Accelerating Large Scale *de novo* Metagenome Assembly Using GPUs

Muaaz Gul Awan, Steven Hofmeyr, Rob Egan, Nan Ding, Aydin Buluc, Jack Deslippe, Leonid Oliker
Lawrence Berkeley National Laboratory
Berkeley, California, USA
mgawan@lbl.gov

Katherine Yelick
University of California
Berkeley, California, USA
Lawrence Berkeley National Laboratory
Berkeley, California, USA
kayelick@lbl.gov

ABSTRACT

Metagenomic workflows involve studying uncultured microorganisms directly from the environment. These environmental samples when processed by modern sequencing machines yield large and complex datasets that exceed the capabilities of metagenomic software. The increasing sizes and complexities of datasets make a strong case for exascale-capable metagenome assemblers. However, the underlying algorithmic motifs are not well suited for GPUs. This poses a challenge since the majority of next-generation supercomputers will rely primarily on GPUs for computation. In this paper we present the first of its kind GPU-accelerated implementation of the local assembly approach that is an integral part of a widely used large-scale metagenome assembler, MetaHipMer. Local assembly uses algorithms that induce random memory accesses and non-deterministic workloads, which make GPU offloading a challenging task. Our GPU implementation outperforms the CPU version by about 7x and boosts the performance of MetaHipMer by 42% when running on 64 Summit nodes.

CCS CONCEPTS

• **Computing methodologies** → *Self-organization*; **Self-organization**; • **Applied computing** → **Computational genomics**; **Bioinformatics**; **Computational genomics**.

KEYWORDS

metagenomic, genomic, GPU, CUDA, sequence assembly, sparse data structures, graph algorithms

ACM Reference Format:
Muaaz Gul Awan, Steven Hofmeyr, Rob Egan, Nan Ding, Aydin Buluc, Jack Deslippe, Leonid Oliker and Katherine Yelick. 2021. Accelerating Large Scale *de novo* Metagenome Assembly Using GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3458817.3476212

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476212

1 INTRODUCTION

With the rapid development of genome sequencing technologies it is now possible to sample and study genomes at an unprecedented scale [5]. This includes the study of uncultured micro organisms which have to be sampled as a group from their environment [27]. The majority of micro organisms cannot be cultured in the lab and have to be sampled directly from the heterogeneous communities of micro organisms that are found living in different environments [15]. The study of these microbial communities has unveiled important and complex relationships that exist within these communities as well as the associations they form with their environments. Among other benefits, metagenomic studies pave way for better antibiotics [15], understanding the impact of climate change [18], forensics [29] and understanding different environments and their impact on life [30] [10].

Collective sampling of micro organisms leads to very large and complex datasets which cannot be processed and analyzed using regular genome-assembly approaches that have been developed for the study of single genomes [5][13][21]. A typical metagenomic workflow involves sampling DNA reads (short strands of DNA obtained using sequencing technologies) from an environmental sample and trying to rebuild the contiguous regions of the underlying genomes. DNA reads are redundant, repetitive, error prone and when obtained from an environmental sample they may contain proportional bias depending upon the population of different organisms in the sample. This makes metagenome assembly significantly different and more complex than single genome assembly, where the sample contains genetic material from only a single cultured organism. Since the majority of single-cell organisms have never been sequenced there is no reference genome present and this process has to be done *de novo*, which further compounds the problem.

With the advent of modern sequencing technologies the speed of metagenomic data generation has far exceeded the computational capabilities of both software and hardware. Metagenome datasets are very large in size and assembling even a moderately-sized sample exceeds the memory capacity of a typical shared-memory server [9]. The bioinformatics software community has attempted to overcome these challenges through the development of memory efficient methods combined with techniques like multiassembly (assembling parts of a dataset separately and then taking consensus), but these result in long run times and poor assembly quality [13] [9]. To overcome these challenges, MetaHipMer was introduced as a large-scale metagenome assembler that can leverage the large memory and compute capacities of supercomputers to coassemble terabase-scale

datasets [7]. MetaHipMer's performance and accuracy in comparison with other state-of-the-art metagenome assemblers have been studied at length [9].

Originally, MetaHipMer was written in UPC and MPI, but it has subsequently been completely rewritten in UPC++; this new version is called MetaHipMer2. Throughout this paper we use the term *MetaHipMer* when referring to MetaHipMer in general (including all versions) and we use the term *MetaHipMer2* when we are talking about the specific implementation, since we used the recent most version for all the experimentation.

MetaHipMer's continuous development has kept it on par with the increasing challenges posed by the ever-increasing metagenome data sizes. With the ongoing development of next-generation supercomputers, which derive a significant portion of their computational power from Graphics Processing Units (GPUs), it has become important to evolve MetaHipMer to exploit GPU processing. Many bioinformatics algorithms (such as the majority in metagenome assembly) are not amenable to GPUs because of their sparse and irregular nature. Furthermore, lack of support for dynamic containers like hash tables, strings and vectors on GPUs makes the implementation of bioinformatics algorithms on GPUs a challenge [31].

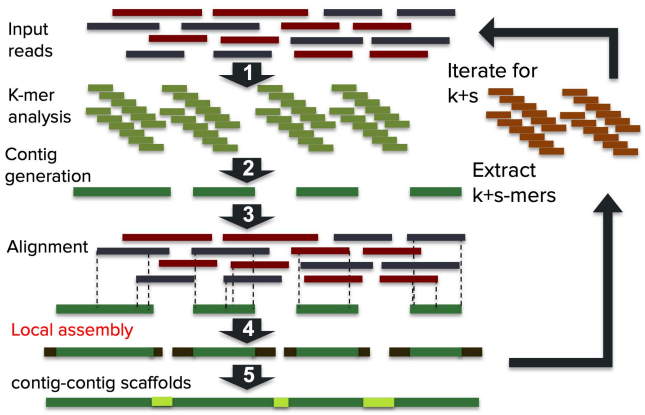


Figure 1: An overview of the MetaHipMer2 pipeline. The Local Assembly module (shown in red) takes up the largest percentage of total run time for many datasets.

As shown in Figure 1, MetaHipMer has a complex workflow that consists of independent components performing different stages of the assembly pipeline. A breakdown of a single MetaHipMer run in Figure 2a shows that about one third of the total execution time is spent in the *Local Assembly* module, which, true to its name, is performed locally on each node, in contrast to other modules, which spend most of the time in communication across processes. The local assembly module involves constructing thousands of hash tables in parallel and then using those hash tables to perform contig (contiguous strand of DNA) extension walks. The CPU implementation of local assembly relies heavily on dynamic structures like hash tables, vectors and strings, which are a challenge to implement on GPUs. In addition, the algorithmic motif of local assembly induces a random memory-access pattern with a non-deterministic amount of work, which makes implementing this module on GPUs a very challenging problem. In this paper, we present an optimized GPU implementation of local assembly that is integrated in the MetaHipMer pipeline. Our experiments on a real-world dataset show 7x faster performance when using GPUs and an overall pipeline speedup of 42%.

2 BACKGROUND

2.1 Literature Review

Depending on the type of study and the availability of a reference genome, sometimes it is possible to align the DNA reads against a reference genome. This process at its core involves a dynamic programming technique that has been shown to do well on GPUs because of a predictable and recurring memory access pattern [1] [16] [3] [26]. For metagenomics studies, reference genomes are rarely available and hence assemblies are done *de novo*. Assembling a genome of cultured organisms *de novo* without the help of a reference genome is a well studied problem. Typically this is performed using assembly-graph approaches, which can take two forms [25]. The first is the de Bruijn graph approach, which builds a graph using short sub-strings of DNA reads known as *k-mers* [23], where each k-mer is represented by a vertex in the graph and an edge connects two vertices when the two corresponding k-mers overlap. The underlying genome can be assembled by finding a path in the de Bruijn graph subject to certain constraints. The second approach is the overlap graph, which uses full DNA reads for vertices, instead of k-mers, and edges correspond to an overlap between the two reads [14]. A path through the overlap graph corresponds to the underlying genome. DNA reads are error prone, redundant and may not cover the complete genome. This makes the task of finding a path through an assembly graph very hard. Typically, the overlap approach is only used for long reads, because the computational cost of doing an all-to-all alignment of all reads is prohibitive for datasets that have many more shorter reads.

There have been multiple approaches to accelerating genome assembly using GPUs. Some of the popular GPU-accelerated genome assemblers include LaSAGNA [8], which uses the overlap string-graph approach and utilizes GPUs for operations like sorting and prefix scan, but performs most of the assembly-graph related computations on the CPU. It achieves a performance improvement of about 3x over a CPU-only parallel assembler. Another genome assembler, GPU-Euler [19], uses the de Bruijn graph approach and performs most of the assembly-graph related operations on GPUs. GPU-Euler demonstrated a speedup of about 5x against a serial CPU approach. GAGM [11] is another GPU-accelerated assembler, which also uses the de Bruijn graph approach and implements an almost end-to-end GPU-offloaded workflow, with some CPU operations. It showed a speedup of about 7x over a CPU-based parallel assembly approach that was run using a quad core CPU. Some of the assembly approaches perform an additional step of aligning different parts of intermediate contigs using sequence-alignment algorithms; sequence alignment is an expensive process but more amenable to GPUs than the rest of the graph-based algorithms. GrassHopper [28] is one such assembler: it keeps most of the assembly pipeline on the CPU and offloads the sequence-alignment portion to GPUs. Another GPU-accelerated genome assembler uses bi-directed de

Bruijn graphs and performs assembly-graph construction on GPUs [17], and comparing only the graph construction phases of CPU-only approaches against their GPU approach, they demonstrated a speedup of between 2 to 7x. The CPU-only assemblers used for comparison took advantage of multiple CPU cores.

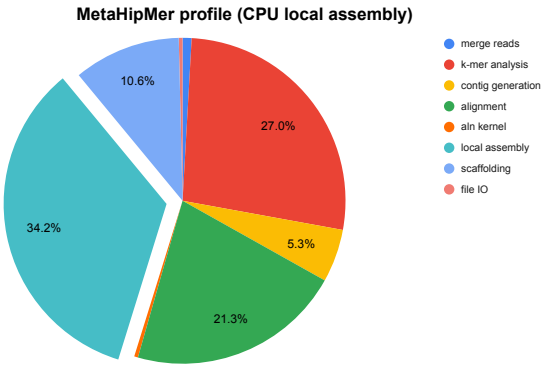
All of these GPU-accelerated genome assemblers were challenged by the nature of algorithms involved and the data and memory-intensive nature of the assembly problem. This is due to the irregular memory-access patterns induced by assembly-graph construction, graph traversal and hash-table construction and probing. Furthermore, the limited memory on GPUs poses another challenge due to low CPU-GPU communication bandwidths. Despite these challenges, GPU-accelerated genome assemblers have demonstrated performance improvements over the CPU-only versions. However, the set of challenges for metagenomic assembly go beyond those discussed above and are further compounded because of the larger and more complex datasets involved.

Metagenome assembly requires a more specialized approach and the existing genome-assembly approaches cannot be used here. Some of the popular metagenome assemblers include MetaHipMer [7], Megahit [13] and MetaSpades [21]. Megahit and MetaSpades are shared-memory assemblers and are not capable of processing datasets that exceed the memory requirement of a single node. MetaHipMer can scale across multiple nodes and is capable of handling terabase-scale datasets [9]. Because of the challenges in GPU-acceleration of the algorithms involved, little work has been done in porting metagenome assembly to GPUs. Megahit demonstrated usage of GPUs in one of its earlier versions by offloading the *k-mer* counting phase to GPUs; however, in the recent work it has been shown that a well optimized CPU-only version of Megahit can outperform the earlier GPU-accelerated version [13]. By contrast, MetaHipMer has seen clear performance gains from the use of GPUs (in alignment operations) [3].

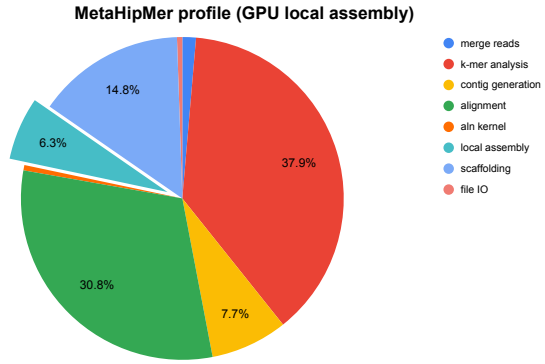
2.2 MetaHipMer

MetaHipMer is the only known distributed-memory metagenome assembler that can scale to thousands of nodes while matching the quality of state-of-the-art shared-memory assemblers [9]. MetaHipMer uses an iterative de Bruijn graph approach to build contiguous strands of DNA, known as *contigs*, from the input DNA reads [24]. It also uses a specialized scaffolding method that stitches together multiple contigs to increase the lengths of final contigs.

The MetaHipMer pipeline consists of multiple components that perform the complete assembly as shown in Figure 1. It starts by breaking down the input reads into shorter sub-strings of length k known as k-mers. After filtering out erroneous k-mers (those that occur only once), these are then used for constructing a de Bruijn graph. Unambiguously connected paths in the resulting graph are traversed to obtain contiguous regions of genomes. This is followed by the alignment stage, where contigs are aligned against reads to find reads that can be used to further extend the contigs. The reads that align to the ends of contigs are then used for extending the contigs in both directions using the local assembly module. The local assembly module also performs de Bruijn graph construction and traversal, but does so separately for the reads that align to each contig end. And finally, the contigs generated in the previous stages are connected together in the scaffolding phase to further increase the contiguity.



(a) MetaHipMer2 run time breakdown when using CPU local assembly. Total time 2128 **seconds**



(b) MetaHipMer2 run time breakdown when using GPU local assembly. Total time 1495 **seconds**

Figure 2: Pie charts here show run time breakdown in percentage for different phases of MetaHipMer2 when running on 64 nodes of Summit system using the marine communities dataset [12].

Previously, only a part of the alignment phase in MetaHipMer had been offloaded to GPUs [3]. A time breakdown of MetaHipMer in Figure 2a shows that 34% of total time is spent in the local assembly for one particular dataset on 64 OLCF (Oak Ridge Leadership Computing Facility) Summit nodes. The exact fraction of time spent in any one stage depends on the dataset and the concurrency, but the time taken in local assembly is always a significant part of the overall run-time. Though not the most GPU-amenable part of the pipeline, most of local assembly is performed locally on each node, in contrast to the other phases, which are distributed in nature and are dominated by network communication. Thus local assembly is a good candidate for GPU offloading.

2.3 Local Assembly Module

Metagenome samples are complex, containing populations of different organisms with varying abundance, which leads to sequencing machines sampling some organisms more than others. Sequencing errors and commonality of DNA fragments across organisms can result in the erroneous overlap of paths in the global de Bruijn graph, producing unresolvable forks. Local assembly addresses this issue by performing the de Bruijn graph traversal entirely locally, using only the reads that map to the end of a contig to extend that end.

Local assembly is performed in an embarrassingly parallel manner in the current CPU-only version. It is a two-step process which is iteratively performed until the correct termination condition is met. Each process picks a local subset of contigs and obtains the candidate reads that align to each contig. Reads for right and left sides of the contigs are kept separate. The same algorithm is repeated for both right and left sides of each contig.

Algorithm 1 kmer hash table construction

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

Algorithm 2 DNA walks

```
1: c ← contig
2: walk ← empty
3: walk_state ← empty
4: kmer ← get_kmer(c)
5: for i = 0 to i < max_walk_len do
6:     if loop_exists(kmer) then
7:         end_walk
8:     end if
9:     ext ← kmer_ht.lookup(kmer)
10:    if ext == end||ext == fork then
11:        walk_state ← ext
12:        break
13:    end if
14:    walk+ = ext
15:    kmer = next_kmer(kmer, ext)
16: end for
```

Once all the data are available locally, the first step is to build a k-mer hash-table from all the reads, where k-mers are used as keys and the values are extensions. An extension object contains the base (nucleotide character) that follows the k-mer and also contains information about the quality of that base and counts (number of occurrences) for that k-mer. The process of inserting k-mers in the hash table is repeated for all the candidate reads for each contig; pseudo code for this can be seen in Algorithm 1. The second step is to perform *mer-walks*, as discussed in [6] and [7]. A mer-walk slices a k-mer from the end of a contig and looks that up in the hash table, and if the k-mer is available, the corresponding extension base is appended to the end of contig. This is repeated until a dead-end or a fork is encountered (see Algorithm 2). If a fork is encountered **k** (the length of the k-mer) is increased or up-shifted and the whole process starting from the first step is repeated; in case of a dead-end k is downshifted. The mer walk phase terminates when a fork is encountered after downshifting or when a dead-end is met after up-shifting the value of k.

2.4 Challenges in GPU offloading

The local assembly module posed three types of challenges for GPU offloading:

- There is a lack of support for dynamic containers on GPUs. As discussed above, the underlying algorithm relies heavily on hash tables and string resizing, both of which require dynamic memory allocation, which is not supported on GPUs.
- There is limited memory available on GPUs. The combined memory of all six GPUs on a Summit node is 96 GBs, while the total DRAM available to the CPUs on the same node is 512 GB [22]. This limits the amount of work that can be offloaded to GPUs.
- The memory access patterns and work distribution in local assembly are not well-suited to GPUs. Hash tables introduce a memory access pattern that is random in nature and leads to non-coalesced memory accesses on GPUs, which incur a large performance penalty. Furthermore, *mer walks* need to be done sequentially and their final lengths cannot be determined beforehand. This creates a highly random workload which leads to warp stalling on GPUs.

These challenges make local assembly and metagenome assembly in general a hard problem for GPUs, both from the perspective of implementation complexity and difficulty of obtaining good performance.

3 GPU-ACCELERATED LOCAL ASSEMBLY

In this section we discuss our approach to offloading the local assembly module to GPUs and address the above challenges and describe the solutions that we employed. For this work we used NVIDIA's CUDA API for GPU offloading. An overview of the GPU-accelerated local assembly can be seen in Figure 4; various aspects of our implementation are detailed below.

3.1 Data binning for better load balance

Each contig from the upstream pipeline can have a varying number of candidate reads used to extend it in the local assembly phase. This number can range from zero to up to 3000 (an empirical upper limit). The larger the number of candidate reads, the more work is required. Offloading all the contigs without any type of sorting would lead to load imbalance across warps, which will cause a few threads of the warps to stall the remaining, and result in poor performance. As a solution, we sort the contigs on the CPU side into three bins based on the number of candidate reads. Bin limits were chosen based on an empirical analysis: the first bin contains those contigs which have zero reads; the second bin contains all those contigs

which have fewer than 10 reads; and all the remaining contigs go into the third bin. Our studies showed that typically the majority of contigs have zero reads while the percentage of contigs assigned to the second bin may vary between 10% and 30%, and the third bin gets less than 1% of contigs. An example contig distribution across bins (for the arcticsynth [9] dataset) can be seen in Figure 3.

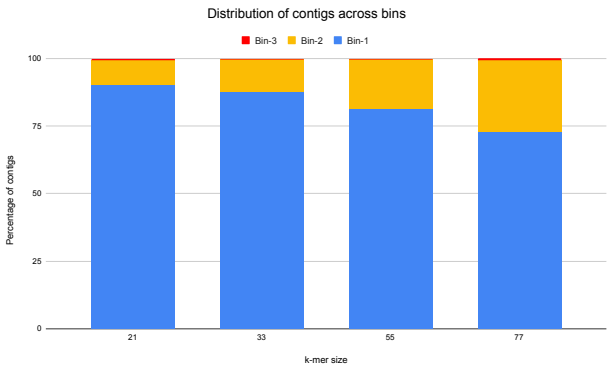


Figure 3: Distribution of contigs across three bins for the arcticsynth [9] dataset. It can be seen that Bin-3 consistently gets less than 1% of total contigs while Bin-2 varies between 10% and 30%. Larger k-mer size leads to larger number of contigs having candidate reads greater than zero.

The first bin is not offloaded to GPUs and is returned without performing any extensions since there are no reads for the contigs to be extended against. For the second and third bins, separate kernel loops are launched. This sorting ensures that contigs which can be processed quickly are launched on a separate kernel and do not face long warp-stalling delays while the contigs that require the longest time are grouped together on a separate kernel. Note that even though the third bin contains less than 1% of the total contigs, it still might take up most of the computational time.

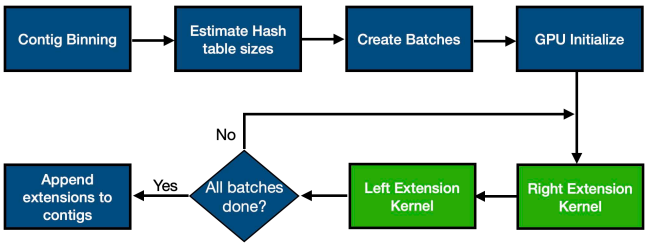


Figure 4: Overview of GPU local assembly module.

3.2 Minimizing device memory usage

The total size of all the k-mers in a read is much larger than the memory occupied by a single read. A read of length l will occupy l bytes, assuming each character is stored in a byte. The total memory in bytes occupied by the k-mers of the same read would be $(l - k + 1) * k$. When performing local assembly on GPUs it is important to conserve memory so that maximum amount of data and hence work can be offloaded. This is particularly important because local assembly is a memory-bandwidth limited algorithm and one of the ways to get performance is to offload as much work to the GPUs as possible. This is also helpful because of the latency-hiding nature of GPUs i.e. while data for one warp is not available, another warp can be scheduled to advance.

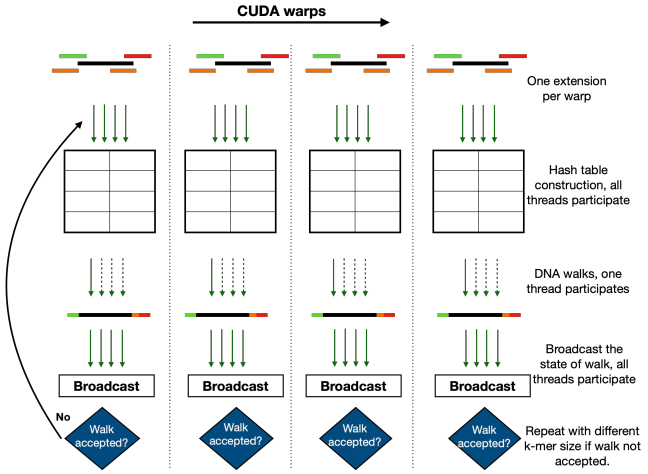


Figure 5: Overview of GPU local assembly extension kernel. Dotted green arrows show threads that have been masked out. One contig is assigned per CUDA warp.

To minimize memory usage and maximize the amount of work that is offloaded we used measures to re-use data that was stored in global memory. Our analysis showed that largest amount of memory on the GPUs was being used by the hash tables. The algorithm uses two different hash tables for extending each contig: one for storing k-mers and extensions, and another for keeping track of k-mers visited (to avoid cycles). Most implementations of hash tables require them to resize depending on their load factor. A load factor is the percentage of hash table slots that have been filled. A higher load factor may lead to increased hash collisions and consequently longer insertion and access times. On the GPU we could not enable the resizing of hash tables because of limits on dynamic memory allocation. A naive way would be to compute the size of largest possible hash table for all the extensions and reserve that much memory for each extension but this quickly takes up all the GPU global memory. As a workaround, we compute the exact size of hash table that would be needed by any contig extension and store all the sizes in an array called *ht_sizes*. Then we compute the total memory used by all the hash tables and allocate that on the GPU. With the help of the *ht_sizes* array we can keep track of the start and end of each hash table and minimize the overall memory-usage. The size of a hash table depends on the number of maximum unique k-mers that exist in a set of candidate reads: assuming that each k-mer is unique, the size of hash table would be $(l-k+1)*r*H$, where l is the maximum read-length, k is the k-mer size, r is the total candidate reads for a contig, and H is the size of each entry in the hash table. In order to prevent the hash table from reaching high load factors we use $l * r * H$ as the hash table

size. This limits the hash-table load factor to a maximum of about 0.93. Using this method we were able to minimize the total GPU memory allocated for hash tables while maximizing the hash-table performance. The load factor for the above discussed strategy can be computed as:

$$\text{Load Factor} = \frac{l - k + 1}{l - k}$$

To compute the worst-case load factor, we replace l with the longest-possible read length and k with the shortest-possible k-mer length. Short reads have a maximum length of 300 while the shortest k-mer length for reasonable accuracy is 21, giving:

$$\text{Worst Case Load Factor} = \frac{300 - 21 + 1}{300} = 0.93$$

As discussed above, the constituent k-mers of a read use up more space than the read itself. However, k-mers can also be obtained directly from the reads if the start location and length of a k-mer is known. As shown in Figure 6, instead of storing complete k-mers in the hash table on GPUs, we just store the pointer to the start location of the k-mer and the length of the k-mer. For instance, to store a k-mer of length 77, same number of bytes will be required. However, to store the same k-mer using the technique in Figure 6 would require just 5 bytes using about 15x less memory per k-mer. This helps free a significant amount of memory overall. We use a similar strategy in the DNA-walks phase by storing only the pointers to different k-mers in reads instead of storing complete k-mers.

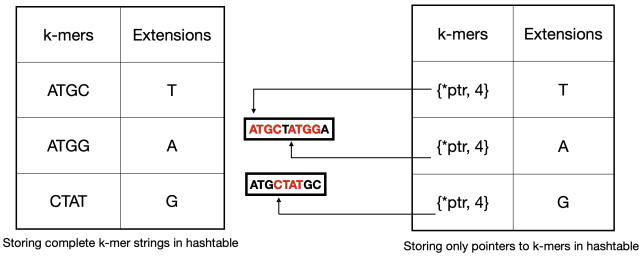


Figure 6: The table on the left stores complete k-mers while the table on the right stores only the pointers to the k-mer starting position inside the stored read along with the length of k-mer. The k-mers in the right table are same as those in the table on the left.

3.3 Warp Local Hash table construction

The first step in local assembly is the construction of the k-mer hash table for each contig using its candidate reads. This is performed using the method outlined in Algorithm (Pseudo Code 1). For the GPU implementation we assigned one CUDA warp to construct one hash table as shown in Figure 5. A CUDA warp is the smallest unit of threads that is scheduled onto the GPU hardware. Each warp on NVIDIA hardware consists of 32 threads. There are several advantages that can be exploited when working on a warp level, for instance threads of a warp can communicate with each other through register-to-register direct data transfers and threads can be synchronized within a warp.

We start the hash-table construction by mapping the threads of a warp to the first candidate read such that the threads point to contiguous k-mers in that read, as shown in Figure 7. Each thread inserts the k-mer assigned to it into the hash table using the murmurhash2 function [2]. During hash table insertions there are couple of scenarios that can occur; these are shown in Figure 7 and discussed below:

- **Hash Collisions:** In case a thread runs into a hash-table entry that is already occupied, we perform a key-to-key comparison where the keys are the k-mer being inserted and the k-mer that is already present. If the keys are a match we increment the k-mer count and update the quality metric for its extension. If the keys do not match, that is a hash collision and is resolved using linear probing where the key is inserted in the very next available empty slot.
- **Thread Collisions:** This happens when two threads get the same k-mer, for example in Figure 7. This creates a unique situation where two threads are competing to insert the same k-mer at the same location. To resolve such a situation an exclusive region is needed, however, exclusive regions cannot be created on a GPU. As a solution we use CUDA's compare and swap (CAS) atomic operation to mark an entry as occupied exclusively by a thread. The CAS atomic ensures that only one of the colliding threads will be able to mark the entry as occupied; call this thread the *winning* thread. Next we use CUDA's *match_any_sync* operation to obtain a *mask* suggesting which of the threads in a warp are colliding and we synchronize them. Immediately after the synchronization we allow the winning thread to initialize the entry of the hash table inside an *if* block and then again synchronize the colliding threads after the *if* block using the *mask* from above. Once the entry has been added by the winning thread, the remaining threads involved in the collision can access the entry and update the values atomically.

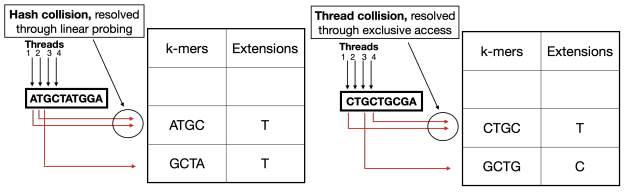


Figure 7: In the table on the left thread 1 and 2 get a different k-mer each but they both hash to the same location. This type of collision is resolved through linear probing, i.e. finding the very next empty slot. In the table on the right threads 1 and 4 get the same k-mer, which leads to a thread collision that is resolved using exclusive regions.

3.4 DNA walks

Once the k-mer hash table has been constructed, one thread among the 32 threads (which constructed the hash table) of a warp traverses the table to perform the DNA walks as described in section 2.3 and shown in Algorithm 2. For each warp, all the threads except the first one are masked out for this step since walks cannot be performed

in parallel (see Figure 5). Local assembly is an iterative process that repeats until an acceptable walk is found. If an acceptable walk is not found at the end of walk, the k-mer hash-table is reconstructed with a different k value to redo the DNA walks. The computation to check if a walk has been accepted or rejected is performed by the same thread that performs the walk. It is important to broadcast the information about the state of the walk (accepted or rejected) to other threads of the warp so that they can be masked or unmasked depending on if the hash table needs to be reconstructed. For each extension, the state of the walk is stored as a variable in a register and then that value is shared with other threads of the warp using shuffle instructions that enable inter-thread communication within the warp as discussed in section 3.3. An overview of the kernel implementation with details about thread participation can be seen in Figure 5.

4 PERFORMANCE ANALYSIS

To analyze the performance of the GPU-accelerated local assembly module we first studied its standalone performance and then integrated the module in MetaHipMer's most recent release *MetaHipMer2* to study the overall improvement in performance of MetaHipMer.

4.1 Test systems and datasets

The CUDA API from NVIDIA was used for the development of the GPU local-assembly kernel, and CUDA version 10.1.2 was used for building all the CUDA modules. UPC++ release 2021.3.0 was used for building MetaHipMer from source. For all the following studies we obtained the source code of MetaHipMer2 from the master branch at git commit 80*ad*090. CPU-only runs were performed by using that exact same version while for GPU integration we started from the above version.

For the standalone performance-analysis we used the Cori system's GPU partition [20]. Each node on the Cori GPU partition contains eight V100 NVIDIA GPUs with two sockets of Intel Skylake processors with 20 CPU cores each. The total available DRAM on each node is 384GB while each NVIDIA V100 has a total global memory of 16GB. For the large-scale studies we used the Summit supercomputer [22]. Each node of Summit consists of two POWER9 processors and six NVIDIA V100 GPUs. Total RAM available to CPUs is 512GB while each GPU contains 16GB of global memory.

For small-scale tests on fewer nodes, we used the arcticsynth dataset [9], which contains 32 million synthetic reads of length 150 bp, and for large-scale tests we used the WA dataset, which is a collection of marine microbial communities from the Western Arctic Ocean and consists of 813 GB of $2,465,328,090$ Illumina HiSeq paired-end reads of length 150 [12].

For standalone runs we used the arcticsynth [9] dataset and processed it through the MetaHipMer pipeline to dump the contigs and their candidate reads that are input to the local assembly module. This data dump was then used to evaluate the performance of the GPU local-assembly kernels.

4.2 Roofline Analysis

To understand the device utilization and the algorithm's limitations we made use of the Instruction Roofline [4]. The Instruction Roofline model is a visually-intuitive method to understand the performance of a given kernel based on a *bound-and-bottleneck* analysis approach. It allows us to analyze instruction throughput and quantify efficiency of memory accesses. Generally, the Instruction Roofline model characterizes a kernel's performance in billions of instructions (GIPS, y-axis) as a function of its instruction intensity (II, x-axis). *Instruction Intensity* is defined as warp instructions per memory transaction of memory traffic. In an instruction roofline, the plotted instruction dots show the performance and utilization by a kernel. The further these solid dots (instruction throughput) are to the upper right corner of the figure, the better performance/utilization that kernel achieves. For the open dots (memory pattern), the closer to the rightmost memory wall they are, the higher the memory-access efficiency of the kernel.

We evaluated two different versions of the GPU local-assembly module that were obtained from its development cycle. The first version (referred to *v1*) used only a single CUDA thread for k-mer hash-table construction (see Figure 8), while the second version (referred to as *v2*) used a single warp (thirty-two threads) per k-mer hash-table (see Figure 9). For both versions the second step of DNA walks is performed by a single thread per extension.

The v2 kernel extracts k-mers from a read such that threads of a warp access contiguous memory locations, as shown in Figure 7, which reduces the number of global memory transactions. It is apparent from Figures 8 (v1) and 9 (v2) that the L1 dot moves in the upper-right direction when moving from v1 to v2. This demonstrates that v2 achieves a higher instruction throughput and a better instruction intensity by reducing the number of global memory transactions and global memory load/store instructions. Figure 10 visualizes the instruction breakdown of the two implementations, and it can be observed that the number of global memory instructions is significantly reduced. In addition, the Roofline plots also show that v2 utilizes the memory bandwidth better than v1. However, this does not improve the memory access patterns while inserting in the hash table, but parallel insertions do improve performance. It can be further observed that both versions are close to the Stride-1 memory wall due to the nature of the random access of the hash table.

The dotted line in the Roofline figure shows the number of non-predicated warp instructions that can be achieved by the underlying kernel. The gap between the red dot and the dotted line indicates the presence of thread predication. This shows that both v1 and v2 kernels suffer from thread predication. Most of the thread predication occurs in second step of the algorithm, where the majority of the threads have to be masked out (Figure 5) to perform sequential DNA-walks and the amount of work is non-deterministic and varies greatly across threads. For instance, a DNA walk can be up to 300 steps long for some threads while for another it might terminate right at the start. For v2, thread predication decreases moderately because multiple threads are utilized for building hash tables.

In conclusion none of the versions achieve close to peak performance, but with v2 achieving a higher peak performance of 14.4 GIPS. This is mostly due to the nature of the algorithm, with randomly-accessed global memory and large usage of local memory. Approximately 70% of L1 memory transactions and L1 memory instructions are from local memory, which in this case limits the performance.

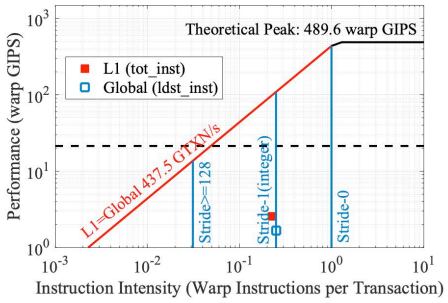


Figure 8: Instruction roofline for single thread version of extension kernel (v1)

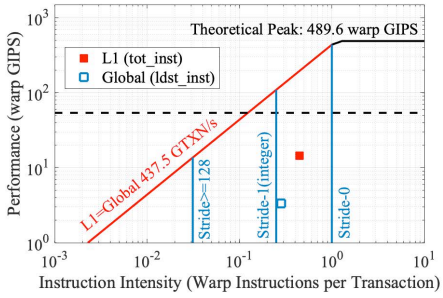


Figure 9: Instruction roofline for per warp version of extension kernel (v2)

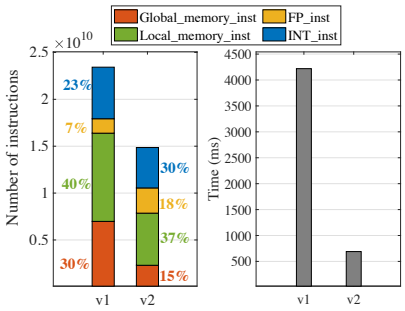


Figure 10: Performance insights breakdown

4.3 Integration in MetaHipMer

For testing the impact of GPU offloading of the local assembly phase on MetaHipMer's performance, we used the most recent version i.e. MetaHipMer2, which is written in UPC++. The GPU module of local assembly was integrated into MetaHipMer2 using a driver function that performs all the CPU-side data packing, device-to-rank mapping, and launches device kernels. Use of a driver function allows for complete isolation of GPU and CPU codes which proves very helpful given that UPC++ and CUDA codes need to be built separately before being linked in. An overview of GPU local-assembly integration in MetaHipMer2 can be seen in Figure 11.

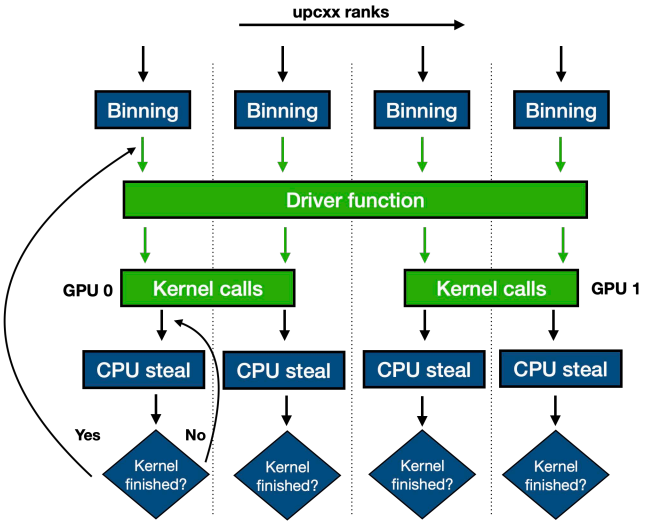


Figure 11: Integration of the GPU local-assembly module in MetaHipMer2. The green boxes show GPU related calls, while the green arrows represent the thread that runs in the background.

After all the needed data from multiple nodes has been collected, binning of the contigs is performed based on candidate read counts as described in section 3.1. Once the bins are available, the driver function call is made inside a new thread and the *third* bin containing contigs with the larger number of candidate reads is passed to the driver (Figure 11). Using a separate thread to launch GPU calls allows for the control to be returned to the parent thread where the CPU can start performing local assemblies on the *second* bin. As soon as the GPU code returns, the *second* bin is also offloaded to the GPU. In Figure 11 arrows colored in green show the newly forked thread that runs in background.

The reason to launch the third bin on GPU first was based on an empirical study which demonstrated that GPUs fair better when using contigs with larger numbers of reads i.e. when the amount of work is larger. This is true because GPUs are latency-hiding devices and more work allows them to cater for the random memory access patterns introduced by the underlying algorithm.

4.4 Performance Improvement

To study the performance improvement in the MetaHipMer2 pipeline with the newly integrated GPU local-assembly, we used the two datasets mentioned above. We obtained the previously mentioned version of MetaHipMer2 and used that for performing runs with CPU local-assembly and then changed the local assembly module as described in previous sections and redid the runs using the same datasets. MetaHipMer2 makes use of all the available cores on a node when using the CPU local-assembly module, and uses all the available GPUs when using the GPU-accelerated local assembly module.

For small-scale runs we used two summit nodes. Figure 12 shows that the GPU local-assembly outperforms the CPU local-assembly module by about 4.3x and leads to an overall run-time improvement of about 12%. Note that the amount of time spent in each module of MetaHipMer is dependent on the dataset used. For the arcticsynth dataset the overall time spent in the Local Assembly phase is about 14%, which is much smaller than when using the WA dataset as shown in Figure 2a.

Figure 13 shows the comparison between the run times of CPU and GPU versions of local assembly, which were extracted from the complete runs of MetaHipMer2 when processing the WA dataset. It can be seen that the GPU version performs more than 7x faster when running on 64 nodes, but the performance advantage deteriorates with increasing numbers of nodes (although it is still 2.65x faster at 1024 nodes). This is because of a decrease in the amount of work that can be offloaded to one GPU when running at larger scale, which causes larger GPU overheads.

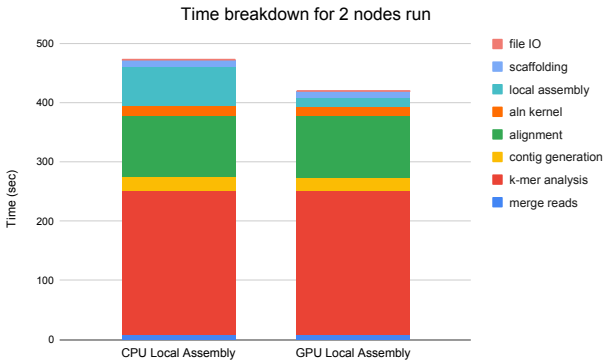


Figure 12: Run time comparison for two node runs using arcticsynth [9] dataset. Local assembly phase speeds up by about 4.3x when offloaded to GPUs.

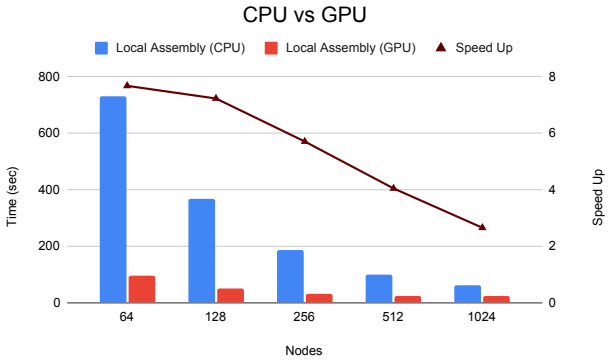


Figure 13: Comparison of run times for CPU and GPU versions of local assembly module. These run times were obtained from complete runs of the MetaHipMer2 pipeline on the Summit supercomputer.

Figure 14 shows the complete MetaHipMer2 pipeline run-time comparisons with and without GPU local-assembly. GPU local-assembly provides a peak performance improvement of about 42% at up to 128 nodes. This performance improvement decreases as the pipeline becomes dominated by communication with increasing numbers of nodes. At larger scale, the amount of work available for the GPUs also goes down and the advantage gained by offloading the local assembly phase to GPUs is decreased, which is expected since we are strong scaling. The sudden drop in the speedup when going from 512 to 1024 nodes is due to the fluctuation in the time spent in communication-heavy portions of the code. Averaging over multiple runs would give a more linear drop in speedup; however, because of limited resources, we performed only single runs for 512 and 1024 nodes. The expected variation was inferred by performing smaller runs multiple times.

Figure 2b shows that overall for 64 nodes the local assembly portion is reduced to just 6% of the total run time from 34% of total when it was CPU-only as shown in Figure 2a.

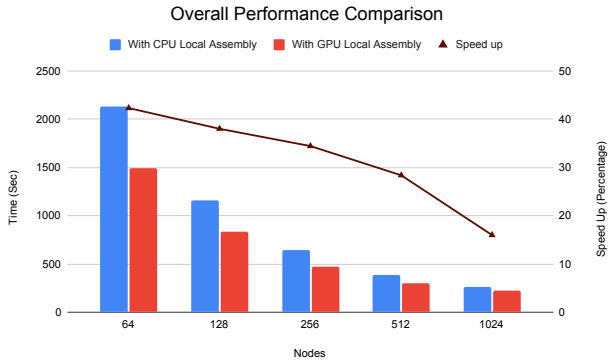


Figure 14: The total run time of the MetaHipMer2 pipeline when run with and without GPU local assembly support.

5 CONCLUSION AND FUTURE WORK

In this paper we presented a first-of-its-kind effort of accelerating a large-scale metagenomic-assembly pipeline using GPUs. Metagenomic assembly relies on algorithms that are irregular and sparse in nature, which makes them a challenging problem for GPUs. With modern genome-sequencing technologies, large and complex metagenomic datasets are being generated so rapidly that soon the data will outpace the current state-of-the-art software tools. This makes a strong case for exploiting exascale era supercomputers, which will utilize GPUs for the bulk of their computational capabilities. To this end, we identified and accelerated one of the most computationally dominant portions of MetaHipMer, which is the only large-scale metagenomic assembler available to the community. The local assembly module of the MetaHipMer pipeline uses hash tables, vectors and string resizing to perform DNA walks to extend the contiguous regions of genomes. We used CUDA's warp level intrinsics and efficient methods of inter-thread communication to implement fast warp-local hash-tables and minimized the use of GPU global memory to offload the maximum amount of work to GPUs. This allowed us to better exploit the GPU's latency-hiding nature. We achieved speedups of up to 7x for the local assembly module when running at scale on the Summit supercomputer against MetaHipMer's own CPU implementation. We also demonstrated integration of a GPU-accelerated local assembly module in the MetaHipMer pipeline to obtain a performance boost of up to 42% when running on the Summit supercomputer.

We have demonstrated that by leveraging a GPU's latency-hiding nature and low-level programming intrinsics we can gain significant performance improvements in metagenomic analysis software. For future work we are moving towards offloading other modules of MetaHipMer to GPUs which pose different sets of challenges for GPUs, such as distributed data structures, sparse graphs and dominant communication times.

ACKNOWLEDGMENTS

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) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 and the resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Nauman Ahmed, Tong Dong Qiu, Koen Bertels, and Zaid Al-Ars. 2020. GPU acceleration of Darwin read overlapper for de novo assembly of long DNA reads. *BMC bioinformatics* 21, 13 (2020), 1–17.

[2] Appleby Austin. [n.d.]. *Murmurhash2.* https://sites.google.com/site/murmurhash

[3] Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. 2020. ADEPT: a domain independent sequence alignment strategy for gpu architectures. *BMC bioinformatics* 21, 1 (2020), 1–29.

[4] Nan Ding and Samuel Williams. 2019. An instruction roofline model for gpus. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 7–18.

[5] RA Leo Elworth, Qi Wang, Pavan K Kota, CJ Barberan, Benjamin Coleman, Advait Balaji, Gaurav Gupta, Richard G Baraniuk, Anshumali Shrivastava, and Todd J Treangen. 2020. To Petabytes and beyond: recent advances in probabilistic and signal processing algorithms and their application to metagenomics. *Nucleic acids research* 48, 10 (2020), 5217–5234.

[6] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. Hipmer: an extreme-scale de novo genome assembler. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[7] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. 2018. Extreme scale de novo metagenome assembly. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 122–134.

[8] Sayan Goswami, Kisung Lee, Shayan Shams, and Seung-Jong Park. 2018. Gpu-accelerated large-scale genome assembly. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 814–824.

[9] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex C Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydın Buluç, et al. 2020. Terabase-scale metagenome coassembly with metahipmer. *Scientific reports* 10, 1 (2020), 1–11.

[10] Curtis Huttenhower, Dirk Gevers, Rob Knight, Sahar Abubucker, Jonathan H Badger, Asif T Chinwalla, Heather H Creasy, Ashlee M Earl, Michael G FitzGerald, Robert S Fulton, et al. 2012. Structure, function and diversity of the healthy human microbiome. *nature* 486, 7402 (2012), 207.

[11] Ashutosh Jain, Anshuj Garg, and Kolin Paul. 2013. GAGM: Genome assembly on GPU using mate pairs. In *20th Annual International Conference on High Performance Computing*. IEEE, 176–185.

[12] JGI. 2021. *Marine microbial communities from Western Arctic Ocean.* https://gold.jgi.doe.gov/biosamples?id=Gb0192059

[13] Dinghua Li, Ruibang Luo, Chi-Man Liu, Chi-Ming Leung, Hing-Fung Ting, Kunihiko Sadakane, Hiroshi Yamashita, and Tak-Wah Lam. 2016. MEGAHIT v1. 0: a fast and scalable metagenome assembler driven by advanced methodologies and community practices. *Methods* 102 (2016), 3–11.

[14] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, et al. 2012. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics* 11, 1 (2012), 25–37.

[15] Losee L Ling, Tanja Schneider, Aaron J Peoples, Amy L Spoering, Ina Engels, Brian P Conlon, Anna Mueller, Till F Schäberle, Dallas E Hughes, Slava Epstein, et al. 2015. A new antibiotic kills pathogens without detectable resistance. *Nature* 517, 7535 (2015), 455–459.

[16] Yongchao Liu and Bertil Schmidt. 2013. CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. *IEEE Design & Test* 31, 1 (2013), 31–39.

[17] Mian Lu, Qiong Luo, Bingqiang Wang, Junkai Wu, and Jiuxin Zhao. 2013. Gpu-accelerated bidirected de bruijn graph construction for genome assembly. In *Asia-Pacific Web Conference*. Springer, 51–62.

[18] Chengwei Luo, Luis M Rodriguez-R, Eric R Johnston, Liyou Wu, Lei Cheng, Kai Xue, Qichao Tu, Ye Deng, Zhili He, Jason Zhou Shi, et al. 2014. Soil microbial community responses to a decade of warming as revealed by comparative metagenomics. *Applied and environmental microbiology* 80, 5 (2014), 1777–1786.

[19] Syed Faraz Mahmood and Huzefa Rangwala. 2011. Gpu-euler: Sequence assembly using gpgpu. In *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, 153–160.

[20] NERSC. [n.d.]. *Cori GPU node configurations.* https://docs-dev.nersc.gov/cgpu/hardware/

[21] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel A Pevzner. 2017. metaSPAdes: a new versatile metagenomic assembler. *Genome research* 27, 5 (2017), 824–834.

[22] OLCF. [n.d.]. *Summit node configurations.* https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[23] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. 2012. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences* 109, 33 (2012), 13272–13277.

[24] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. 2012. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics* 28, 11 (2012), 1420–1428.

[25] Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, and Paola Bonizzoni. 2019. Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era. *Quantitative Biology* 7, 4 (2019), 278–292.

[26] Edans Flavius de O Sandes and Alba Cristina MA de Melo. 2011. Smith-waterman alignment of huge sequences with gpu in linear space. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1199–1211.

[27] Thomas J Sharpton. 2014. An introduction to the analysis of shotgun metagenomic data. *Frontiers in plant science* 5 (2014), 209.

[28] Aleksandra Swiercz, Wojciech Frohmberg, Michal Kierzynka, Pawel Wojciechowski, Piotr Zurkowski, Jan Badura, Artur Laskowski, Marta Kasprzak, and Jacek Blazewicz. 2018. GRASShopPER—An algorithm for de novo assembly based on GPU alignments. *PloS one* 13, 8 (2018), e0202355.

[29] Silvana R Tridico, Dáithí C Murray, Jayne Addison, Kenneth P Kirkbride, and Michael Bunce. 2014. Metagenomic analyses of bacteria on human hairs: a qualitative assessment for applications in forensic science. *Investigative genetics* 5, 1 (2014), 1–13.

[30] Thomas R Turner, Karunakaran Ramakrishnan, John Walshaw, Darren Heavens, Mark Alston, David Swarbreck, Anne Osbourn, Alastair Grant, and Philip S Poole. 2013. Comparative metatranscriptomics reveals kingdom level changes in the rhizosphere microbiome of plants. *The ISME journal* 7, 12 (2013), 2248–2258.

[31] Katherine Yelick, Aydın Buluç, Muaaz Awan, Ariful Azad, Benjamin Brock, Rob Egan, Saliya Ekanayake, Marquita Ellis, Evangelos Georganas, Giulia Guidi, et al. 2020. The parallelism motifs of genomic data analysis. *Philosophical Transactions of the Royal Society A* 378, 2166 (2020), 20190394.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran the the GPU local assembly module that has been developed as a result of this study as a stand alone software on Cori's GPU cluster. We used only a single GPU on one of the nodes to perform roofline analysis. For building the CUDA kernels we used CUDA 10.1.2. For this study we used ArcticSynth dataset that was developed for a controlled study performed in this paper: Hofmeyr, Steven, et al. "Terabase-scale metagenome coassembly with metahipmer." Scientific reports 10.1 (2020): 1-11.

Availability of all the datasets is mentioned at the end.

For large scale studies we used OLCF's Summit supercomputer. We obtained the recent most version of MetaHipMer2 software that is available on the master branch here: https://bitbucket.org/berkeleylab/mhm2/src/master/ and used it as the baseline. We then integrated our newly developed GPU kernels in the same version of MetaHipMer2 and merged the two versions. The merged version is available on the same link on branch: "GPU", commit id f9c2d0f. This new version with GPU kernels integrated was then used for performing performance analysis studies. UPC++ release 2021.3.0 along with CUDA versions 10.1.2 was used for building both the versions of MetaHipMer2 (with and without GPU local assembly module).

Update 1: we have updated the branch name and commit id above, this was because we have been continuing the development on other parts of the software but the results for the local assembly module discussed in this paper should still be reproducible from this new branch and commit id.

Update 2: to provide the reviewers with the exact commit of the software that we used for CPU and GPU runs, we have created a separate repo (separate from the production repo referenced above) here: https://github.com/mgawan/mhm2_staging . The gpu_locassem branch at this repo contains the exact version we used for testing our GPU local assembly module and the cpu_locassem branch contains the base version of MetaHipMer2 that contains the corresponding CPU local assembly module. Reviewers can run both these versions (available on separate branches) and then compare the timings of local assembly module from the log file to observe the performance improvement. This staging repo is behind a DOI which is also linked below.

datasets availability:

For large scale performance studies the dataset used was a metagenomic dataset from marine communities of western arctic that is available here: https://gold.jgi.doe.gov/biosamples?id=Gb0192059

The ArcticSynth dataset is not available publicly, we are making it available through Globus here: https://app.globus.org/file-manager?origin_id=e4794a66-c4c1-11eb-87e2-559da91cd9a3&origin_path=%2F

Deploying MetaHipMer2:

Instructions on how to build and deploy MetaHipMer2 on different supercomputers and linux servers can be found here: https://bitbucket.org/berkeleylab/mhm2/src/ae57a4ce84d33ca0b723a53e63675e47d1ccc331/docs/mhm_guide.md

After building, the experiments can be run using below lines in a job script. Below instructions are for running MetaHipMer2 on Summit as was done for experiments in this paper. If reviewers want to run it on a different machine please look the mhm_guide available on the link provided above.

export MHM2_BUILD_ENV=<mhm2_directory>/mhm2/contrib/environments/s
source <mhm2_directory>/mhm2/contrib/environments/summit/gnu.sh

cd <mhm2_directory>/mhm2/install/bin

./mhm2.py -r –checkpoint=off -o <output_directory> –pin=none –ranks-per-gpu=7

where mhm2_directory is where the MetaHipMer2 repo is located.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪  https://zenodo.org/badge/latestdoi/393159384
Artifact name: MetaHipMer2 with GPU accelerated local
↪  assembly module
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: OLCF's Summit supercomputer (https://docs.olcf.ornl.gov/systems/summit_user_guide.html), NERSC's Cori System's GPU partition (https://docs-dev.nersc.gov/cgpu/), GPUs used in the both machines were NVIDIA's V100 devices.

Compilers and versions: CUDA 10.1.2, UPC++ 2021.3.0, gcc 7.4

Applications and versions: MetaHipMer2

Libraries and versions: UPC++ 2021.3.0

Key algorithms: de bruijn graph construction and traversal