Needletail: Scalable GPU-Accelerated Global Long Read Alignment

University of Michigan
EECS 570 - Parallel Computer Architecture
Undecided_Group_Name_1
(Authors Are Listed Alphabetically By Surname)

Tim Dunn
timdunn@umich.edu

Nathan Ozog
ozog@umich.edu

Sanjay Singapuram
singam@umich.edu

Nicholas Wendt
nwendt@umich.edu

I. Abstract

The Needleman-Wunsch alignment algorithm was developed in 1970 as a solution to the global alignment problem [1]. This dynamic programming solution, and its derived variants, remain in widespread use today, most commonly to compare biological sequences. Alignment is a compute intensive process, accounting for substantial portions of industry standard genomics tools such as Minimap2 (33.2% runtime) and BWA-MEM (28.4% runtime). In this paper we present Needletail [2], a scalable GPU accelerated long read alignment approach to Needleman-Wunsch. We classify long reads as a query and target length greater than or equal to 2.5kbp. For long reads Needletail has shown $226.54\times$ speedup over an elementary CPU implementation, $19.68\times$ speedup over Parasail [3], and $3.57\times$ over GASAL2 [4].

II. Introduction

The Needleman-Wunsch algorithm is primarily applied in bioinformatics for pairwise alignment of protein or nucleotide sequences, particularly when the quality of global alignment is application-critical.

A. Applications in Bioinformatics

The most notable application of Needleman-Wunsch is the NW-like alignment step in Minimap2 [5], one of the popular genetic sequence alignment tools in use today. Minimap2 utilizes a vectorized dynamic programming strategy similar to that of Parasail [3], which will serve as our parallel CPU baseline. While Minimap2 is flexible enough to support application-specific parameters, in addition to the dynamic programming matrix and backtrack steps, this was determined to be beyond the scope of our project. We have developed a highly-scalable parallel GPU-based dynamic programming solution for long read alignment.

B. Base Algorithm Overview

First an N by M matrix is initialized by filling in the first row with $x_i = i \cdot gap_score,\ i \in [0, tlen+1)$. Similarly, the first column is initialized with $y_i = i \cdot gap_score,\ i \in [0, qlen+1)$. $tlen$ and $qlen$ refer to the target and query string lengths, respectively. The dynamic programming algorithm then begins with cell index $(1,1)$ and continues left to right, row-by-row, until the bottom right cell has been computed. Each cell is computed with the following, where S is the application's similarity matrix, D the deletion score, and I the insertion score:

$$M_{i,j} = \max \begin{cases} M_{i-1,j-1} + S(Q[i], T[j]) \\ M_{i,j-1} + D \\ M_{i-1,j} + I \end{cases} \tag{1}$$

Once the dynamic programming matrix has been filled, the backtrack step begins. First, the bottom right value ($M_{qlen,tlen}$) is saved, as this represents our optimal alignment score. Next, we perform the backtracking algorithm to find the optimal alignment between the query and reference strings. Starting from $M_{qlen,tlen}$, until we reach the top left cell ($M_{0,0}$) we perform:

```
// Match.
if (M(i,j) == M(i-1,j-1) + S(Q[i-1], T[j-1])) {
  Q_Alignment = Q[i-1] + Q_Alignment
  T_Alignment = T[j-1] + T_Alignment
  i = i-1
  j = j-1
}
// Insertion.
else if (M(i,j) == M(i-1,j) + I) {
  Q_Alignment = Q[i-1] + Q_Alignment
  T_Alignment = "-" + T_Alignment
  i = i-1
}
// Deletion.
else {
  Q_Alignment = "-" + Q_Alignment
  T_Alignment = T[j-1] + T_Alignment
  j = j-1
}
```

After the backtrack step has completed we are left with two strings that represent the optimal alignment for the query and target sequences. Note, it is possible for multiple paths to result in the same maximal alignment score. In such an instance, all paths are considered optimal or correct. For validating correctness, we ensure our model follows the same path as our elementary CPU implementation. The key takeaways from the algorithm, with respect to its acceleration, are:

- The memory requirement is $O(qlen \cdot tlen)$.
- Serial runtime is $O(qlen \cdot tlen)$.
- The backtrack step runtime is $O(tlen + qlen)$
- The backtrack step has 3 possible directions at each cell.

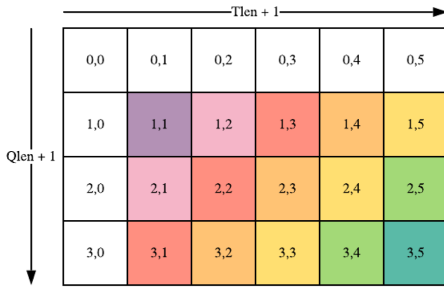

Fig. 1: Wavefront Parallelism

C. Smith–Waterman and Banding

Smith–Waterman and Needleman-Wunsch are very similar in their modeling, and we will briefly explain how our model could be adapted to suit SW applications. For matrix initialization: instead of a gap-score dependent first row and column, SW sets both to zero. Scoring: SW does not allow negative scoring and instead clamps values to zero if they would otherwise be negative. Traceback: SW only backtracks, starting from the bottom right cell, until a zero is encountered.

A popular optimization is to enable banding, where cells outside some bandwidth k off the diagonal of the matrix are ignored. A problem with this approach, when targeting optimal global alignment, is that it is plausible this alignment does not reside within this band. For the purposes of this GPU implemenation we did not explore banding.

III. Design

To write an efficient GPU implementation we will walk through several optimizations. These relate to algorithmic optimizations to exploit parallelism, and hardware-specific optimizations with respect to GPU execution of our design. Note that while our design is targeted towards CUDA specifically, an OpenCL implementation is certainly possible with language modifications. All of the ideas exhibited in this section will also work for an OpenCL design.

A. The Wavefront Model

A key algorithmic optimization to notice is how "Read-After Write" (RAW) hazards are represented in Needleman-Wunsch. Any given cell requires the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$ to be completed before proceeding. This allows us to imagine optimal parallelism waves depicted in Figure 1.

In this small example we do not achieve much parallelism. However, standard matrix sizes for most long read global alignment implementations are on the order of 1000 to 25000 for each dimension [6], [7]. Assuming there are sufficient worker threads available (which should always be the case for a GPU), we can transform the runtime from $O(qlen \cdot tlen)$ to $O(tlen + qlen)$.

A downside to this approach is that the irregular strided memory access pattern is detrimental to GPU memory throughput. As memory coalescing is not exhibited for either reads or writes, and memory transactions are completed per-warp, only $\frac{1}{32}$ of the available memory bandwidth will be used to send useful information. In the next section we will address how to transform our memory space to exhibit perfect GPU memory coalescing for both reads and writes.

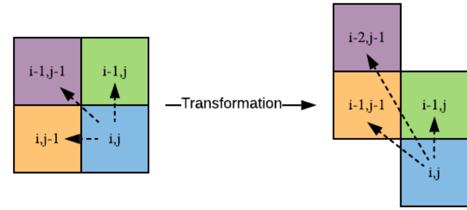

Fig. 2: Sheared Matrix Transformation

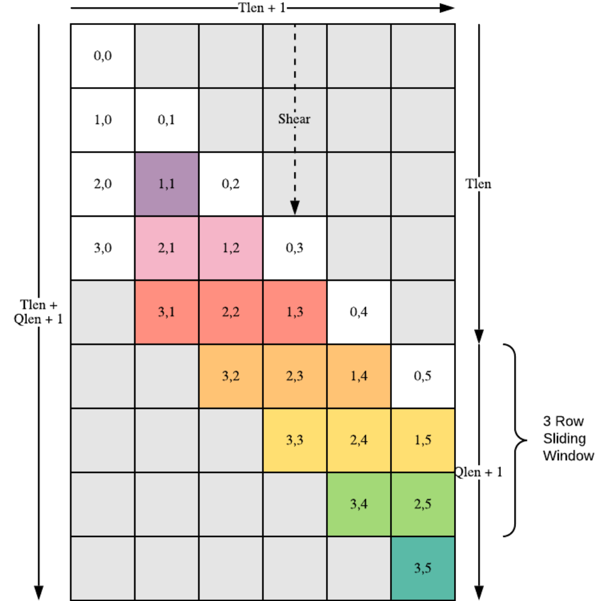

Fig. 3: Full Sheared Matrix

B. The Shear Factor

We can offset our matrix by an i-offset relative to its j index to place all reads for a compute band within a contiguous region of memory. The mapping from a given cell to its transformed counterpart in the sheared matrix, along with its RAW dependencies, are shown in Figure 2.

Note this mapping allows sequential threads to read from sequential memory addresses. Since GPUs must coalesce reads into large memory requests, this means that all of the information read from global memory for a given request will be useful during the current wavefront computation. A full sheared matrix is displayed in Figure 3.

Although storing our data in sheared matrix format significantly improves memory throughput, in its most naive form shearing will waste substantial quantities of memory. In the figure above, we require $(tlen + qlen + 1) \cdot (tlen + 1) \cdot sizeof(int)$ bytes to store only $(qlen+1) \cdot (tlen+1)$ useful values. When $tlen \approx qlen$ (as is typically the case for global alignment), half of the available memory is wasted. Referring back to our transformation mapping in Figure 2, we can see that our DP computation depends only on the previous two

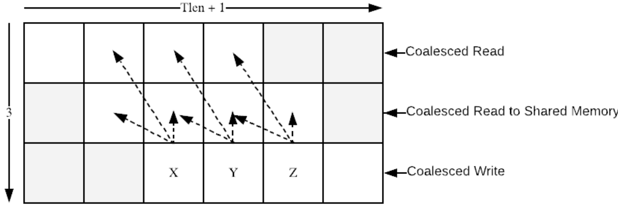

Fig. 4: Sliding Window

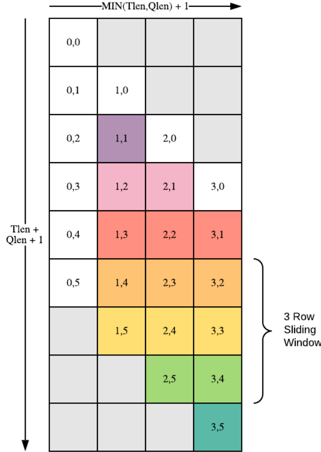

Fig. 5: New Sheared Matrix

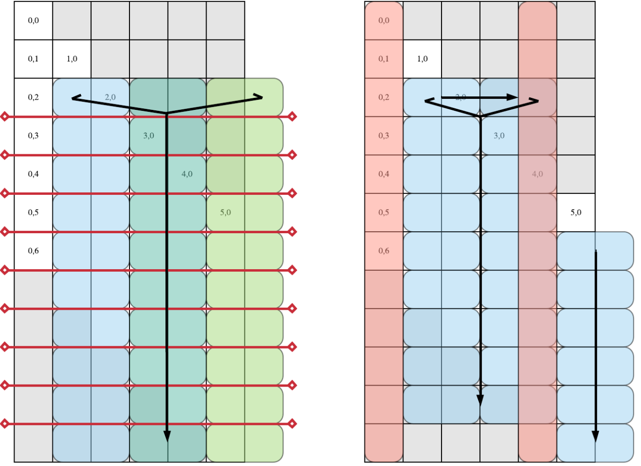
Fig. 6: Work Allocation, old (left) and new (right)

rows. This space optimization assumes we are storing the information necessary for backtracking elsewhere, which will be addressed in Section III-E.

Using a sliding window of three rows, we can also leverage shared memory to reduce the number of global memory accesses. Observe that the middle row of our sliding window has overlap: cell X requires a value that cell Y also requires. Performing a coalesced read into shared memory prior to computation reduces our global memory bandwidth by $2\times$ for this row. Once a row is computed, the buffer pointers are passed in a circular fashion (0 to 1, 1 to 2, 2 to 0) in preparation for computing the next row.

C. Improving The Sheared Matrix

It was observed that when $tlen \neq qlen$, the sheared matrix can be more compactly represented by storing the longer of t and q along the first dimension and the shorter of t and q along the second. Figure 5 demonstrates this modified transformation, an improvement in space requirements over Figure 3. To avoid complications in indexing, kernel logic, and matrix computation resulting from this conditional swap, it was performed prior to kernel execution. This allows the kernel to remain oblivious to the optimization and assume $tlen \geq qlen$.

D. Work Allocation

Initial profiling of our kernel revealed that kernel launching accounted for over 50% of our runtime. To fix this, we re-designed our high-level parallelization strategy and assigned more work to each kernel. Figure 6 demonstrates our initial approach, where many blocks (shown in various shades of blue and green) are responsible for computing a single row of the matrix simultaneously. Each thread computes a single element, and then a grid-wide synchronization (shown as a red line) must be performed to avoid race conditions. Both implicit kernel-relaunching synchronization and utilizing CUDA “Co-operative Groups” were found to add approximately the same overhead, which was unacceptably high.

The simplest way to increase the amount of work per kernel launch would be to compute the entire matrix using a single block, proceeding in row-major order. This eliminates synchronization overhead entirely, but is fairly slow due to repeated global memory accesses.

To remedy this, we utilized Shared Memory. Unfortunately, GPU Shared Memory is a limited resource (maximum 49152 bytes per SM), so we cannot store the three rows required for our sliding window approach (see Section III-B for details) if the matrix width $(\min(tlen, qlen))$ exceeds $49152/(3 * sizeof(\texttt{int})) = 4096$ elements.

Instead of computing all columns of the matrix before proceeding to the next row, our kernel block strides across the matrix and computes exactly 4096 elements per row, storing the results in Shared Memory. The remaining columns in the full sheared matrix are dependent upon only the last column which has just been computed, and so this column is stored in Global Memory. This pattern is shown in Figure 6 for a single block: striding across until 4096 elements are computed, doing so for all rows of those selected columns, and repeating this for the next chunk of 4096 columns until the entire matrix has been computed. Values stored in Global Memory are shown in red; all other computation occurs in Shared Memory, and no grid-wide synchronization is necessary.

E. Backtrack Matrix Storage

As noted previously, the asymptotic complexity of memory storage required for backtracking $(O(qlen \cdot tlen))$ exceeds the asymptotic storage requirements of computing the matrix $(O(qlen + tlen))$. Therefore, in order to decrease the total memory required to perform alignment (and therefore increase the number of simultaneously executing alignment kernels),

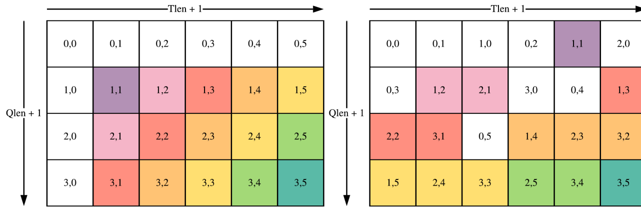

Fig. 7: Coalescing Output Writes

we spent considerable effort reducing the constant factor in memory storage requirements.

Ordinarily, a full scoring matrix is computed during the forward DP step, and the backtracking step is performed as described in Section II-B using the scoring matrix. We noted, however, that the parent of a given cell (see Figure 2, the cell which will be traversed next during the backtracking step) is already known during computation, since it's the cell which results in the maximum score (as calculated in Equation 1). Therefore, for each cell in the matrix, we need only store which of the three possible cells was the parent cell. Thus, instead of storing a 4-byte integer for each cell, we can store this information in a 2 bit pointer, representing a $16\times$ reduction in memory storage requirements. At the time of writing, our code stores this information in one byte for simplicity, still retaining a $4\times$ improvement in storage requirements.

Our initial implementation output this pointer matrix naively, in the expected format. Note that due to the inherent wavefront parallelism structure, writing information in this manner will lead to many uncoalesced writes to global memory, which accounted for 50% of our kernel's execution time. Figure 7 shows how this structure can be improved upon to allow for greater memory throughput, although it does significantly increase the complexity of accessing the correct memory addresses for each cell during the backtracking calculation.

F. Modified Backtracking

Because backtracking is not GPU friendly due to its substantial branching and unique memory access patterns, we have left this compute to the host. A powerful CPU core will fare much better as there is no inherent parallelism. Each backtrack step is dependent on the last and we can utilize vastly more powerful CPU prefetching/branch prediction mechanisms than our GPU has to offer. Because we have modified the way the backtrack matrix is stored, relative to II-B, we need to utilize the closed form function:

```
ij_to_z(i, j, tlen, qlen) {
  z = 0;
  jpi = j + i;
  if (jpi <= qlen)
    z = jpi * (jpi + 1) / 2 + i;
  else if (jpi <= tlen)
    z = qlen * (qlen + 1) / 2
      + (qlen + 1) * (jpi − qlen) + i;
  else
    z = (tlen + 1) * (qlen + 1)
      − (tlen + qlen + 1 − jpi)
      * (tlen + qlen + 2 − jpi) / 2
      + i − (jpi − tlen);
  return z;
}
```

This function allows to to map (i, j) from our original, untransformed, matrix to our compressed form as outlined in Figure 7. The original matrix is split into three matrix segments: top left triangle where the compute bandwidth is increasing with each compute iteration, a middle segment where the compute bandwidth is held constant, and the bottom right triangle where the compute band is shrinking. It is important to note: this function only applies if qlen is less than or equal to tlen. For a generalized application our GPU algorithm swaps query and reference, prior to compute, if this condition is not met. Once backtrack is completed we swap the aligned strings back to their original configuration.

G. Custom GPU Memory Management

The CUDA operations normally used for memory management synchronize all other CUDA operations [8]. For example, if `cudaMalloc()` is called to allocate device memory for a new kernel launch, the function will block the caller and synchronize all running kernels (even if they are in different streams) before performing the allocation. Since our implementation allocates and deallocates memory on an alignment-by-alignment basis (to efficiently handle varying target and query lengths), this behavior greatly reduces concurrency and hurts performance.

To alleviate this issue, our implementation allocates two large pools of memory at startup: one of device memory for kernel use and one of pinned, host memory for returning completed matrices. We then use pool manager objects to handle mallocs and frees within each of these pools. This way, we can avoid using implicitly-synchronizing CUDA memory operations while kernels are in-flight.

The pool manager's `free()` implementation simply pushes the given pointer into a list of pending frees and sets a condition variable to indicate that a pending free exists (see Listing 1).

Listing 1: Pseudocode for `free()`

```
free(ptr) {
  mutex_lock(frees_lock);
  frees.push_back(ptr);
  cond_signal(frees_exist);
  mutex_unlock(frees_lock);
}
```

Our `malloc()` implementation first pushes any pending frees into the pool manager's free-segment-tracking data structures. Next, it attempts to find a best-fit free segment in the memory pool. If this fails, the caller is blocked on a condition variable so that it waits until `free()` is called by another thread. Once this happens, the thread in `malloc()` updates the segment data structures and reattempts the allocation. When a segment of pool memory is successfully reserved, the function returns a pointer to the segment. Listing 2 is pseudocode for our `malloc()` implementation.

CPU	Intel Xeon E5-2697 v3 @ 2.60GHz (x2)
Logical Cores	28
L1 I&D Cache	32 KB Instruction — 32 KB Data
L2 Cache	256 KB
L3 Cache	35.8 MB
Memory	65.7 GB
GPU	**NVIDIA TITAN Xp @ 1.58GHz**
CUDA Version	10.1
CUDA Cores	3840
Memory	12.2 GB GDDR5X
Memory Speed	11.4 Gbps
Memory Bandwidth	547.7 GB/s
Bus Support	PCIe 3.0

TABLE I: Testing Infrastructure

Listing 2: Pseudocode for `malloc()`

```
malloc(size) {
  mutex_lock(malloc_lock);

  mutex_lock(frees_lock);
  free_pending_frees();
  mutex_unlock(frees_lock);

  do {
    ptr = get_best_fit(size);
    if (ptr == NULL) {
      mutex_lock(frees_lock);

      while (frees.empty())
        cond_wait(frees_exist, frees_lock);
      free_pending_frees();

      mutex_unlock(frees_lock);
    }
  } while (ptr == NULL);

  mutex_unlock(malloc_lock);
  return ptr;
}
```

H. SM Utilization

To increase streaming multiprocessor (SM) utilization we utilize a block size of 512. With this block size we allow four blocks to run on a single SM before we hit the hardware limit of 2048 threads per SM. This grants our SMs enough contexts to hide memory accesses and block-level synchronization operations. Block sizes greater than 512 led to substantial performance degradation in our analysis ($\approx 1.25 - 2\times$ slower runtime depending on matrix size).

IV. Methodology

A. Testing Procedure

All testing procedures are conducted on server grade hardware as seen in Table I. In our analysis we utilize a generic "-1" insertion or deletion penalty for all scoring algorithms. We fix a batch size of 10k matrices to compute for every input batch. Each batch has varied query and target lengths ($qlen/tlen$) to examine how each algorithm performs under different circumstances. Reported runtimes include both computation and backtracking.

B. Parasail

We compare Needletail to Jeff Daily's vectorized CPU implementation, Parasail [3], as a comparison to efficient CPU implementations of Needleman-Wunsch. Parasail was adapted to fit our testing infrastructure to eliminate any variables relating to I/O. Parasail is able to utilize its vectorized algorithm to provide different speedups relative to matrix cell bitwidths. In our testing we fix Parasail to 32-bit integers to reduce confounding variables because our GPU implementation, in its current state, only performs operations on 32-bit integers. It is crucial to note, Parasail's performance will scale linearly based on the bitwidth of the vector operands. As expected, with 8-bit integers Parasail runs $4\times$ faster than its 32-bit integer counterpart.

Prior to matrix computation, it is reasonable to estimate an upper bound on the maximum alignment score in any cell. This would enable selecting the minimally sufficient vector operand width, in order to increase performance. However, in practical applications 8-bit integers are unlikely to be used, so we place them out of the scope of this comparison.

C. GASAL2

We utilize the GASAL2 testbench provided on the master branch of the GASAL2 repository. Global alignment mode is set and the appropriate scoring penalties to match Needletail are assigned. Our input batch generation software is modified to pre-process identical batches in fastq format. Using the 56 logical threads in our testing infrastructure, we begin filling in data-table cells one by one. If GASAL2 fails due to running out of GPU memory we reduce the number of outstanding GPU streams, to the maximum value possible, before GPU memory is exhausted. In addition, the GASAL2 binary is remade for each row of the data table. This is a result of compiler-time optimizations made by GASAL2, based on the maximum possible query length.

D. Correctness Verification & Testing Infrastructure

We utilize a naive CPU implementation of Needleman-Wunsch as a correctness checking mechanism. Both Needletail and this elementary application are guaranteed to follow the same backtracking paths in circumstances where multiple optimal alignments are possible.

Testing and benchmarking batches were created using our custom string generation software called Batchgen. Batchgen works by generating arbitrary genomics strings for alignment given input parameters such as: target length ranges, query length ranges, number of targets, number of queries per target, etc. Generally, alignment occurs once a heuristic has determined a region where a query is expected to align. Our simulated alignment does not follow this model, however the work done is identical.

V. Results

A. Needletail vs Parasail

Observe runtime comparisons with respect to Parasail in Figure 8. In this comparison we note Parasail's efficiency for small matrix sizes. Because Parasail does not have to transfer data over the high-latency PCIe bus, we see a substantial performance benefit to using Parasail for short read

Needletail Speedup (w.r.t. Parasail)							
qlen/tlen	100	500	1000	2500	5000	7500	10000
100	0.15	0.30	0.64	1.26	1.59	1.74	1.65
500		3.10	3.11	5.74	5.64	5.69	5.67
1000			15.87	7.80	6.37	6.01	11.26
2500				22.20	13.15	11.91	11.69
5000					26.36	15.08	14.19
7500						27.06	22.49
10000							32.71

Fig. 8: Needletail vs Parasail Speedup

Needletail Speedup (w.r.t. GASAL2)							
qlen/tlen	100	500	1k	2.5k	5k	7.5k	10k
100	0.03	0.08	0.09	0.12	0.22	0.01	0.11
500		0.21	0.45	0.35	0.33	0.43	0.38
1k			0.58	0.50	0.52	0.45	0.60
2.5k				1.24	1.20	1.19	1.46
5k					1.47	2.06	3.07
7.5k						3.24	12.28
10k							8.49

Fig. 9: Needletail vs GASAL2 Speedup

alignment vs Needletail. As discussed previously, for these smaller matrices we can employ smaller bitwidths as needed for increased CPU-SIMD parallelism. Parasail would be best utilized for GASAL2 or Needletail's spare host CPU cycles. Both algorithms have idle host treads while GPU compute occurs. For maximum throughput it may be worth exploring a heterogeneous solution by applying Parasail in these circumstances.

B. Needletail vs GASAL2

GASAL2 comparison with Needletail is observed in Figure 9. Likewise with Parasail, GASAL2 is undeniably better for short read alignment. GASAL2 utilizes a global alignment algorithm that allocates a single GPU thread for one entire matrix. This approach is fantastic for these matrix sizes as it fully utilizes the SMs on the GPU, the PCIe bus, reduces the compute space to registers, and minimizes synchronization. However, we analyze that this approach does not scale well. It places unnecessary burden on the programmer for long read alignment. One must ensure not to launch enough threads to exhaust GPU memory resources. In such a scenario, the number of outstanding matrix computations plummets and per-thread matrix computation becomes a bottleneck. Here, a vectorized approach like Needletail or Parasail would be preferable. Note that as matrix size increases GASAL2 has to limit its host threads, and outstanding GPU streams, to reduce memory exhaustion.

C. Takeaways

Much like everything in computer science, a heterogeneous solution is the best approach. Prior computation should determine which algorithm to utilize. GASAL2 is substantially better than both Needletail and Parasail for small alignment. However, scalability issues plague GASAL2 because of its underlying implementation. Per-thread matrix computation performs exceptionally well for these small alignments, incurs no synchronization overhead, and utilizes GPU resources more efficiently. However, as matrix size increases to long read alignment magnitude, storing a matrix per compute thread is impractical as underlying hardware memory limitations are met. It is here we find Needletail's application and propose a heterogenous solution to alignment: GASAL2-like algorithms for short reads and Needletail-like algorithms for long reads.

VI. Related Work

Genomics acceleration has become popular in the last 5 years. The two primary avenues researchers have focused on is GPU and FPGA acceleration.

GPU acceleration grants programmers substantial flexibility and scalability in their designs, bugs are easier to fix and test, and adoption into applications is easier. GPU based algorithms are aided by vast amounts of money and R&D for GPU hardware. A few notable GPU genomics applications that were not explored are Parabricks [9] and NVBIO [10]. Unfortunately Parabricks is closed source so we cannot explore a possible comparison with its underlying algorithm. NVBIO was chosen not to be compared against as GASAL2 is shown to have better performance [4]. A more exhaustive study would have larger breadth than the two comparisons we explored in this paper.

FPGA and ASIC acceleration has also proven very promising with DRAGEN [11], Darwin [12], GenAx [13], and ERT [14]. Designs such as these have shown substantial performance benefits over their GPU counterparts, however, their production and adoption is significantly more difficult. ASIC designs are very costly and current market demand does not justify their cost. FPGA designs are more flexible but are still plagued by long design runs for even slight design changes. Cloud computing services such as AWS have given these custom designs more flexibility, but mass adoption has not yet occurred.

VII. Conclusion

As the volume of raw genomics data grows, finding increasingly faster solutions to common secondary analysis algorithms will be a catalyst for the next generation of healthcare. Higher quality long read sequencers will place increasing importance on their secondary analysis. In this term-paper we exhibit techniques programmers can employ to write hardware-conscious algorithms for the increased long read alignment throughput. We open-source our project, for future adaptations, here https://github.com/nate-ozog/eecs570_project_3. It is our hope that the methods exhibited for increased long read alignment throughput can find their way into de facto algorithms such as: BWA-MEM2 and Minimap2 [5], [15].

VIII. Contributions

Tim Dunn: Optimization and implementation of core compute kernel (memory usage, tiling, write coalescing, algorithm modifications).

Nathan Ozog: Batching and multithreading support, core compute kernel, base serial algorithm for verification.

Sanjay Singapuram: Initial batching and multithreading support, GASAL2 benchmarking.

Nicholas Wendt: Batching and multithreading support, memory pool management, Parasail implementation, testing infrastructure.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[2] "White-throated needletail," https://en.wikipedia.org/wiki/White-throated_needletail.

[3] J. Daily, "Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments," *BMC bioinformatics*, vol. 17, no. 1, p. 81, 2016.

[4] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data," *BMC bioinformatics*, vol. 20, no. 1, p. 520, 2019.

[5] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[6] S. L. Amarasinghe, S. Su, X. Dong, L. Zappia, M. E. Ritchie, and Q. Gouil, "Opportunities and challenges in long-read sequencing data analysis," *Genome Biology*, vol. 21, no. 1, p. 30, Feb 2020. [Online]. Available: https://doi.org/10.1186/s13059-020-1935-5

[7] M. O. Pollard, D. Gurdasani, A. J. Mentzer, T. Porter, and M. S. Sandhu, "Long reads: their purpose and place," *Human Molecular Genetics*, vol. 27, no. R2, pp. R234–R241, 05 2018. [Online]. Available: https://doi.org/10.1093/hmg/ddy177

[8] S. Rennich, "Cuda streams and concurrency." [Online]. Available: developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[9] "Parabricks," https://developer.nvidia.com/nvidia-parabricks.

[10] "Nvbio," https://developer.nvidia.com/nvbio.

[11] R. McMillen and M. Ruehle, "Bioinformatics systems, apparatuses, and methods executed on an integrated circuit processing platform," https://www.google.com/patents/US9014989, Apr. 21 2015, uS Patent 9,014,989.

[12] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 199–213, 2018.

[13] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 69–82.

[14] A. Subramaniyan, J. Wadden, K. Goliya, N. Ozog, X. Wu, S. Narayanasamy, D. Blaauw, and R. Das, "Accelerating maximal-exact-match seeding with enumerated radix trees," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/03/25/2020.03.23.003897

[15] V. Md, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," *arXiv preprint arXiv:1907.12931*, 2019.