Efficient High-Performance Computing Strategies for the Legendre Pairs Search

Amirhossein Sojoodi[1,2*], Ilias S. Kotsireas[3] and Ahmad Afsahi[1*]

[1]Department of Electrical and Computer Engineering, Queen's University, Ontario, Canada.
[2]Distributive Corp., Kingston, Ontario, Canada.
[3]Department of Physics and Computer Science, Wilfrid Laurier University, Ontario, Canada.

*Corresponding author(s). E-mail(s): amir.sojoodi@queensu.ca; ahmad.afsahi@queensu.ca;
Contributing authors: i.kotsireas@wlu.ca;

Abstract

In this paper, we present a comprehensive study on the parallelization and optimization of Legendre Pairs (LPs) construction using High-Performance Computing (HPC) techniques. We address the computational challenges associated with LPs, particularly for large lengths, and propose a highly effective multi-GPU solution. Throughout this journey, we present our findings, insights, and contributions including the parallelized implementation of LP search using OpenMP, MPI, and CUDA, along with systematic optimization strategies for both CPU and GPU. We demonstrate significant performance improvements, achieving over 6500x and 1900x speedup on NVIDIA H100 and A30 GPUs, respectively, compared to the serial execution on Intel Xeon Gold 6338 CPU.

Keywords: High-Performance Computing, MPI, OpenMP, CUDA, Legendre Pairs

1 Introduction

High-Performance Computing (HPC) is an essential tool in modern scientific computing, enabling researchers to tackle complex problems that are computationally intensive and require significant processing power. The ability to perform large-scale simulations, data analysis, and algorithmic computations has revolutionized various

fields, including physics, chemistry, biology, and mathematics. In particular, HPC has become a critical enabler for exploring mathematical conjectures and constructing complex mathematical objects that would be infeasible to handle with traditional computing resources.

Legendre Pairs (LPs) represent fundamental combinatorial entities, which are introduced in [1]. They play a crucial role in the systematic construction of Hadamard matrices, and their properties have been studied by recent research work [2–5]. As stated in previous studies, the existence of LPs for every odd order is conjectured, a proposition that, if proven true, would imply the validity of the Hadamard conjecture. This conjecture has important theoretical and practical implications, with applications from image analysis, signal processing, to cryptography.

The construction of LPs generally follows two primary methodologies: for lengths that are odd prime numbers, they can be systematically constructed via the Legendre Symbol, with precise definitions [6]. For composite lengths, a more generalized approach involves the theory of compression of complementary sequences, as detailed in [7].

The compression methodology for LPs operates on the principle that for any divisor d of an order n, an $LP(n)$ can be derived from "seed" sequences of order n/d, and termed candidate pairs of compressed sequences. The generation of these candidate compressed sequences is generally a more tractable computational task than directly computing an $LP(n)$. However, the subsequent "decompression" process, which transforms each candidate pair into a full $LP(n)$, is computationally intensive. Despite this significant computational burden, the benefits of the compression/decompression method distinctly outweigh those of alternative direct construction methods. This inherent computational complexity associated with the decompression phase establishes a critical bottleneck, underscoring the necessity for advanced computational strategies to make the discovery of LPs for larger lengths feasible.

The construction of LPs, particularly for composite lengths, often presents an intractable computational challenge when approached via brute-force search or direct methods, primarily due to the immense size of the search space. The utility of the compression method is well-illustrated by its application in solving previously open problems. For instance, $LP(77)$ was successfully constructed in [5] using simultaneous decompression with divisors 7 and 11, notably resolving the smallest open order for LPs for over two decades. Similarly, the construction of $LP(87)$ in [4] involved decompression with divisor 3, necessitating the parallel decompression of several thousands of candidate 3-compressions of length 29. These examples highlight the practicality of the compression method, even with its high computational demands.

The extreme need for HPC becomes strongly evident when considering the remaining open problems in LPs. The list of odd integers less than 200 for which the existence of LPs remains unconfirmed, as stated in [4], includes several values, half of which are multiples of 5, indicating that the conjectures and methodologies discussed in [4] are directly applicable to a substantial subset of these outstanding challenges. The case of $LP(117)$ exemplifies the extreme scale of the problem: its search space encompasses approximately 3.1×10^{18} independent test cases. Using sampling and extrapolation, we estimate that a brute-force search for $LP(117)$ on a single CPU core requires an

impractical 1.2×10^{10} years. This staggering computational requirement transforms the problem from merely slow to fundamentally infeasible within a practical timeframe without the application of advanced computational techniques. Therefore, the inquiry of these mathematical goals relies heavily on the development and application of efficient HPC strategies. It is worth mentioning that the given estimation applies to the the generic form of $LP(117)$ algorithm. For instance, there is one pair found in the previous study [3]. However, in this paper, we are looking for a particular $LP(117)$ with a specific property, which will be explained later.

The computational bottleneck highlighted by problems with massive yet parallelizable search spaces necessitates the utilization of multi-core CPUs and Graphics Processing Units (GPUs). Although CPUs have traditionally been the primary target for HPC application distribution, the advent of GPUs created new opportunities for accelerating parallel workloads. These devices and their architecture is very well suited for workloads with high data-level parallelism, including our specific problem, the LP algorithm. The programming model for GPUs, Compute Unified Device Architecture (CUDA) [8], allows for the execution of thousands of threads in parallel, making it the ideal paradigm for tackling the LP problem. However, the CUDA programming model should be paired with a deep understanding of the GPU architecture to achieve optimal performance. This includes managing memory hierarchies, minimizing branch divergence, and minimizing resource contention, all of which are critical for maximizing throughput and efficiency in GPU-accelerated applications.

This study addresses these challenges by developing and evaluating a series of HPC strategies for the construction of LPs. The specific contributions of this work are outlined as follows:

- **Comprehensive CPU Parallelization:** We develop and evaluate both shared-memory, Open Multi-Processing (OpenMP) [9], and distributed-memory, Message Passing Interface (MPI) [10], parallelization strategies for the LP problem. Our analysis demonstrates significant speedups and scalability on multi-core and multi-node architectures.
- **Advanced GPU Acceleration:** We design and develop a highly optimized CUDA-based solution, in a multi-layered approach to overcome common GPU performance bottlenecks. This includes techniques such as removing branch divergence, optimizing memory access patterns (constant and shared memory), applying loop transformations, tuning kernel parameters, utilizing CUDA Graphs, and minimizing bank conflicts and register pressure, resulting in substantial overall speedups.
- **Improve Resource Utilization:** We explore and use various GPU optimization strategies to improve resource utilization, including exploiting NVIDIA's Multi-Process Service (MPS) and Multi-Instance GPU (MIG) technologies. With these modern GPU features, our approach achieves a high degree of throughput, and more efficient resource management.
- **Hybrid MPI-CUDA Scalability:** We integrate MPI and CUDA to enable a multi-GPU, multi-node solution. Our weak scaling evaluation demonstrates the effectiveness of this hybrid approach in distribution, showcasing near-optimal scalability with minimal inter-GPU communication overhead.

- **In-depth Performance Analysis and Optimization Insights:** We provide a detailed empirical analysis of each parallelization and optimization step, offering quantitative data and practical insights into the performance characteristics of the LP algorithm across diverse HPC platforms. This includes profiling results and an analysis of GPU occupancy.
- **Enabling New Mathematical Discovery:** The methodologies and optimization techniques developed in this paper significantly reduce the computational time required for LP, thereby paving the way for their discovery and contributing directly to the ongoing computational search for these challenging mathematical objects.

The rest of the paper is organized as follows: Section 2 provides the necessary background and reviews the related work. Section 3 discusses the OpenMP and MPI designs and implementations, while Section 4 presents the GPU implementation and optimization strategies. We have covered the performance evaluation and analysis of each approach in their respective sections. Finally, Section 6 concludes the paper and outlines future work.

2 Background and Related Work

2.1 Legendre Pairs: Definition and Properties

LPs represent a specific type of combinatorial sequence, distinct from Legendre's Conjecture related to prime numbers or Legendre polynomials. These pairs were introduced in [1] as a means to construct Hadamard matrices systematically. Some of their properties were recently studied in [2–5]. It is well-known that LPs of order an odd prime number can be constructed via the Legendre Symbol. For the precise definition and properties of the Legendre Symbol see [6]. To construct LPs of order a composite number, one can use the general theory of compression of complementary sequences [7]. The general idea of compression, as it applies to the case of LPs, is that for every divisor d of the order n of the LP, one can construct $LP(n)$ starting with some pair of *seed* sequences of order n/d, that we call candidate pairs of compressed sequences. The point is that the construction of candidate compressed sequences is generally a much easier task than computing $LP(n)$. Subsequently, each candidate pair of compressed sequences has to be decompressed, into a $LP(n)$.

The decompression process is computationally expensive, but its benefits clearly outweigh the alternative direct method of construction of LPs. An illustration of the usefulness of the compression method for LPs can be found in previous work [4, 5]. In [5], the authors construct $LP(77)$ using simultaneous decompression with the divisors $d = 7$ and $d = 11$. Note that $LP(77)$ was the smallest open order for LPs for more than 20 years. In [4], the authors construct $LP(87)$ using decompression with the divisor $d = 3$. The construction of $LP(87)$ required to construct several thousands of candidate 3-compressions of length 29 and subsequently decompress each one of them in parallel.

Given the quite fertile application of compression/decompression in the construction of LPs, it is reasonable to try to speed up the currently available C code [4], using

modern technologies available in HPC. The currently available C code for decompression is automatically generated using meta-programming in Maple, which guarantees a correct and bug-free solution.

In order to define Legendre Pairs, we first mention the well-known definitions of Periodic Autocorrelation Function (PAF) and Power Spectral Density (PSD) [11].

Definition 1 The PAF of a finite sequence of length n, $A = [a_0, \ldots, a_{n-1}]$ is defined as:

$$PAF(A, s) = \sum_{k=0}^{n-1} a_k a_{k+s}, \ s = 0, \ldots, n-1$$

where ($k + s$ is taken mod n).

A well-known PAF symmetry property is that

$$PAF(A, s) = PAF(A, n-s), s = 1, \ldots, n-1$$

which essentially tells us that we need to know $\frac{n}{2}$ or $\frac{n-1}{2}$ PAF values, according to whether n is even or odd.

Definition 2 The PSD of a finite sequence of length n, $A = [a_0, \ldots, a_{n-1}]$ is defined as:

$$PSD(A, k) = |\mu_k|^2 = \Re(\mu_k)^2 + \Im(\mu_k)^2$$

where the μ_k are the elements of the Discrete Fourier Transform (DFT) sequence associated to $[a_1, \ldots, a_n]$, i.e.

$$DFT_{[a_0,\ldots,a_{n-1}]} = [\,\mu_0, \ldots, \mu_{n-1}\,], \text{ with } \mu_k = \sum_{i=0}^{n-1} a_i\, \omega^{ik}, \ k = 0, \ldots, n-1$$

with $\omega = e^{\frac{2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right)$ a primitive n-th root of unity.
Since the PSD values are sums of two squares, they are always nonnegative. In contrast, the PAF values may be positive or negative.

Definition 3 A Legendre Pair of (odd) order $n > 1$, denoted by $LP(n)$, consists of two sequences $A = [a_0, \ldots, a_{n-1}]$ and $B = [b_0, \ldots, b_{n-1}]$, both of length n, with $\{-1, +1\}$ elements, that satisfy:

$$PAF(A, s) + PAF(B, s) = -2, \forall s = 1, \ldots, \frac{n-1}{2}.$$

For an $LP(n)$, the normalizations $a_0 + \cdots + a_{n-1} = 1$ and $b_0 + \cdots + b_{n-1} = 1$ are typically imposed, which leads to the upper bound on the number of potential A,B sequences: $\binom{n}{\frac{n+1}{2}}$. This is because in each of the two sequences A, B, we must have $\left(\frac{n+1}{2}\right)$ $+1$ elements and $\left(\frac{n-1}{2}\right)$ -1 elements. Another important consequence of the definition of $LP(n)$ is the PSD constancy property:

$$PSD(A, s) + PSD(B, s) = 2n + 2, s = 1, \ldots, \frac{n-1}{2}.$$

In summary, LPs are characterized by the constancy of their PAF and PSD values.

As mentioned earlier, we are interested in constructing an $LP(117)$ whose "9-compression" is given by the formula in Listing 1. For the specific details of `LegendreSymbol` and the mathematical background behind this formula, please refer to [6].

```
> with(NumberTheory):
> p:=13; q:=3;
> L:=q*[seq(LegendreSymbol(i,p),i=1..p-1)];
L:=[3, -3, 3, 3, -3, -3, -3, -3, 3, 3, -3, 3];

> a:=[1,op(L)];
> b:=[1,-op(L)];
a := [1, 3, -3, 3, 3, -3, -3, -3, -3, 3, 3, -3, 3]
b := [1, -3, 3, -3, -3, 3, 3, 3, 3, -3, -3, 3, -3]
```

Listing 1: A Maple snippet to show the 9-compression formula

2.2 Parallel Processing in Legendre Pairs Search Problem

To the best of our knowledge, there is no previous work that specifically addresses the parallelization of the LP search problem using HPC techniques. The existing literature primarily focuses on the mathematical properties and theoretical aspects of LPs, with limited attention to computational methods for their construction. The work by [4] discusses the use of compression techniques for constructing LPs, but does not provide parallelization or optimization strategies.

However, beyond LPs, the application of parallel processing to similar exhaustive search problems in combinatorial design, Boolean Satisfiability, and cryptographic key searches demonstrates the immense power of these techniques [12–15]. Furthermore, maximizing performance in complex parallel and hybrid environments requires careful optimization, including data locality management, memory footprint reduction, and, in some cases, the development of hardware-aware methods [16, 17].

2.3 Parallel Programming Models, OpenMP, MPI, and CUDA

OpenMP is a widely adopted, directive-based Application Programming Interface (API) designed for parallelization on shared-memory architectures [9]. It utilizes a fork-join model in which a master thread creates a group of threads to execute some regions of the code in parallel. The ease of use of OpenMP has made it a popular choice for parallelizing applications, particularly those involving loop-centric tasks where data can be shared efficiently among the threads.

On the other hand, MPI provides the API for parallel programming on distributed memory systems, enabling communication between independent processes. It supports point-to-point and collective operations, with well-known implementations like MPICH [18], MVAPICH [19], and Open MPI [20].

GPUs, initially created only for graphics purposes, have evolved into high throughput hardware chips for extreme computing power in HPC and machine learning

applications. Their architecture, including thousands of cores, are very well suited for data-parallel tasks such as scientific simulations, machine learning, and data analysis. Over the past decade, this capability have allowed numerous studies to demonstrate significant speedups for highly parallelizable workloads, when using GPUs compared to the CPUs [8, 21–23].

Leveraging GPUs in HPC applications, often involves hybrid MPI+X models, such as MPI+CUDA and MPI+NVIDIA Collective Communications Library (NCCL) [8, 24]. These models combine the strengths of MPI for distributed memory parallelism with CUDA for intra-node parallelism, enabling efficient scaling across multiple nodes with multiple GPUs per node. This hybrid approach is particularly effective for large-scale problems, allowing for fine-grained parallelism within nodes while maintaining the ability to scale across the cluster [16, 17, 25]

2.3.1 GPU-Specific Optimization Strategies

Despite their computational power, GPUs optimization are very challenging due to their distinct architecture compared to CPUs. Key bottlenecks often stem from memory latency, resource contention, and the management of parallel execution flow [26]. Effective GPU utilization demands close management of the memory hierarchy, which includes global memory, shared memory, constant memory, and registers [27–29].

Minimizing costly data transfers between the host (CPU) and device (GPU) is important to the extend that is possible, and overlapping computation with communication for the cases where data transfer is unavoidable [30–32]. Code that transfers data for brief use by a small number of threads will see little or no performance benefit, emphasizing the need for substantial work per data element transferred [33]. Various loop transformations, such as loop unrolling, can also significantly enhance performance by reducing the overhead of loop control and increasing instruction-level parallelism [34].

Further challenges include thread divergence among warps, where conditional branches can result in serialized execution, or the careful resource utilization (e.g., registers and shared memory) in order to maximize occupancy [35]. Optimizing GPU kernels is a complex task requiring detailed understanding of architecture, extensive profiling, and iterative experimentation [36]. This inherent complexity is a consequence of the GPU's hierarchical memory organization, and their Single Instruction Multiple Data (SIMD) execution model [37]. More interestingly, some recent studies have shown that manual utilization of GPU Streaming Assembler (SASS) or Parallel Thread Execution (PTX) can lead to better performance, underscores a persistent gap between high-level programming abstractions and low-level hardware efficiency [38–40]. Consequently, this intrinsic complexity of GPU architecture, has driven the continuous development of advanced profiling tools and, more recently, the emergence of AI-driven automated optimization frameworks designed to abstract away or automate portions of this low-level complexity [41].

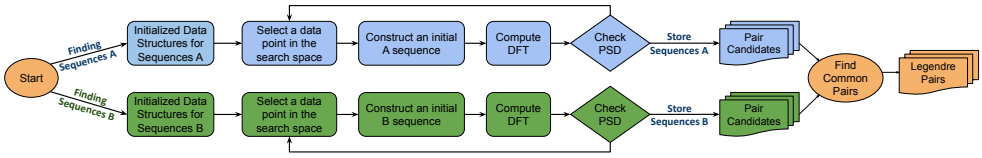

Fig. 1: General Legendre Pairs (LPs) Algorithm Design and Generation Process: generating LPs through the iterative construction of sequences A and B and finding the common members.

3 CPU-Based Approaches

3.1 Experimental Framework

As we provide various parallelization strategies and optimizations for the LP algorithms, we provide the evaluation and discussion of each approach in their respective sections. This helps to better understand the performance characteristics of each approach. All the experiments were conducted on the Daisy cluster at Queen's University. Each node in the cluster has two-socket and is equipped with two Intel Xeon Gold 6338 CPUs, each having 32 cores and supporting two threads per core, resulting in a total of 64 physical cores and 128 logical threads per node. Moreover, each node is equipped with 256 GB of DDR4 memory, and four NVIDIA A30 GPUs with 24 GB of GDDR6 memory each. The GPUs are connected to the CPUs via Peripheral Component Interconnect Express (PCIe) Gen4 links. For the implementations, we utilized OpenMP v4.5, Open MPI v5.0.1, and CUDA v12.0.

3.2 Baseline: Single-Core Implementation

The main approach to generate the LPs is to find common members of the two groups of sequences, A and B. Figure 1 illustrates the general algorithm design and generation process. The algorithm iteratively constructs sequences A and B, checking for common members to identify the LPs. The sequences are generated based on their specific initial parameters. This brute-force approach explores the parameter space, and for any candidate, its DFT is calculated and checked for PSD validation. If the PSD is valid, the candidate is saved as a potential LP. The algorithm continues until all possible combinations of the parameters are explored. It is important to note that we cannot predict where in the parameter space the LPs will be found, so we must explore all possible combinations before we can find the common members.

Algorithms 1, 2, and 3 show the process of searching for sequences A in more detail. These algorithms are based on the ones described in [4] with minor modifications to improve data structure and locality. In Lines 2-5, the Algorithm 1 takes the input parameters to construct the initial data structures. For sequences B, the initial parameters are different (the code is available on its GitHub repository[1]). As shown in Lines 5-9, the algorithm constructs the initial sequence using a series of nested loops, iterating through all possible combinations of the parameters. For length 45, there are

[1] https://github.com/amirsojoodi/LeGendre-Pairs-Algorithm

Algorithm 1: Legendre Pairs (length 45): Sweeping the search space and looking for valid sequences A.

```
1  Function GenerateSequence():
2      PairLength ← 45 , PSDCount ← 23
3      Precompute ω^k for k ∈ [0,PairLength)
4      Op1 ← 1 , Op3 ← {2,5} , Om3 ← {3,4}
5      for xp1 ← 1 to 126 do
6          for xp3[0] ← 1 to 84 do
7              for xp3[1] ← 1 to 84 do
8                  for xm3[0] ← 1 to 84 do
9                      for xm3[1] ← 1 to 84 do
10                         A ← ContructA(Op1, Op3, Om3, xp1, xp3, xm3)
11                         dft ← ComputeDFT(A, ω)
12                         if CheckPSD(dft, ω_powers) then
13                             Save valid sequence
14                         end
15                     end
16                 end
17             end
18         end
19     end
20 end
```

about 6.27×10^9 independent test-cases for either sequences, which is computationally very expensive.

Moreover, as depicted in Algorithm 2, the algorithm constructs the sequences based on the input parameters through a series of switch cases (Lines 3-19). Then, based on the properties and formulation described in 2.1, the algorithm computes the DFT of the sequence using the `ComputeDFT` function in Algorithm 3 (Lines 1-12). Then it checks if the computed DFT is valid by checking whether it satisfies the PSD conditions, in the `ComputePSDs` function (Lines 13-25). One improvement to the function `ComputePSDs` is to check for PSD_{15} first, as most of the sequences will fail this condition (Lines 17-20 of Algorithm 3). It is out of the scope of this study to provide the reason behind this phenomenon, and more details can be found in [4]. Once again, the conditions are different for sequences A and B.

3.2.1 Performance Analysis of the Baseline Implementation

While the LP-45 algorithm is computationally expensive, it is still feasible to run on a single-core CPU. However, the LP-117 algorithm is significantly more complex and requires a more efficient approach. The brute-force search for LP-117 on a single core is not practical due to the sheer number of combinations, which is in the order of 10^{18} different test cases. Table 1 provides a comparison of the two algorithms, highlighting the differences in their search space sizes, number of switch cases, and execution times, measured on a single core of the Intel Xeon Gold 6338 CPU. LP-117 algorithm is not provided here, but it follows a similar structure to the LP-45 algorithm, with different

Algorithm 2: Legendre Pairs (length 45): Sequence A construction.

```
1  Input: Op1[ ], Op3[ ], Om3[ ], xp1[ ], xp3[ ], xm3[ ]
2  Output: A[ ]
3  Function ContructA():
4      Initialize array A[PairLength + 1]
5      foreach i ∈ {1, 2} do // Update A based on xp3_i
6          switch xp3_i do
7              case 1 do
8                  A[Op3s[i]] = −1
9                  A[Op3s[i] + 5] = −1
10                 A[Op3s[i] + 10] = −1
11                 A[Op3s[i] + 15] = 1
12                 A[Op3s[i] + 20] = 1
13                 A[Op3s[i] + 25] = 1
14                 A[Op3s[i] + 30] = 1
15                 A[Op3s[i] + 35] = 1
16                 A[Op3s[i] + 40] = 1
17             end
               // ... 83 more cases
18         end
19     end
20     foreach i ∈ {1} do // Update A based on xp1_i
21         switch xp1_i do
               // 126 different cases
22         end
23     end
24     foreach i ∈ {1, 2} do // Update A based on xm3_i
25         switch xm3_i do
               // 84 different cases
26         end
27     end
28     return A
29 end
```

initial parameters, conditions, DFT/PSD computations, and more importantly, the size of the address space. The source code for the LP-117 algorithm is available at its GitHub repository [2].

We profiled the execution time of the LP-45 using gprof and perf tools, and the results suggest that the algorithm is mainly compute-bound. While all the test cases are independent, the preparation of the initial sequences and their DFT computations are the most time-consuming parts of the algorithm (Lines 10-11, of Algorithm 1). For LP-45, very few test cases satisfy all the PSD condition to be a valid sequence. In fact, out of the 6.27×10^9 test cases, only 246651 and 129651 unique sequences are valid for sequences A and B, respectively. We expect that the LP-117 algorithm will have a similar behavior, but with a larger search space and more complex conditions.

[2] https://github.com/amirsojoodi/LeGendre-Pairs-Algorithm

Algorithm 3: Legendre Pairs (length 45): Compute DFT and filter based on PSD thresholds.

```
Function ComputeDFT(A, ω_powers, step, t):
    dft ← A[0] · ω_powers[0]  // Initialize DFT sum
    k ← t
    for j ← 1 to PairLength/step do
        dft += A[j] · ω_powers[k]
        k += t
        if k ≥ PairLength then
            k −= PairLength
        end
    end
    return dft
end
Function ComputePSDs(A, ω_powers):
    for t ← 1 to PSDCount − 1 do
        dft ← ComputeDFT(A, ω_powers, 1, t)
        PSD_t ← Re(dft)^2 + Im(dft)^2
        if t = 15 then
            if PSD_t ≠ 28 then
                return False
            end
        else if PSD_t > 92 then
            return False
        end
    end
    return True
end
```

Table 1: Comparison of LP-45 and LP-117 algorithms.

Parameter	**LP-45**	**LP-117**
Sequence length	45	117
Size of the search space	6.27×10^9	3.1×10^{18}
Switch cases per test-case	462	2142
PSDs to compute per test-case	23	59
Execution time (on a CPU core)	$\approx$ 38 minutes	$\approx 1.2 \times 10^{10}$ years
Lines of code in the algorithm	$+7K$	$+13K$
Unique valid sequences A	246651	Unknown
Unique valid sequences B	129651	Unknown
Unique pairs found	192	Unknown

As can be seen in Table 1, the LP-45 algorithm is computationally feasible, taking about 38.8 minutes to run on a single core. However, as mentioned earlier, the brute-force search algorithm for LP-117 is not practical, as it would take approximately 12

billion years to complete on a single core. This highlights the need for parallelization and optimization strategies to make the LP-117 algorithm more efficient. The LP-45 algorithm is a good starting point for understanding the structure and performance characteristics of the LPs algorithms, and as we progress, we will provide various parallelization strategies and optimizations to improve the performance of the both lengths.

For evaluations, we will provide the results for both lengths when they exhibit different behaviors, otherwise we will only provide the results for one of them. We also provide the performance evaluations of both sequences separately, as they have different initial parameters and conditions, which can lead to different performance characteristics. The performance results for the entire application (both sequences) can be estimated by adding the execution times of the individual sequences when they are executed sequentially. Furthermore, to evaluate the optimizations of the LP-117 algorithm, we select a subset of the address space, which allows us to approximate the performance of the algorithms and optimizations without running them on the entire address space, which is unfeasible.

3.3 Utilizing OpenMP

The first step of the parallelization process is to identify the independent test cases, which in this case are every iteration of the inner-most loop of the algorithm. This means that each iteration can be executed independently without any data dependencies. However, to keep the number of forks/joins between the threads at the minimum, the strategy is to parallelize at the outermost loop level, as depicted in Algorithm 4. Line 5 illustrates the scheduling policy, which is set to `dynamic` to allow for dynamic load balancing among the threads. On Line 16, the `critical` section is used to ensure that only one thread at a time can save the valid sequence. As the number of valid sequences is very small compared to the total number of test cases, this critical section will not be a bottleneck. However, we optimized this section to temporarily store the valid sequences in a local array, which is then merged into the global array at the end of the computation. However, this optimization is not shown in the algorithm for simplicity, and the performance impact is negligible.

To evaluate the performance of the OpenMP implementation, we executed the algorithm on a single node of the Daisy cluster. The results are shown in Figure 2 and Figure 3. Speedup computation consists of dividing the execution time of the single-core version by that of the multi-core version. The efficiency metric or *parallel efficiency* of the application is the ratio of the speedup to the number of threads, mathematically represented as $E = \frac{S}{P}$, where S stands for speedup and P stands for the number of threads. As shown in Figure 2, the speedup increases with the number of threads, but the performance improvement degrades for 128 threads, which is due to the overhead of thread management and memory accesses on different Non-Uniform Memory Access (NUMA) nodes within the same process. The efficiency plot in Figure 3 suggests that the algorithm is highly parallelizable as the efficiency remains above 90% for up to 64 threads.

Results for LP-117 are provided in Figure 4 and Figure 5. The speedup and efficiency figures follow the same trend as the LP-45 algorithm, but the speedup is lower

Algorithm 4: Legendre Pairs (length 45) - Work distribution on CPU cores using OpenMP

```
1  Function GenerateSequence():
2      PairLength ← 45 , PSDCount ← 23
3      Precompute ω^k for k ∈ [0,PairLength)
4      Op1 ← 1 , Op3 ← {2,5} , Om3 ← {3,4}
5      #pragma omp parallel for schedule (dynamic)
6          shared (Op1, Op3, Om3, omega_powers)
7          private (A, dft, xp1, xp3, xm3)
8      for xp1 ← 1 to 126 do
9          for xp3[0] ← 1 to 84 do
10             for xp3[1] ← 1 to 84 do
11                 for xm3[0] ← 1 to 84 do
12                     for xm3[1] ← 1 to 84 do
13                         A ← ConstructA(Op1, Op3, Om3, xp1, xp3, xm3)
14                         dft ← ComputeDFT(A, ω)
15                         if CheckPSD(dft, ω_powers) then
16                             #pragma omp critical
17                             Save valid sequence
18                         end
19                     end
20                 end
21             end
22         end
23     end
24 end
```

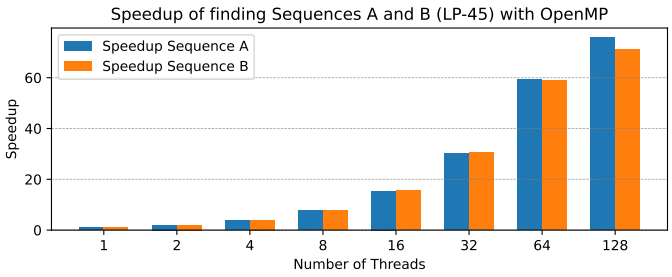

Fig. 2: Speedup of the OpenMP implementation of LP-45 algorithm on a single node of Daisy cluster comparing with the single-core execution.

due to the more complex conditions and larger amount of data to be processed. The speedup is still significant, reaching up to 40x for 64 threads, and the efficiency remains above 80% for up to 32 threads. This shows that the LP-117 is also highly parallelizable and scalable.

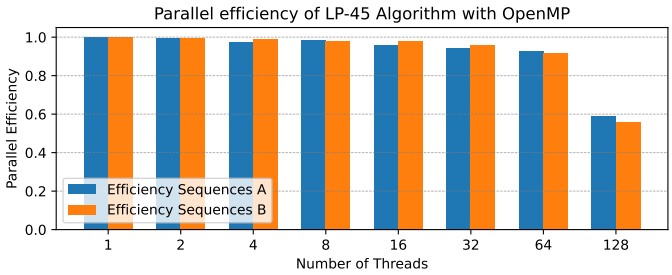

Fig. 3: Efficiency of the OpenMP implementation of LP-45 algorithm, executed on a single node of Daisy cluster.

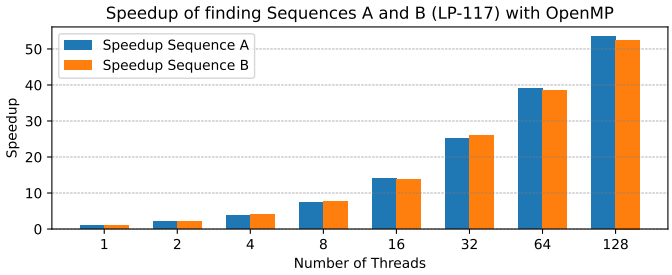

Fig. 4: Speedup of the OpenMP implementation of LP-117 algorithm on a single node of Daisy cluster comparing with the single-core execution.

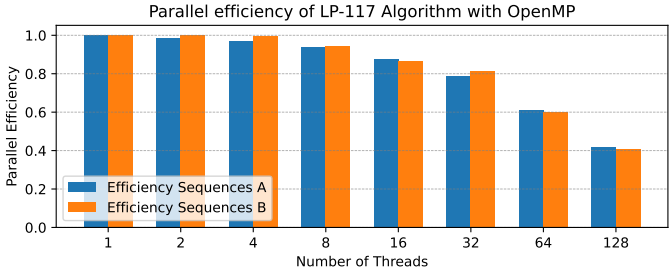

Fig. 5: Parallel efficiency of the OpenMP implementation of LP-117 algorithm, executed on a single node of Daisy cluster.

3.4 Utilizing MPI

Now that we have a single-node parallel implementation of the LP algorithms, we can further scale the implementation to multiple nodes using MPI. The MPI implementation of the LP algorithm is similar to the OpenMP implementation, but it uses a master-worker model, where the leader process (rank 0) collects the results from the

Algorithm 5: Legendre Pairs (length 45) with MPI: Sequence generation and workload distribution

```
1  Function GenerateSequence():
2      MPIInit()  // Initialize MPI
3      if rank = 0 then // Start listener pthread on the leader
4          Launch background thread to ListenForResults()
5      end
6      PairLength ← 45 , PSDCount ← 23
7      Precompute ω^k for k ∈ [0,PairLength)
8      Op1 ← 1 , Op3 ← {2,5} , Om3 ← {3,4}
       // Distribute the workload among the worker processes
9      total_tasks ← 126 * 84
10     for task ← rank to total_tasks by process_count do
11         xp1[0] ← task/84
12         xp3[0] ← task%84
13         for xp3[1] ← 1 to 84 do
14             for xm3[0] ← 1 to 84 do
15                 for xm3[1] ← 1 to 84 do
16                     A ← ConstructA(Op1, Op3, Om3, xp1, xp3, xm3)
17                     dft ← ComputeDFT(A, ω)
18                     if CheckPSD(dft, ω_powers) then
19                         SaveValidSequence(PSDs)
20                     end
21                     if Local buffer is full then
22                         SendResults()  // to the leader process
23                     end
24                 end
25             end
26         end
27     end
28     MPIBarrier()  // Synchronize before shutdown
29     if rank = 0 then
30         Signal listener thread to FlushFinalResults() and exit
31     end
32     MPIFinalize()
33 end
```

worker processes (Algorithm 5). As Algorithm 6 shows, the leader process utilizes a listener thread that communicates with the workers using `MPI_Iprobe` and `MPI_Recv` functions. The worker processes execute the same algorithm as the single-core implementation, and once their local buffers are full of valid sequences, they send the results back to the leader process by the `MPI_Ssend` function to make sure that the leader process has collected all the results successfully and to avoid extensive memory usage in the workers (Line 22 of the Algorithm 5).

Note that the leader process also participates in the computation, and it is not idle while waiting for the results from the worker processes. This is done by interleaved execution between the listener and the main thread to share the CPU resources. That

Algorithm 6: LP Algorithm with MPI, the listener thread from the leader rank: receiving and flushing results from worker processes in a background thread.

```
1  Function ListenForResults(received_results, run_flag):
2      while run_flag ≠ 0 do
3          MPIIprobe(AnySource, AnyTag, probe_flag, status)
4          if probe_flag = 0 then // No awaiting messages
5              Sleep for 1ms
6              continue
7          end
           // A new message is available, getting count from the status
8          MPIGetCount(status, count)
           // Receive results from the worker process
9          MPIRecv(received_results, status.source)
10         FlushToFile()
11         if status.tag = TAG_RESULT then
               // Remaining results flushed from rank status.MPI_Source
12         else if status.tag = TAG_BUFFER_FULL then
               // Buffer is full, from rank status.MPI_Source
13         end
14     end
       // received the stop signal from the leader process
15     FlushRemaining()  // Flush any remaining results to file
16     ExitThread()
17 end
```

is why the listener thread has a sleep time of 1ms (Line 10 of Algorithm 6) to avoid busy-waiting.

In the listener thread, the leader probes for any incoming messages from the worker processes. If a message is received, it gets the message size and receives the results from the worker process (Lines 8-9 of Algorithm 6). The leader process then flushes the results to a file whenever the buffer is full or when the worker processes finish their computations (Line 10 of the Algorithm 6). At the end of the computation, the leader process signals the listener thread to flush any remaining results and exit (Lines 15-16 of Algorithm 6).

To evaluate the performance of the MPI implementation, we executed the algorithm in two different scenarios: (1) on a single node, and (2) four nodes of the Daisy cluster, with increasing number of ranks per node. The results for the single-node execution of LP-117 are depicted in Figure 6 and Figure 7. Similar to the OpenMP implementation, the speedup increases with the number of ranks, but the performance degrades for 128 ranks per node. The efficiency plot in Figure 7 suggests that the algorithm is highly parallelizable as the efficiency remains above 95% for up to 64 ranks.

Figure 8 and Figure 9 illustrate the results for the multi-node version of LP-117 executed on four nodes of the Daisy cluster. Similar to the single-node version, the

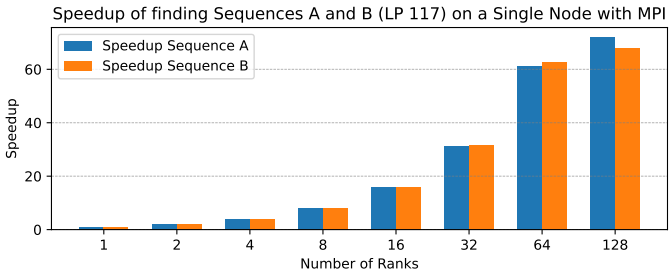

Fig. 6: Speedup of the MPI implementation of LP-117 algorithm, executed on a single node of Daisy cluster, comparing with the single-core execution.

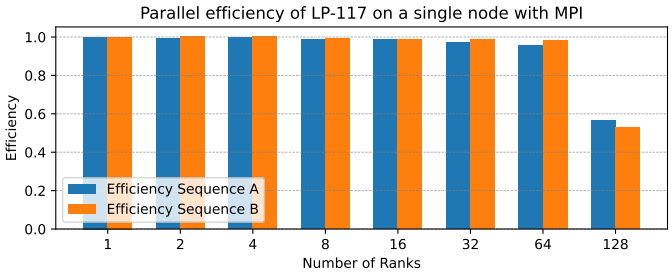

Fig. 7: Parallel efficiency of the MPI implementation of LP-117 algorithm, executed on a single node of Daisy cluster.

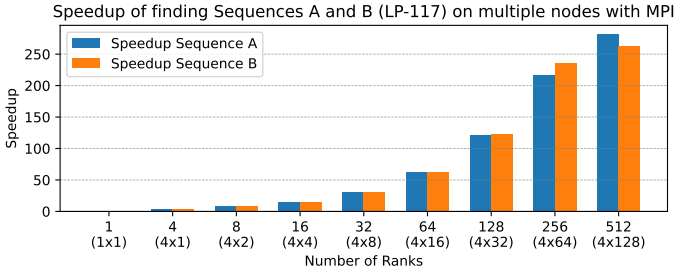

Fig. 8: Speedup of the MPI implementation of LP-117 algorithm, executed on four nodes of Daisy cluster, comparing with the single-core execution.

results suggest that the algorithm parallel efficiency remains above 90% for up to 64 ranks per node, where the speedup reaches up to 230x for 256 ranks.

To analyze the ratio of the communication time to the computation time, we executed both LP-45 and LP-117 algorithms in four scenarios:

- 64 ranks on a single node (Figure 10.a and Figure 10.e)

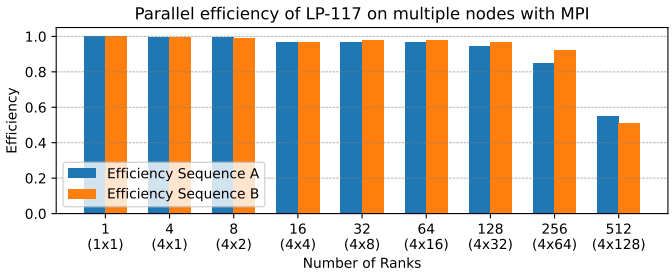

Fig. 9: Parallel efficiency of the MPI implementation of LP-117 algorithm, executed on four nodes of Daisy cluster, comparing with the single-core execution.

- 128 ranks on a single node (Figure 10.b and Figure 10.f)
- 256 ranks on four nodes - 64 ranks per node (Figure 10.c and Figure 10.g)
- 512 ranks on four nodes - 128 ranks per node (Figure 10.d and Figure 10.h)

As shown in Figure 10, communication time is negligible compared to the computation time for both algorithms, especially for LP-117. This is due to the increase of search space and computation time per rank, which makes the communication overhead less significant. For LP-45, the communication time is slightly more noticeable, but it remains below 10% of the total execution time in all scenarios except for the last one. We can also observe that communication overhead becomes slightly noticeable (15% of the execution time) for LP-45 on multiple nodes and 128 ranks per node (Figure 10.d). Comparing to Figure 10.c and Figure 10.b, we can infer that this noticeable overhead is due to contention on the intra-node interconnect between the NUMA nodes, not the inter-node interconnect.

4 GPU Acceleration - Design and Evaluation

As we discussed in Section 3, the LP algorithm is embarrassingly parallel, making it very suitable for GPU acceleration. In this section, we will explore the design and evaluation of a GPU-based implementation of the LP algorithm, focusing on performance optimizations and scalability. Once again, for simplicity, we will only provide a high-level design and approach to the LP-45 version of the algorithm, but the same principles have been applied to the LP-117 version as well, and the results are presented for both versions.

4.1 Baseline GPU Implementation

We will start with a baseline GPU implementation of the LP algorithm, which serves as a foundation for further optimizations. The baseline implementation is designed to be straightforward, focusing on the core functionality of the LP algorithm while leveraging the parallel processing capabilities of GPU architectures. Algorithm 7 provides a high-level overview of the main function and the GPU kernel. The main function initializes the necessary data structures, allocates memory on the GPU, and launches

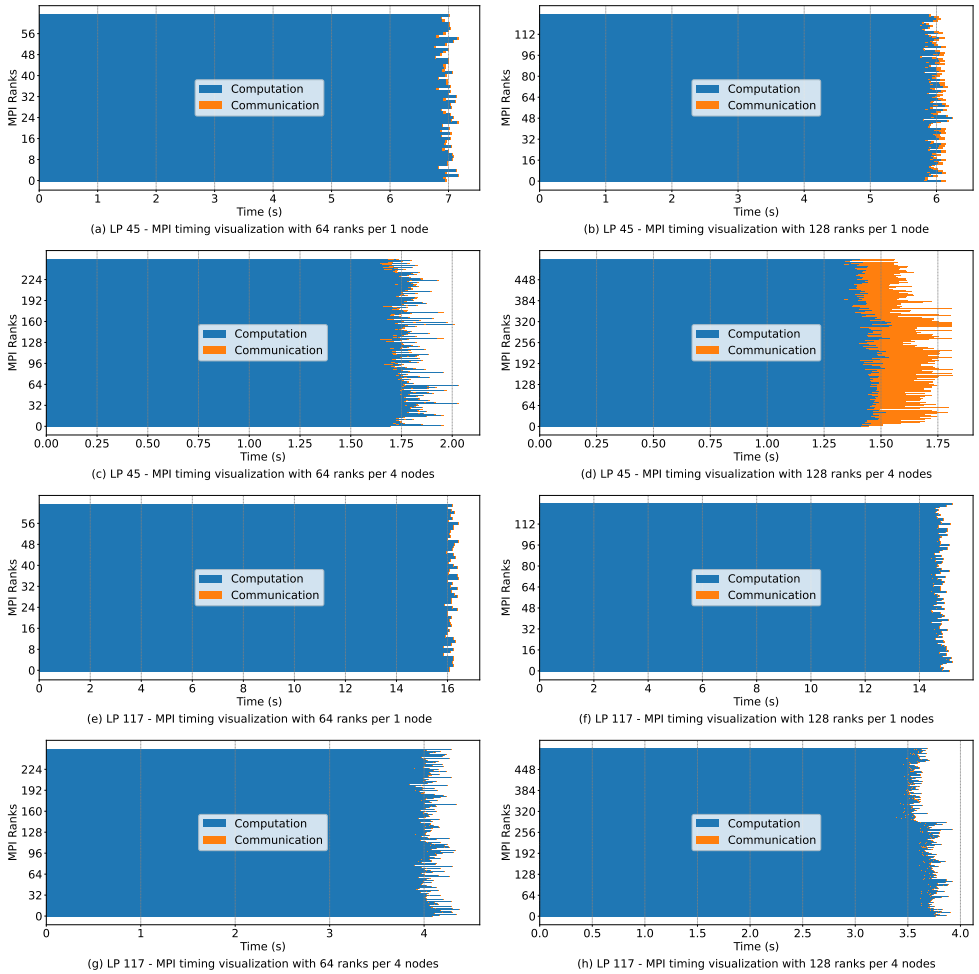

Fig. 10: Communication and computation time for the MPI implementation of LP-45 and LP-117 algorithms, executed on a single node and four nodes of Daisy cluster.

the GPU kernel for the LP-45 version of the algorithm. The functions `ContructA`, `ComputeDFT`, and `CheckPSD` in this kernel are the `__device__` versions of the same functions we discussed in Algorithm 1.

As depicted, in this version, every GPU thread handles a single data point, while the workload is distributed across 126 blocks of 84 threads each ($126 \times 84 = 10584$ threads). Once a thread finds a valid sequence, it atomically increments a counter to ensure thread safety when updating the results array (Line 10 of Algorithm 7). The results are then stored in a global array, which is later copied back to the host memory for further processing (Line 26 of Algorithm 7).

In comparison to the LP-45, the LP-117 version of the algorithm follows a similar design, with adjustments made to accommodate the increased complexity and size of

Algorithm 7: LP-45, CUDA-v1: Naive GPU Kernel Implementation of finding and storing A sequences

```
1  ___global___ legendre_kernel(results, count, omega_powers, Op1s, Op3s, Om3s):
2      xp1[0] ← blockIdx.x  // 126 blocks per grid
3      xp3[0] ← threadIdx.x  // 84 threads per block
4      for xp3[1] ← 1 to 84 do
5          for xm3[0] ← 1 to 84 do
6              for xm3[1] ← 1 to 84 do
7                  A ← ConstructA(Op1s, Op3s, Om3s, xp1, xp3, xm3)
8                  dft ← ComputeDFT(A, ω_powers)
9                  if CheckPSD(dft) then
10                     idx ← atomicAdd (count, 1)
11                     Store sequence at results[idx]
12                 end
13             end
14         end
15     end
16 end
17 Function main():
18     Initialize and precompute data  // Similar to Algorithm 1
19     cudaMalloc() and cudaMemcpy() input data to device
20     cudaMalloc(results, count)  // Allocate device memory to store results
21     cudaMemset(count, 0)  // Initialize count to zero on device
22     grid_size ← 126  // First loop of Algorithm 1 assigned to 126 blocks
23     block_size ← 84  // Second loop is assigned to 84 threads per block
24     LaunchKernel(legendre_kernel, args)
25     cudaDeviceSynchronize()
26     cudaMemcpy(results, count)  // from device to host
27     FlushToFile(results)
28 end
```

the data structures involved. The LP-117 kernel encompasses a larger computational workload, and the launch itself is inside an eight-fold nested loop (the details are skipped for clarity).

5 GPU-based Optimizations

The evaluations on single GPU were conducted on an NVIDIA A30 GPU, and the results are compared with the baseline CPU implementation on a single core of an Intel Xeon Gold 6338 CPU. The baseline GPU version of the LP-45 and LP-117 outperforms their CPU counterparts, achieving a speedup of 21.5× and 18.95×, respectively. The same improvement is observed for both sequences A and B. Analyzing the baseline implementation, we can identify several opportunities for performance improvements, which we will explore in the following sections.

As we discussed in Section 3.2, the LP algorithm is highly parallelizable, but the observed speedup from the GPU version is relatively modest considering the potential

Table 2: Some of the relevant specifications of the NVIDIA A30 GPU

Specification	**Value**
CUDA Version	12.3
CUDA capability	8.0
GPU Max clock rate	1440 MHz (1.44 GHz)
L2 Cache Size	25165824 bytes
Maximum number of threads per block	1024
Memory clock rate	1215 MHz
Memory bus width	3072-bit
Multiprocessors	56
CUDA Cores/MP	64
Total amount of constant memory	65536 bytes
Total amount of global memory	24061 MBytes
Total amount of shared memory per block	49152 bytes
Total number of registers per block	65536
Warp size	32

of the A30 GPU. Although several factors contribute to performance improvements, the GPU specifications in Table 2 suggest that the naive GPU implementation is unable to fully utilize the available CUDA cores. As depicted, the NVIDIA A30 GPU has 56 multiprocessors, each with 64 CUDA cores, resulting in a total of 3584 CUDA cores, and the speedup of 18.95× is not even close to the potential peak performance of the GPU.

5.1 Removing Branch Divergence

One of the primary performance bottlenecks in the baseline GPU implementation is the presence of branch divergence, particularly in the `switch cases` where each thread might take a different execution path based on the value of `Op1s, Op3s, and Om3s` variables.

In the SIMD execution model, threads execute in lockstep, meaning that if some threads take a different path due to a conditional statement, the GPU must serialize the execution of those divergent paths. In GPUs, this issue occurs when threads within a warp (a group of 32 threads) take different execution paths, leading to performance degradation as the GPU must wait for all threads in the warp to complete their execution.

To address this issue, we can refactor the code to utilize bit-wise operations inside the kernel to completely avoid the `switch cases`. As depicted in Algorithm 8 and 9, we can precompute the masks for all the threads, and based on their thread IDs, we can identify how the threads should initialize their variables without branching. This approach allows us to eliminate the `switch cases` and ensure that all threads follow the same execution path, thereby minimizing thread divergence and improving performance. The same approach is applied to the LP-117 version of the algorithm

Algorithm 8: LP-45, CUDA-v2: Recursively generate the masks for GPU threads with *bits*-many *0*s in a 9-bit field

```
1  Function intToBinarySequence(a):
2      switch a do
3          case 1: return 0b000000001
               // ...
4          case 9: return 0b100000000
5      end
6  end
7  Function intToBinarySequenceNegated(a):
8      switch a do
9          case 1: return 0b111111110
               // ...
10         case 9: return 0b011111111
11     end
12 end
13 Function populateMasks(masks, bits, depth, length, index, entry, count):
14     if depth ≥ bits then
15         masks[count] ← entry
16         count ← count + 1
17         return
18     end
19     for i ← index to length − (bits − depth) do
20         entry ← entry & intToBinarySequenceNegated(i + 1)
21         populateMasks(masks, bits, depth + 1, length, i + 1, entry, count)
22         entry ← entry | intToBinarySequence(i + 1)
23     end
24 end
25 Function main():
       // ...
26     Initialize masks_84_3bits_h, masks_84_6bits_h, and masks_126_4bits_h
        arrays
27     mask ← 0b111111111  // 9-bit mask with all bits set to 1
28     mask_count ← 0
29     populateMasks(masks_126_4bits_h, 4, 0, 9, 0, mask, mask_count)
30     populateMasks(masks_84_3bits_h, 3, 0, 9, 0, mask, mask_count)
31     populateMasks(masks_84_6bits_h, 6, 0, 9, 0, mask, mask_count)
       // Copy masks to global memory
       // The rest of the main function, similar to Algorithm 7
32 end
```

as well with a relative speedup of 10.42× compared to the baseline implementation (Section 4.1).

Algorithm 9: LP-45, CUDA-v2: Sequence A construction.

```
1  ___device___ ContructA():
2      Initialize array A[PairLength + 1]
3      foreach i ∈ {1, 2} do // Update A based on xp3_i
4          mask ← masks_84_6bits[xp3_i − 1]
5          for j ← 0 to 8 do
6              A[Om3s[i] + j × 5] ← (mask&(1 << j))?1 : −1
7          end
8      end
9      foreach i ∈ {1} do // Update A based on xp1_i
10         mask ← masks_126_4bits[xp1_i − 1]
11         for j ← 0 to 8 do
12             A[Op1s[i] + j × 5] ← (mask&(1 << j))?1 : −1
13         end
14     end
15     foreach i ∈ {1, 2} do // Update A based on xm3_i
16         mask ← masks_84_3bits[xm3_i − 1]
17         for j ← 0 to 8 do
18             A[Om3s[i] + j × 5] ← (mask&(1 << j))?1 : −1
19         end
20     end
21     return A
22 end
```

5.2 Utilizing *Constant Memory*

Constant memory is a fast, limited, and read-only GPU memory space, especially designed for efficient access to broadcast data. When all threads in a half-warp access the same constant memory address, the value can be read in one operation, optimizing bandwidth and caching. Subsequent accesses by other half-warps to the same address utilize the constant cache, resulting in less memory traffic. This memory is ideal for storing small, uniformly accessed read-only values, like lookup tables, or configuration parameters.

Therefore, in addition to removing branch divergence, we can further optimize the GPU implementation by utilizing *constant memory* for frequently accessed data. As depicted in Table 2, the NVIDIA A30 GPU has a total of 64KB of *constant memory*. By storing variables including `Op1s`, `Op3s`, `Om3s`, and `omega_powers` as well as the created `masks` from the previous step, in *constant memory*, we can reduce *global memory* accesses and improve performance. This improvement leads to a relative speedup of 1.59× for the LP-117 version of the algorithm, compared to Section 5.1.

5.3 Utilizing *Shared Memory*

Another common optimization in GPU computation is the utilization of *shared memory*, which is a fast, on-chip memory that is shared with threads within the same block. By storing frequently accessed data in *shared memory*, we can further reduce the number of accesses to *global memory*. However, as we have assigned each data

point of the address space to a single thread, which is independent of other threads, utilizing *shared memory* seems counterintuitive. In the current version, the candidate sequences A and B are stored in each thread's local memory which is stored in the *global memory*. Also, as the memory accesses are not cache friendly, the load and store operations are high-latency.

To alleviate the intensive *global memory* accesses, we can store the candidate sequences in the *shared memory* to force low-latency accesses for the active warps. This optimization demonstrate a relative speedup of 2.07× for LP-117 compared to the previous optimization (Section 5.2).

5.4 Loop Unrolling, Loop Fission, and Loop Reordering

Loop unrolling is a common optimization technique that can improve performance by reducing the overhead of loop control and increasing instruction-level parallelism. In the context of the LP algorithm, we can apply loop unrolling to the nested loops in the GPU kernel and the `ConstructA` function (Algorithm 9). Moreover, due to the threads task assignment, we have several redundant iterations of the nested loops that can be moved to the outer loops, where the `xp3` and `xm3` variables are modified. Therefore, using *loop fission* and *loop reordering* techniques, we can integrate the `ConstructA` function into the GPU kernel at different levels of the nested loops. This optimization leads to a relative significant speedup of 1.52× compared to the previous optimization (Section 5.3).

5.5 Optimizing *Shared Memory* Usage

Although utilizing *shared memory* is a great technique to improve GPU kernels performance, overusing it leads to lower occupancy. Occupancy is the ratio of the number of active warps to the maximum number of warps per Streaming Multiprocessor (SM). While higher occupancy does not necessarily translate to higher performance, lower occupancy often limits the device peak potential due to inability to hide latencies. As a rule of thumb, an occupancy of at least 50% is considered an efficient work distribution on the GPU [8]. Moreover, large differences between the theoretical and achieved occupancy shows that the workload is highly imbalanced.

Similarly in LP algorithm, overusing *shared memory* leads to performance degradation due to decreased occupancy. Considering the amount of *shared memory* and the size of sequences A and B per thread, reducing the sequences data type from `int` to `short int`, or even to `char`, increases the occupancy thereby improving the performance by 1.26× compared to the previous version (Section 5.4). Note that this data type representation does not result in data loss, as the sequence indices are populated with either 1 or -1 (See Algorithm 9).

5.6 Tuning Kernel Parameters

Another attribute of GPU occupancy is the *thread block* size and dimension. While larger block size helps with low-latency synchronization among higher number of threads, it decreases the number of resident *thread blocks* per SM due to lack of enough resources for all the threads. Changing the *thread block* and *grid* size from

$(1, 1, 84) \times (1, 32, 32)$ to $(84, 84, 84) \times (1, 1, 32)$ improves the performance by 1.21× compared to the previous version (Section 5.5).

5.7 Decreasing Memory Stalls by Increasing Work Per Thread

In the previous steps, we have optimized the GPU kernel to reduce memory stalls and improve performance. However, we can further enhance the performance by increasing the work done per thread to hide more memory stalls. In the LP-117 version, we can achieve this by moving another loop inside the kernel, which allows each thread to perform more computations before waiting for memory accesses. This optimization leads to a relative speedup of 1.02× compared to the previous version (Section 5.6).

5.8 Minimizing *Shared Memory* Bank Conflicts

Profiling the GPU kernel reveals that the *shared memory* accesses are not fully optimized, because of the high number of bank conflicts. As discussed in Section 2.3.1, *shared memory* is divided into 32 banks, and each bank can be accessed by multiple threads simultaneously. However, if multiple threads within the same warp attempt to access different addresses that map to the same memory bank at the same time, a *bank conflict* occurs, leading to serialization of memory accesses and performance degradation.

One way to minimize bank conflicts is to ensure that threads access different banks by introducing padding in the *shared memory* arrays [29]. However, finding the correct padding size can be challenging, as it depends on the number of active threads, the size of the *shared memory* arrays, and the access patterns. In our case, Algorithm 10 finds the optimal padding size, considering the mentioned parameters. This approach can be applied on various lengths of LP algorithms with different configurations. The best padding size determined by Algorithm 10 is 7, and it is used as a `__shared__` buffer with the type and size in the GPU kernel $char\ \ A[blockSize][pairLength + padding]$. This optimization leads to a relative speedup of 1.14× compared to the previous version (Section 5.7).

5.9 Optimizing Register Usage

Registers are the fastest memory available on the GPU, and optimizing their usage is crucial for performance. However, excessive register usage can lead to register pressure, spilling registers to local memory, which is significantly slower. To optimize register usage, we first need to analyze the register usage of our GPU kernel. Utilizing compiler option `-Xptxas="-v"` allows us to see the number of registers used per thread. We can also utilize the `nvprof` or `Nsight Compute` tools to analyze the register usage and identify potential bottlenecks.

In our case, we found that the GPU kernel uses a significant number of registers, mostly to store PSDs and their intermediate results (112 per thread). To reduce the register usage, we lowered the precision of the PSDs, and we also store the PSDs lazily once they are validated. Moving PSDs to *shared memory* helps with reducing the register usage, but it increases the memory usage and leads to lower occupancy.

Algorithm 10: Finding Optimal Shared Memory Padding to Minimize Bank Conflicts

```
1  Function CalculateBankConflicts(T, W, P, C, B):
       // T: threads per block, W: width, P: padding, C: accessed column, B: number of banks
2      for b ← 0 to B − 1 do
3          bank_conflicts[b] ← 0
4      end
5      W_p ← W + P   // Padded width
6      for t ← 0 to T − 1 do
7          address ← t × W_p + C
8          bank ← ⌊address/4⌋ mod B
9          bank_conflicts[bank] ← bank_conflicts[bank] + 1
10     end
11     return bank_conflicts
12 end
13 Function FindBestPadding(T, W, C, B):
14     min_conflicts ← T
15     best_padding ← 0
16     for P ← 0 to B − 1 do
17         conflicts ← CalculateBankConflicts(T, W, P, C, B)
18         max_conflict ← max(conflicts)
19         if max_conflict < min_conflicts then
20             min_conflicts ← max_conflict
21             best_padding ← P
22         end
23     end
24     return best_padding
25 end
26 Function Main():
27     T ← 32 , W ← 117 , B ← 32
       // Traversing all cases for computing PSDs: 13 × 9 = 117
28     for col ← 0 to 13 do
29         for case ← 0 to 9 do
30             C ← col + 13 × case   // Absolute accessed column
31             P* ← FindBestPadding(T, W, C, B)
32             Append P* to best_paddings
33         end
34     end
       // best_paddings now contains the optimal padding for each column
35 end
```

This optimization reduces the register usage to 32 per thread, and it improves the performance by 1.06× compared to the previous version (Section 5.8).

5.10 Removing Ternary Operations

Ternary operations in the Algorithm 9 (Lines 6, 12, and 18) can cause unnecessary stalls in the GPU pipeline, as they introduce additional branches that can lead to

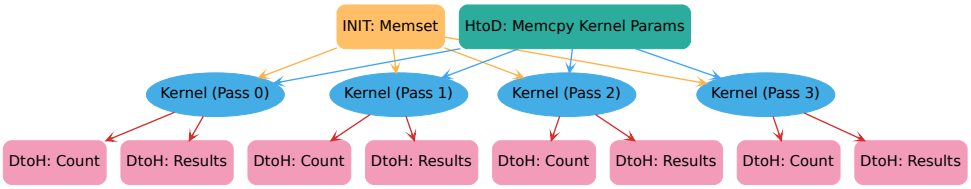

Fig. 11: Utilizing CUDA Graphs to encapsulate the entire sequence of kernel launches and memory operations (four passes of pipelined execution) in the LP-117 algorithm (Sequences A).

thread divergence. To optimize the code further, we can replace the ternary operations with simple arithmetic binary operations. More specifically, we can replace instructions like:

- $A1[local\_id][Om3s_d[i]] = (mask\&1)?1 : -1$; with:
- $A1[local\_id][Om3s_d[i]] = -1 + (((mask >> 0)\&1) << 1)$;

This change introduces two additional bitwise operations plus one *add* operation. Our tests show that even with the added operations, the new instructions still outperform the ternary operations. Although this improvement is a relatively minor optimization, but it can still contribute to performance gains by reducing the number of branches and improving instruction-level parallelism and pipeline stalls. As a result, we can achieve a relative speedup of 1.01× compared to the previous version (Section 5.9).

5.11 CUDA Graphs Utilization

CUDA Graphs allow us to create a sequence of CUDA operations and execute them as a single entity. This can help reduce the overhead of launching multiple kernels and improve performance. In the LP algorithm, we can utilize CUDA Graphs to encapsulate the entire sequence of kernel launches and memory operations, thereby reducing the overhead of multiple kernel launches. As depicted in Algorithm 11, we can create a CUDA Graph that includes all the memory operations and kernel launches required for the pipelined execution of the LP algorithm. Other unrelated parts of the code, including the CUDA kernels are omitted for clarity. Note that the same approach is applied to both sequences A and B.

In more detail, in the `SetupGraph` function, we create a new CUDA Graph and add the necessary memory operations and kernel launches (with dependencies) for each pass of the pipelined execution (Lines 2-10). Once the graph is set up, we can instantiate it and upload it to the device (Line 11 and Line 12). In the `LaunchGraph` function, we simply launch the graph on the corresponding stream and synchronize the device to ensure completion (Lines 15-18). Finally, in the `Main` function, we initialize the data structures and parameters, set up the graph, and instead of launching individual kernels, we launch the entire graph for execution (Lines 21-28).

Figure 11 illustrates the structure of a four-pass CUDA Graph for the LP-117 created by Algorithm 11. In this graph, the first nodes on top are the memory operations

Algorithm 11: Utilizing CUDA Graphs to encapsulate the entire sequence of kernel launches and memory operations in the LP algorithm (Sequences A).

```
Function SetupGraph():
    Initialize a new CUDA Graph // cudaGraphCreate
    Add memory initialization node to the graph // cudaGraphAddMemsetNode
    Add Host-to-Device (H2D) memory operations to transfer input data
     // cudaGraphAddMemcpyNode
    for i ← 0 to num_passes − 1 do
        Create a kernel node for pass i, configured with appropriate parameters
        Add the kernel node to the graph, as a successor to the initial memory
         operations. // cudaGraphAddKernelNode
        Create Device-to-Host (D2H) memcpy nodes for transferring results back to
         the host
        Add the memcpy nodes as successors to the corresponding kernel nodes
         // cudaGraphAddMemcpyNode
    end
    Instantiate and store the graph // cudaGraphInstantiate
    Upload the graph to the device // cudaGraphUpload
    return the created graph
end
Function LaunchGraph(graph):
    Launch the CUDA Graph on a corresponding stream // cudaGraphLaunch
    Synchronize the device to ensure completion // cudaStreamSynchronize
end
Function Main():
    Initialize data structures and parameters
    graph ← SetupGraph()
     // ...
     // 8-fold nested loops for different configurations
    Launch the graph for execution LaunchGraph(graph)
    Save the results (similar to LP-45 in Algorithm 7)
end
```

required to set up the algorithm. Then, on the next level, we have four kernel nodes, each representing one pass of the pipelined execution. Finally, we have the memory operations required to retrieve the results from the GPU back to the host.

We have evaluated various configurations of CUDA Graphs, including different numbers of passes up to 1024 resulting in a CUDA Graph of more than 3072 nodes. However, the performance improvement from utilizing CUDA Graphs is observed to be marginal, due to the fact that the overhead of launching multiple kernels is already low comparing to the overall execution time of the LP algorithm. Moreover, the non-CUDA Graph version of the LP algorithm is already designed to overlap computation and memory transfers using multiple streams, which further reduces the potential benefits of CUDA Graphs. However, this approach is not scalable with increasing number

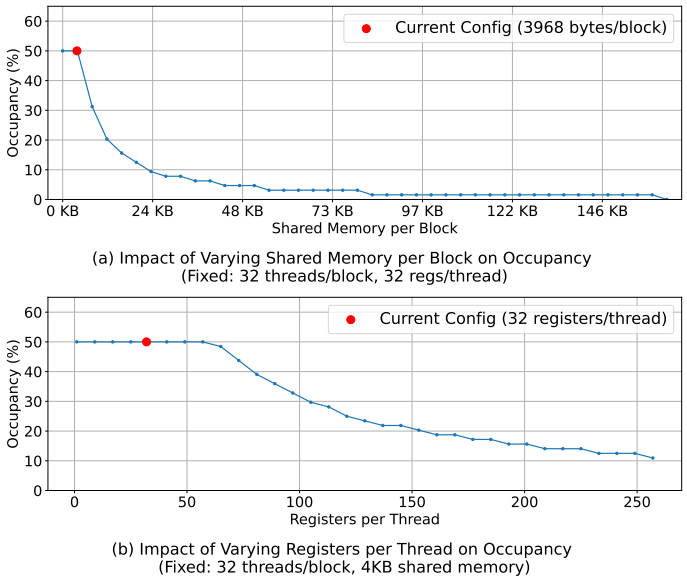

Fig. 12: Occupancy of the LP-117 CUDA kernel on the A30 GPU with respect to *shared memory* and *register* utilization. The current implementation is marked with a red dot.

of passes, as managing multiple streams and overlapping operations becomes increasingly complex. Nevertheless, this optimization still contributes to performance gains by reducing the overhead of multiple kernel launches. As a result, we achieve a relative speedup of 1.01× compared to the previous version (Section 5.10). We speculate if CUDA Graphs were applied in earlier stages of the optimizations, they would have shown more noticeable performance improvement comparing to the previous steps.

5.12 Profiling and Performance Analysis

In order to ensure that there is no more room for performance improvements, we profiled the GPU kernel after each optimization to identify performance bottlenecks and understand how the GPU resources are utilized. We used *Nsight Compute* and *nvprof* to analyze the performance of the GPU kernel. Moreover, we developed an offline algorithm to predict GPU occupancy based on the *shared memory* and *register* usage, which is provided in Algorithm 12.

As shown in Figure 12, the occupancy of the LP-117 CUDA kernel on the A30 GPU is at 50% with respect to *shared memory* and *register* usage. A30 GPU can schedule up to 64 warps per SM, and as discussed before, aiming for an occupancy of at least 50% is a good practice to ensure that the GPU can hide latencies effectively. As Figure 12.a depicts, the occupancy cannot be further improved as the *shared memory* per block is the limiting factor; and as Figure 12.(b) shows, the threads can use more registers (up to 64) without affecting the occupancy. However, the current version of

Algorithm 12: Predicting GPU Occupancy based on device specifications, as well as Shared Memory and Register Usage

Function `predictOccupancy`($threadsPerBlock, sharedMemPerBlock, registersPerThread$):
  // Constants for the A30 GPU extracted at runtime
  $threadsPerWarp \leftarrow 32$
  $maxWarpsPerSM \leftarrow 64$
  $maxThreadsPerSM \leftarrow 2048$
  $maxBlocksPerSM \leftarrow 32$
  $registersPerSM \leftarrow 65536$
  $maxRegistersPerThread \leftarrow 255$
  $maxRegistersPerBlock \leftarrow 65536$
  $maxSharedMemPerBlock \leftarrow 167936$
  $sharedMemPerSM \leftarrow 167936$
  $warpsPerBlock \leftarrow \lceil threadsPerBlock / threadsPerWarp \rceil$
  // Warp per thread limits
  $maxBlocks \leftarrow \min\left(maxBlocksPerSM, \frac{maxThreadsPerSM}{threadsPerBlock}, \frac{maxWarpsPerSM}{warpsPerBlock}\right)$
  // Register limit
  $regsPerBlock \leftarrow registersPerThread \times threadsPerBlock$
  if $regsPerBlock > maxRegistersPerBlock$ **then**
    $maxBlocks \leftarrow 0$
  else
    $maxBlocks \leftarrow \min(maxBlocks, registersPerSM / regsPerBlock)$
  end
  // Shared memory limit
  if $sharedMemPerBlock > maxSharedMemPerBlock$ **then**
    $maxBlocks \leftarrow 0$
  else
    $maxBlocks \leftarrow \min\left(maxBlocks, \frac{sharedMemPerSM}{sharedMemPerBlock}\right)$
  end
  $totalThreads \leftarrow maxBlocks \times threadsPerBlock$
  $occupancy \leftarrow totalThreads / maxThreadsPerSM$
  return $occupancy \times 100$
end

the algorithm and selected datatype cannot be altered in anyways to optimize the register usage further.

5.13 Summary of Optimizations

Until now, we have discussed various optimization techniques applied to the GPU implementation of the LP algorithm. Each optimization step is designed to address specific performance bottlenecks and improve the overall efficiency of the algorithm on the GPU architecture. Table 3 summarizes the performance improvements achieved through each optimization step for the LP-117 version of the algorithm, compared to the baseline CPU implementation on a single core of an Intel Xeon Gold 6338

Table 3: Performance of the LP-117 CUDA algorithms tested on a single NVIDIA A30 GPU, compared with the baseline CPU implementation on a single core of Intel Xeon Gold 6338 CPU.

Version	**Brief Description**	**Relative Speedup**	**Overall Speedup**
LP-117-v1 (Section 4.1)	Naive CUDA implementation	18.95×	18.95×
LP-117-v2 (Section 5.1)	Remove branch divergence	10.42×	197.56×
LP-117-v3 (Section 5.2)	Utilizing *constant memory*	1.59×	314.27×
LP-117-v4 (Section 5.3)	Utilizing *shared memory*	2.07×	652.55×
LP-117-v5 (Section 5.4)	Loop unrolling and loop reordering	1.52×	997.6×
LP-117-v6 (Section 5.5)	Reducing *shared memory* usage	1.26×	1,257.59×
LP-117-v7 (Section 5.6)	Tuning kernel parameters	1.21×	1,523.92×
LP-117-v8 (Section 5.7)	Increasing work per thread	1.02×	1,559.63×
LP-117-v9 (Section 5.8)	Minimize *shared memory* bank conflicts	1.14×	1,781.73×
LP-117-v10 (Section 5.9)	Decreasing register pressure	1.06×	1,900.36×
LP-117-v11 (Section 5.10)	Replacing ternary operations with binary	1.01×	1,912.42×
LP-117-v12 (Section 5.11)	Utilizing CUDA Graphs	1.01×	1,927.53×

CPU. The overall speedup achieved by the final optimized version (LP-117-v12) is approximately 1,927.53×.

5.14 Multi-GPU Implementation

After optimizing the single-GPU version, we combined the GPU implementation with the MPI implementation to provide a multi-GPU version of the algorithm. The multi-GPU implementation uses the same MPI approach as the CPU-only version (discussed in Section 3.4), but with each MPI process assigned to a single GPU. To evaluate the scalability of the multi-GPU implementation, we conducted weak scaling evaluation, where the overall problem size increases proportionally with the number of GPUs to maintain a constant workload per device.

Figure 13 demonstrates how the algorithm scales: as the number of GPUs increases from 1 to 16, the total execution time increases slightly, indicating that the parallelization strategy effectively distributes the workload and minimizes inter-GPU

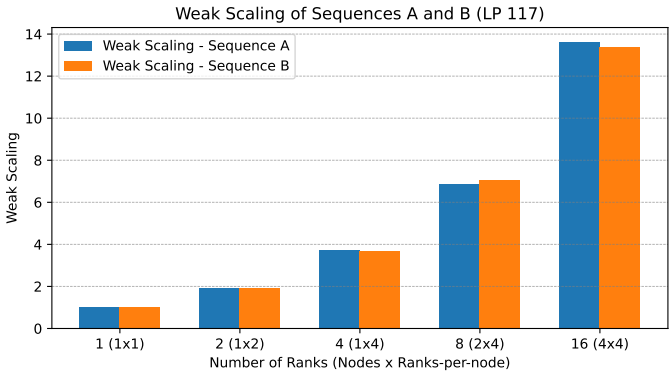

Fig. 13: Weak scaling of the LP-117 multi-GPU implementation on a single node with up to 16 NVIDIA A30 GPUs. Each rank has a constant search space to sweep.

communication overhead. This confirms that the MPI+CUDA implementation is well-suited for large-scale deployments.

5.15 Improving Resource Utilization by harnessing MIG and MPS

Although we have achieved significant performance improvements through various optimizations so far, there is still room for further enhancements by saturating the GPU resources more effectively. This is especially true for modern GPUs, which have a larger number of SMs and higher memory bandwidth compared to older models. A common pattern is to oversubscribe the GPU, placing multiple MPI processes per GPU, to fill the execution and memory resources more effectively. Moreover, oversubscribing the GPUs can improve the application-level throughput and save costs.

However, placing multiple MPI processes per GPU can lead to resource contention and increased overhead due to context switching. When multiple processes target the same GPU, the driver creates a separate *context* for each. Because the GPU hardware can only execute one *context* at a time, the *contexts* must be time-sliced, which can introduce additional latency and reduce overall performance. This overhead can negate the benefits of increased resource utilization if not managed properly.

To address this challenge, we can utilize NVIDIA MPS [42], which is designed to improve the performance of multiple processes sharing a single GPU. By utilizing NVIDIA MPS, we can manage the *contexts* more efficiently and reduce the overhead associated with context switching. MPS consists of a server process that manages multiple client processes, allowing them to share the same *context* on the GPU. This allows the MPI processes to connect to the MPS server instead of creating their own *contexts*. By using MPS, we can replace the driver-level time-slicing with a lightweight spatial-slicing.

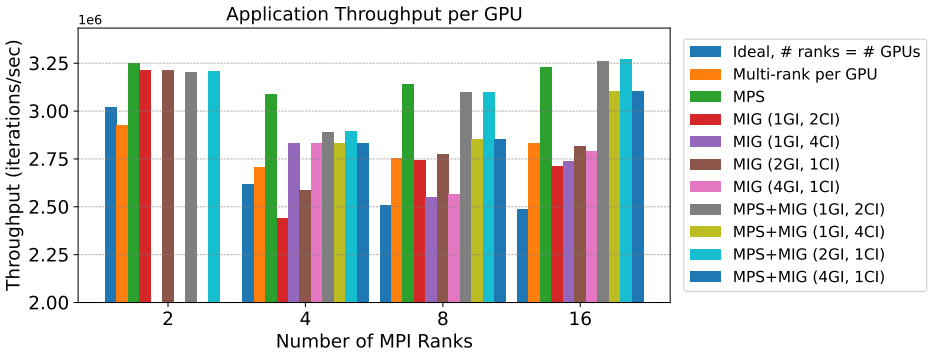

Fig. 14: Application throughput per GPU (iterations/sec) for varying number of MPI processes running LP-117. While the *Ideal* case maps one rank to one GPU, all other configurations oversubscribe a single physical device. Higher values indicate better resource utilization and cost-efficiency

Another approach to improve resource utilization on modern GPUs is to use NVIDIA MIG [43], which allows partitioning a single physical GPU into multiple isolated instances. Depending on the slicing method, each MIG instance has its own dedicated resources, including SMs, memory, and cache, allowing multiple workloads to run concurrently on the same physical GPU without interfering with each other. By using MIG, we can create multiple smaller GPU instances that can be assigned to different MPI processes, thereby improving resource utilization and reducing contention. While MPS provides concurrency, MIG provides isolation by guaranteeing a predictable Quality of Service (QoS) for each MPI process.

Furthermore, these two technologies can be combined to further enhance resource utilization. In this way, we can create multiple MIG instances on a single physical GPU, and then run multiple MPI processes on each instance using MPS. MIG provides a coarse-grained, hardware-level partitioning of the GPU, while MPS provides a fine-grained, software-level sharing of the GPU resources within each MIG instance. This allows us to take advantage of the benefits of both technologies, improving concurrency while maintaining isolation between workloads.

Figure 14 illustrates the application throughput per GPU (iterations/sec) for varying number of MPI processes running the LP-117 algorithm on a single NVIDIA A30 GPU. The *Ideal* case maps one rank to one GPU, while all other configurations oversubscribe a single physical device. The other configurations include: (1) *Multi-rank per GPU*, where multiple MPI processes share the same GPU without any special management; (2) *MPS*, where multiple MPI processes share the GPU using NVIDIA MPS; (3) *MIG*, where multiple MPI processes run on separate MIG instances without MPS; and (4) *MPS +MIG*, where multiple MPI processes run on separate MIG instances using MPS. Various configurations of MIG instances are included: 1-GI-2-CI (1 GPU Instance (GI) with 2 Compute Instance (CI)), 1-GI-4-CI, 2-GI-1-CI, and 4-GI-1-CI.

Moreover, Figure 15 presents the speedup of the multi-GPU implementation on a single NVIDIA A30 GPU relative to the *Multi-rank per GPU* baseline, using different

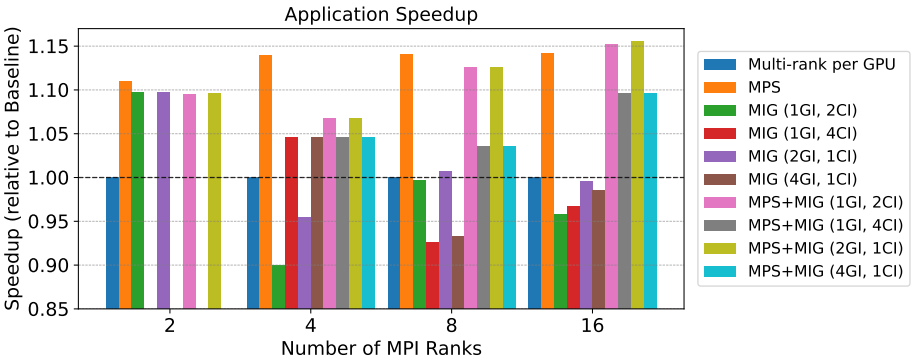

Fig. 15: Speedup of multi-GPU implementation on a single NVIDIA A30 GPU relative to the *Multi-rank per GPU* baseline, using different configurations of MPI processes, MIG instances, and MPS. Each rank has a constant search space to sweep.

configurations of MPI processes, MIG instances, and MPS. Based on the results, we can make the following observations:

- **Oversubscription significantly improves resource utilization.** A primary observation from Figure 14 is that oversubscribing the GPU is substantially more efficient than a 1:1 rank-to-GPU mapping. While the *Ideal* case yields the lowest per-GPU throughput, all 16-rank oversubscribed configurations achieve higher throughputs, confirming that the 1:1 mapping underutilizes the hardware.
- **MPS consistently mitigates context-switching overhead.** As shown in Figure 15, enabling MPS provides a consistent performance improvement of over the baseline. This illustrates the performance penalty of the default driver-level time-slicing and demonstrates that MPS effectively eliminates this overhead.
- **MIG-only configurations degrade performance when oversubscribed.** Conversely, employing MIG partitions without MPS (the *MIG* configurations) results in a consistent performance degradation relative to the baseline when number of ranks exceeds the number of MIG instances (Figure 15). This suggests that LP-117 is already efficient enough that the strict hardware partitioning of MIG, when combined with driver-level time-slicing within each instance, introduces more resource contention or scheduling inefficiencies that are more detrimental than the baseline.
- **The most effective configuration is the combination of MPS and MIG**. The *MPS +MIG* configurations achieve the highest throughput (Figure 14), demonstrating that the two technologies complement each other well in our implementation. By leveraging the fine-grained resource management of MPS within the isolated environments provided by MIG, we can maximize the utilization of the underlying hardware when oversubscribing the GPU.

Table 4: Comparison of key specifications between NVIDIA H100 and A30 GPUs.

Feature	**NVIDIA H100 (SXM)**	**NVIDIA A30**
GPU Architecture	Hopper	Ampere
SM Count	132	56
CUDA Cores per SM	128	64
Total CUDA Cores	16896	3804
Transistors	80 Billion	54.2 Billion
Memory Size	80 GB HBM3	24 GB HBM2
Memory Bandwidth	3.35 TB/s	933 GB/s
Memory Bus Width	5120-bit	3072-bit
L2 Cache Size	50 MB	24 MB
Shared Memory per SM	228 KB	164 KB
Shared Memory per Block	48 KB	48 KB
Registers per Block	65536	65536

5.16 Scalability on Modern GPUs

To evaluate the scalability of our optimized GPU implementation on modern hardware, we tested the performance of the LP-117 algorithm on a more recent GPU model, the NVIDIA H100, on Nibi cluster from Compute Canada. The H100 is based on the Hopper architecture and offers significant improvements in terms of computational power, memory bandwidth, and overall efficiency compared to the A30.

As shown in Table 4, the H100 has about 4.4× more SMs and CUDA cores, as well as 3.6× higher memory bandwidth compared to the A30. These enhancements are expected to yield substantial performance gains, and as expected, our evaluation shows a speedup of 3.4× when running the LP-117 algorithm on the H100 compared to the A30. This means that the H100 implementation achieves an overall speedup of $6,569\times$ compared to the baseline CPU implementation on a single core of Intel Xeon Gold 6338 CPU. This demonstrates that our optimized GPU design and implementation scales well with modern GPU technologies, making it suitable for future hardware platforms.

6 Conclusions and Future Work

In this study, we highlighted the critical role of HPC in addressing the immense computational demands of constructing LPs. The infeasibility of solving large orders like LP-117 on single-core systems, with estimates exceeding 10 billion years, necessitated advanced parallelization and optimization strategies.

We showed CPU-based approaches demonstrate strong scalability. OpenMP effectively utilized multi-core CPUs up to 64 threads, though diminishing returns were observed at 128 threads due to NUMA effects. The MPI implementation extended scalability across nodes, achieving over 90% parallel efficiency and up to 230× speedup (strong scaling) with 256 ranks, while analysis revealed that for over-subscribed configurations, intra-node contention was the main bottleneck, not inter-node bandwidth.

GPU acceleration yielded substantial performance gains by exploiting the algorithm's parallelism. Key optimizations included:

- Removing branch divergence for 10.42× speedup.
- Leveraging constant/shared memory for 1.59× and 2.07× gains.
- Applying loop transformations for 1.52× improvement.
- Managing resources to achieve 1.26× speedup.
- Employing fine-tuning strategies (e.g., register optimization, shared memory bank conflict avoidance) for additional cumulative gains.
- Utilizing CUDA Graphs to reduce kernel launch overhead, achieving 1.01× speedup.
- Utilizing MPS and MIG for better resource utilization, showing 1.3× improvements in application throughput per GPU.

Collectively, GPU optimizations achieved over $6,500\times$ and $1,900\times$ speedup on NVIDIA H100 and A30 GPU, respectively, versus the single-core baseline executed on an Intel Xeon Gold 6338 CPU. Profiling indicated a 50% occupancy, bounded by shared memory. The hybrid MPI+CUDA solution confirmed significant weak scaling (close to 14× for 16 GPU devices) and minimal inter-GPU overhead, validating its scalability. For future work, we believe further optimizations could be achieved through:

- Exploring other architectures, like FPGAs, for acceleration.
- Investigating alternative parallelization strategies (e.g., task-based models).
- Enhancing load balancing across heterogeneous systems.
- Utilizing more advanced GPU features, such as *Tensor Cores* to accelerate specific computations or CUDA *AsyncCopy* to improve data transfer to the device *shared memory*.

Acknowledgements. This research was supported by Distributive Corp. Canada (distributive.network), and the authors would like to thank them for their support. This research was enabled in part by support provided by the Digital Research Alliance of Canada (alliancecan.ca).

References

[1] Fletcher, R.J., Gysin, M., Seberry, J.: Application of the discrete Fourier transform to the search for generalised Legendre pairs and Hadamard matrices. Australas. J. Combin. **23**, 75–86 (2001)

[2] Arasu, K.T., Bulutoglu, D.A., Hollon, J.R.: Legendre G-array pairs and the theoretical unification of several G-array families. J. Combin. Des. **28**(11), 814–841 (2020) https://doi.org/10.1002/jcd.21745

[3] Kotsireas, I., Koutschan, C.: Legendre pairs of lengths $\ell \equiv 0 \pmod 3$. J. Combin. Des. **29**(12), 870–887 (2021)

[4] Kotsireas, I.S., Koutschan, C., Bulutoglu, D.A., Arquette, D.M., Turner, J.S., Ryan, K.J.: Legendre pairs of lengths $\ell \equiv 0 \pmod 5$. Spec. Matrices **11**,

20230105–15 (2023)

[5] Turner, J.S., Kotsireas, I.S., Bulutoglu, D.A., Geyer, A.J.: A Legendre pair of length 77 using complementary binary matrices with fixed marginals. Des. Codes Cryptogr. **89**(6), 1321–1333 (2021)

[6] Apostol, T.M.: Introduction to Analytic Number Theory. Undergraduate Texts in Mathematics, p. 338 (1976). https://doi.org/10.1007/978-1-4757-5579-4

[7] Djokovic, D.Z., Kotsireas, I.S.: Compression of periodic complementary sequences and applications. Des. Codes Cryptogr. **74**(2), 365–377 (2015) https://doi.org/10.1007/s10623-013-9862-z

[8] CUDA. https://docs.nvidia.com/cuda/index.html. https://docs.nvidia.com/cuda/index.html [Accessed: 2025-10-01] (2025)

[9] OpenMP. https://www.openmp.org/ (2024)

[10] MPI Forum. https://www.mpi-forum.org/ (2024)

[11] Golomb, S.W., Gong, G.: Signal Design for Good Correlation: for Wireless Communication, Cryptography, and Radar, p. 438 (2005)

[12] Colmenares, J., Galizia, A., Ortiz, J., Clematis, A., Rocchia, W.: A combined mpi-cuda parallel solution of linear and nonlinear poisson-boltzmann equation. BioMed Research International **2014**(1), 560987 (2014) https://doi.org/10.1155/2014/560987 https://onlinelibrary.wiley.com/doi/pdf/10.1155/2014/560987

[13] Yang, C.-T., Huang, C.-L., Lin, C.-F.: Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. Computer Physics Communications **182**(1), 266–269 (2011) https://doi.org/10.1016/j.cpc.2010.06.035 . Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009

[14] Venetis, I.E., Saltogianni, V., Saltogianni, V., Stiros, S.C., Gallopoulos, E.: Multivariable inversion using exhaustive grid search and high-performance gpu processing: a new perspective. Geophysical Journal International **221**, 905–927 (2020) https://doi.org/10.1093/GJI/GGAA042

[15] Eum, S., Song, M., Kim, S., Seo, H.: Efficient gpu parallel implementation and optimization of aria for counter and exhaustive key-search modes. Electronics **14**(10) (2025) https://doi.org/10.3390/electronics14102021

[16] Shreeyansh, V., Gad, A.S., Rao, B.A., Kini, N.G.: Parallelizing exponential search algorithms with mpi and cuda for high-performance computing. In: 2025 International Conference on Next Generation Communication and Information Processing (INCIP), pp. 285–288 (2025). https://doi.org/10.1109/INCIP64058.2025.11019520 . IEEE

[17] Lončar, V., Young-S., L.E., Škrbić, S., Muruganandam, P., Adhikari, S.K., Balaž, A.: Openmp, openmp/mpi, and cuda/mpi c programs for solving the time-dependent dipolar gross-pitaevskii equation. Computer Physics Communications **209**, 190–196 (2016) https://doi.org/10.1016/j.cpc.2016.07.029

[18] MPICH. https://www.mpich.org/ (2024)

[19] MVAPICH. https://mvapich.cse.ohio-state.edu/ (2024)

[20] Open MPI. https://www.open-mpi.org/ (2024)

[21] Li, Y., Zhou, B., Zhang, J., Wei, X., Li, Y., Chen, Y.: Radik: Scalable and optimized gpu-parallel radix top-k selection. In: Proceedings of the 38th ACM International Conference on Supercomputing. ICS '24, pp. 537–548 https://doi.org/10.1145/3650200.3656596, New York, NY, USA (2024). https://doi.org/10.1145/3650200.3656596 . Association for Computing Machinery

[22] Iqbal, U., Davies, T., Perez, P.: A review of recent hardware and software advances in gpu-accelerated edge-computing single-board computers (sbcs) for computer vision. Sensors **24**(15) (2024) https://doi.org/10.3390/s24154830

[23] Sojoodi, A.H., Salimi Beni, M., Khunjush, F.: Ignite-GPU: a GPU-enabled in-memory computing architecture on clusters. Journal of Supercomputing, 1–28 (2020) https://doi.org/10.1007/s11227-020-03390-z

[24] NVIDIA Collective Communications Library. https://github.com/NVIDIA/nccl (2024)

[25] Prat, R., Carrard, T., Amarsid, L., Richefeu, V., Doncecchi, C., Lafourcade, P., Latu, G., Vanson, J.-M.: ExaDEM: a HPC application based on exaNBody targeting scalable DEM simulations with complex particle shapes. Journal of Open Source Software **10**(106), 7484 (2025)

[26] Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B., Bal, H.E.: Optimization Techniques for GPU Programming. ACM Computing Surveys **55**(11), 1–81 (2023) https://doi.org/10.1145/3570638

[27] Yi, X.: A Study of Performance Programming of CPU, GPU accelerated Computers and SIMD Architecture. arXiv, 1–19 (2024) arXiv:2409.10661

[28] Choi, H., Lee, J.: Efficient use of gpu memory for large-scale deep learning model training. Applied Sciences **11**(21) (2021) https://doi.org/10.3390/app112110377

[29] Berney, K., Sitchinava, N.: Eliminating bank conflicts in gpu mergesort. In: Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures. SPAA '25, pp. 158–170 https://doi.org/10.1145/3694906.3743337, New York, NY, USA (2025). https://doi.org/10.1145/3694906.3743337 . Association

for Computing Machinery

[30] Lee, S., Oh, J., Kim, J., Go, S., Park, J., Mahajan, D.: Characterizing Compute-Communication Overlap in GPU-Accelerated Distributed Deep Learning: Performance and Power Implications. https://arxiv.org/abs/2507.03114 (2025)

[31] Srikanth, A., Trigila, C., Roncali, E.: Gpu optimization techniques to accelerate optigan—a particle simulation gan. Machine Learning: Science and Technology **5**(2), 027001 (2024) https://doi.org/10.1088/2632-2153/ad51c9

[32] Khan, A.H., Al-Mouhamed, M., Al-Mulhem, M., Ahmed, A.F.: Rt-cuda: A software tool for cuda code restructuring. International Journal of Parallel Programming **45**(3), 551–594 (2017) https://doi.org/10.1007/s10766-016-0433-6

[33] Kaur, R., Asad, A., Al Abdul Wahid, S., Mohammadi, F.: A survey of advancements in scheduling techniques for efficient deep learning computations on gpus. Electronics **14**(5) (2025) https://doi.org/10.3390/electronics14051048

[34] Murthy, G., Ravishankar, M., Baskaran, M., Sadayappan, P.: Optimal loop unrolling for gpgpu programs, pp. 1–11 (2010). https://doi.org/10.1109/IPDPS.2010.5470423 . IEEE

[35] Matsuo, R., Degawa, Y., Irie, H., Sakai, S., Shioya, R.: Flexible Approximate Computing for Mitigating Branch Divergence in GPUs . IEEE Micro **45**(02), 90–100 (2025) https://doi.org/10.1109/MM.2024.3504261

[36] Werkhoven, B.: Kernel tuner: A search-optimizing gpu code auto-tuner. Future Generation Computer Systems **90**, 347–358 (2019) https://doi.org/10.1016/j.future.2018.08.004

[37] Zhao, C., Gao, W., Nie, F., Zhou, H.: A Survey of GPU Multitasking Methods Supported by Hardware Architecture. IEEE Transactions on Parallel and Distributed Systems, 1–13 (2022) https://doi.org/10.1109/TPDS.2021.3115630

[38] Cui, C.: Acceleration of tensor-product operations with tensor cores. ACM Transactions on Parallel Computing **11**(4), 1–24 (2024)

[39] Sakdhnagool, P., Sabne, A., Eigenmann, R.: Regdem: Increasing GPU performance via shared memory register spilling. CoRR **abs/1907.02894** (2019) 1907.02894

[40] NVIDIA: Advanced NVIDIA CUDA Kernel Optimization Techniques with Handwritten PTX. Accessed: 2025-01-01

[41] Andrews, M., Witteveen, S.: GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. https://arxiv.org/abs/2506.20807 (2025)

[42] Multi-Process Service. https://docs.nvidia.com/deploy/mps/ (2024)

[43] Multi Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/ (2024)