Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Final Report for 15-618 Final Project

“Sequence Alignment Using AWS Trainium”

Summary

We implemented the Smith–Waterman algorithm for sequence alignment on the AWS Trainium accelerator. Taking advantage of Trainium’s highly heterogeneous architecture, we demonstrated several levels of parallelism:

- SIMD-parallelism on the Vector and Scalar Engines
- Pipeline parallelism between the Vector, Scalar, and Tensor Engines
- Systolic-array-based matrix multiplication on the Tensor Engine

Additionally, our algorithm is capable of SPMD-parallelism across multiple Neuron Cores if run on larger instances[1].

Background

Our project implements the Smith–Waterman (SW) algorithm for local sequence alignment. This algorithm has significant importance in computational biology as it is used for identifying common parts of different biological sequences, like DNA/RNA strings. Compared to global alignment algorithms, such as Needleman-Wunsch, which try to find the way to best match two entire strings, Smith-Waterman focuses on finding the parts of two strings that match the best.

Specifically, in this project, we focus on the variation with an affine gap penalty. The SW algorithm technically supports any function W(k) for calculating a gap penalty, but in practice, affine gap penalties in the form $W(k) = \alpha k + \beta$, are used most often, and allow for the simplification of the general SW algorithm.

Inputs

1. Two sequences of a given alphabet Σ: $Q=(q_{1, \dots,} q_m)$, $S=(s_{1, \dots ,} s_n)$. These alphabets vary depending on the specific application, but they are usually just letters
2. Substitution penalty matrix of size |Σ| x |Σ| - “cost” of mismatch of two given characters in aligned sequences. There are a variety of penalty matrices, again depending on the use case, but we test both a simple version and BLOSUM62, a widely used matrix
3. Gap penalties α and β - local alignment specific parameters; the cost of “opening” and “extending” a gap, respectively

[1] Large instance with Systolic Array of Neuron Chips, trn1.32xlarge, is not available for experiments (pre-booked capacity only)

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Key data structures

SW is a dynamic programming problem. At a high level (further detail in the next section), the algorithm constructs a scoring matrix, H, and iteratively fills in each cell of H, by looking at the cells above and to the left of it. It uses two other matrices, E and F, to hold intermediate data.

DP subproblems:

- *H(i,j)*: The optimal local alignment score of the sequences ending at (i,j) for the first and second sequences, respectively
- *E(i,j)*: The best score ending at (i,j) where the alignment ends with a gap in sequence Q
- *F(i,j)*: The best score ending at (i,j) where the alignment ends with a gap in sequence S
- Substitution score matrix between i character in Q and j character in S: $\sigma(q_i, s_j)$

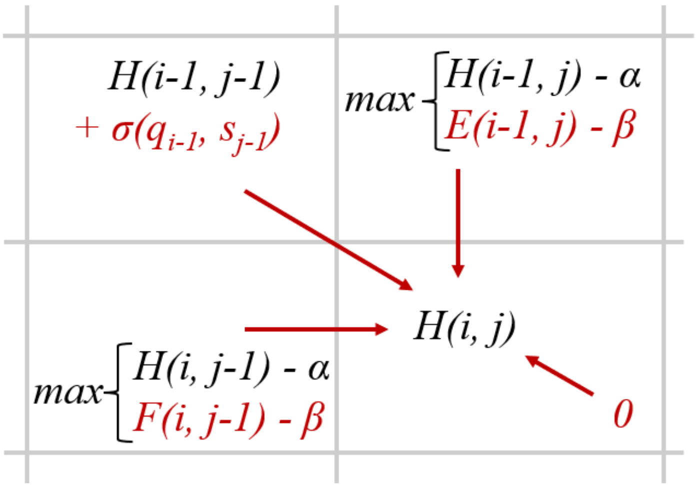
Algorithm

Recurrence dependencies:

$$H(i, j) = \max \{ H(i-1, j-1) + \sigma(q_{i-1}, s_{j-1}) \\ E(i, j) \\ F(i, j) \\ 0$$

$$E(i, j) = \max \{ E(i-1, j) - \beta \\ H(i-1, j) - \alpha$$

$$F(i, j) = \max \{ F(i, j-1) - \beta \\ H(i, j-1) - \alpha$$


$H(i-1, j-1)$
$+ \sigma(q_{i-1}, s_{j-1})$
$\max \begin{cases} H(i-1, j) - \alpha \\ E(i-1, j) - \beta \end{cases}$
$H(i, j)$
$\max \begin{cases} H(i, j-1) - \alpha \\ F(i, j-1) - \beta \end{cases}$
0

Boundary conditions:

$H(0,0) = H(i,0) = H(0,j) = 0;$ $E(0,0) = F(0,0) = F(i,0) = E(0,j) = -\infty$

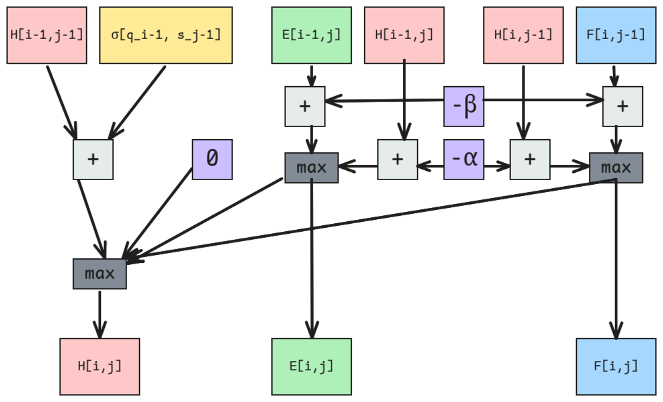
$E(i,0) = -\alpha - (i - 1) \cdot \beta;$ $F(0,j) = -\alpha - (j - 1) \cdot \beta$

After solving the DP problem by filling out these matrices, the algorithm executes a Traceback:

- Finding the maximum value of H(i, j)

- Traversing back through the matrices H, E, and F to find the start of the optimally aligned sequences. We mostly omit the details of the traceback algorithm, as this part can run O(m+n) time, and is generally ignored in terms of parallelization
- The output from this traceback is two sets of indices, one for each string, indicating the parts of the strings that most closely match. Using the traceback process, we can annotate the indices to show where there is a substitution or where there are gaps

Key bottleneck



Data dependency diagram for the above algorithm, during one iteration

The main operations involved are:

- **Matrix initialization** for H, E, and F following the boundary conditions
- **Matrix filling** according to the recurrence relations above
- **Traceback** from the highest-scoring H(i,j) to reconstruct the optimal local alignment

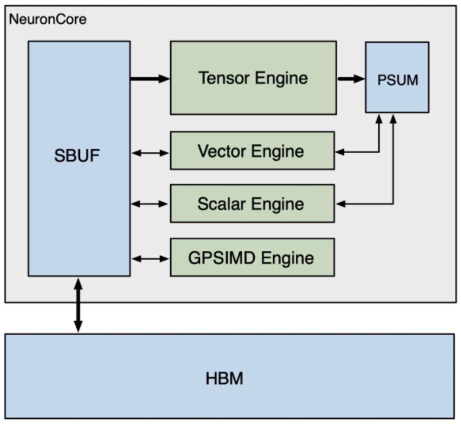
The **computational bottleneck** lies in filling the matrices, which requires O(mn) operations and heavily relies on element-wise max and addition operations. Moreover, the algorithm is inherently iterative. As seen in the above diagram, to compute one element, we need the element above it, to the left, and diagonally left and up of it, meaning the usual SIMD parallelism is not obviously ideal, as we cannot simply loop over either rows/columns. Thus, we focused on parallelizing this part of the algorithm. But *before we talk about parallelization opportunities* for this particular algorithm, we need to give an overview of the parallelization opportunities provided by Trainium.

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Approach

Overview of Trainium

AWS Trainium is part of a new class of machine learning accelerators designed with a highly heterogeneous architecture to maximize computational throughput for ML workloads. Its key computational element is a Neuron Core, which consists of:

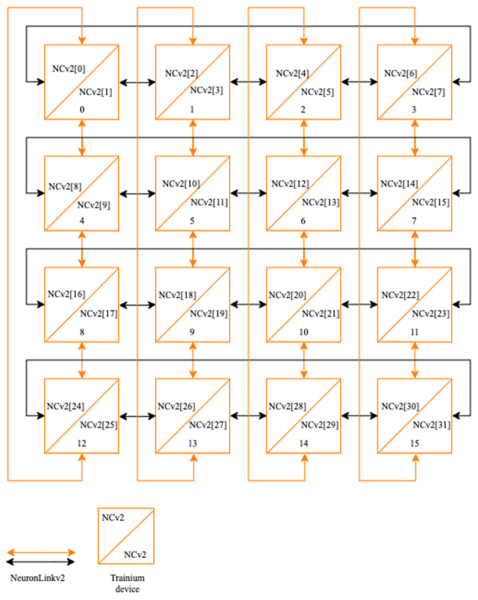


- 4 engines: Tensor Engine, Vector Engine, Scalar Engine, and GPSIMD Engine
- 2 on-chip memory buffers: SBUF (shared buffer) for inter-engine communication and fast local data access, and PSUM unit for accumulating partial sums generated by the Tensor Engine.

The Tensor engine is the most performant, operating at 2x the clock speed of the other engines.

Neuron Cores are backed by a high-bandwidth shared HBM memory pool, enabling efficient large-scale data movement.

Trainium devices combine Neuron Cores into larger compute structures:



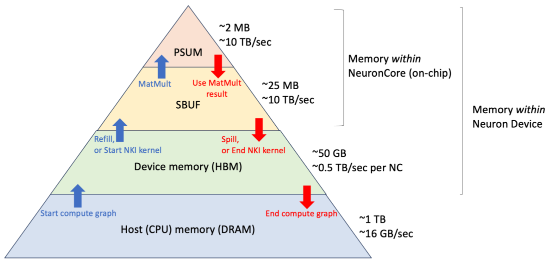
- Each Trainium chip integrates two Neuron Cores connected via shared memory (32 GB HBM)
- Full Trainium instances are built as 4×4 systolic arrays of chips (16 chips total) linked with a high-speed interconnect NeuronLink-v3

This architecture **enables various types of parallelism**:

4. Intra-core computational parallelism
    a. Tensor Engine: Systolic-array-based matrix multiplication
    b. Vector Engine: SIMD-parallel operations on vectors of up to 128 elements
    c. Scalar Engine: SIMD-parallel operations between vectors and scalars
    d. GPSIMD Engine: Custom CUDA-style programmable SIMD kernels
    e. Pipeline parallelism between different engines

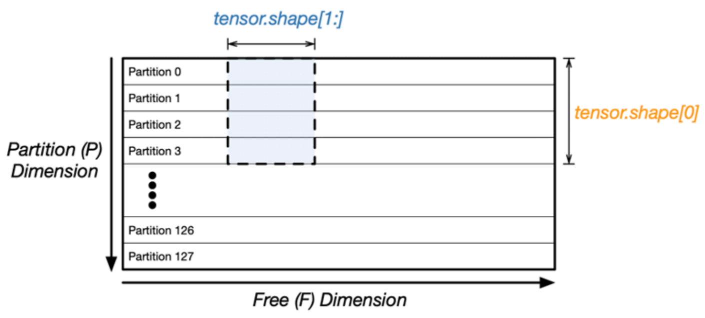
5. Row-level data-parallelism via 2D memory layout
6. SPMD-parallelism between Neuron Cores (NC) reinforced by fast interconnect:
    a. 2 NC on one Trainium chip with 32 GB of shared memory
    b. 4x4 systolic array of Trainium chips forming a large Trainium instance

A notable feature of the NeuronChips is their **unique memory layout and hierarchy**. In contrast to the analysis and intuition we've built up in class, Trainium does a few things differently.



In the NeuronChips, there are 4 levels to the memory hierarchy: DRAM, HBM, SBUF, and PSUM. For the most part, DRAM is invisible to the programmer, and HBM is where all kernels initially read from and finally write to. Both of these are 1D memory - so, like in class, we have the same sort of access issues and patterns to keep in mind, for example, a value in the same column but next row might be very far away in memory. One other thing to note is that HBM is both very big and fairly fast, meaning that most memory is kept here, with only what we need imminently being in SBUF.

SBUF and PSUM are the areas of memory meant for computation, with PSUM being specifically for MatMult results (having the property of being very easy to compute Partial SUMs). Both of these have a **2D memory layout**, provided by the hardware. This means a value in the next row is actually *guaranteed* to be close. Moreover, the hardware provides row-parallel access to the data. The below image, from the docs, demonstrates how a 2D tensor is stored - it is physically placed over multiple rows. The (128 at most) different rows are called *partitions*, leading to the rows being called the *partition dimension* and the columns being called the *free dimension*. Inherently, we get free row-parallel access, albeit with some limitations, discussed later.



Overview of Neuron Kernel Interface

AWS Trainium was originally designed to expose only high-level interfaces for machine learning frameworks such as PyTorch, TensorFlow, and JAX. As recently as September

2024, Neuron Kernel Interface (NKI) was introduced as a low-level programming API for Trainium.

NKI gives developers fine-grained control over Trainium's engines, allowing custom kernel development optimized for specific workloads beyond standard ML.

Due to its novelty, at this stage of development, NKI **faces several limitations**:

1. No support for custom operations on the Tensor Engine – only matrix multiplications are allowed
2. No single-row operations on the Vector Engine – only chunked operations with sizes divisible by 32. This is a property of the row-parallel hardware
3. GPSIMD Engine is not programmable via NKI
4. No support for custom pipelining across different engines
5. NKI compiler is heavily optimized for ML workloads, leading to extremely slow compile times due to exhaustive analysis.

Another notable limitation of AWS Trainium (not related to NKI) that is relevant for our work is that **large instances with systolic arrays of Neuron Chips are not available for experiments** and require pre-booked capacity.

NKI is a Python-like language, with two main interfaces - nki.language and nki.isa. The former is a high-level interface to program, providing simple functions that get compiled to nki.isa operations. These isa operations provide much lower-level access to the actual Instruction Set Architecture of the NeuronChips, at the cost of being more complicated. Most of the syntax is based on numpy-like operations.

The NKI compiler is a big part of the framework, compiling the high-level operations by determining all memory layouts in advance, pre-allocating space, and simulating parts of the program to accurately determine what memory is needed. This partially means final parallelism is up to the compiler - it will perform both ILP and parallel execution of different data-independent computations, as long as it can detect and support them, which is partly still in development.

Finally, an important part of NKI is its capability for *masking*. For the rest of this document, we will consider the simple square matrix case, where tiles evenly divide the whole matrix. However, in reality, this is usually not the case. Masking allows the programmer to specify to an operation a valid range of the input tile to process; the compiler will, at compile time, determine which parts of the tensor to accordingly actually compute on. Masking is especially interesting in that the compiler will actually *omit* those instructions.

Final Description of the algorithm and mapping to Neuron Core

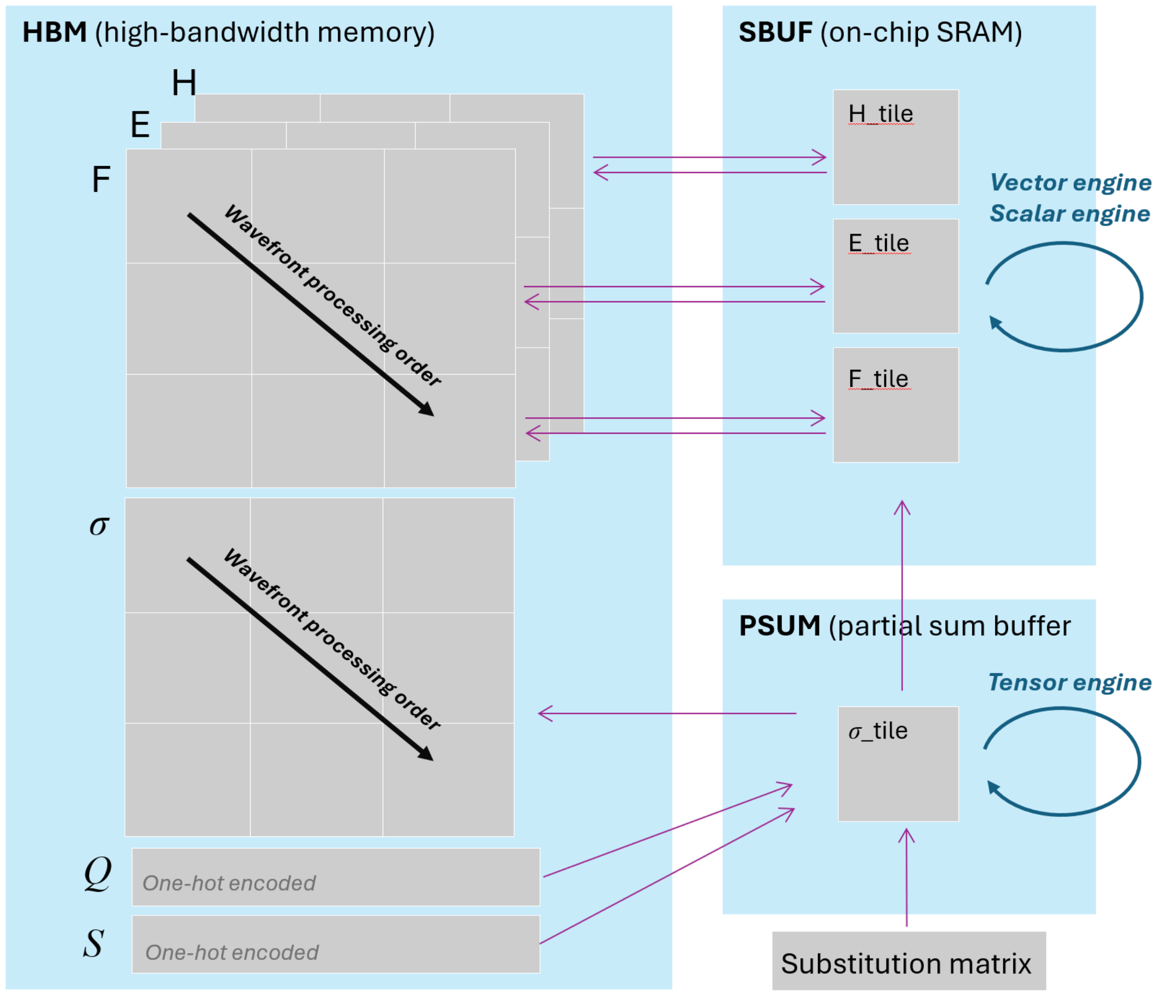
1. Memory allocation:
    - Create arrays for storing the full matrices H, E, F, and the substitution score matrix inside HBM memory
    - These matrices have padding: each is allocated with dimensions $(m+1)\times(n+1)$, where m and n are the lengths of the input sequences Q and S
2. Boundary condition initialization:
    - Loop over 128-row chunks of the arrays E and F (maximum amount of rows that could be stored at SBUF)
    - Each chunk is loaded into SBUF
    - Boundary conditions are computed: first column of E initialized with $-\alpha-(i-1)\times\beta$ (gap open + extension penalty); first row of F initialized with $-\alpha-(j-1)\times\beta$
    - After the update, the modified chunks are stored back in HBM
3. Looping over tiles in a wavefront pattern:
    - Partition H, E, and F matrices into (tile_size+1) chunks to carry boundary conditions
    - Loop over tiles in wavefront pattern - that is, all tiles along the same anti-diagonal are computed together and can be parallelized
4. Loading tiles with boundary overlap:
    - Before processing each tile, load a tile of H, E, and F into SBUF
    - The tile is loaded with overlap: the top row comes from the last row of the tile directly above, the leftmost column comes from the last column of the tile to the left. This guarantees that boundary conditions are properly passed between tiles without additional synchronization
5. Efficient substitution scores tiles computation on Tensor engine:
    - One-hot encodings of sequence chunks and substitution matrix are loaded to PSUM, and multiplied on the Tensor engine to produce a tile of the substitution score matrix
    - Calculated tile of substitution score matrix is loaded to SBUF
6. Per-tile DP calculation:
    - For each tile, compute H(i,j), E(i,j) and F(i,j) based on the standard Smith-Waterman recurrence
    - Store the calculated tile back in HBM.

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

7. Once the full matrices H, E, and F are computed, a traceback is performed

- Find the maximum element in matrix H
- At each step, select to move up, to the left, or diagonally based onthe original recurrence formulas
- Continue the traceback until reaching a cell with zero score, reconstructing the best local alignment

The visual scheme of the algorithm is depicted below:



Note that data movement between PSUM and HBM is represented in a simplified way. In Neuron Core, this data movement happens through the SBUF.

Parallelism in the algorithm

1. Row-parallel computation of H_tile, E_tile and F_tile
2. Systolic-array parallelized calculation (matmult) of σ_tile
3. Pipeline parallelism between Vector, Scalar, and Tensor engines

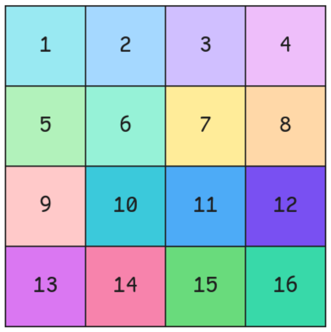
Potentially, this algorithm carries another source of parallelism - parallel computation of tiles on the same wavefront on different Neuron Chips. This parallelism is possible due to the wavefront pattern of the loop over tiles. Unfortunately, a large instance with Systolic Array of Neuron Chips, trn1.32xlarge, is not available for experiments since it requires pre-booking capacity.

Journey (Iterations of the algorithm)

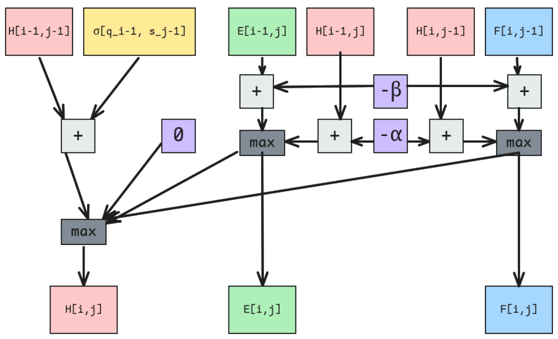
In this section, we'll detail the journey and versions of the algorithm we went through to highlight the parallelism opportunities we took advantage of, but also caveats of NKI that both limit parallelism and enable NKI-specific speed up.

To demonstrate the differences between the various algorithms and showcase parallelism, we have a series of diagrams showing their operation on a 4x4 scoring matrix, but our code can generalize to any number, by doing tiling, which we will explain later. In those diagrams, the number represents the serial timestamp at which that cell will be calculated, and the color represents a logical timestamp of when it will. That is, two squares with the same color but different timestamp could be executed in parallel at the same time, but may not actually be due to lack of resources. In addition, in some diagrams, we use darker shades of colors to represent operations that happen "later" in one logical timestep.

Baseline



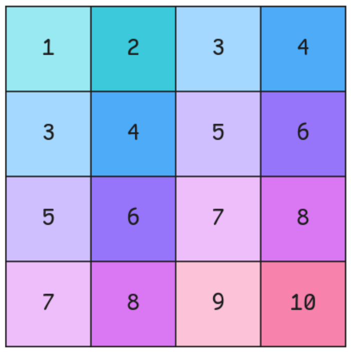
First, we wanted to implement a simple baseline/reference solution to compare our optimized solution against. We chose to do this in Python, using numpy, to quickly and efficiently simulate an algorithm similar to the one we would implement in NKI. NKI itself is based on Python and array accesses are made to be as similar to numpy as possible, so this allowed us to implement a similar style of algorithm.



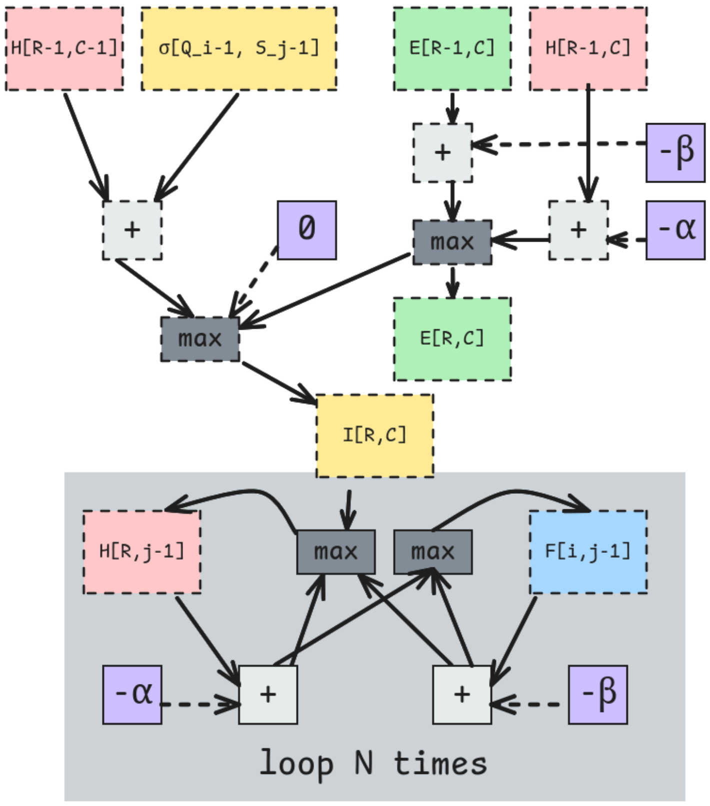
In this algorithm, we simply iterate over the rows, iterating through the columns in each row. This is the regular, standard DP version of the algorithm, with no parallelism. This works because when calculating any cell, we know we calculated the previous row already (because we loop over rows sequentially) and we know we calculated the cells in the preceding columns (because we loop within rows sequentially). This takes 16 timesteps.

The actual calculation of a cell is done as in the algorithm above. More specifically, we again show the data dependency diagram for calculating the new values for a specific cell (i,j).

The maximum length of sequential operations that must happen to calculate the value of a cell is 3 - see the flow from E[i-1,j] or F[i,j-1] to H[i,j]. In our baseline code, however, we do all the operations fully sequentially, rather than parallelizing anything.



To start to put this in a form NKI is more compatible with, we first took inspiration from CUDASW++ 4.0, an existing CUDA library for running the Smith-Waterman algorithm in parallel. In their algorithm, they separate the matrix into stripes of columns; here, we have two stripes - the first two columns, and the second two columns. Starting with the first stripe, in the first logical timestep, we sequentially compute across the first row in that stripe, doing #1 and #2. In the next logical timestep, we can do the next row in the first stripe, but notice that we now have everything we need to calculate the first two cells in the second stripe, so we also do those in this logical timestep. We can continue this pattern, where in the second stripe, we are calculating the cells in the row previous to the first stripe, ensuring that we have calculated the cells in the second column, which we need before calculating any cells in the second stripe. This method allows us to complete the matrix in 10 timesteps.



With this in mind, we put together our baseline sequential version in Python, with numpy. Again, our goal was to have something we could fairly easily translate to NKI. The data dependency diagram is shown on the left.

Dashed boxes represent vectors, dashed operators represent element-wise operations, and dashed arrows represent the broadcasting of constants to vectors.

In this example, let R refer to some specific row number and let C refer to all the elements in the current stripe, respectively. For example, H[R-1,C-1], means the elements in the current stripe in the previous row, but shifted one column to the left. To give a more concrete example, if we are currently looking at the 3rd row, 2nd stripe, where the stripe length is 4, then we are updating the elements H[2,4:8] and H[R-1,C-1] is H[1,3:7]. We discuss boundary conditions above, which allows us to assume we do not go out of bounds of the matrix, but we will also discuss some specifics in a later section.

In this version of the algorithm, we iterate over the rows, and for each row, iterate the over the stripes of columns. For each stripe, we take slices (the top-most vector boxes) out of H, SIGMA, and E and can compute the new values for E and an intermediate vector I that is almost the new part of H, but is missing the inclusion of the F values. We can do this computation in an element-wise way, for efficiency, and we also thought that we could easily parallelize this on NKI.

However, things become tricky when you try to calculate F. Here, we cannot compute the elements all at once doing row-wise operations because F[i,j] depends on H[i,j-1], but H[i,j-1] depends on F[i,j-1], and etc. As such, we must loop N times, where N is the number of elements in the stripe and compute both together. This is inefficient as you must fully sequentially loop over the elements in each stripe for each row.

A First NKI Implementation

Drawing from our baseline implementation above, we started to write our first NKI implementation. Our base idea was the exact same, except for planning to use some built-in NKI ISA instructions (tensor_tensor_scan) to simplify the F/H dependency.

Pseudo-Code

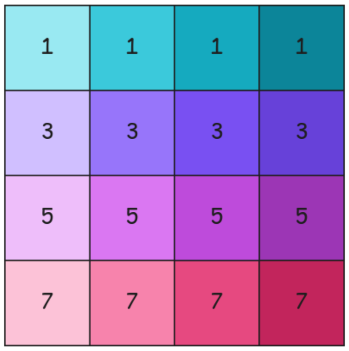
```
4. Read Inputs (sequences and substitution matrix) from file
5. Convert sequences from letters to one-hot vectors
6. Launch Kernel
    a. Initialize H, E, F (see Algorithm)
    b. Given some M_TILE_SIZE (number of rows) and N_TILE_SIZE (number
       of columns), loop over them, row-first
        i. For each tile, load that location of H, E, and F from HBM
        ii. Load the two portions of the one-hot sequences, and calculate
            SIGMA
        iii. For each row in the rile
```

1. Calculate the H/SIGMA term based on the previous row (element-wise)
2. Calculate the new E term based on the previous row (element-wise)
3. Calculate the new F and H terms using the scan

iv. Store the tile back to memory

7. Perform Traceback

As in the diagrams above, this pseudo-code has a flow like the following:



Here, we compute each row in one overall logical step, since we can compute 2 out of the 3 terms in parallel, with the 3rd (the F/H) terms being computed sequentially, leading to the darker colors as you go to the right in one row. It takes roughly two time steps to process each row - one to do both the parallel operations and one to do the scan (in actuality this is a bit more complicated).

NKI provides a simulate_kernel function that allows for development and validation of the computation by simulating the architecture and execution of the program on a CPU. We ran this implementation on that simulation and got the same results as our baseline. However, once we got access to the machines, we ran into multiple issues, which deepened our understanding of NKI.

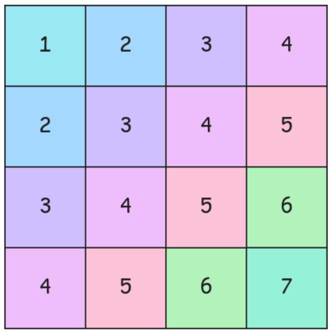
Firstly, both due to hardware and design reasons, it does not make sense to loop row-wise on NKI. In terms of hardware, this is just not supported. You cannot arbitrarily write to random, non 32 (or 64) row-aligned locations in a tensor, due to the row-parallel architecture of NKI. In terms of design reasons, this makes sense, as the point of the 2D memory is to be row-parallel, and writing to a single partition is a waste of parallelism.

Secondly, computation of F and H with nki.isa.tensor_tensor_scan is not easily doable due to the linked computation of both tensors at the same time. We also considered writing a C++ implementation of this operation with the fast NKI C++ Custom Operators, however they are not currently supported by NKI. Finally, we tried converting this algorithm to work with 32-row chunks, and using masking to write the rows accordingly, but this was not supported by NKI nor was it very efficient.

With these setbacks, we went back to the drawing board to come up with an algorithm more suited for the 2D row-parallel architecture that also made use of the different engines.

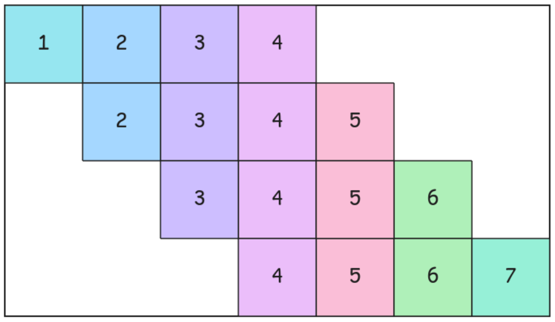
Embracing Wavefronts

Our eventual plan was to use wavefronts at a larger scale to run the tiles in parallel (discussed later), but we realized we could also make use of them now. The wavefront pattern is a parallelism pattern that works well with systolic arrays and dynamic programming problems.



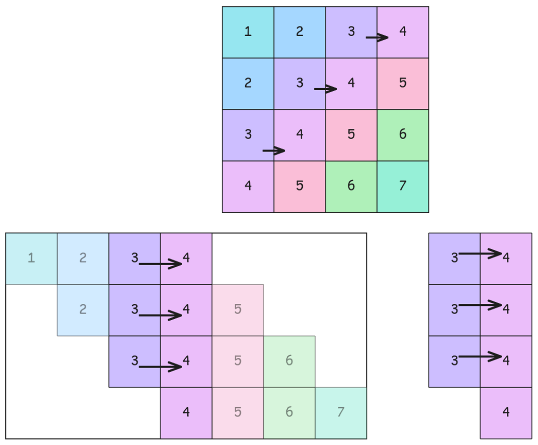
The diagram on the left visualizes this parallel pattern. This pattern is distinguished by the "wavefronts" in which we can run cells in parallel. Note that when we have finished computing cell 1 in this diagram, both cells labeled two can be computed independently. They both only depended on 1 - and the same goes for the cells labeled 3. Once both cells labeled 2 are finished, all cells labeled 3 depend on cells labeled 2 and 1, as they compose as cells to the left (2), above (2) and diagonally to the left and above (1). Thus, all cells in a given wavefront can be run *at the same time*! This provides a significant amount of parallelism (the amount of serial steps comes down to 7), although the amount of cells that can be run in parallel does vary with the actual length of the wavefront. Generally, for a tile of M rows and N columns, there are M+N-1 wavefronts, with each having varying length (a formula for calculating these lengths can be found in the code).

This provides us with significant parallelism, and note that it is actually somewhat similar to the CUDASW++ diagram above, but their wavefronts are further stretched out. This is partly because accessing one wavefront is an expensive operation. Each cell in a different row and column, and on the usual 1D architecture this can be extremely slow as almost every cell in a wavefront would be in a different cache line. This is partly why the CUDASW++ version does not do this - generalizing to stripes of columns has a better memory access pattern.

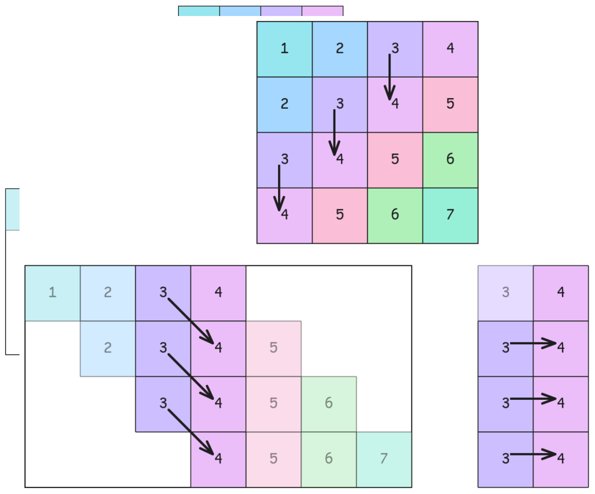


However, this is where the uniqueness of the 2D memory layout in Trainium comes in. Say, somehow, we are able to efficiently rearrange the wavefronts to align every wavefront in one column. We'd get something like the diagram on the left. Our square tile went from M x M to M x (2M-1), but now each column consists of one wavefront. This is a perfect scenario for the NeuronChips - since we can access this memory in row-parallel, we can loop over the columns of a tile and every step compute every value across all rows of that column **in parallel**.

How, exactly, do we do this computation? Can we actually perform these accesses, keeping in mind we are operating in a vectorized memory layout? We can break this down into 3 cases, one for each original recurrence relation. Assume in the following diagrams that H, E, F, and SIGMA have all been converted to above wavefront form, and see the Appendix for specific information regarding header rows/OOB accesses/indexing.



First we consider the computation of the F term. As a reminder, the F term is based on the values in the previous column, as shown in the top part of the diagram. In the lower-left of the diagram we show how this maps into our "wavefronted" tensor - we just need to look directly at the previous elements in the same row, and perform the computation. This is a simple vector access of the previous column, along with masking to omit the last row.



To compute E, we need to look at the previous row. Looking at where those values are in our wavefronted matrix, they are in the previous column, but are shifted up by one row. Since each row is in its own vector lane, and we cannot access starting from an arbitrary row, we had a bit of trouble figuring how to pair up the values to compute element-wise parallel. We then realized we could make use of the TensorEngine - since this is 2x faster, it made sense to utilize it as much as possible, and moreover, its systolic array based architecture is perfect for this operation. We can multiply the previous column by a matrix with 1s on the lower subdiagonal, and we will get a new vector with the elements shifted down, which we can quickly and in parallel, compose with the current column.

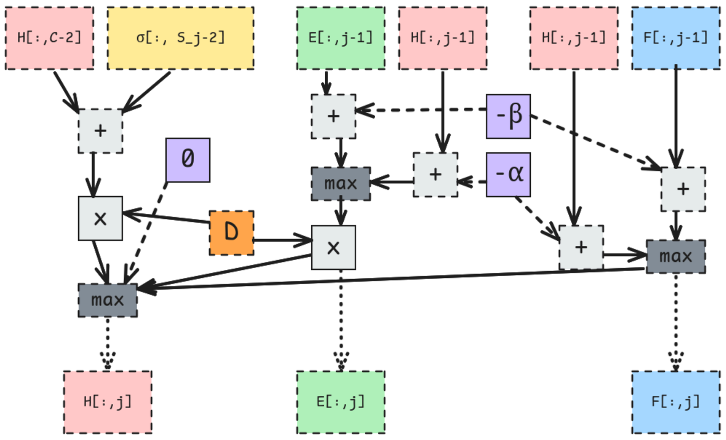
Finally, the last access pattern is accessing the elements diagonally to the left and above. In our wavefronted matrix this becomes access to the elements two columns previous and shifted up by one. We can apply the same trick as with the E matrix, and use a matrix multiplication to shift them down. We mask out the top element, which gets shifted down, and the bottom element, which does not exist.

With this all figured out, we rearchitected our solution to fit this paradigm. The pseudo-code looks like the following:

Pseudo-Code

```
1. Read Inputs (sequences and substitution matrix) from file
2. Convert sequences from letters to one-hot vectors
3. Launch Kernel
    a. Initialize H, E, F (see Algorithm)
    b. Given some M_TILE_SIZE (number of rows) and N_TILE_SIZE (number
       of columns), loop over them, row-first
        i.   For each tile, load that location of H, E, and F from HBM
        ii.  Load the two portions of the one-hot sequences, and calculate
             SIGMA
        iii. Convert them to be "wavefronted"
        iv.  For each column in the tile
            1. Calculate the H/SIGMA term based on the previous
               column(s) (element-wise)
            2. Calculate the new E,F,E term based on the previous
               columns(s) (element-wise)
        v.   Store the tile back to memory, after converting back to
             non-wavefront form
4. Perform Traceback
```

In this algorithm, note that all operations in 3.b.iv can be done in parallel (mostly). To illustrate this, we have the following data dependency diagram.



Here, we are computing entire columns at once. We are performing broadcasts of the constants, but as we will cover in the next section, this can be made very efficient. D is the downshifting matrix, and its inclusion on the dataflow path for E increases the max serial steps to 4 operations to calculate all of the next column. In addition, the path from the old H to the new H has increased by one operation as well. The dotted arrows below indicate masking is done to prevent the overwriting of header rows (see the Appendix).

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

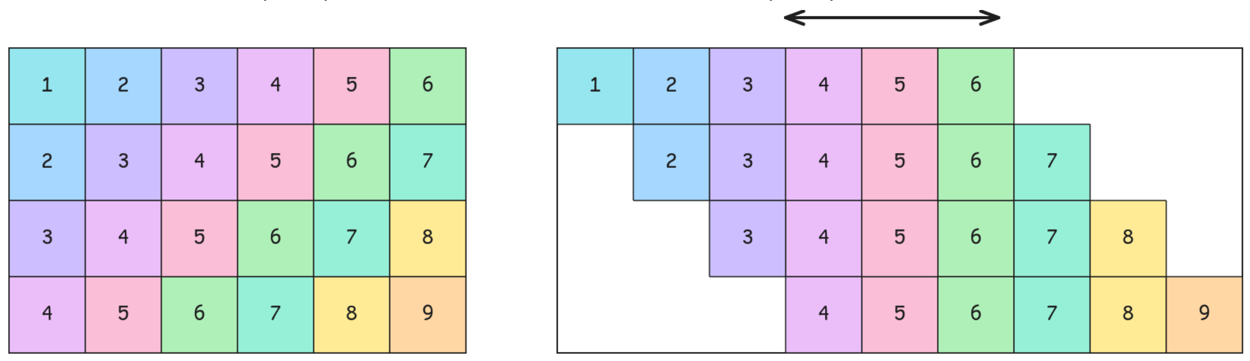
This algorithm is extremely parallel. In proportion to M+N-1 (number of wavefronts), we can compute all the new values, doing every operation in element-wise parallel. However, there is still significant room for optimization.

Optimizations

There are three major optimizations we explored: long tiles, simplifying ISA operations, and wavefront tiles, as well one notable failed one.

For the first, long tiles, let's take a look at the previous wavefront diagram. As mentioned above, the varying amount of parallelism in the wavefront pattern is not ideal. This alone will lead to unideal speedup, as in a square matrix, only 50% of the wavefronted tile has useful information. This is how we came up with our first explored optimization - long tiles.

Consider a rectangular tile with shape M by N. When converting this to a wavefronted form, it has dimension M by M+N-1. It takes M-1 columns for the diagonal to reach the bottom and all above elements to be useful, and the final M-1 columns are the remaining diagonal winding down, which is the right edge of the original matrix. When M+N-1 > 2(M-1), we have M+N-1-2(M-1)=N-M+1 full columns.

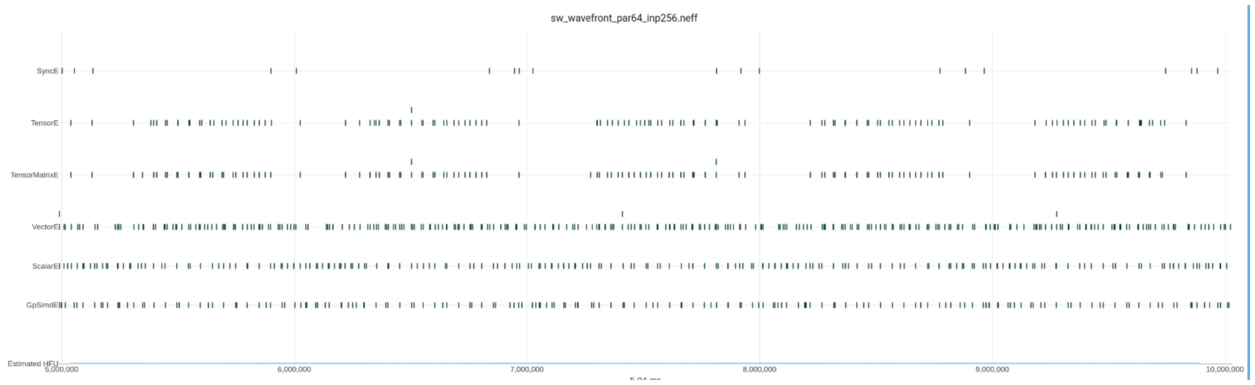


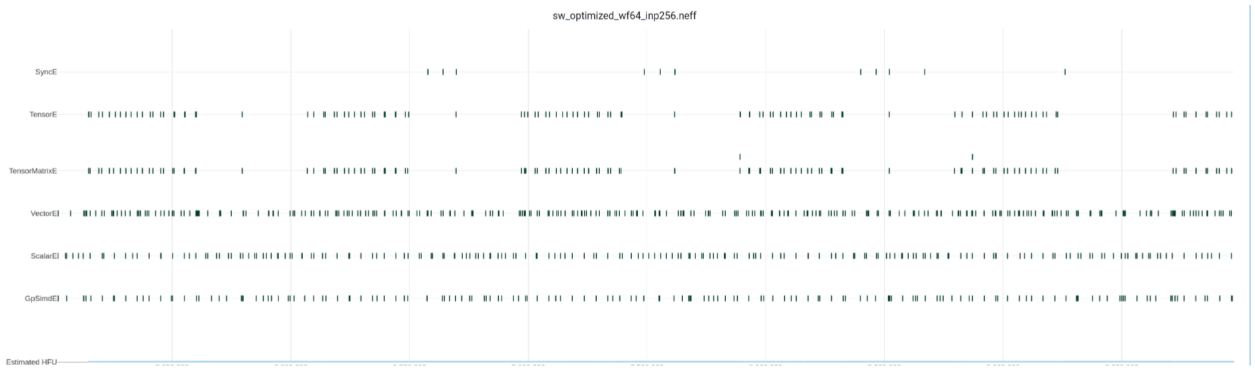
In the above example, we have a matrix of 4 rows and 6 columns, and in the wavefronted matrix we have 6-4+1=3 full columns. It is important to have full columns, especially since we are computing these in row-parallel - it means we have better vector utilization. However, one caveat to keep in mind is that at a larger scale, longer tiles means that we have less overall tiles, reducing the amount of inter-tile parallelism possible, as the amount of sequential operations increase. Running the NKI profiler, we see than when on an input of size 128 by 128, jumping from a tile size of 64 x 64 to a size of 64 x 128 leads the utilization of the vector engine to go from 63% to 65% and the the utilization of the scalar engine (which is also SIMD, just vector-scalar) to jump from 35% to 45%.

Next, we have tile wavefronting. As mentioned earlier, we were originally planning to use the wavefronts at a larger scale - running the tiles in parallel. Like in the original

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

wavefront diagram shown, instead of running the tiles by first iterating over rows and then iterating over columns, we can iterate over wavefronts of tiles and run all tiles in a wavefront in a parallel, and each tile will compute its elements like the above pattern. This is equivalent to changing 3.b to say loop over the wavefronts for the tiles, and then within each wavefront.

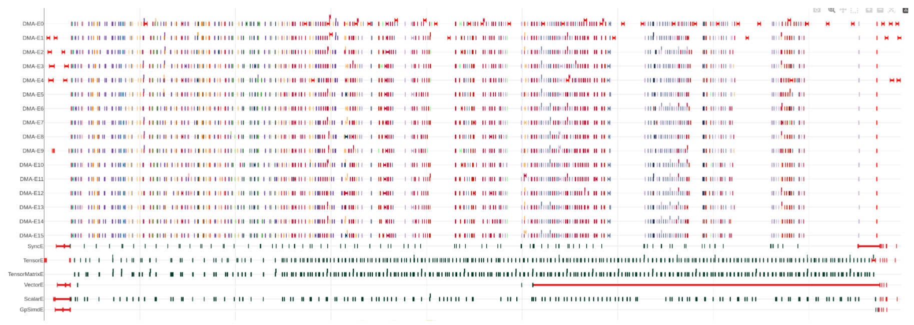




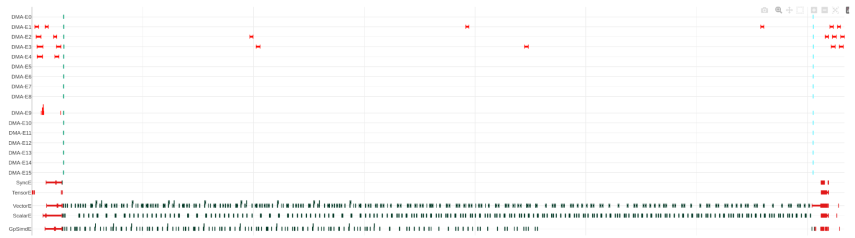
The parallel implementation is the previous picture, and while there are still gaps of idle time in the TensorE execution, we can see these gaps have shrunk and the operations have been spread out in the parallel implementation, implying better pipelining. However, this pipelining is not that much of a boost because we are already maximizing the amount of computation we can do, and the bottleneck is not intra tiles.

Then, there are the ISA optimizations. We were originally planning to do much more of these optimizations, but due to the restructuring of our code, we naturally ended up already converting many nki.language instructions into nki.isa instructions or the compiler did them for us. One notable one was identifying the amount of CAST instruction operations in the profiler. This was when we were using fp16 to represent our data, but, as a hardware restriction the matrix multiplication engine only operates in fp32. This means that the NKI compiler was inserting CASTs between those two data types everywhere, and upon removal, this brought down execution time by 10%. In addition, we wanted to ensure the vector-scalar operations like adding constants alpha and beta were done on the scalar engine, but the compiler implemented these for us.

Finally, one optimization we tried to implement unsuccessfully was a different version of conversion to wavefronts. Currently our wavefront conversion functions loop over the square matrix and for N iterations, we shift the bottom N rows one to the right (or left if converting back). One shift for N rows can be done completely in parallel, so a round trip for conversion takes about N serial steps. However, a significant amount of masking must be done to do this, which leads to some slowdown. We came up with a different method that relied solely on matrix multiplication, in hopes of shifting more computation to the Tensor Engine, to maximize parallel engine execution and use the engines faster utilization. Since we need to shift every row of the matrix by a different amount, we cannot do this with a 2D matrix multiplication, so we broadcasted the matrix into 3D, with each 2D slice a copy of the original matrix, and then, in parallel, shifted every slice by an increasing amount, before using the PSUM buffer to accumulate all of them. However, as seen in the profiling results below, this ended up being a slowdown because of the extra memory accesses needed to compute the 3D matrix - we are spending a lot more time loading data (that is the large amount of loads happening on the top rows called DMA).



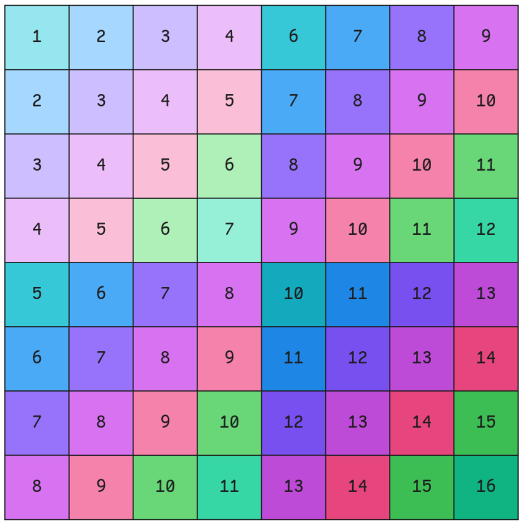
Compare this to the existing implementation, which has poorer tensor engine utilization, but faster overall speed as a result:



Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Tile Pipelining

When we originally introduced Trainium, we discussed how there is an instance (trn1.32xlarge) with a 4x4 grid of NeuronChips, with fast interconnects between them. We originally planned to scale our implementation up to use these chips in parallel, since the memory speeds between them make it feasible for another level of parallelism, but, unfortunately due to AWS EC2 constraints (they must be pre booked and you must wait for availability if/when it is there to our understanding), we were not able to actually implement this. However, we believe the algorithmic aspect is still an interesting avenue of parallelism to discuss, and hopefully implement in the future.



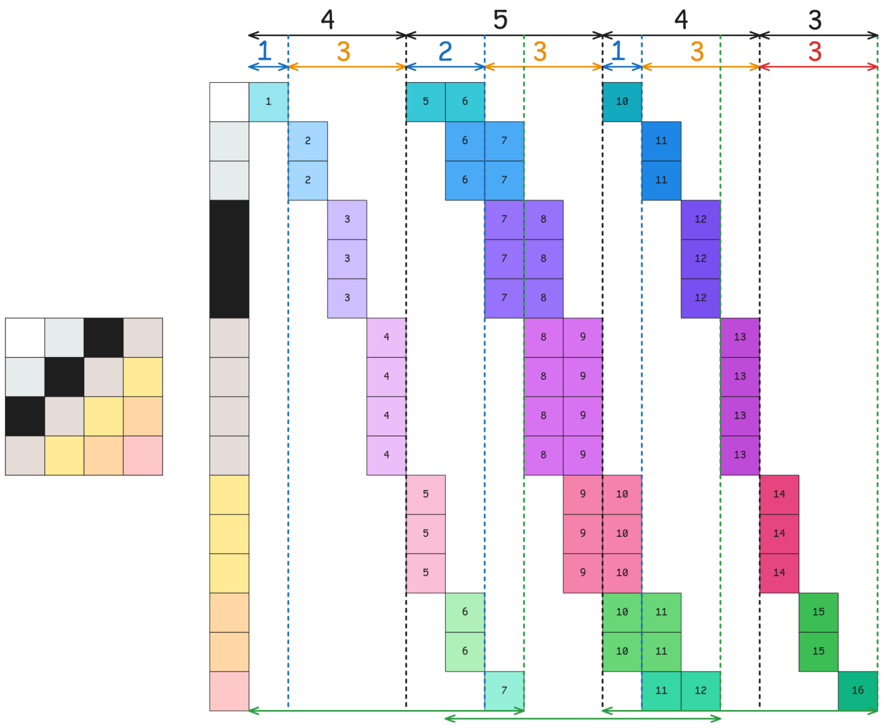
1	2	3	4	6	7	8	9
2	3	4	5	7	8	9	10
3	4	5	6	8	9	10	11
4	5	6	7	9	10	11	12
5	6	7	8	10	11	12	13
6	7	8	9	11	12	13	14
7	8	9	10	12	13	14	15
8	9	10	11	13	14	15	16

For example, say we have a 8x8 matrix, and we want to make use of this 4x4 grid of processing elements (PE). In the original implementation of tiles, we loop over tiles sequentially, and each tile takes 7 serial steps, for a total of 28 serial steps. Even with the tile-level wavefront parallelism above, we saw that one chip did not have enough resources to effectively run everything in parallel. With an 8x8 grid of processing elements, we could run each of the wavefronts in parallel, to finish in 8+8-1=15 serial steps (where one step is the time taken to compute a tile). But, we are limited to a 4x4 grid - how do we most effectively use it?

Here, with a 4x4 grid, we can pipeline the elements in a dilated wavefront way, in order to process all elements as fast as possible. We run the upper-left 4x4 by tile as normal, mapping a cell in that tile to the PE at that location in the 4x4 grid, until we get to the 5th wavefront. Here, the upper left PE in the grid is unused, and both upper-left cells of the bottom-left and upper-right tiles are ready to be computed, so we can start computing them - they make up a dilated wavefront. We can continue this pipelining through the rest of these matrices, making use of as many processing elements as possible, and running the upper-right tile's cells after the bottom-left's. This pattern continues into the bottom-right tile, waiting for the bordering cells to be done.

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Expanding this out into a pipeline diagram, we can get a better idea of what is going on:



Here, we show the PE grid on the left, with colors to represent each wavefront in the PE grid. Then, we expand the grid out into a line, and show which tile from the previous diagram is executing at any given time. Remember, each tile is being computed with a wavefront inside as well. We have 3 separate waves of computation, one for each wavefront. Any wave takes 7 serial tile steps to finish computing fully, but we can start the next wavefront after computing all tiles' cells up their minor diagonal, which is 3 steps (4-1, where 4 is the size of the PE grid, a constant) plus the amount of cells in that timestep. For the final wave, those last 3 iterations have nothing to overlap with.

This produces a total of 3(2M-1)+2(M-1)(M)-M+3 serial tile steps generally, where the first term comes from the number of wavefronts, plus the time taken exclusively by that wavefront. The second term is the sum of the number 1+2+...M+...+1, representing the number of tiles in each wavefront and the 3rd accounts for the last wavefront's sole execution time. M here represents the number of tiles per row and column. This formula simplifies to 2M^2-3M total serial tile steps, which is as pipelined as possible.

Results

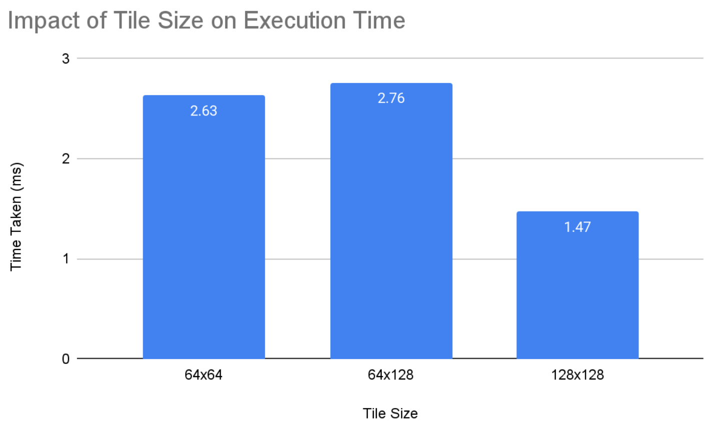
Experimental Setup

We ran all our experiments on an AWS trn1.2xlarge instance, which has 32GB RAM, and one Trainium NeuronChip attached. We ran all of our experiments on one NeuronCore. We validated our programs against our baseline pure-python implementation, and profiled all the kernels with neuron-profile, which provides low-level data about a kernel. One important note is that neuron-profile is the only way to profile/time these kernels and that neuron-profile does not actually provide the data created by the program to the kernel during profiling, but generates random data. This is a limitation of neuron-profile, but should not have an impact on our results as our algorithm's time complexity is data-independent, just based on the size of the input.

Input sequences were generated by a python program we wrote to generate random sequences based on the real alphabet used by the industry standard BLOSUM62 scoring matrix. We wanted to try to run on realistic input sequence lengths, but due to the NKI compiler being suited for machine learning workloads, the compiler makes running high-length workloads prohibitive. A sequence length of 1024x1024 was tested, but this took 5 hours to compile and used 103 GB of RAM. We hope that using a bigger instance in the future will mitigate this issue.

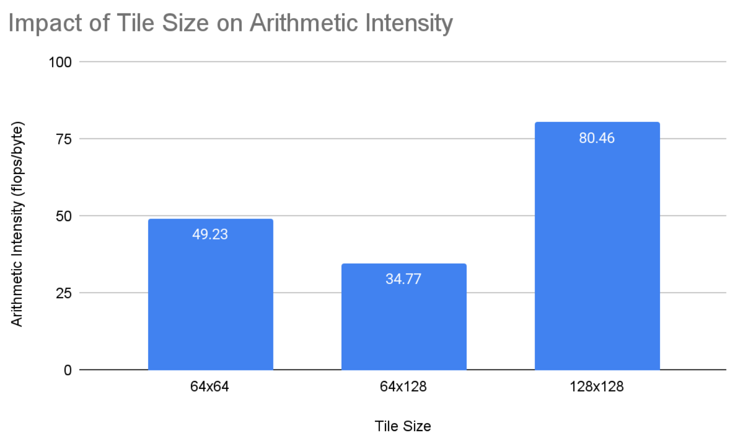
In addition, due to our indexing of the sequence lengths (tile size is one less than what is reported in each dimension to take into account the header rows), we adjust the dimensions of the input to still be the same factor as if the tile size was not adjusted to ensure proper scaling. That is, a tile of 64x64 is an effective 63x63 so for an input size of 128x128 we adjust the input to be 126x126.

Impact of Tile Size



First, we explore the impact of changing the tile size on an input of size 128x128 characters. We look at two key metrics - execution time, measured in milliseconds and arithmetic intensity, measured in flops per byte, which is calculated by neuron-profiler.

We start with the 64x64 tile - it has an execution time of 2.63 milliseconds, and an arithmetic intensity of 49.23 flops/byte. If we



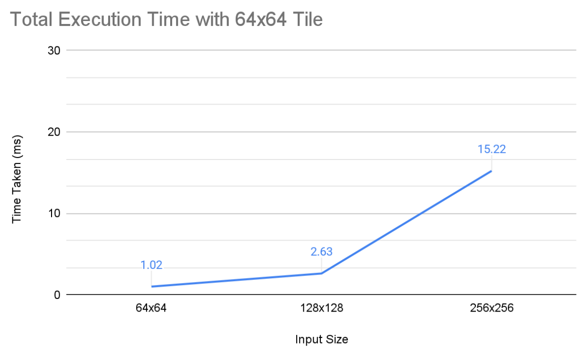
increase the tile width, as mentioned in the long tiles section, we might expect the time, due to better utilization of the SIMD lanes - however, we see the opposite. Interestingly, the arithmetic intensity also heavily decreases.

From examining the profiler, we can see the big difference is that for the 64x128 tile, the Vector Engine takes 0.2 ms longer - this is because we have increased the free dimension, which while it increases utilization, is now inherently a bottleneck for parallelism. Those elements in the free dimension must be executed sequentially, which is especially a big issue for our algorithm when it has to convert those tiles to wavefronts, making the wavefronted tile 64x191, versus 64x127. This means there are more sequential operations that must happen, and with a bigger wavefronted tile, this also explains a lower arithmetic intensity.

For the 128x128 tile size, we can now load the entire input in one tile. In addition, we make full use of the 128-width vector lanes, explaining why our arithmetic intensity jumps up a significant amount - with the same amount of operations, we can do all of the cells in one tile. We also see a 2x speedup, because of the “free” SIMD parallelization we get. We are now looping over the columns of 128x128 matrix, and while we still have to do 128 loop iterations sequentially, we can do both the top 64 and bottom 64 rows at once - which is where the 2x speed up comes from.

Impact of Input Size

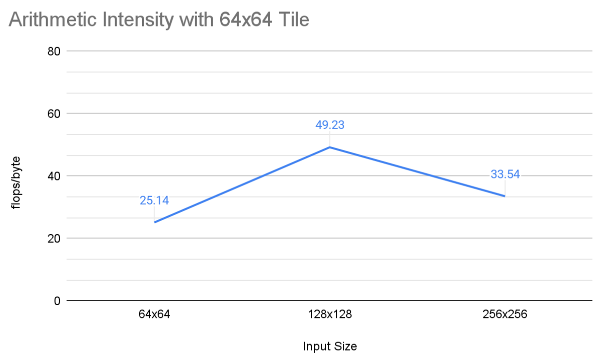
We evaluated the performance of our Smith-Waterman implementation across different input sizes and tile sizes, measuring total execution time and arithmetic intensity (FLOPs/byte).



Key observations for the tile size of 64 x 64 are the following:

- As input size increases, the execution time, at first, grows linearly with input size: 1.02 ms → 2.63 ms (x2.57). It happens because growing lengths of the first sequence to the maximum size of the partition dimension provides "free" SIMD parallelism as described above

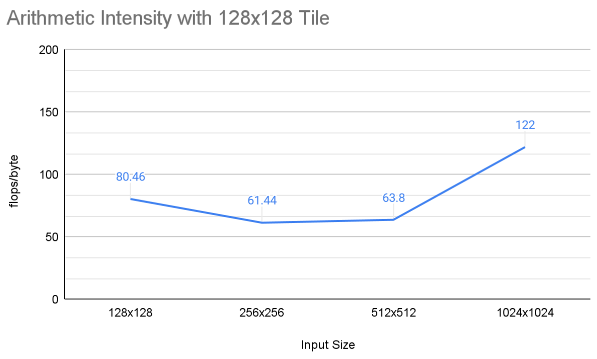
- As input size increases beyond the allowed 128 for the partition dimension, execution time increases more rapidly: 2.63 ms → 15.22 (x5.78). It exceeds expected quadratic growth (number of tiles grows by factor of 4) and is explained by reduced arithmetic intensity

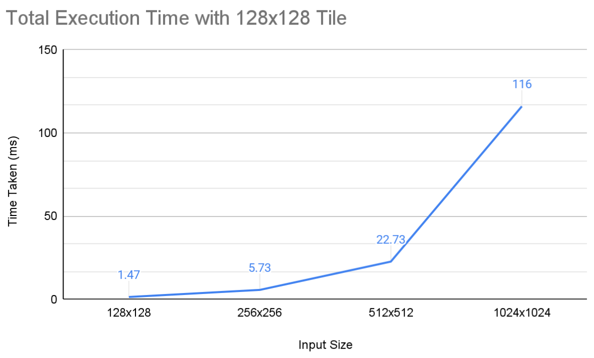


- Arithmetic intensity, at first, grows from 25.14 to 49.23. It happens because of the computation of the substitution score matrix: we load 2 times more data, but produce 4 times as many computations.
- When moving from 128x128 input size to 256x256, the arithmetic intensity decreases. Looking at profiler results, we see that it happens because of the denominator in the arithmetic intensity formula: while the amount of computation grows 5 times (from 16 Mflops to 80 Mflops), the amount of data loaded grows 7 times, from 300 Kbits to 2 Mbits. Profiler hides the exact breakdown of where this growth is coming from, except for the fact that we load chunks of one-hot-encoded sequences more than expected. Thus, we have a hypothesis that while executing transposing (at the calculation of substitution score matrices), the compiler writes some data to HBM. matrices E and F, since smaller tile size causes frequent HBM accesses (they are also redundant, since the access pattern underutilizes vector engine computation capacity). Unfortunately, due to the compiler's automatic pipelining, we do not have access to determine what exactly the issue may be.

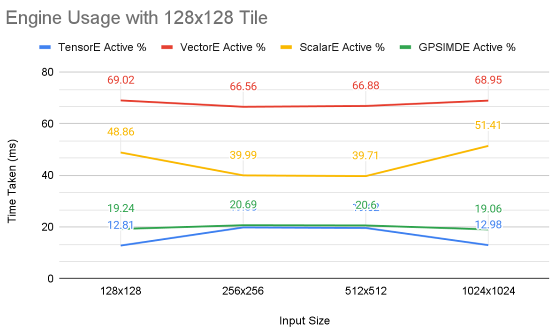
For the tile size of 128 x 128, key observations are:

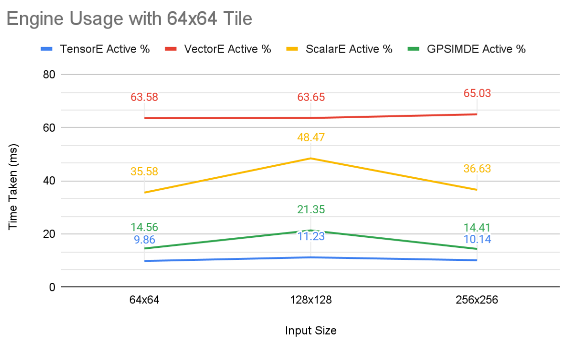
- Scaling input size from 128×128 to 1024×1024 leads to a clear quadratic increase in execution time: from 1.47 ms → 5.73 ms (x3.9) → 22.73 ms (x 4.0) → 116 ms (x 5.1). This increase is fully predictable, since the number of tiles per input grows quadratically.
- Arithmetic drops for 256x256 input size with the reasons similar to those described above for 64 x 64 tile size





Engine Usage Across Input/Tile Size





Since the Neuron Core processors are heterogeneous chips, it is important for us to ensure that we are utilizing each engine as much as possible without sacrificing speed.We see that across all tests, we have good utilization of the Vector Engine, and alright utilization of the scalar engine. We do not utilize the GPSIMD engine as much, since of our core operations (max,addition,masking) the only one that can run efficiently on the GPSIMD engine is masking, which already runs there.

We also do not achieve great utilization on the tensor engine, but this still leads to the fastest runtimes out of everything we've tested. This is because in NKI, performing a

Tensor Engine operation is composed of two operations, loading the weights and then actually performing the multiplication. As mentioned in the optimization section, when we tried to distribute extra work to that engine by doing wavefronting of matrices on it, we saw an overall slow down, partly become of the memory accesses, but diving deeper into the profiler showed an alternating of loading weights and doing matrix multiplications in two different places in the code. The engine was thrashing between the two operations, slowing everything down. As such, out of everything we've tested, despite there being low matrix multiplication usage, we only use it on the hotpath for the downshifting of vectors, meaning the one load can be used for multiple multiplies.

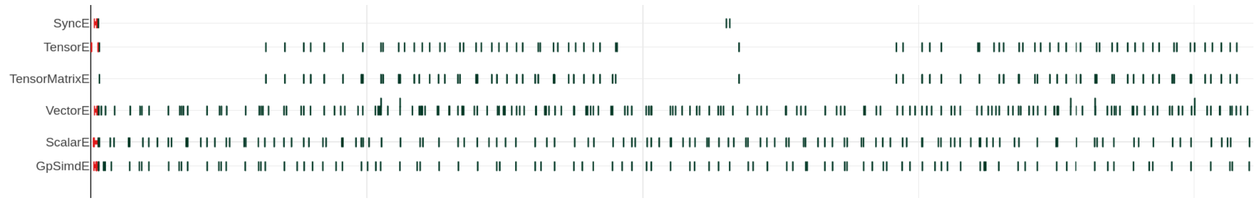
Looking at the overall trends, we see that the vector engine usage stays mostly constant across tile sizes and input sizes, as a significant portion of our computation happens here (element-wise addition of columns), decreasing the possible variance.

Anton Efremov <aefremov@andrew.cmu.edu>
Kandasamy Chokkalingam <kchokkal@andrew.cmu.edu>

Final Bottlenecks & Imperfect Speedup

Overall, we achieve mostly linear speedup for the sizes we have tested. However, we cannot generalize to input sizes beyond, which is where most industry applications are. There are a few execution-specific concerns for imperfect speedup.

The first is that this idea of "wave-fronting" tiles is not as efficient as possible. We see fairly good results, but looking at the profiles, we can see that this is still a bit of a bottleneck:

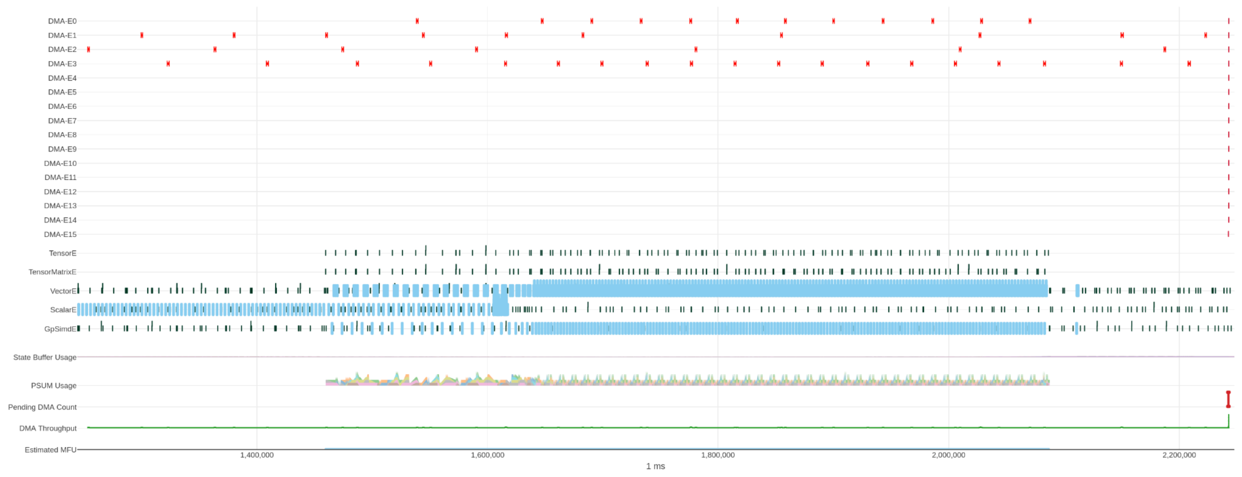


Notice the unused TensorEngine time on either side of the two "blocks" of utilization. Hovering over them, we can see those blocks' operations come from mostly the utilization of the downshifting matrix, and hovering on the vector/scalar/GPSIMD operation between them, we can see they are mostly in the to_wavefront_sbuf and from_wavefrom_sbuf functions. They do not take more time than actual computation itself, but they do take up a significant amount of time as it - about ¾. Improving this time would allow us to achieve more parallelism by allowing the pipelining of these computations, especially between tiles within a wavefront.

Another part of imperfect speedup is the interdependence of the different engines on instructions between them. We can see in the profiler screenshots, all of the engines other than the tensor engine usually do not have idle time. This is good in that we are using the engines mostly fairly, but on the other hand, checking the dependencies between them shows we tend to be waiting for results from other engines very often, which, if we can simplify them, would remove that unnecessary waiting time.

From a platform speedup, not being able to explicitly adjust the pipelining and which computations run asynchronously is a limiting factor in what optimizations we can do. However, as mentioned before and seen in the screenshots of the profiler, we still do see very good instruction level parallelism with the engines being constantly busy, and pipelining in that computations for H, E and F are able to be done in parallel. This is shown in the profiler by hovering through instructions and see non-linear progression through the source code.

In the below screenshot, we've highlighted all instructions that happen on line 52 of the source, which is "X = nl.maximum(nl.subtract(F[:,w-1],beta),H[:,w-1]-alpha)". The scalar operations (the subtraction of alpha and beta), are pipelined with the vector operations (maximum), since the highlights on the two engines overlap.



Finally, it is interesting to note some things that are *not* speedup issues that might traditionally be. We do not have to deal with synchronization/communication, nor do we have divergence because we have very few (just one, triggered only once per block) conditionals. Moreover, cache access patterns are not a concern for us either due to the NKI memory hierarchy - the software managed memory means we can efficiently and in parallel access all of our matrices, and the profiler shows no “spills”, which is the NKI term for running out of memory in SBUF for a computation and having to store intermediate data in the HBM memory.

Conclusion

Overall, we believe that we achieved the goals we set up for this project, despite some setbacks with being able to run problems on the larger machines. We achieved a scalable algorithm that makes good use of the NKI functions and methodology, and saw that our algorithm can effectively execute this SW algorithm in parallel using this wavefront style with SIMD parallelism and the heterogenous engine structure.

Appendix

I. REFERENCES

[1] AWS Neuron Documentation. *AWS Neuron SDK Documentation*. [Online]. Available: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/index.html

[2] J. Xiang, Y. Guo, W. Lin, et al. *"Accelerating Smith–Waterman sequence alignment with sparse dynamic programming and adaptive banded wavefront"*, BMC Bioinformatics, 2024. [Online]. Available: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-024-05965-6

[3] National Center for Biotechnology Information (NCBI). *BLOSUM62 Substitution Matrix*. [Online]. Available: https://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62

II. LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

The total credit for the project is distributed equally, 50%-50%, between the two team members. The work was carried out through a series of joint brainstorming sessions, pair coding efforts, and passing partially completed algorithms back and forth. As a result, there was no strict division into separate workstreams — the development process was highly collaborative at every stage.

III. SPECIFIC ALGORITHMIC DETAILS

In the above document, we mostly skim over the specifics of index accesses, masking, and positioning we do in the code to make the algorithm work, but in some cases (tiling, memory access rules, etc), this is a relevant detail. In this section, we provide a quick overview of those details for more context.

In terms of indexing, in the SW algorithm, there is both a header row and column. We need to ensure we do not actually overwrite these values if they stay constant, so we set up our masks to avoid overwriting the first header row in the wavefronted matrix and everything below and including the major diagonal.

Also as a result, our tiles are effectively one less in each dimension, since that first header row and column is data that must have been computed previously, and which enables us to start compute the cells in this tile. Thus, a 128x128 tile is actually a 127x127 tile. This means our tile loops jump by 128 each time in each dimension, but they get the previous row and column headers, so that they only calculate 127x127 new values.