once for the entire computation. Similarly, for the strip mining algorithm that uses scratchpad memory to store and read the cost matrix values, the total number of memory reads and writes is $O\left((\frac{m}{w}n)+2n+m\right)$ where $(\frac{m}{w}n)$ comes from reading T, $2n$ comes from reading and writing to linear space cost matrix and m comes from reading S. The running time of this algorithm with w row workers is: $T_w = \Theta((\frac{m}{w}n)+w\frac{m}{w}) = \Theta((\frac{m}{w}n)+m)$ where the second term comes from the synchronization cost of w row workers at the end of each strip. Hence, the running time with infinite number of row workers (as well as m row workers) is: $T_\infty = \Theta(m+n)$.

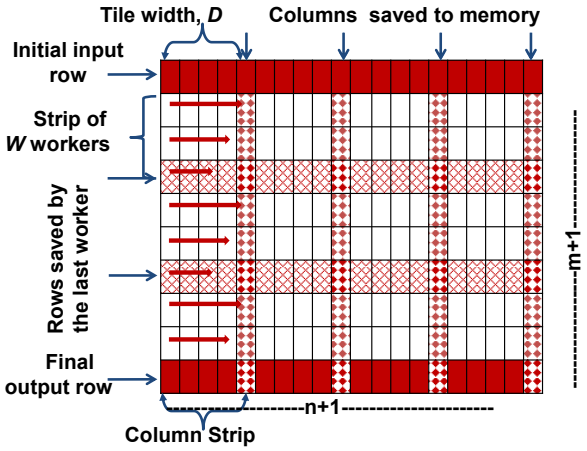

Fig. 10: Tiled approach. Computation occurs on a column strip of tiles. Initial and final rows are read from and written to memory, while intermediate rows use scratchpad memory. Column results are saved in memory.

E. Tiling

In the tiling algorithm, we virtually divide the two-dimensional cost matrix into tiles of size $w \times D$, where w denote the number of row workers (also height of the tiles in this case), and D denote the width of the tiles. We solve the column of tiles (*column strips*) one by one starting from the leftmost column of tiles, ending with the rightmost column of tiles. For the tiling algorithm we use linear $O(n)$ memory space to store the cost matrix values from which the first row worker reads its *top* inputs and the last row worker writes its computed cost values. Additionally we use two other $O(m)$ sized memory arrays to store the left most column of a column strip and the right most column of a column strip. These two arrays work as input and output in alternative iterations (column strips). All w row workers first compute values for the first column strip of $\frac{m}{w}$ tiles of size $w \times D$, each of which ends at the column $D+1$ of the original cost matrix M. Then the row workers compute the next column strip consisted of $\frac{m}{w}$ tiles as shown in Figure 10 and so on.

The computation for a single column of tiles (column strip) is similar to the strip mining algorithm using scratchpad memory with a few exceptions.

- Each row worker starts from the 1^{st} column, instead of the 0^{th} column. Therefore, each row worker needs to read $left$ and $diagonal$ cells for the very first column of each tile from memory.
- Each row worker of a tile needs to store the last value of its row to memory, so that they can be used as input (*left* and *diagonal* cells) for the next column strip.
- Only the first row worker of a column strip reads a segment of string T from memory at the beginning of that column strip, and all other row workers receive T from the previous row worker and forward T to the next row worker. The last row worker stores T in local scratchpad memory, so that for the next tile the first row worker does not need to read T from memory.

Therefore, over all the tiles, T needs to be read only once (cost $(O(n))$ to compute all $O(mn)$ cells of the cost matrix.

Figure 10 shows how the tiling based algorithm works. The two-dimensional cost matrix has been divided into tiles where the hash-patterned cells along the rows are stored in scratchpad memory and the checkerboard-patterned cells along the columns are stored in memory and are used as input ($left$, $diagonal$) for the next column of tiles. Note that we have a choice on whether the intermediate rows, intermediate columns, or both dimensions should be saved in scratchpad memory based upon each dimension's size and the amount of available scratchpad memory. In general the maximum tile width $D_{max} = min(Total\ Scratchpad\ Memory\ Size/2, n/2)$.

Memory Loads/Stores and Time Complexity:

To show that the tiling approach has better theoretical memory bandwidth utilization than that of the strip mining approach, we count total number of memory reads and writes required by the tiled approach. The total number of memory reads in this approach is $O(\left(3(n\frac{m}{D})\right) + 2n)$, where the $O\left(3(n\frac{m}{D})\right)$ comes from reading S, $left$ and $diagonal$ cells at the beginning of each tile. There are m memory reads for each column strip, and in total we have $\frac{n}{D}$ column strips/iterations. On the other hand, the $2n$ term comes from reading T and the cost matrix of size n from memory. In addition, there are $O(n)$ writes to memory for writing the final cost values (hashed-patterned cells in Figure 10) and $O\left(n\frac{m}{D}\right)$ writes for writing the end cells (rightmost column) for each column strip (checkerboard-patterned cells in Figure 10). Therefore, in the tiled approach, we have $\left(4n\frac{m}{D} + 3n\right)$ memory operations. Clearly, the tiled based approach will perform better than the strip mining with scratchpad memory based approach iff $\left(4n\frac{m}{D} + 3n\right) < \left((n\frac{m}{w}) + 2n + m\right) \implies D_{min} > 4w$ (considering $n = m$). As w is constrained by the number of PEs in the spatial architecture and D is constrained by the total aggregated scratchpad memory for all PEs, D will satisfy the condition trivially. The running time for this algorithm with w row workers is $T_w = \Theta\left(\frac{n}{D}(\frac{mD}{w} + w\frac{m}{w})\right) = \Theta\left(mn/w + mn/D\right)$.

F. Linear Memory Space Trace Back Path

In both of our strip-mining and tiled algorithms we use linear memory space for computing the score. In addition to that, we reduce the number of actual memory reads and writes in the spatial architecture by utilizing direct PE-to-PE communication, something not possible in a general-purpose processor architecture. However, the specific edits required to achieve the minimum edit distance are often needed alongside the scores. Storing the edits for each cell in the cost matrix often requires quadratic memory space which is very expensive for large string inputs. Fortunately, there are algorithms that can reproduce the edits without storing the edits initially (Hirschberg [12] and Chowdhury [14]) and without the requirement for quadratic memory space. However, such algorithms require extra $O(mn)$ work to do so. The algorithm in [14] is a divide and conquer based recursive algorithm which executes the edit distance algorithm in two passes: the forward pass and the backward pass. The algorithm assumes that it is given input boundaries (the 1st row and 1st column of the cost matrix) and at the end it will produce output boundaries (rightmost column and bottommost row). In the forward pass, the algorithm computes the score by recursively dividing a

PEs	32 Total - Each with 16 Instructions 8 local registers, 8 predicates
Network	Mesh (1 cycle link latency)
Scratchpad	8KB (distributed)
L1 Cache	4KB (4 banks, 1KB/bank)
L2 Cache	24 KB shared slice
DRAM	200 cycle latency
Estimated Clock Rate	3.4 GHz

TABLE I: Block Architectural Parameters

virtual two-dimensional cost matrix into four quadrants and keeps dividing until it reaches a small base case size when it solves for edit distance using the standard dynamic programming algorithm using linear memory space. During the forward pass, the algorithm saves additional information about where the path from each cell of the output boundary (rightmost column and bottommost row) intersects the input boundary (leftmost column and topmost row). In the backward pass, it recursively executes the edit distance in backward direction to reconstruct the path information. The algorithm decides which quadrant to explore based on where the path from the bottom-right point intersects the input boundaries and hence solves only those segments required to reconstruct the edits/paths.

As we have theoretically shown that use of spatial architecture reduces the total memory footprint by several times, it is easy to predict that all the traditional linear memory space algorithms (Hirschberg's or Chowdhury's) can realize a huge performance boost by using our optimized linear memory space edit distance row workers to compute the cost as well as the required edits on spatial architectures.

IV. Experimental Setup

A. Spatial Architecture Performance and Power Modeling

We evaluate our edit distance implementations on a cycle-accurate performance model that simulates the triggered instruction-based scalable spatial architecture (TIA) in [3]. The performance model is developed using Asim, an established performance modeling infrastructure [16]. We model the detailed microarchitecture of each TIA PE in the array, the mesh interconnection network, L1 and L2 caches, and DRAM. The architectural organization of our evaluation architecture is found in Figure 2. The architecture is built from an array of TIA PEs organized into blocks. Each block contains a grid of interconnected PEs, a set of scratchpad memory slices distributed across the block, a private L1 cache, and a slice of a shared L2 cache that scales with the number of blocks on the fabric. Table I provides the parameters that we use in our evaluation. The TIA PEs use 32-bit integer/fixed-point datapaths, and do not include hardware floating point units. As a reference, 12 blocks (each including PEs, caches, etc.) are about the same size as our baseline Intel®Core™i7-3770 core (including L1 and L2 caches), normalized to the same technology node. Also note that the length of a clock cycle of both a high-end x86 core and TIA are estimated to be the same [3]. We used one block of PEs (32 PEs in total) to conduct all our experiments, and then extrapolated the results to 12 blocks. As in other accelerators, a general-purpose processor is used as the host device responsible for setup of the kernel on the TIA architecture. The interface between the two devices is shared memory managed using cache coherence to transfer data, eliminating much of the communication overhead

```
Module MatchCost()
{
  predicate match;
  predicate p0 = false, p1 = false;
  doCMP_Si_Tj:
        when (!p0 && !p1 && %Si_In.tag != EOL && %Tj_In.tag != EOL) do
        cmp.ge match, %Si_In.data, %Tj_In.data (deq % Si_In, p0:=1)

   Forward_Tj:
        when (p0 && !p1) do
          enq %TjOut, % Tj_In.data (deq % Tj_In, p1:=1)

    Write_Match_Cost:
        when (p0 && p1 && match && %diag.tag != EOL  ) do
          enq %score, % diag.data (deq % diag, p0 := 0, p1=0)

  Write_Substitute_Cost:
        when (p0 && p1 && !match && %diag.tag != EOL  ) do
        enq %score, ADD(% diag.data, 1)  (deq % diag, p0 := 0, p1=0)
}
```

Fig. 11: Sample code for a module that matches S[i] with T[j] and computes the cost of cell M[i][j] based on the *diagonal* cell M[i-1][j-1].

to transfer the initial strings as input and the path and score as output.

We modeled energy consumption on the TIA-based spatial architecture using synthesis and CACTI with a CMOS 22nm manufacturing process and a 0.8 Volt operating voltage [17]. CACTI is used to model memory structures including instruction memory, scratchpad memory, the caches (L1, L2), while synthesized Verilog is used for all other major PE structures such as local registers, functional units, pipeline latches, and scheduling logic. We leverage ITRS reports on 22nm to develop our wiring model. We leverage area and physical dimension estimates for a TIA PE, and we use those dimensions to find wire lengths for local connections between PEs and between PEs and memory, assuming global wiring tracks. In addition to generating the runtime of each benchmark, the performance model also outputs the activity for each component model in the block, measured by counting the number of accesses performed to the structure. Reads and writes are counted for storage structures, while the number of active cycles is counted for functional units. We then link the energy costs derived from CACTI and synthesis with architectural activity factors to generate the total dynamic energy consumption. Total energy consumption is calculated by combining this dynamic energy value with leakage energy based upon the runtime.

B. Spatial Edit Distance Implementation

We translate the data flow diagram of the row worker to triggered instruction code by mapping each step of the data flow diagram to one or more rules and their corresponding guard conditions that fire those rules. A sample code snippet that calculates the match/substitute cost is shown in Figure 11 that follows a similar convention to code in [3]. In this example, the predicates help to define states and guard conditions that determine relevant program state transitions and enable specific instructions to fire. For example, a PE first checks whether it is in state 0 and both the channels Si_In (S[i]) and Tj_In (T[j]) have inputs, then the PE compares Si_In with Tj_In, decides whether there is a match, dequeues the Si_In channel, and moves to state 1 (p1=0, p0=1). In state 1 the

PE forwards Tj_In to the next row worker through its Tj_Out channel, dequeues the Tj_In channel and moves to state 3 (p1=1, p0=1). Finally, based on whether the PE found a match or mismatch in state 0, it computes the match/substitute cost from either *diagonal* or *(diagonal + 1)* and again moves to state 0.

For integration with the host processor, we set up the input and output datasets and leverage control registers within the TIA architecture. The host sets up the memory space for the cost matrix and writes the memory pointers of the cost matrix and input strings into the TIA architecture registers, then the host signals the TIA architecture to start computation. The host waits for the TIA to signal that the row workers have finished their computation before leveraging cache coherence hardware to collect the resultant data.

C. x86 Comparison

For our comparisons to a general-purpose processor, we performed real runtime and power measurements using an Intel®Core™i7-3770. The i7-3770 is four-core, eight-thread processor which operates at a 3.4 GHz frequency (3.9 GHz Turbo Boost) and is manufactured in 22nm technology. To capture runtime for each experiment, we looped over the computation with enough iterations so that the caches were properly warmed up and so that execution took on the order of seconds in wall time, which enabled us to use operating system timers. To capture energy consumption, we utilized the LIKWID tool set which reads registers included in the Intel®Core™i7-3770 processor to read energy consumption and power [18]. Note that while we do not model power for DRAM accesses, we eliminated most DRAM accesses using code optimizations.

For our x86 software version, we started with a C++ naïve implementation of edit distance and progressively applied both source-level and compiler-level optimizations to improve performance and energy consumption. We evaluate the performance and energy benefit of each of these optimizations in detail in the next section. To enable parallelization of the x86 implementation of edit distance to multiple threads, we tiled computation and used OpenMP. Tiling divides the two-dimensional cost matrix into blocks, and then uses multiple threads to compute blocks of cells along a diagonal starting from the top-left diagonal and ending to the bottom-right diagonal. Each diagonal wavefront of blocks can be computed in parallel, with synchronization occurring between wavefronts. To improve scalability of our x86 implementation for edit distance to larger string sizes, we performed an additional source-level transformation to our tiled version to enable the use of linear memory space and reduce the size of the cost matrix [14]. In practical terms, this often reduces the memory footprint requirements from gigabytes of memory to kilobytes, which further improves cache behavior and avoids energy expensive DRAM accesses. For the final source-level transformation, we transformed each tile computation to enable vectorization by the compiler. As parallelism is found along the diagonals, we transformed each tile into the shape of a diamond so that parallel work exists horizontally in memory. For our compiler, we used the Intel®*icc* compiler version 14.0 and experimented with optimization levels $-O0$ through $-O3$ and AVX vectorization.

Algorithm	Platform	Workers / PEs per Block	Cells / cycle	Speedup over x86	Area-Normalized Speedup over x86
Score and Path	**x86 Optimized**	–	0.36	1.0	1.0
	TIA StripMem	16 : 32	1.11	3.08	37.0
	TIA StripSP	14 : 30	1.53	4.25	51.0
	TIA Tiled	10 : 22	1.12	3.11	37.3
Score Only	**x86 Optimized**	–	0.48	1.00	1.00
	TIA StripMem	16 : 32	2.14	4.45	53.5
	TIA StripSP	14 : 30	1.80	3.75	45.0
	TIA Tiled	14 : 30	1.88	3.92	47.0

TABLE II: Performance of Edit distance on the spatial architecture and a comparison with a typical modern processor

V. Experimental Results and Analysis

A. Overview of Performance Results

Table II presents the overview of the performance results for a fixed input string size of 1024 for both S and T. While in this paper we focus on computing the score matrix, the results for computing the score and path edits together are also presented. Scores are calculated using $O(n)$ memory space and edits are saved using $O(nm)$ memory space. Recall that it is possible to reconstruct the edits by recomputing a smaller subset of the scores in the backward direction. Therefore, showing that computing the scores on TIA is significantly faster than computing the score on x86 is still of great interest. The x86 implementation presented uses all the optimizations previously mentioned. Though the TIA implementations are scalable with the total number of PEs available, we limit the computation fabric to a single block and normalize to the area of an x86 core by multiplying by a factor of 12 (approximately 12 blocks consume the area of an Intel®Core™i7-3770 core).

From Table II we can see that strip mining using scratchpad memory is the fastest (achieves a 51 times speedup with respect to x86) among the algorithms that compute both score and path. We were only able to use 10 row workers (22 PEs) for the tiled based approach as we were resource constrained by the number of communication channels used by the row workers to read to and write from memory. Despite these constraints, the tile-based approach was slightly better than strip mining using memory in this case. For algorithms that only compute the score, the performance of all algorithms improved because the overall computational work was reduced by not saving $O(nm)$ edits to memory. When computing the score only, all our algorithms on TIA also performed at least 45 times faster than the x86 based version. Note that our extrapolated performance when scaling TIA to 12 blocks is conservative, as we assume the number of row workers is kept constant for each block. For example, strip mining using scratchpad memory requires two control PEs to work as a multiplexer and de-multiplexer attached to the first and last row workers, effectively limiting the maximum number of row workers to 14 instead of 16. However, this control overhead need not be replicated when scaling to 12 blocks, and performance would therefore be better than our extrapolated number.

B. Analysis of x86 optimizations

Before comparing the two platforms in detail with each other, we first seek to understand the impact of the coding

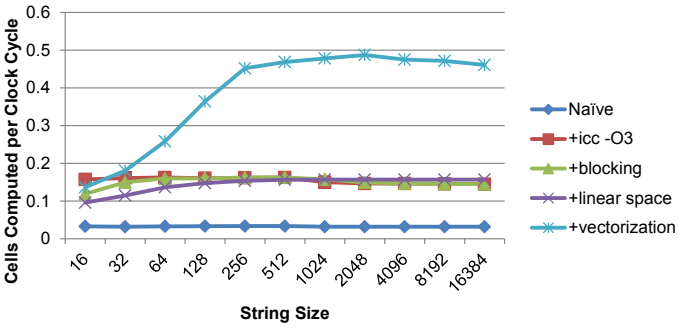

Fig. 12: Throughput performance for x86 optimizations.

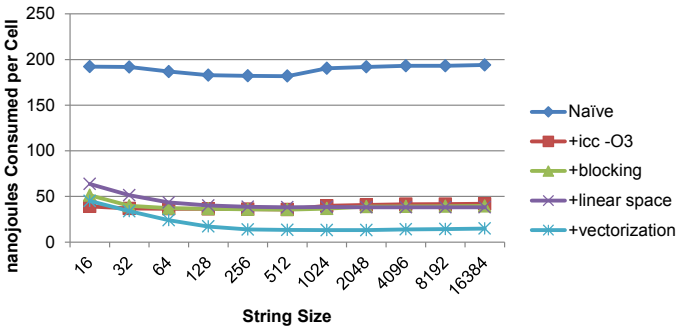

Fig. 13: Energy consumption for x86 optimizations.

effort to optimize the x86 version of edit distance. For each of the optimizations mentioned in Section IV, we analyzed the performance and energy consumption properties of progressively adding compiler optimizations, blocking, linear memory space, and vectorization. Figure 12 presents the performance improvement using the throughput metric cells/cycle while Figure 13 presents the energy consumption in nanoJoules per cell computation.

Overall, we find that the *icc* compiler $-O3$ optimization pass provides a 4.8× performance improvement by reducing a substantial amount of dynamic instructions required, increasing the number of cells calculated per clock cycle from 0.03 to 0.16. Similarly, the $-O3$ optimization pass provided a 5.1× reduction in energy consumption. The blocking and linear memory space optimizations are required to enable multi-threaded parallelization (evaluated later in this section) and reduce the memory footprint needed for the cost matrix. We find that the overhead required for each of these optimizations is sufficiently large to reduce performance at string lengths less than 256. Vectorization provides a substantial improvement, as performance is increased over the prior optimizations by 2.8×, while energy consumption is reduced by 2.6×. Furthermore, we find that given our vectorization strategy, larger input strings more readily take advantage of the performance improvement potential, with performance leveling off at string lengths above 256. Finally, our best performing single-threaded x86 implementation with all optimizations considered is able to calculate 0.48 cells per cycle, while consuming 13.1 nanoJoules of energy per cell.

C. Comparison of TIA to x86

We compare our TIA edit distance implementations to the best x86 implementation. Leveraging our blocking optimization, we compare from one to four OpenMP x86 threads

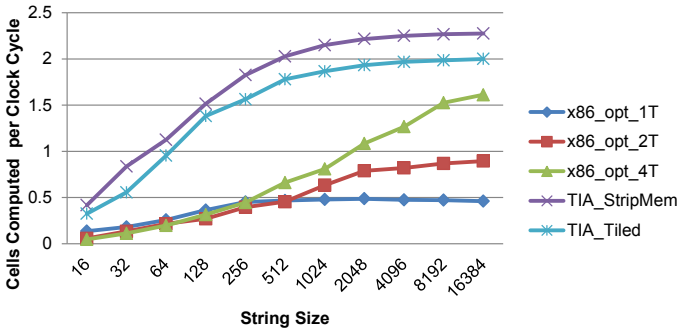

Fig. 14: Throughput performance comparison. Results are not area normalized. Presented are 1-, 2-, and 4- core x86 configurations and a TIA Block (1/12 area of x86 core).

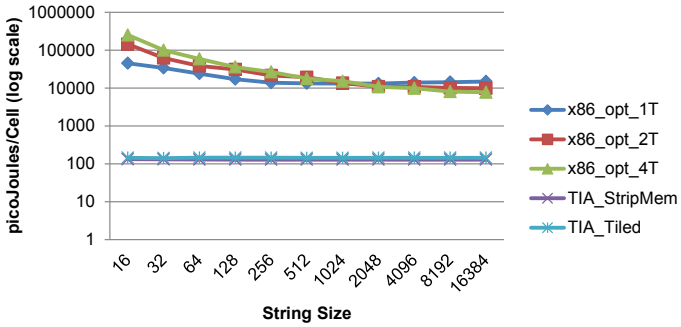

Fig. 15: Energy consumption comparison.

working together on a single edit distance problem. Each thread is bound to a separate core, enabling the private use of both the L1 and L2 caches. Note that the TIA versions are implemented on a single block, which is approximately 1/12 the area of a single x86 core.

Figure 14 illustrates that our TIA implementations have much better performance than the optimized x86 implementation. When compared to a single x86 thread, our one block TIA implementation (in 1/12 the area) has 4.9× better throughput, 0.48 cells per cycle versus 2.2 cells per cycle. Note that because our implementation works on a single edit distance problem, this performance also translates to a 4.9× reduction in runtime. Even when compared to a four-core x86 implementation (single thread per core), we find that the one block TIA implementation maintains a 39% advantage. Scaling TIA to the same area as four x86 cores results in a TIA performance that is more than 58× better than the x86 implementation. Overall, TIA and spatial architectures in general are able to express and exploit significant amounts of fine-grained parallelism unavailable to non-spatial architectures.

The energy efficiency advantage of the TIA implementation is nearly twice the performance advantage. Figure 15 shows that the energy consumption per cell calculation of our TIA implementations is more than two orders of magnitude less than the x86 version. This significant benefit corresponds both to the low complexity of a PE relative to an x86 core, which results in much lower energy cost per operation, and the aggressive conversion from expensive memory operations into inexpensive local communication.

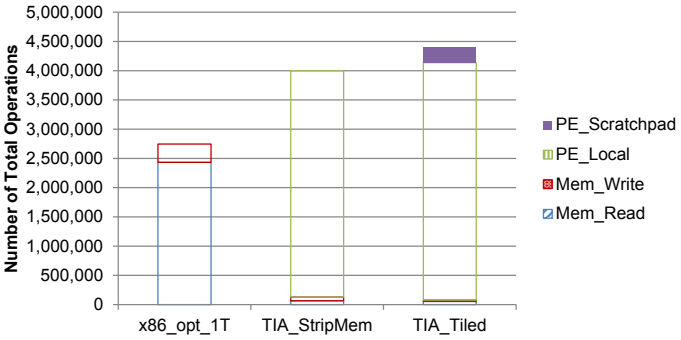

Fig. 16: Comparison of memory, local communication, and scratchpad memory accesses between x86 and TIA. While register file activity is not shown, TIA local communication numbers include what would be register file accesses on x86.

D. Memory References and Communication

Figure 16 compares the number of memory accesses, local communication between PEs, and scratchpad memory accesses between x86 and TIA. In this analysis, we compare our fully-optimized single-threaded x86 implementation with two of our TIA implementations on a dataset where strings S and T are both of length 1024. Activity numbers for TIA were gathered from the performance model, while the number of memory accesses on x86 were gathered using cachegrind, a performance instrumentation tool from Linux [19]. We find that over 95.2% and 97.3% of memory accesses in both TIA implementations are successfully converted into less expensive local communication between PEs and scratchpad memory accesses. The dramatic reduction in memory accesses is a key benefit in accelerating applications using spatial architectures. Note that the amount of local communication is on the order of the number of memory accesses in the x86 version. While register file accesses are not explicitly shown, the TIA local communication numbers include accesses that would be facilitated on x86 using the register file. This conversion of memory accesses to local communication corroborates well with the organization of our TIA implementations, where local communication is primarily used to send scores between PEs.

E. Coding Effort Analysis

To further compare the two platforms, we analyzed the code footprint of the x86 and TIA kernels. Performing this analysis provides further insight into the natural mapping of edit distance onto spatial architectures. Table III presents the lines of source code for x86 (C++) and TIA (Assembly). For this analysis, we only included lines in each kernel containing real work, and excluded lines entirely devoted to comments, whitespace characters, and control characters such as braces {}. For the C++ x86 version, we further optimized code footprint by aggressively modularizing code into reusable functions that could be inlined by the compiler.

We find that the number of lines of TIA assembly is nearly on the order of the fully optimized C++ x86 version. This is an interesting result, given the superior expressibility of C++ for computation. However, the effort required to optimize the x86 version for scalability and performance adds significant code complexity. While scaling the number of row workers is trivial in TIA, x86 must change the algorithm and implement blocking and OpenMP support to facilitate parallelization. Similarly, implementing vectorization on x86 results in an algorithmic change and corner cases that must be handled with additional code. Overall, we feel that this data analysis reflects the natural mapping of edit distance onto TIA.

Platform	Version	Lines of source code
x86 (C++)	Naive	10
	+blocking	65
	+linear memory space	144
	+vectorization	168
TIA (Assembly)	StripMem	222
	Tiled	313

TABLE III: Coding effort for edit distance (code footprint)

VI. Related Work

There is much prior research in developing and optimizing edit distance and its several variants on general-purpose processors such as x86. Hirschberg [12] first discovered a linear-space algorithm for sequence alignment, which was then popularized and extended by Myers and Miller [13]. There are also several multicore based implementations of edit distance as well as sequence alignment. In [14] the authors present a cache-oblivious divide-and-conquer algorithm for multicores, where the cost matrix is divided into four quadrants to be solved recursively, and the diagonal quadrants are solved in parallel. Other prior research on edit distance has focused on parallelization targeting both MIMD [20] and SIMD [21] architectures. Vectorization of the sequence alignment problem by reshaping the cost matrix has been described in [22].

With the recent surge of research on accelerator architectures, edit distance and its variants have also been mapped to GPUs, FPGAs, and reconfigurable hardware. This prior research focuses on mapping to the data-parallel nature of the architectures, with optimization utilizing inter- and intra- task parallelism, tiling, pruning, and approximate solutions. Given the difficulty in vectorizing edit distance, much of this prior research finds parallelism by performing computation across multiple sets of small gene sequences [23], [24]. However, there exists some research which focuses on improving performance for a single large sequence alignment problem [25]. Some FPGA and reconfigurable hardware research leverage the strong dataflow properties within the algorithm in order to exploit parallelism [26], [27], [28]. While much of this research focuses on improving throughput performance, some work also emphasizes reducing logic footprint [29].

In contrast to this paper, no prior research focuses on analyzing energy or power consumption of edit distance, or compares fully-optimized implementations on two platforms with different architectural properties. We develop edit distance algorithms for the emerging class of coarse-grained spatial architectures, and leverage best-known techniques to develop an optimized x86 implementation of edit distance. We concentrate on developing a scalable implementation for spatial architectures that operates on arbitrarily large source and target strings while minimizing memory bandwidth by optimizing communication among the PEs. We also optimize for a single instance of edit distance operating on a single pair of strings, rather than gaining parallelism through multiple computations. Finally, both x86 and TIA implementations are evaluated on high-end evaluation platforms to further ensure a fair comparison.

VII. Conclusion

In this paper we demonstrate the ability of a triggered instruction-based spatial architecture to solve the edit distance problem, a broadly used dynamic programming problem. We exploit inherent dataflow properties and pipelined parallelism to implement edit distance on a previously proposed spatial architecture. We show that it is possible to get more that 50× performance improvement in runtime and a 100× reduction in energy consumption compared to a highly optimized x86 based implementation run on a high-end Intel®x86 processor. This huge acceleration essentially comes from the exploitation of very fine-grained parallelism and the dramatic reduction of memory reads and writes. Our experiments show that this proposed spatial architecture has a tremendous potential for providing high performance for applications with local communications. We conclude that for applications where vectorization is not straightforward or inefficient due to horizontal and vertical dependencies between the computation elements, it is possible to map them to a spatial architecture more efficiently than an x86 processor.

Acknowledgments

We greatly appreciated the suggestions of Michael Pellauer, Angshuman Parashar, Mohit Gambhir, and Matthew Frank during development of the edit distance implementations used in this paper. We thank the VSSAD team for their gracious feedback on experimental results and early versions of this paper. Finally, we thank the anonymous reviewers for the useful comments in developing the final version of the paper.

References

[1] G. E. Blelloch and M. Reid-Miller, "Pipelining with futures," in *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1997, pp. 249–259.

[2] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang, "On-the-fly pipeline parallelism," in *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 2013, pp. 140–151.

[3] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: a control paradigm for spatially-programmed architectures," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 142–153.

[4] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey Of Systems and Software," *ACM Computer Survey*, vol. 34, no. 2, pp. 171–210, 2002.

[5] A. Marquardt, V. Betz, and J. Rose, "Speed and Area Tradeoffs in Cluster-Based FPGA Architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 84 –93, Feb. 2000.

[6] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1996, pp. 157–166.

[7] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997, pp. 12–21.

[8] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2003, pp. 61–70.

[9] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor, "PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 2002, pp. 63–66.

[10] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011.

[11] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar Architecture," *ACM Transactions on Computer Systems*, vol. 25, no. 2, pp. 4:1–4:54, May 2007.

[12] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[13] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Computer Applications in the Biosciences*, vol. 4, no. 1, pp. 11–17, 1988.

[14] R. A. Chowdhury, H.-S. Le, and V. Ramachandran, "Cache-oblivious dynamic programming for bioinformatics," *Transactions on Computational Biology and Bioinformatics*, vol. 7, no. 3, pp. 495–510, 2010.

[15] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete algorithm*, 2006, pp. 591–600.

[16] J. S. Emer, P. S. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. L. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework." *IEEE Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[17] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proceedings of the International Symposium on Microarchitecture*, 2007, pp. 3–14.

[18] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the International Conference on Parallel Processing Workshops*, 2010, pp. 207–216.

[19] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[20] X. Huang, W. Miller, S. Schwartz, and R. C. Hardison, "Parallelization of a local similarity algorithm," *Computer applications in the biosciences: CABIOS*, vol. 8, no. 2, pp. 155–165, 1992.

[21] J. A. Grice, R. Hughey, and D. Speck, "Parallel sequence alignment in limited space." in *Proceedings of Intelligent Systems for Molecular Biology*, 1995, pp. 145–153.

[22] A. Heilper and D. Markman, "Vectorization of sequence alignment computation using distance matrix reshaping," no. US Patent 7343249, 2008.

[23] R. Farivar, H. Kharbanda, S. Venkataraman, and R. H. Campbell, "An algorithm for fast edit distance computation on gpus," in *Proceeding of Innovative Parallel Computing*, 2012, pp. 1–9.

[24] E. F. de O.Sandes and A. C. M. de Melo, "Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 1009–1021, 2013.

[25] M. Korpar and M. Sikic, "Sw# - gpu enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, 2013.

[26] S. Dydel and P. Bała, "Large scale protein sequence alignment using fpga reprogrammable logic devices," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*. Springer, 2004, pp. 23–32.

[27] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman, "A run-time reconfigurable system for gene-sequence searching," in *Proceedings of the 16th Annual VLSI*, 2003, pp. 561–566.

[28] B. Sahoo, T. Swarnkar, and S. Padhy, "Implementation of parallel edit distance algorithm for protein sequences using reconfigurable accelerator," in *Proceedings of the International Conference on Advances in Computing, Communication and Control*, 2009, pp. 26–29.

[29] K. B. Kent, R. B. Proudfoot, and Y. Zhao, "Parameter-specific fpga implementation of edit-distance calculation," in *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, 2006, pp. 209–215.