Performance Modeling and Analysis of a *de Bruijn* Graph Based Local Assembly Kernel on Multiple Vendor GPUs

LeAnn M. Lindsey
Kahlert School of Computing
University of Utah
Salt Lake City, UT USA
leann.lindsey@utah.edu

Nan Ding
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA USA
nanding@lbl.gov

Jack Deslippe, Muaaz Awan
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA USA
jrdeslippe,mgawan@lbl.gov

***Abstract*—Bioinformatics workloads differ significantly from traditional scientific computing and AI workloads because they consist primarily of integer-only operations and string comparisons rather than floating-point operations. The underlying algorithms usually have low arithmetic intensity, irregular memory access patterns, and non-deterministic workloads. Local Assembly is an essential step in large-scale genome assembly software and is typically implemented using de Bruijn graphs. This paper examines the performance, portability, and productivity of a local assembly GPU kernel from a metagenome assembly pipeline implemented using hash table data structures on NVIDIA, AMD, and Intel GPUs. We focus on the challenges of achieving portability while maintaining performance for a complex bioinformatics GPU kernel that relies on hardware-specific optimizations. In this paper, we evaluate the local assembly kernel's performance and portability across different GPU architectures, identify performance bottlenecks, and propose modifications in existing tools and methods for performance modeling and analysis of integer-heavy bioinformatics application kernels.**

***Index Terms*—performance modeling, portability, genome assembly, de Bruijn Graph, CUDA, HIP, SYCL**

I. Introduction

The recent deluge of genomic sequencing data has led bioinformatics workloads to be an increasingly important sector in the field of scientific computing. Even with the tremendous advancement in genome sequencing technologies in the past ten years, sequencing read lengths are still significantly smaller than even a bacterial genome, and as sequence length increases, accuracy tends to decrease [1]. As a result, computational algorithms are required to reconstruct the genome, and correct errors in reads through read consensus. If the sequencing reads come from a well-studied organism, reads can be aligned to a reference genome using a sequence alignment algorithm, however, when the reference genomes are unknown, genomes need to be constructed *de novo*. Such techniques often rely on de Bruijn graph based methods [2, 3] to discover contiguous pieces of the genome (contigs). These methods involve connecting overlapping pieces of the genome (input reads) to find the longest contiguous region of the DNA that fulfills certain conditions.

Exascale Computing Project (17-SC-20-SC)

As open data becomes increasingly popular among scientists to promote research reproducibility and integrity, it's also crucial to adapt bioinformatics workflows to various computational resources. The recent large-scale adoption of GPUs for AI and scientific workloads has led to GPUs being the dominant computing resource for large-scale computing. This adds to the significance of exploring software portability across GPUs from different vendors. Bioinformatics kernels are particularly challenging when it comes to portability because of the peculiar characteristics that make them less amenable to GPU offload, hence requiring hardware-specific optimizations and frequent use of vendor-specific intrinsics [4, 5].

Alignment algorithms such as Smith-Waterman are typically implemented as dynamic programming algorithms, which have data dependencies in each iteration that limit the amount of parallelism that can be used while also presenting a data access pattern that does not allow much caching or memory coalescing on GPUs. The previous work on evaluating the portability of a sequence alignment kernel [5] (another frequently used bioinformatics kernel) focuses only on a dynamic programming based alignment algorithm, which is quite different than the local assembly kernel [5]. Local Assembly uses a de Bruijn graph-based algorithm implemented as thousands of dynamically allocated hash tables. While constructing hash tables can be highly parallelized, the memory access pattern in a hash table can be highly random, which leads to poor utilization of much of the GPU's memory subsystem. Similarly, the graph traversal algorithms are forced to limit parallelism as relatively short graph walks are faster if done serially.

Bioinformatics application kernels differ from traditional scientific computing tasks, which rely heavily on floating-point operations, because they rely primarily on integer operations and string comparisons. Another challenge that plagues both of the above mentioned bioinformatics kernels is that of large memory requirements. GPUs typically have relatively smaller main memory which limits the amount of work that can be offloaded to the device, limiting the utilization of the large number of computational units that GPUs offer.

In this paper, we assess the performance portability of the

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00150

local assembly GPU kernel from MetaHipMer [4], which is a widely used large scale metagenome assembler [6]. MetaHipmer utilizes GPUs and can scale to thousands of nodes. For this paper we focus on the local assembly phase of the MetaHipmer software pipeline and study the performance and portability of this kernel on NVIDIA Ampere GPU (A100), AMD Instinct GPU (MI250X), and Intel Data Center Max 1550 GPU (MAX1550). The contributions in this paper include:

- Porting the optimized CUDA code for local assembly kernel to HIP and SYCL. We demonstrate that algorithms like the local assembly can have better code and performance portability if programming models provide a fine-grained way of controlling GPU threads and have more uniform cache sizes.
- Evaluating the achieved performance and identify performance bottlenecks on the three different GPU architecture using the Instruction Roofline Model [7]. We demonstrate that the local assembly kernel is sensitive to cache size when operating for larger k-mer sizes, thus, we see GPUs with relatively larger cache sizes performing better for large k-mer sizes.
- Comparing the performance across the three platforms using the Pennycook portability metric [7, 8] to quantify the portability of local assembly kernel across multiple vendor GPUs and programming models. We show that our implementation of the local assembly kernel achieves a good performance portability across the three GPUs.
- We also demonstrate an analysis of the local assembly kernel using the architecture oblivious potential speedup plot to project performance improvements for different devices given ideal theoretical performance. This helps inform the hardware improvements that could be beneficial for a kernel like local assembly.

Our study is based on the optimized CUDA implementation of local assembly that was implemented initially keeping in mind an NVIDIA ecosystem, this kernel was then ported to HIP programming model to run on AMD GPUs and ported to SYCL to run on Intel GPU architectures.

II. Background

A. de Bruijn Graphs

De Bruijn graphs [9–11] are directed graphs made of overlapping sequences of characters and were first used to assemble genomic sequences in 1995 [12]. Each node in the graph represents a character sequence of fixed length, and a directed edge represents an overlap where the suffix of the first node overlaps with the prefix of the next node [13] (see Figure 1). An Eulerian path through the graph produces a genomic contig. Typically, several increasing lengths of k are used iteratively to construct de Bruijn graphs to resolve loops and other complexities in the graph. Hash tables are an effective way of implementing de Bruijn graphs in practice. The choice of hash tables as a data structure in this case also simplifies most algorithms that apply to these graphs.

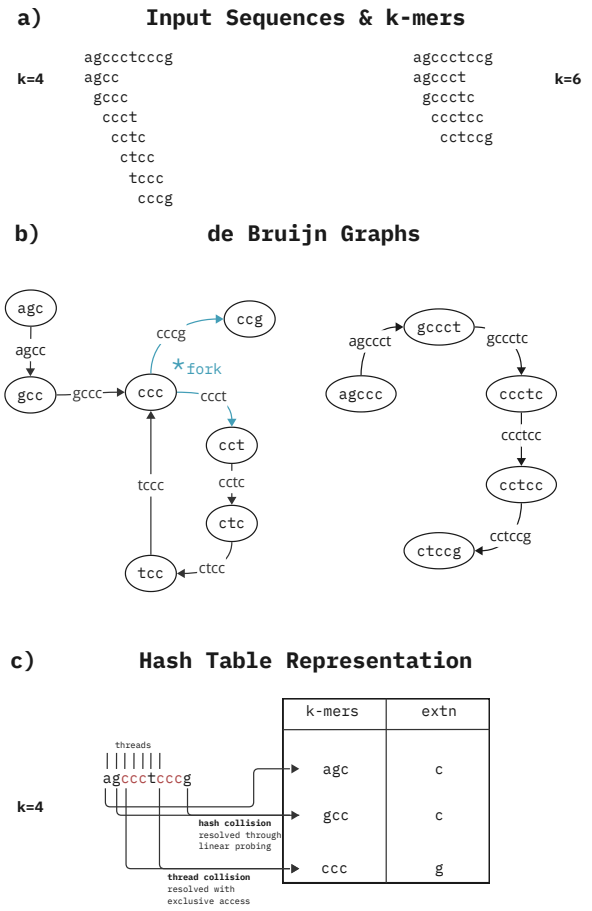

Fig. 1. de Bruijn Graph Overview with k=4 and k=6. a) Each input sequence is segmented into overlapping k-mers. b) The nodes of the de Bruijn graphs are made up of the k-1 prefix and suffix of each k-mer. The edges of the graph are the k-mers. Walking the directed edges of the graph gives the original sequence. Using longer k-mers resolves forks in the graph, as shown here, where using k=6 resolves the fork in the k=4 graph. c) A hash table is created with the k-mer prefix as a key and the extension character as the value. The hash table is created in parallel, with each thread corresponding to a k-mer. Thread collisions are resolved with an atomicCAS instruction.

B. MetaHipMer

Metagenomics is the study of the collective genome of microbial communities found in various environments. This is more complicated than the study of a single organism's genome (genomics) as metagenomics involves taking a sample directly from an environment and analyzing it without any pre-processing. A metagenomic sample may contain genomes of thousands of organisms with varying populations, thus making the problem of metagenome assembly a much more challenging task. [4]. MetaHipMer is a *de novo* assembler, optimized for metagenomic data, and uses UPC++ to distribute its workload over thousands of nodes [6][14]. The localized portions of work on each node are offloaded to GPUs in an asynchronous manner [15] [4] hence efficiently utilizing

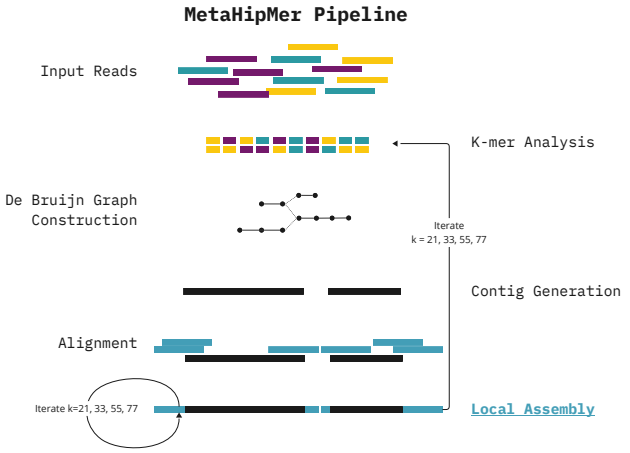

Fig. 2. **MetaHipMer Pipeline:** MetaHipMer is a *de novo* metagenome assembler. It uses de Bruijn graphs to construct assemblies of the genomes present in a metagenomic sample, and then estimate the proportion of each species that is present based on the number of reads that align back to each genome. The local assembly phase takes as input reads that align to the edge of each contig and creates a smaller de Bruijn graph which is used to extend the edges of each contig.

CPU and GPU resources. MetaHipmer has many advantages over the shared memory metagenome assemblers; its ability to scale across thousands of nodes allows it to co-assemble large datasets that otherwise would have to follow a multi-assembly style workflow [6]. For example, water samples from different locations in a lake over a period of one year can be co-assembled to analyze the bacterial and viral species present and how the quantities of each species change over time. Co-assembly, or assembling all samples together in a single assembly, has been shown to have several advantages, including recovering a larger fraction of the rare species present, but co-assembly is extremely memory-intensive. The scaling properties of MetaHipmer allow it to use the memory on as many nodes as it is allocated, and thus, it has been able to co-assemble metagenomic datasets in the order of terabytes that were previously too large to assemble.

MetaHipMer has an iterative workflow (see Figure 2) that starts with creating k-mers from each of the input reads, filtering out likely erroneous reads (those that occur only once), and then contigs are generated via a global de Bruijn graph construction. Sequencing error and homology between organisms can cause unresolvable forks in the global de Bruijn assembly graph, thus limiting the lengths of contiguous regions of DNA (or contigs) that can be obtained in this step. Local assembly performs a local de Bruijn graph traversal using only the reads that align to each end of a contig, extending the edges to resolve errors in the global graph and providing much longer contigs.

C. Local Assembly Module

All the reads and the contigs to which they align are localized on the same nodes, this allows the local assembly

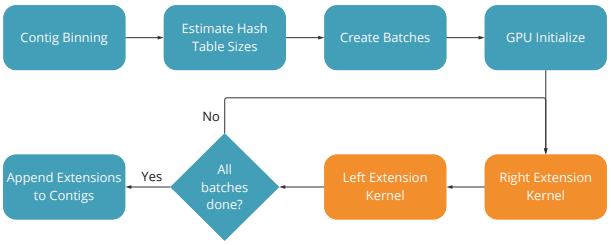

Fig. 3. Overview of the GPU local assembly workflow

phase to be offloaded to GPUs without being interrupted by off node communications. The local assembly phase involves breaking down all the reads into k-mers and constructing a de Bruijn graph using hash tables as the underlying data structure. The next step is to traverse the de Bruijn graph starting from the end of the contig that is to be extended. The process of traversing the de Bruijn graph is called mer-walk and has been described at length in [4] and [16]. This step essentially involves reading a k-mer from the end of a contig and performing a look up in the hash table, each hit in the hash table results in a character being appended to the end of the contig.

The local assembly module is called in each iteration of the MetaHipmer workflow to extend the contigs (see Figure 1), and constitutes a significant portion of the overall runtime. Moving local assembly to the GPU sped up this portion of the workflow by 7x [4]. The local assembly module is given a list of contigs and a corresponding set of reads that align to the ends of the contigs. Since the underlying algorithms and data structures are not ideal for performance on a GPU, the local assembly module exploits the latency-hiding nature of the GPU by inundating each GPU with as much work as can be offloaded in a single kernel call. To this end, the local assembly's GPU version uses a pre-processing phase that accurately estimates the upper limit of each hash table's size and reserves memory accordingly (Figure 3). Another step in this pre-processing phase is the binning of the contigs based on the number of reads that are assigned to each contig. This is important because the graph traversal phase of the algorithm has a non-deterministic amount of work, and because multiple graph walks are performed in parallel, warp stalling can be avoided if a similar amount of work gets offloaded together (i.e., all walks terminate after a similar number of steps). Here, the binning method enables offloading the contigs with an estimated similar amount of work together.

Each bin of contigs, along with their corresponding reads, is offloaded in a separate kernel call. Within each kernel call, each contig and its corresponding reads are assigned to one warp containing 32 threads. Each warp then constructs the de Bruijn graph in parallel, using a hash table as the underlying data structure. Each hash table uses the k-mer as the key and the value contains the extension nucleotide, as well as a voting metric to determine the number and quality of the reads that determined the extension nucleotide (see Algorithm 1).

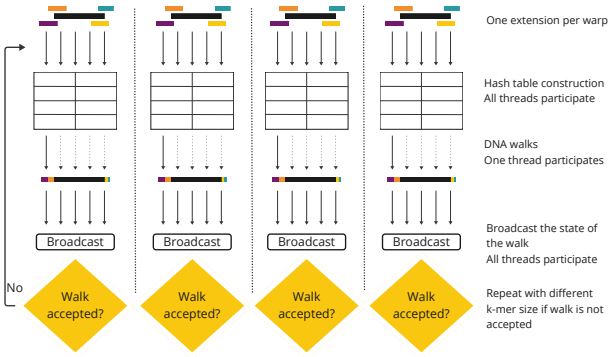

Fig. 4. Overview of the GPU local assembly kernel. One contig is assigned per warp. Dotted arrows show threads that are masked out during the merwalk.

After the hash table construction, each end of the contig is extended by entering the graph traversal phase, using the k-mer from the end of the contig as the first key. The DNA walk continues until the extension value in the hash table is either a fork, a loop or there is no nucleotide to continue the walk (see Algorithm 2). The graph traversal phase is performed by a single thread of a warp, and the state of the walk (fork, loop, or termination) is then broadcast to the other threads. An overview of the local assembly kernel can be seen in Figure 4.

III. Algorithm Implementations

Local assembly code was initially implemented in CUDA and optimized for NVIDIA GPUs using the low level intrinsics available. Below, we provide an overview of how the CUDA code differs from the SYCL and HIP counterparts. We also include the effort involved in porting from the initial CUDA code to the SYCL and HIP variants to help understand developer productivity offered by the different programming models when porting a complex bioinformatics GPU kernel.

A. CUDA

When implementing the de Bruijn graph construction phase of the local assembly kernel, threads of each warp access consecutive k-mers in each read, e.g. in Figure 1, the consecutive k-mers are inserted by consecutive threads of a warp doing parallel insertions in a hash table. Parallel insertions can result in two types of collisions, hash collisions, which are resolved using linear probing and thread collision. A thread collision occurs when multiple threads run into identical k-mers but each k-mer is located at a different index. In such a case, all the k-mers must be added to the same location in the hash table but in an atomic manner. In the CUDA programming model this is achieved by using the instruction `__match_any_sync` to synchronize only those threads in the warp that are colliding and then using `atomicCAS()` to update the entry in an atomic manner.

In the second phase of the algorithm, the de Bruijn graphs are traversed with only one thread in each warp performing the walk. These DNA walks are called mer-walks. At the end of the walk if the walk is not accepted, the hash tables are reconstructed using a different k-mer size. Due to the iterative nature of the algorithm, the state of the walk needs to be communicated to all the threads of the warp, this is done using shuffle instructions provided by CUDA that allow the threads to communicate data through direct register to register transfer.

B. HIP

The original CUDA code was converted to HIP for AMD GPUs using the hipify tool provided as part of ROCm toolkit. A large portion of the code was converted with ease, however since the kernel relied heavily on CUDA particular intrinsics, significant manual intervention was required. Below are some major modifications that were performed manually:

- The implicit assumption in the CUDA code is that a warp is 32 threads wide, this is not true on AMD GPUs, hence we had to manually change the code to correct this.
- Since HIP on AMD GPUs does not have the `__match_any_sync` instruction available, the code for ensuring atomic access in case of a thread collision had to be modified. This was done by leveraging implicit synchronization of threads in a warp (a wavefront on AMD GPUs) and adding a flag to keep track of threads that had already inserted a value in the hashtable.
- Other minor code modifications were needed to fix issues such as changes to warp shuffle functions and lack of explicit warp synchronizations on AMD hardware.

C. SYCL

The CUDA code was converted to SYCL code using the SYCLOMATIC migration tool [17]. Workload distribution changes were made because of the difference in warp and thread sub-group sizes between the architectures. CUDA has a fixed warp size of 32, while SYCL allows variable sizes of sub-groups. We experimented with several sub-group sizes and found that the sub-group size of 16 had the most consistent and optimal performance.

Similar to the HIP implementation, the atomic insertion function needed to be completely rewritten in SYCL. The SYCL implementation uses a work-group barrier to ensure that all insertions are complete before continuing (see Appendix B). The sub-group barrier implementation was tested in both CUDA and HIP to ensure correctness of results because of the implementation change.

Algorithm 1 Construct hash tables

```
1: C ← contigs
2: for each contig c in C do
3:     for each read r in get_reads(c) do
4:         for each k-mer k in r do
5:             k-mer_ht.insert(k)
6:         end for
7:     end for
8: end for
```

TABLE I
HPC ARCHITECTURES, COMPILERS AND LANGUAGES

HPC System	Accelerator	Programming Model	Compiler
Perlmutter (NERSC)	Nvidia A100	CUDA	CUDA 12.0
Frontier (OLCF)	AMD MI250x	HIP	ROCm 5.3.0
Sunspot (ALCF)	Intel Max 1550	SYCL	Intel DPC++ 2023

Algorithm 2 DNA walks

```
c ← contigs
walk ← empty
walk_state ← empty
k-mer ← get_k-mer(c)
for i = 0 to i < max_walk_len do
    if loop_exists(k-mer) then
        end_walk
    end if
    ext ← k-mer_ht.lookup(k-mer)
    if ext == end || ext == fork then
        walk_state ← ext
        break
    end if
end for
```

IV. EXPERIMENTAL SETUP

A. System Architecture and Compilers

We evaluate our implementation on three different GPU architectures (shown in Table I). The CUDA implementation was tested on an Nvidia A100, the HIP implementation was tested on an AMD MI250x, and for the SYCL implementation, we used Intel MAX1550. Note that the MI250X has two graphics compute dies (GCDs) per GPU, for this study we used only one GCD. Similarly, the Intel Max 1550 GPU has two tiles, and for this study we used a single tile. The compiler toolchains and their versions used for each implementation are shown in Table I. It must be noted that the Sunspot system used for the Max 1550 GPU was using pre-production software which may impact performance/results.

B. Datasets

In this study, we used the same test data that was used to assess the performance of the GPU local assembly kernel in [4]. However, in this case, since we only want to study performance on a single device at a time, we extracted much smaller datasets from the data in [4] such that it allows us to replicate the conditions of a production case yet providing small run times for convenient profiling. The local assembly module is called multiple times in the MetaHipmer workflow, and each time it is called with a successively larger k-mer size. To understand the effects of varying k-mer size, we used four datasets, each corresponding to k-mers of length 21, 33, 55, and 77 (k-mer sizes that are used by MetaHipmer in production workflow). Since these were extracted from intermediate data produced by MetaHipMer in the course of *de novo* assembly, the size of each dataset varies, depending on the number and size of reads that align to each contig. Characteristics of each dataset can be found in Table II. It must be noted that, since we use test data extracted from real-world datasets, the conclusions from this study can be applied to the production application.

V. RESULTS AND ANALYSIS

As described before, the local assembly kernel relies only on integer operations. Therefore, we utilize the instruction roofline [7] to understand the performance across different GPU architectures. We then create a modified version of the theoretical speed-up plot used in [18] to demonstrate the normalized performance relative to the sustained machine peak. We leverage the insights from these tools to understand the performance limiters and variations in performance across different GPU architectures. In addition, we demonstrate performance portability using the Pennycook portability metric [8, 19]. Finally, we draw conclusions that may apply to a wider set of bioinformatics applications that rely on similar algorithmic motifs.

A. Time-to-solution Performance

The raw kernel execution times for each architecture can be seen in Figure 5. It must be noted that larger k-mer sizes do not necessarily mean more work, since work is dependent upon the number of reads assigned to the contig, as well as the length of the final extension. It can be observed in the figure that the run-times for the AMD MI250x device show a unique pattern, i.e., for the larger k-mer sizes, there is a drastic increase in runtime, which is different from both the other devices, which more or less center around a similar average runtime. Figure 5 also illustrates that for varying k-mer sizes, the local assembly kernel utilizes device resources differently across all devices, hence a large difference in performance on the same datasets. To better understand the behavior of this kernel across architectures we do an in-depth analysis in the following sections.

B. Roofline Performance Analysis

The Instruction Roofline Model [7] enables the roofline analysis capability for integer-heavy applications. The Instruction Roofline Model characterizes a kernel's performance as billions of instructions per second, which is a function of peak machine bandwidth, the Instruction Intensity of the kernel, and the machine peak measured in Giga Instructions Per Second (GIPs). On an NVIDIA GPU the "Instruction Intensity" is defined as warp-based instructions per global memory transaction.

In this paper, we only measure the total integer operations (INTOPs) instead of the total integer instructions. Considering

TABLE II
DATASET CHARACTERISTICS.

k-mer size	total contigs	total reads	average read length	total hash insertions	average extn length	total extns
21	14195	74159	155	10,011,465	48.2	684100
33	4394	20421	159	2,593,467	88.2	387283
55	3319	13160	166	1,473,920	161.0	534206
77	2544	7838	175	775,962	227.0	577496

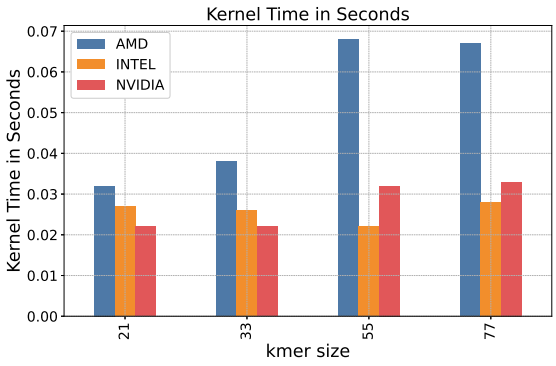

Fig. 5. Kernel Execution Time Comparison.

only INTOPs for modeling the performance has two advantages, 1) this makes the model more portable as not every device architecture may measure instructions per transactions and 2) this will allow us to model pure arithmetic workload making the model application and compiler independent. Therefore, we simplified the instruction roofline model by substituting the integer operations peak for the total instruction peak and converting the "Instruction Intensity" from the original Instruction Roofline Model to "Integer operations per byte" which we call "INTOP Intensity (II)".

Similar to the traditional Roofline model, the ridge point on the roofline is called the "Machine Balance" and kernels that fall to the left of this line are memory bound, while points that fall to the right of this line are compute bound.

The instruction roofline plots, shown in Figure 6, indicate that the local assembly kernel is compute-bound for all of the k-mer sizes on the A100 GPU while it is memory-bound on the MI250x and Max 1550 GPUs. It is interesting to see three different trends on the three GPUs as the k-mer size increases. Recall that in a roofline plot, the more the location of a marker is to the upper right, the better performance it achieves (i.e. kernel is not limited by memory bandwidth).

It can be observed that on the A100 GPU, the marker moves to the left as the k-mer size increases from 21 to 33; it then goes to the upper right with a k-mer size of 55 and then moves to the lower right direction with a k-mer size of 77. As the k-mer size increases, the thread predication (load imbalance) starts becoming dominant because the possibility of a successful graph walk becomes more limited for larger k-mer sizes. This effect can be observed when considering that the k-mer size 33 marker moves slightly to the left. However, for even larger k-mer size (55) we observe that the pointer is moving back to the right, this is because larger k-mer sizes benefits more from bigger cache available in the A100 device. Similar to the earlier trend, when the k-mer size is 77, the negative thread predication effect is revealed again hence the marker for k-mer size 77 moves to lower right. Correlating this discussion with Figure 5, we see A100 GPU achieves its shortest run time at $k = 55$.

For the MI250x GPU, it can be seen that the markers for all k-mer sizes move to the lower left as the k-mer size increases. This is a combined effect of thread predication and a smaller cache size available on the MI250x GPU. The larger k-mer size benefits greatly from a large cache size, however in this case increasing k-mer size results in more cache misses leading to poorer performance. This can be again be correlated with the longer run times on MI250x GPU in Figure 5, in particular for the larger k-mer sizes.

In contrast to MI250x GPU profile, the markers on the Max 1550 GPU roofline plot move to the upper right direction as the k-mer size increases. In this case, there are multiple factors at play. First, on the Max 1550 GPU, the subgroup size of 16 is the smallest across all the GPUs, which helps to reduce the effect of predication for larger k-mer sizes, secondly, Max 1550 GPU also has the largest L2 cache of the three GPUs being considered here (see Table III), this helps minimize any cache misses associated with larger k-mer sizes thus keeping data movement consistent with increasing compute requirements for larger k-mers. Hence, we see that for larger k-mers the arithmetic intensity consistently increases with markers moving to the right.

C. Performance Correlation Comparisons

The Instruction Roofline plots show the achieved performance using INTOPs and INTOPs/byte, which amortizes the total amount of bytes moved. To better understand the bounding factor of performance, we performed a head-to-head comparison across devices. Figure 7 plots the performance in GINTOPs/s (Figure 7a) and total bytes moved (Figure 7b) on the NVIDIA and AMD devices. The diagonal line is included to observe the pattern and relationship between programming models. In Figure 7a, the empirical dots above the dotted diagonal indicate that the NVIDIA device has a higher performance than the AMD device for all the k-mer sizes. While if we look at Figure 7b, it can be observed that more bytes are

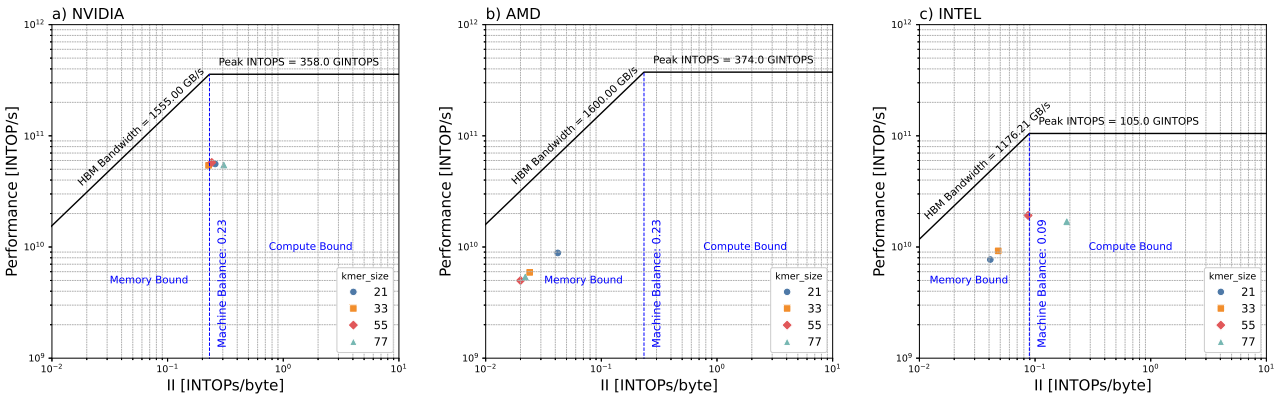

Fig. 6. Integer Operations Roofline Model Comparison. The three GPUs show three trends (empirical markers) as the k-mer size increases. The MI250x GPU has a smaller L2 cache size (8MB per die) than the NVIDIA GPU (40MB). Therefore, the empirical markers move to the lower left direction for AMD GPU as the k-mer size increases. On the A100 GPU, the markers move to the left first and then move to the right, but not the upper right direction. This is due to a combination of thread predication and increasing k-mer size. The markers move to the upper right direction on the Max 1550 GPU because of a combination the smaller warp size (16 threads) that reduces the load imbalance impact and a much larger L2 cache (204MB per die).

TABLE III
COMPARISON OF ARCHITECTURAL FEATURES.

Board	Compute Units	L1 Cache	L2 Cache	Memory
Nvidia A100	108 SMs	192 KB/SM	40 MB	40 GB
AMD MI250X	220 CUs	16 KB/CU	16 MB	128 GB
Intel MAX1550	128 X^e - cores	64 MB	408 MB	128 GB

moved on the AMD device. This further reinforces the idea that the smaller cache on the AMD device leads to many more global memory accesses, especially for larger k-mer sizes.

Similarly, Figure 8 shows a comparison of INTOPs/s performed and bytes moved for the local assembly kernel on the Intel device and the NVIDIA device. It is clear that the NVIDIA device achieves much higher performance (Figure 8a). However, Figure 8b shows that the Intel device moves a smaller number of bytes in comparison to the NVIDIA device. This again can be attributed to the significantly larger L2 cache, it can be observed that increasing k-mer size takes advantage of the larger cache in the Intel device. This behavior also explains how the Intel device is able to achieve better time to solution in Figure 5.

D. Performance Portability

We use the Pennycook Performance Portability metric [8, 19] to understand the performance portability of local assembly kernel across different GPU architectures. This formula uses the harmonic mean to provide a metric that allows us to quantify the degree to which a kernel or application is portable, using any measurable property of correct application performance on a platform.

For a given set of platforms H, the performance portability $\mathcal{P}$ is:

$$\mathcal{P}(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i\in H} \frac{1}{e_i(a,p)}} \\ 0, \end{cases}$$

TABLE IV
ARCHITECTURAL EFFICIENCY
THE PERFORMANCE PORTABILITY METRIC $\mathcal{P}$ BASED ON THE FRACTION OF THE INTEGER OPERATIONS ROOFLINE MODEL.

dataset k-mer size	Nvidia A100 CUDA	AMD MI250X HIP	Intel Max 1550 GPU SYCL	$\mathcal{P}_{arch}$
21	12.8%	15.1%	15.6%	14.4%
33	14.9%	15.8%	17.3%	15.9%
55	14.5%	18.8%	16.1%	16.3%
77	15.6%	16.1%	15.3%	15.6%
Average $\mathcal{P}_{arch}$				15.5%

where $e_i(a,p)$ is the performance efficiency of application a solving problem p on platform i.

We consider two performance portability metrics, first, *architectural efficiency*, defined as the fraction of achieved performance on the integer operations roofline model, and second, the *algorithm efficiency*, defined as the fraction of achieved performance against the ideal or theoretical INTOP Intensity (II) that can be obtained by the algorithm on an ideal architecture.

1) Architectural Efficiency : The architectural efficiency on each platform is listed in Table IV along with the average architectural efficiency. It can be observed that the local assembly kernel's average architectural efficiency across devices as well as datasets does not vary significantly, thus indicating good portability for the implementation across multiple GPU architectures.

2) Algorithm Efficiency : The metric of algorithm efficiency was introduced in [18] as a method to compare the empirical arithmetic intensity to a theoretical ideal based on assuming that every GPU has infinite memory capacity and a fully associative cache. We modified this metric by measuring the theoretical Integer Operations Intensity (II) and comparing the

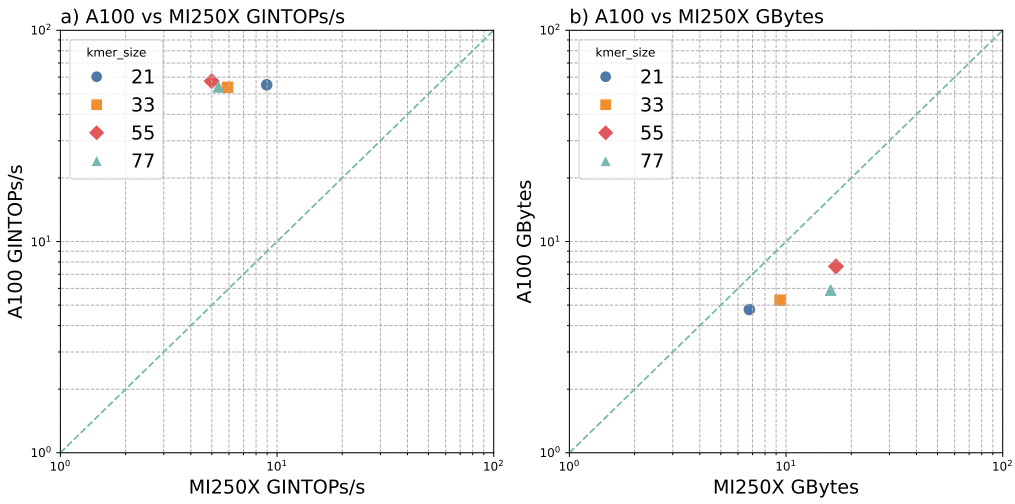

Fig. 7. Performance (left) and GBytes accessed (right) correlation between CUDA and HIP implementations on NVIDIA and AMD devices. The CUDA implementations consistently outperforms HIP by achieving higher GINTOPs and moving less data.

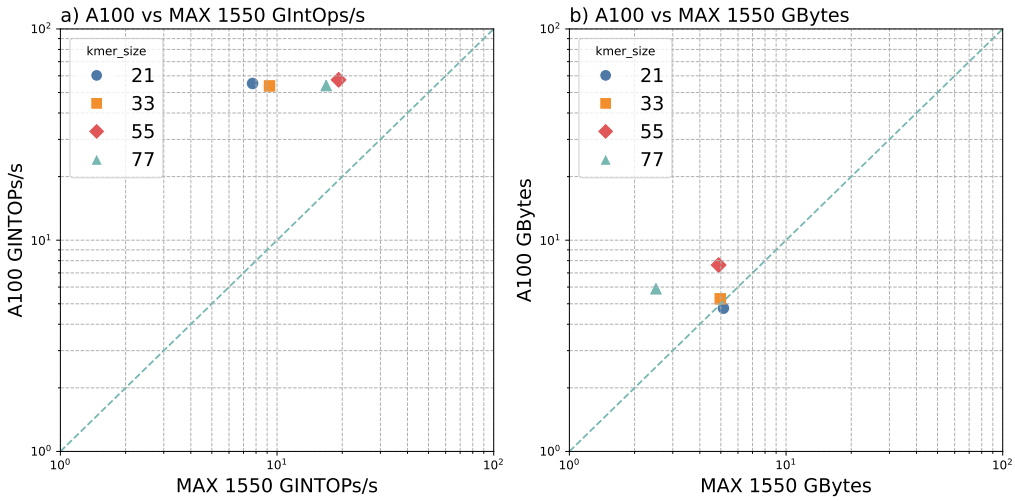

Fig. 8. Performance (left) and GBytes accessed (right) correlation between CUDA and SYCL implementations on NVIDIA and Intel devices. With a close amount of data movement, CUDA outperforms SYCL in time-to-solution due to a higher integer throughput for small k-mer size of 21 and 33. As the k-mer size increases to 55 and 77, SYCL has a shorter run time due to fewer data movement.

empirical II to the theoretical II. Since the local assembly kernel is significantly more complex than a stencil kernel, we have outlined the method by which we obtained the theoretical II.

The theoretical II is calculated as a ratio of the theoretical number of integer operations to the theoretical number of bytes moved. Since it is a ratio, we count the number of integer operations and the number of HBM bytes moved in one iteration of the algorithm 1 and algorithm 2 and take the ratio of instructions to bytes. This removes any dependency of the theoretical II on dataset size.

Definitions:

$$INTOP_1 = \text{Integer operations for Algorithm 1}$$
$$B_1 = \text{HBM bytes accessed in Algorithm 1}$$
$$INTOP_2 = \text{Integer operations for Algorithm 2}$$
$$B_2 = \text{HBM bytes accessed in Algorithm 2}$$

Algorithm 1: K-mer Hash Table Construction. The dominating factor for integer operations in Algorithm 1 is the hash function which is called each time a k-mer insertion occurs. We use the MurmurHashAligned2 hash function [20]

TABLE V
INTEGER OPERATIONS IN THE HASH FUNCTION

Dataset (k-mer size)	21	33	55	77
Initialization	33	33	33	33
Mix Loop	125	200	325	475
Cleanup	31	31	31	31
$INTOP_1$	215	305	457	635

in this implementation, which includes a loop whose length is dependent upon the length of the k-mer used, hence the number of integer operations depend heavily on the k-mer size used.

The number of HBM bytes moved are the bytes required to read in the k-mer and corresponding quality score and the number of bytes needed to insert a k-mer into the hash table. The hash table has a 4-byte pointer as a key, and the value is the extension, which is a 1-byte char. A 4-byte quality score is also recorded, as well as a 4-byte count for the number of reads with this k-mer.

$$B_1 = 2 * k + (3 * 4 + 1) \tag{1}$$
$$= 2k + 13 \text{ bytes} \tag{2}$$

Algorithm 2: DNA Walks The DNA walk in Algorithm 2 starts with slicing the k-mer length from the end of the contig that we wish to extend. This k-mer is used as a key in the hash table, and the extension value is added then the next k-mer is looked up in the table, and this continues until we reach a loop, a fork, the end of the table or `max_walk_len`.

As above, the number of integer operations in the loop is equal to the number of integer operations in the hash function (used for lookup this time).

The number of HBM bytes moved are the bytes required to read in the k-mer, k, a lookup in the hash table is the same 13 bytes written when the table is constructed.

$$B_2 = k + 13 \text{ bytes} \tag{3}$$

The DNA Walk algorithm is run every time the Hash Table Construction algorithm is run, so we can simply sum the INTOP and byte values and then take the ratio to obtain the theoretical II.

$$II = \frac{INTOP_1 + INTOP_2}{B_1 + B_2} \tag{4}$$
$$\tag{5}$$

Table VI shows theoretical II values computed for different k-mer sizes and Table VII shows the algorithm efficiency achieved across different devices. It can be observed in Table VII, except for a few outliers, overall, the local assembly kernel shows good portability across different architectures. Something interesting that can be observed in this table is the increasing algorithm efficiency for NVIDIA and Intel devices for increasing k-mer sizes and the inverse for the AMD device. This again points to the ability of the algorithm to utilize larger caches and achieve closer to theoretical peak performance.

TABLE VI
THEORETICAL II CALCULATIONS

k-mer size	Integer Operations per loop cycle	Bytes per loop cycle	INTOP Intensity (II)
21	430	89	4.831
33	610	125	4.880
55	914	191	4.785
77	1270	257	4.942

TABLE VII
ALGORITHM EFFICIENCY
THE PERFORMANCE PORTABILITY METRIC $\mathcal{P}$ BASED ON THE FRACTION OF ACHIEVED THEORETICAL AI.

dataset k-mer size	Nvidia A100 GPU CUDA	AMD MI250X GPU HIP	Intel Max 1550 GPU SYCL	$\mathcal{P}_{alg}$
21	17.1%	55.4%	13.4%	18.0%
33	17.6%	31.4%	15.8%	20.0%
55	21.1%	26.7%	30.0%	20.3%
77	27.2%	28.9%	60.9%	19.5%
Average $\mathcal{P}_{alg}$				19.38%

E. Potential Speed Up Plot

Unifying the architecture efficiency and the algorithm efficiency on the same plot yields important information as demonstrated in [18], especially because it allows us to analyze a GPU kernel's performance while being oblivious to the underlying device architecture.

In Figure 9 we combine the architecture efficiency (the fraction of sustained performance on the integer operations roofline model, left y-axis) and algorithm efficiency (the fraction of achieved performance against the theoretical INTOP Intensity, bottom x-axis) into a single plot for the local assembly kernel. To quantify the overall algorithmic and implementation performance across different devices, one can define a set of iso-curves of constant potential speedup. In the Figure [18], the right y-axis represents potential speed up that can be gained by improved performance, this can be derived by introducing a new and better suited algorithm for GPU architectures, better compiler code generation or by further optimizing the implementation. Similarly, the top x-axis represents potential speed up that can be gained by improving data locality or by better utilizing the device's memory sub-systems.

Figure 9 shows that most empirical dots for the local assembly kernel are gathered in the bottom left corner, which is very different from the traditional stencil [18] computation usually located in the upper right corner. On all three architectures, we can see there is a lot of potential for speed up, in particular by improving the kernel execution performance (y-axis). The potential for improvement by better utilizing the memory subsystem is significantly high for lower k-mer values however, devices with larger cache (e.g. Intel) end up exploiting memory

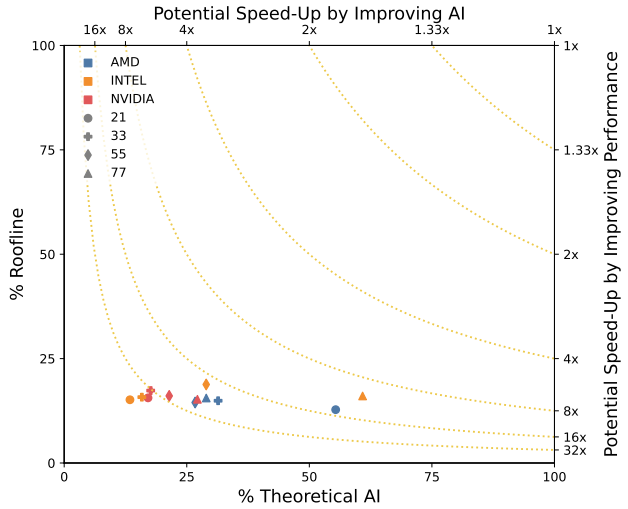

Fig. 9. Potential Speed Up Plot

bandwidth much better for larger k-mer values. It can be observed that on the Intel device, local assembly kernel can achieve up to 60% of the theoretical II. For NVIDIA device, the trend is similar to Intel but it is much localized to the left side of the plot which indicates to larger potential gains by using novel algorithms, more optimized implementation or a data structure with more localized memory access pattern.

Figure 9 provides an insight into the fact that bioinformatics algorithms in their current state are not very well suited for GPU architectures and programming models, i.e., porting from CPU and getting performance gains require significant low-level kernel development and optimization. The reason for this can be attributed to the relatively small presence of bioinformatics workloads when considering the overall utilization of large-scale systems dedicated to science [21]. This has led the GPU designers to favor an architecture that is more optimized for the typical physics and science workloads. However, with GPUs quickly taking over the computing landscape, bioinformatics workloads need to start adapting. Similarly, GPU architectures need to start supporting features that favor bioinformatics workloads, e.g., Intel's introduction of a larger L2 cache allows the local assembly kernel to scale better and achieve higher efficiencies at large k-mer sizes (Figure 9).

VI. Conclusion

With the wide-scale adoption of GPUs as the computing element of choice, most of the existing workloads are being ported to GPUs. The availability of portable programming models has made it much easier and straightforward to have a single code base that works across all the vendor GPUs. However, less common workloads like bioinformatics applications face multiple challenges in this space. These workloads rely on algorithms and data structures that are quite different from the widely used physics workloads, this leads to developers needing to build everything from scratch using low level programming models such as CUDA, HIP and SYCL. In this paper, we present an analysis of porting a complex bioinformatics kernel across multiple vendor GPUs.

The Local Assembly kernel that is part of the popular metagenome assembler (MetaHipMer) relies on hash tables to implement de Bruijn graphs and traversal algorithms for extending DNA contigs. This kernel was initially implemented in CUDA for NVIDIA GPUs and then ported using SYCL and HIP to Intel and AMD GPUs, respectively. In this paper we provide an in-depth analysis of the underlying algorithm's portability across three different vendor GPUs, we study the hardware features that limit this algorithms performance while clearly highlighting the features that may support better scaling of such workloads.

We demonstrate that larger GPU memory along with a memory subsystem with large cache sizes is more suitable for workloads like local assembly, it is promising to see that some of the newer architectures are making such features available. It has been further shown that graph based algorithms are not the best at exploiting the SIMT nature of GPUs since frequent divergence, inter-warp communication and synchronization leads to a complex as well as a sub optimal code, hence leading to poorer peak utilization of the devices in addition to poor portability. In this space, independent thread scheduling may help mitigate the issues by allowing for easier inter-warp communication and simplifying the code [22].

In summary, bioinformatics algorithms can be complex to offload on GPUs and may require the GPU ecosystem to pay more attention to these workloads as they become more and more dominant with rapid development in bioinformatics technologies of data generation.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility.

We would also like to acknowledge help and guidance from Rob Egan, Steven Hofmeyr (the lead developers of MetaHipMer) and the members of ExaBiome project.

References

[1] X. Ma, Y. Shao, L. Tian, D. A. Flasch, H. L. Mulder, M. N. Edmonson, Y. Liu, X. Chen, S. Newman, J. Nakitandwe, Y. Li, B. Li, S. Shen, Z. Wang, S. Shurtleff, L. L. Robison, S. Levy, J. Easton, and J. Zhang, "Analysis of error profiles in deep next-generation sequencing data," *Genome Biology*, vol. 20, no. 1, p. 50, Mar. 2019. [Online]. Available: https://doi.org/10.1186/s13059-019-1659-6

[2] S. Nurk, D. Meleshko, A. Korobeynikov, and P. A. Pevzner, "metaSPAdes: a new versatile

metagenomic assembler," *Genome Research*, vol. 27, no. 5, pp. 824–834, May 2017. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5411777/

[3] R. Rizzi, S. Beretta, M. Patterson, Y. Pirola, M. Previtali, G. Della Vedova, and P. Bonizzoni, "Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era," *Quantitative Biology*, vol. 7, no. 4, pp. 278–292, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s40484-019-0181-x

[4] M. G. Awan, S. Hofmeyr, R. Egan, N. Ding, A. Buluc, J. Deslippe, L. Oliker, and K. Yelick, "Accelerating large scale de novo metagenome assembly using GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476212

[5] M. Haseeb, N. Ding, J. Deslippe, and M. Awan, "Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2021, pp. 68–78.

[6] S. Hofmeyr, R. Egan, E. Georganas, A. C. Copeland, R. Riley, A. Clum, E. Eloe-Fadrosh, S. Roux, E. Goltsman, A. Buluç, D. Rokhsar, L. Oliker, and K. Yelick, "Terabase-scale metagenome coassembly with MetaHipMer," *Scientific Reports*, vol. 10, no. 1, p. 10689, Jul. 2020, number: 1 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/s41598-020-67416-5

[7] N. Ding and S. Williams, "An Instruction Roofline Model for GPUs," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Denver, CO, USA: IEEE, 2019, pp. 7–18.

[8] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," Nov. 2016, arXiv:1611.07409 [cs]. [Online]. Available: http://arxiv.org/abs/1611.07409

[9] C. F. Saint-Marie, "Solution to question nr. 48." *l'Intermediaire des Mathematiciens*, 1894.

[10] N. G. De Bruijn, "A combinatorial problem," *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, vol. 49, no. 7, pp. 758–764, 1946.

[11] I. J. Good, "Normal recurring decimals," *Journal of the London Mathematical Society*, vol. 1, no. 3, pp. 167–169, 1946, publisher: Oxford University Press.

[12] R. M. IDURY and M. S. WATERMAN, "A New Algorithm for DNA Sequence Assembly," *Journal of Computational Biology*, vol. 2, no. 2, pp. 291–306, 1995, _eprint: https://doi.org/10.1089/cmb.1995.2.291. [Online]. Available: https://doi.org/10.1089/cmb.1995.2.291

[13] P. E. C. Compeau, P. A. Pevzner, and G. Tesler, "Why are de Bruijn graphs useful for genome assembly?" *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, Nov. 2011. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5531759/

[14] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil: IEEE, 2019, pp. 963–973.

[15] M. G. Awan, J. Deslippe, A. Buluc, O. Selvitopi, S. Hofmeyr, L. Oliker, and K. Yelick, "ADEPT: a domain independent sequence alignment strategy for GPU architectures," *BMC bioinformatics*, vol. 21, no. 1, p. 406, Sep. 2020.

[16] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "HipMer: an extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2807591.2807664

[17] A. Huang, "SYCLomatic compatibility library: making migration to SYCL easier," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCL '23. New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 1–2. [Online]. Available: https://doi.org/10.1145/3585341.3585349

[18] O. Antepara, S. Williams, H. Johansen, T. Zhao, S. Hirsch, P. Goyal, and M. Hall, "Performance Portability Evaluation of Blocked Stencil Computations on GPUs," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1007–1018. [Online]. Available: https://doi.org/10.1145/3624062.3624177

[19] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[20] A. Appleby, "SMHasher - MurmurHash2." [Online]. Available: https://github.com/aappleby/smhasher

[21] B. Austin, "NERSC-10 Workload Analysis (Data from 2018)," Apr. 2020. [Online]. Available: https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf

[22] Nvidia, "NVIDIA TESLA V100 GPU ARCHITECTURE, WP-08608-001_v1.1," Aug. 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

APPENDIX A
ALGORITHMIC DIFFERENCES BETWEEN CUDA, HIP AND SYCL

The atomic insertion into the hash table had to be rewritten in HIP and SYCL because HIP and SYCL did not have functions equivalent to `__match_any_sync` and `__syncwarp(mask)`

In CUDA, `__match_any_sync` and `__syncwarp(mask)` are used to sync the threads that are in conflict with the hash table insertion to allow for an atomic insertion.

In HIP, this functionality is not possible, so each thread is given a `done` flag, and the while loop runs until all threads have inserted their values.

In SYCL, a sub-group barrier, `sg.barrier()` is used to to synchronize all of the threads in the subgroup inside the loop and `dpct::atomic_compare_exchange_strong` was used for atomic insertion. This same implementation was tested in CUDA with no degradation in performance.

The different versions of `__ht_get_atomic` can be seen in the following code block.

```
// CUDA Implementation of ht_get_atomic()
__device__
loc_ht& ht_get_atomic(loc_ht* thread_ht, cstr_type kmer_key, uint32_t max_size){
  unsigned hash_val = MurmurHashAligned2(kmer_key, max_size);
  unsigned orig_hash = hash_val;

  while(true){
    int prev = atomicCAS(&thread_ht[hash_val].key.length, EMPTY, kmer_key.length);
    int mask = __match_any_sync(__activemask(), (unsigned long long)&thread_ht[hash_val]);

    if(prev == EMPTY){
      thread_ht[hash_val].key.start_ptr = kmer_key.start_ptr;
      thread_ht[hash_val].val = {.hi_q_exts = {0}, .low_q_exts = {0}, .ext = 0, .count = 0};
    }
    __syncwarp(mask);
    if(prev != EMPTY && thread_ht[hash_val].key == kmer_key){return thread_ht[hash_val];}
      else if (prev == EMPTY){return thread_ht[hash_val];}
      hash_val = (hash_val +1 ) %max_size;
      if(hash_val == orig_hash){printf("*hashtable full*\n");}
    }
}

// HIP Implementation of ht_get_atomic()
__device__
loc_ht& ht_get_atomic(loc_ht* thread_ht, cstr_type kmer_key, uint32_t max_size){
  unsigned hash_val = MurmurHashAligned2(kmer_key, max_size);
  unsigned orig_hash = hash_val;
  int done = 0;
  int prev;

  while(true){
    if(__all(done))
      return thread_ht[hash_val];

    if(!done) {
      prev = atomicCAS(&thread_ht[hash_val].key.length, EMPTY, kmer_key.length);
      if(prev == EMPTY){
        thread_ht[hash_val].key.start_ptr = kmer_key.start_ptr;
        thread_ht[hash_val].val = {.hi_q_exts = {0}, .low_q_exts = {0}, .ext = 0, .count = 0};
      }
    }

    if(!done) {
      if(prev != EMPTY && thread_ht[hash_val].key == kmer_key){done = 1;}
        else if (prev == EMPTY){done = 1;}
    }

    if(__all(done))
      return thread_ht[hash_val];
    if(!done) {
      hash_val = (hash_val + 1) % max_size;
      if(hash_val == orig_hash){ printf("*hashtable full*\n"); done = 1; }
    }
  }
}

```

```
//SYCL Implementation of ht_get_atomic()
loc_ht& ht_get_atomic(loc_ht* thread_ht, cstr_type kmer_key, uint32_t max_size,
  const sycl::nd_item<3> &item_ct1,
  const sycl::stream &stream_ct1){
  sycl::sub_group sg = item_ct1.get_sub_group();
  unsigned hash_val = MurmurHashAligned2(kmer_key, max_size);
  unsigned orig_hash = hash_val;

  while(true){
    int prev = dpct::atomic_compare_exchange_strong<sycl::access::address_space::generic_space>(
        &thread_ht[hash_val].key.length, EMPTY, kmer_key.length);

    if(prev == EMPTY){
      thread_ht[hash_val].key.start_ptr = kmer_key.start_ptr;
      thread_ht[hash_val].val = {.hi_q_exts = {0}, .low_q_exts = {0}, .ext = 0, .count = 0};
    }
    sg.barrier();

    if(prev != EMPTY && thread_ht[hash_val].key == kmer_key){return thread_ht[hash_val];}
    else if (prev == EMPTY) { return thread_ht[hash_val];}

    hash_val = (hash_val + 1) % max_size;
    if(hash_val == orig_hash){ stream_ct1 << "*hashtable full*\n"; }
  }
}
// This implementation was designed for and has been tested for correctness on Intel GPUs generation 9 and
    Xe-HPC architectures.
```

APPENDIX B
ARTIFACT DESCRIPTION APPENDIX: PERFORMANCE MODELING AND ANALYSIS OF A *de Bruijn* GRAPH BASED LOCAL ASSEMBLY KERNEL ON MULTIPLE VENDOR GPUS

A. Abstract

This artifact contains the CUDA, HIP, and SYCL code for the local assembly kernel of MetaHipMer, as well as instructions to install and run them on Nvidia, AMD, and Intel hardware. It also includes the commands and scripts used for profiling. The data collected from profiling was used to create Tables 3, 6 and Figures 5-9.

B. Description

1) Check-list (artifact meta information): Fill in whatever is applicable with some informal keywords and remove the rest

- **Algorithm: Local Assembly using de Bruijn Graphs**
- **Program: CUDA, HIP, & SYCL**
- **Compilation: CUDA 12.0, ROCm 5.3.0, Intel DPC++ 2023**
- **Data set: Test datasets are included in the Github repository**
- **Run-time environment: Profiling results were completed on Perlmutter (Nvidia A100), Frontier (AMD MI250x), Sunspot (Intel Max 1550). The SYCL implementation was developed and tested on Intel DevCloud.**
- **Hardware: Nvidia A100, AMD MI250x, Intel Max 1550**
- **Output: Correct output for the sample input is provided in the github repository.**
- **Experiment workflow: Git clone. Switch to correct branch (ie CUDA, HIP, SYCL). Run test program.**
- **Publicly available?: Yes**

2) How software can be obtained (if available): Software can be obtained at the github repository

`https://github.com/leannmlindsey/gpu_local_ht.git`

3) Hardware dependencies: The software was tested on the following platforms: Nvidia Ampere GPU (A100), AMD Instinct GPU (MI250x), Intel Data Center Max 1500 (MAX1550)

4) Software dependencies: CUDA 12.0, Nsight-compute, ROCm 5.3.0, ROCProf, Intel DPC++ 2023, Intel Advisor

5) Datasets: All datasets used for testing and profiling are included in the github repository folder `/locassm_data`

C. Installation

The github repository contains the original optimized CUDA local assembly code, as well as the ported HIP and SYCL versions of the code. The repository has three branches, the `main` branch contains the CUDA code, the `hip` branch contains the HIP code, the `sycl` branch contains the SYCL code. Installation instructions for each branch are included in the `README.md` file in the corresponding branch. Installation instructions include a test script that verifies the results for correctness against a result file.

1) Nvidia/CUDA:

```
git clone
https://github.com/leannmlindsey/gpu_local_ht.git
cd gpu_local_ht
git status # verify you are on the main branch
nvidia-smi # verify you have access to an Nvidia GPU
cd src
mkdir build
./test_script.sh
```

2) AMD/HIP:

```
git clone https://github.com/leannmlindsey/gpu_local_ht.git
cd gpu_local_ht
git checkout hip
git status # To verify you are on the hip branch
rocm-smi # To verify that you have access to an AMD GPU
cd src
mkdir build
./test_script.sh
```

3) INTEL/SYCL:

```
git clone https://github.com/leannmlindsey/gpu_local_ht.git
cd gpu_local_ht
git checkout sycl
git status # To verify you are on the sycl branch
icpx -fsycl -I . *.cpp -o ht_loc
./test_script.sh
```

D. Experiment workflow

After the software is installed and tested on all three platforms, profiling can be completed following the instructions listed below. The profiling data can then be used to create Tables 3, 6 and Figures 5-9.

When profiling, the `<exe>` and `<params>` for each dataset can be found in the file `test_script.sh`. One example is listed below:

```
<exe> <input file> <k-mer length> <output file>
./ht_loc localassm_extend_7-21.dat 21 res_localassm_extend_7-21.dat
```

1) Nvidia/CUDA:

```
ncu -o nvidia_profiling.out --kernel-id iterative_walks_kernel
--metrics "smsp__inst_executed.sum, dram__bytes.sum,
sm__cycles_elapsed.avg, sm__cycles_elapsed.avg.per_second"
<exe> <params>

INTOPs = smsp__inst_executed.sum
HBM Bytes = dram__bytes.sum
Time=(sm__cycles_elapsed.avg/sm__cycles_elapsed.avg.per_second)
```

2) AMD/HIP:

```
rocprof -i rocprof.txt -o amd_profiling.out  <exe> <params>

contents of rocprof.txt:
# IOPS
pmc: SQ_INSTS_VALU_INT32 SQ_INSTS_VALU_INT64
# HBM Bandwidth
pmc: TCC_EA_RDREQ_sum TCC_EA_RDREQ_32B_sum
TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_sum

INTOPs = 64(SQ_INSTS_VALU_INT32 + SQ_INSTS_VALU_INT64)
HBM Bytes = 32* TCC_EA_RDREQ_32B_sum  +
  64 * (TCC_EA_RDREQ_sum-TCC_EA_RDREQ_32B_sum) +
  32 * (TCC_EA_WRREQ_sum{TCC_EA_WRREQ_64B_sum) +
  64 * TCC_EA_WRREQ_64B_sum
```

The kernel time can be taken from the rocprof output file.

3) Intel/SYCL:

```
advisor --collect=roofline --profile-gpu
--project-dir=./gpu_local_ht
-- <exe> <params>
```

The kernel time, INTOPs and HBM Bytes can be taken from the Intel Advisor HTML output file.

E. Evaluation and expected result

The profiling data can be used to calculate the metrics that were visualized in Tables 3, 6, and Figures 5-9. We used warp level INTOPs in our calculations.