CUDA Dynamic Parallelism Performance Evaluation

Lorenzo Sorrentino
Department of Computer Science
University of Salerno
Italy

***Abstract*—CUDA Dynamic Parallelism (CDP) allows GPU kernels to launch other kernels directly from the device, enabling recursive and irregular parallelism. This report evaluates its performance on three benchmarks: Mandelbrot set with adaptive mesh refinement, N-Body simulation with Direct-Sum and Barnes–Hut, and quadtree construction in recursive and iterative forms. Results show that CDP provides substantial benefits for hierarchical and adaptive algorithms, while in regular workloads its launch overhead often makes iterative implementations preferable.**

I. Introduction

CUDA Dynamic Parallelism (CDP) is a powerful feature introduced by NVIDIA that allows GPU kernels to launch additional kernels directly from the device without host-side intervention. This capability has the potential to simplify the implementation of nested or irregular parallelism patterns by removing the need for complex host-device synchronization or multiple kernel launches from the CPU.

However, while CDP increases flexibility and expressiveness, it introduces new runtime overheads, including higher kernel launch latency, additional memory pressure, and possible underutilization of hardware resources if not carefully managed. These trade-offs make it essential to perform a rigorous evaluation of CDP's performance to determine under what conditions it is beneficial compared to traditional flat parallelization approaches.

Motivation. Dynamic parallelism can be particularly useful in algorithms where the amount of parallel work is not known a priori and must be discovered at runtime, such as recursive tree traversals, graph exploration, or adaptive computations. Rather than overprovisioning resources or restructuring algorithms to fit a rigid thread-block grid, CDP enables a more natural expression of the algorithm's logic on the GPU itself. Nonetheless, developers must be cautious, as the overheads associated with nested kernel launches may negate the potential performance gains. Understanding when and how CDP leads to improvements is the central motivation of this study.

Related work. Wang and Yalamanchili [1] conducted one of the earliest and most comprehensive studies on CDP, characterizing its performance in unstructured GPU applications. They observed that dynamically formed pockets of parallelism can be exploited effectively using CDP, achieving up to $2.73\times$ speedup in ideal scenarios. However, they also identified substantial launch overheads, noting an average slowdown of $1.21\times$ in real-world conditions.

Jarząbek and Czarnul [2] extended the evaluation to real applications based on the SPMD and divide-and-conquer paradigms. Their findings suggest that CDP can improve code readability and maintainability, particularly in hierarchical algorithms like adaptive integration or recursive simulations. Nevertheless, the performance benefits were inconsistent, with some workloads performing worse than optimized iterative counterparts.

This report builds upon these prior studies by designing and evaluating several benchmarks specifically tailored to test CDP behavior. The focus is to quantify the impact of dynamic kernel launches in terms of execution time and scalability, and to provide practical recommendations for its effective use.

II. Background

GPU. Graphics Processing Units (GPUs) are massively parallel processors originally designed for graphics rendering, but now widely used for general-purpose computations (GPGPU). Unlike CPUs, which are optimized for low-latency sequential execution, GPUs excel at high-throughput parallelism. A modern GPU consists of thousands of simple cores grouped into Streaming Multiprocessors (SMs), capable of executing many threads concurrently.

Programming for GPUs involves a different computational model compared to CPUs. While CPUs typically execute a few complex threads, GPUs are designed to handle tens of thousands of lightweight threads organized in warps (groups of 32 threads). Efficient GPU programming requires exposing sufficient data parallelism and minimizing control flow divergence and memory access irregularities. This often requires algorithmic reformulation to map the computation onto the Single Instruction Multiple Threads (SIMT) architecture.

CUDA. CUDA (Compute Unified Device Architecture) is NVIDIA's programming model and platform for general-purpose GPU computing. CUDA exposes a C/C++-like language where developers write functions, called *kernels*, that are executed in parallel by many threads on the GPU.

Kernels are launched from the host (CPU) using the triple angle bracket syntax:

```
kernel<<<numBlocks, threadsPerBlock>>>(args
    ...);
```

Each kernel launch defines a grid of thread blocks, and each block contains multiple threads. Threads within the same block can cooperate via shared memory and synchronization primitives, while blocks are independent and scheduled by the GPU runtime.

The classical CUDA execution model is flat and static: kernels are launched from the host, and all parallelism must be defined ahead of time. This model works well for structured applications where the amount of parallel work is known a priori, but becomes cumbersome for irregular or data-dependent computations such as graph traversal or recursive algorithms.

CUDA Dynamic Parallelism. CUDA Dynamic Parallelism (CDP), introduced in CUDA 5.0 for GPUs with Compute Capability 3.5 or higher (Kepler architecture and beyond), extends the traditional execution model by allowing kernels to launch other kernels directly from the device. This enables more flexible and expressive parallel programming, especially for workloads with nested or adaptive parallelism.

With CDP, a parent kernel can dynamically launch child kernels as new work becomes available. These child kernels can themselves launch further kernels, creating a hierarchy of parallel computation. Synchronization between parent and child kernels can be enforced using `cudaDeviceSynchronize()`, though excessive synchronization may reduce performance.

The advantages of CDP include:

- Simplified expression of recursive or irregular algorithms without CPU intervention.
- Better utilization of dynamically discovered parallelism during execution.
- Reduced host-device communication.

However, CDP also comes with important limitations:

- **Kernel launch overhead:** launching kernels from the device is more expensive than from the host.
- **Resource contention:** child kernels share GPU resources with their parents and with each other, which can lead to scheduling delays.
- **Memory visibility:** parent and child kernels cannot share local or shared memory, limiting data reuse and requiring communication through global memory.
- **Limited control:** while CDP enables nesting, it offers less control over launch concurrency and stream management compared to host-side orchestration.

While CDP offers a compelling abstraction for certain classes of problems, it is not a silver bullet. In many cases, flat and iterative kernel designs can achieve better performance, especially when the overhead of nested kernel launches outweighs the benefits of dynamic scheduling. As shown in prior evaluations [1], [2], the effectiveness of CDP is highly application-dependent and must be carefully benchmarked.

III. Benchmark Applications

In this section, we describe in detail the three benchmark applications used to evaluate the performance and practicality of CUDA Dynamic Parallelism (CDP). Each selected application showcases a different class of problems where CDP can either enhance programmability, improve performance, or highlight its overhead compared to flat, host-controlled kernel launches.

The benchmarks span a variety of computational patterns: recursive subdivision in adaptive mesh refinement, hierarchical force computation in N-body simulation, and structured quadtree construction. For each case, we implemented or adapted both a standard CUDA version and a CDP-enabled version to allow direct performance and complexity comparisons.

A. Adaptive Mesh Refinement: Mandelbrot Set

The first benchmark is based on a fractal computation problem: rendering the Mandelbrot set. The version used in this evaluation was adapted from NVIDIA's official CDP tutorial [3]. This application is a canonical example of adaptive mesh refinement, where a region of the domain is recursively subdivided to increase resolution only where needed.

In the standard CUDA implementation the *Escape Time Algorithm* is used, the entire 2D domain is rasterized at a fixed resolution, and each thread independently computes whether its assigned complex coordinate belongs to the Mandelbrot set. This naive approach wastes compute resources in regions that are trivially part of or outside the set.

With CDP the benchmark uses the *Mariani-Silver Algorithm*, the kernel responsible for rendering a region can dynamically launch child kernels to further refine subregions that exhibit boundary complexity. This recursive approach increases resolution only in computationally interesting areas, avoiding unnecessary work. The CDP implementation leverages device-side recursion and kernel launching to build a quadtree-like refinement structure entirely on the GPU. This not only reduces host-device synchronization but also demonstrates the natural expressiveness of CDP in recursive spatial algorithms.

B. N-body Simulation: Direct-Sum vs. Barnes-Hut

The second benchmark explores the classical N-body simulation problem, which models the gravitational interaction among a set of particles. Two algorithms were implemented: the brute-force *Direct-Sum* method and the more efficient *Barnes-Hut* algorithm [4].

Direct-Sum. This approach has $O(N^2)$ computational complexity. Each body computes the net force exerted by every other body. While trivially parallelizable, this method does not scale well with increasing particle count. In the CUDA implementation, one thread is assigned to compute the total force on each body. Shared memory and tiling are employed to maximize memory reuse and reduce global memory accesses.

Barnes-Hut. The Barnes-Hut algorithm reduces complexity to $O(N \log N)$ by hierarchically approximating distant particle groups. The simulation space is recursively partitioned into quadrants (forming a quadtree). Distant regions are approximated by their center of mass, while nearby bodies are treated individually. This requires two key phases:

1) Quadtree construction: recursively inserts each body into its appropriate quadrant.

2) Force computation: traverses the tree to accumulate the total force on each body.

In the CDP version, the quadtree is constructed entirely on the device via dynamic kernel launches. Each node at a given level spawns child kernels for its four quadrants. This top-down recursive strategy maps naturally onto CDP and avoids repeated host-side launches. Moreover, it improves scalability by exploiting fine-grained parallelism at all tree levels.

C. Quadtree Construction: Recursive vs. Iterative

To isolate and directly compare the overhead of CDP, we implemented a standalone quadtree construction algorithm in two variants:

- A CDP-based version originally developed by NVIDIA, which builds the tree recursively using nested device kernel launches.
- An equivalent non-CDP version, developed by transforming the recursive logic into an explicit iterative implementation using an host-side loop.

Both versions execute the same algorithmic steps: partitioning space, assigning elements to quadrants, and processing subregions until a leaf condition is met. However, the CDP version relies on GPU hardware to handle all of this via recursion, while the non-CDP version uses a series of iterative kernel launches to build the quadtree via a level-order scan.

```
build_quadtree_kernel_cdp<<<1,
    NUM_THREADS_PER_BLOCK>>>(nodes, points);
```

Listing 1. Recursive quadtree construction with CDP (pseudocode). The host calls the first kernel, subsequent recursive calls occur directly on the device. Every block in each kernel will call another kernel with 4 blocks, one for each child.

```
for each quadtree level
    build_quadtree_level_kernel<<<
        num_nodes_at_this_level,
        NUM_THREADS_PER_BLOCK>>>(nodes, points
        );
end for
```

Listing 2. Iterative quadtree construction without CDP (pseudocode). The host calls every kernel, one for each level of the quadtree. Each kernel has as many blocks as there are nodes at the current level.

By holding the algorithm constant and only changing the parallelism model, this benchmark offers a clean comparison of performance characteristics. Specifically, it allows us to evaluate whether the CDP approach incurs overhead compared to the iterative one and whether there are situations in which the CDP approach is preferable.

Unlike the Mandelbrot Set and the N-body simulation, where the CDP version uses a more efficient algorithm (Mariani-Silver and Barnes-Hut, respectively) than the flat counterpart (Escape Time and Direct-Sum, respectively), here both versions solve the exact same problem with the same logic. The only difference lies in the parallelism mechanism: recursive CDP vs. iterative. As such, this benchmark is crucial for understanding whether CDP provides practical advantages even when not strictly required for correctness.

IV. Experimental Results

This section reports the performance evaluation of CUDA Dynamic Parallelism (CDP) on three benchmarks introduced in Section III: adaptive mesh refinement for the Mandelbrot set, the N-Body simulation (Direct-Sum vs. Barnes–Hut), and quadtree construction (iterative vs. recursive with CDP). For each benchmark we compare a standard CUDA implementation against a CDP-enabled version, evaluating the impact of device-side kernel launches on both runtime and scalability.

Test platform. All experiments were executed on a laptop-class system featuring an AMD Ryzen 7 6800H CPU (8 cores with SMT, 3.2 GHz base, up to 4.7 GHz boost) and an NVIDIA GeForce RTX 3060 Mobile GPU (3840 CUDA cores, 900 MHz base clock, up to 1425 MHz boost). The operating system was Windows; some experiments required a WSL (Windows Subsystem for Linux) guest due to toolchain incompatibilities. The same GPU and power profile were used across Windows and WSL runs.

Toolchain and compilation. All CUDA codes were compiled with `nvcc` at `-O3`. The `-rdc=true` flag (relocatable device code) was *only* enabled for CDP builds, because device-side kernel launches require device linking. The Windows host used CUDA Compiler version `V12.8.93`, while WSL used `V12.9.41`. Note that enabling `-rdc=true` can inhibit inlining across translation units and marginally increase register pressure, which is a known source of overhead for small kernels.

The Mandelbrot Set and N-Body benchmarks were compiled and executed on WSL, whereas the Quadtree construction was performed on Windows.

Methodology. Each data point is the average of multiple runs after discarding evident outliers. We report absolute runtimes and discuss speedups.

A. Adaptive Mesh Refinement: Mandelbrot Set

We evaluate the NVIDIA Mandelbrot example in a standard (flat) CUDA version and in a CDP version that recursively refines only tiles with complex boundaries, following the approach in [3]. We sweep the image size from 4096^2 up to 32768^2 pixels.

Key results. Measured average runtimes (ms), as shown in Fig. 1, are:

- 4096^2 px (16.8 Mpx): standard 7.47, CDP 30.11 $\Rightarrow$ CDP is 0.25$\times$ the speed of standard (overhead dominates).
- 8192^2 px (67.1 Mpx): standard 23.78, CDP 34.52 $\Rightarrow$ CDP 0.69$\times$ standard.
- 16384^2 px (268.4 Mpx): standard 90.94, CDP 74.08 $\Rightarrow$ CDP 1.23$\times$ standard.
- 32768^2 px (1073.7 Mpx): standard 342.26, CDP 213.56 $\Rightarrow$ CDP 1.60$\times$ standard.

A linear interpolation between the 8192^2 and 16384^2 points places the *crossover* where CDP breaks even around $\sim$ 145 Mpx, i.e., roughly a 12k$\times$12k image. Beyond this

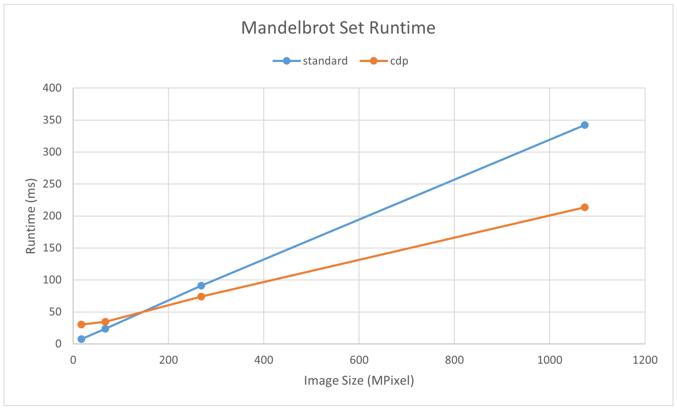

Fig. 1. Mandelbrot image size (Mpx) vs. runtime (ms). (lower is better)

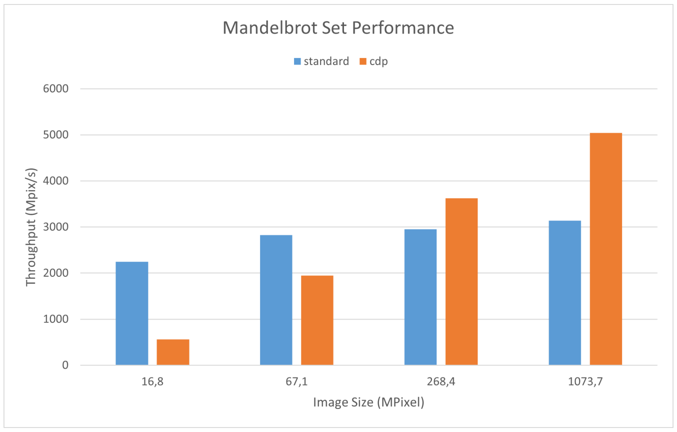

Fig. 2. Mandelbrot image size (Mpx) vs. throughput (Mpx/s). (higher is better)

size, the CDP recursion exposes enough parallel work and saves redundant iterations in homogeneous regions, amortizing launch overheads.

Throughput trend. Expressed as throughput, CDP goes from $\sim$ 563 to $\sim$ 5042 Mpx/s as the image grows, while the standard version goes from $\sim$ 2246 to $\sim$ 3138 Mpx/s; the throughput ratio (CDP vs. standard) increases from $0.25\times$ (small) to $1.61\times$ (largest). This corroborates that CDP benefits scale with problem size and boundary complexity.

Discussion. At small sizes the fixed overheads of device-side launches (and of `-rdc=true`) dominate; the CDP version also performs extra boundary checks that are not offset by refinement savings. For larger images, the recursive subdivision reduces wasted work in uniform tiles, and the avoided host synchronizations/launches become meaningful. Minor run-to-run variation can stem from GPU boost behavior; however, these are not meaningful.

B. N-Body Simulation: Direct-Sum vs. Barnes–Hut

We compare the $O(N^2)$ Direct-Sum (DS) against the $O(N \log N)$ Barnes–Hut (BH) implementation that builds a quadtree on the device using CDP (cf. Section III-B and [4]). We sweep $N \in \{5\times10^3, 5\times10^4, 5\times10^5, 5\times10^6\}$, with Direct-Sum omitted at 5×10^6 due to impractical runtime. Each test has been executed with 10 iterations of the simulation, using the Spiral Galaxy model.

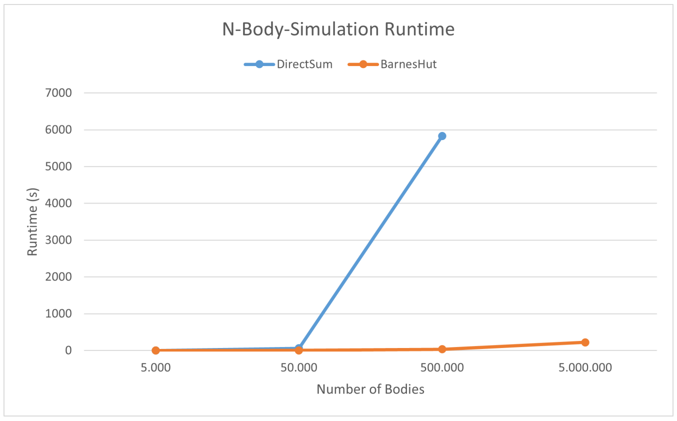

Fig. 3. N-Body N vs. runtime (s) for DS and BH (CDP). (lower is better)

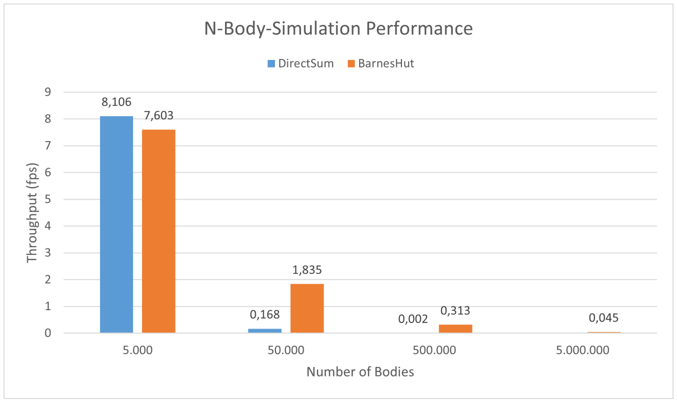

Fig. 4. Performance in fps (frames per second) of BH (CDP) over DS; BH underperforms only at very small N. (higher is better)

Key results. Average runtimes (s), as shown in Fig. 3, are:

- $N = 5{,}000$: DS 1.234, BH (CDP) 1.315 $\Rightarrow$ BH $0.94\times$ DS (tree-build overhead dominates).
- $N = 50{,}000$: DS 59.63, BH 5.45 $\Rightarrow$ BH $10.94\times$ faster.
- $N = 500{,}000$: DS 5827.11, BH 32.00 $\Rightarrow$ BH $182\times$ faster.
- $N = 5{,}000{,}000$: BH 223.57 (DS not measured for convenience reasons, the test would have taken too long).

Discussion. The BH advantage emerges once N is large enough that the cost of building/traversing the tree is outweighed by approximating distant clusters. CDP is instrumental to a clean and efficient device-side recursive build. For very small N, the BH overheads (tree construction, irregular memory accesses, `-rdc=true`) slightly outweigh the benefits, explaining the $N = 5000$ reversal. For large N, the asymptotic improvement dominates and yields the expected orders-of-magnitude speedup.

C. Quadtree Construction: Iterative vs. Recursive (CDP)

To isolate the overhead of CDP *without* changing algorithmic complexity, we implemented the same quadtree builder

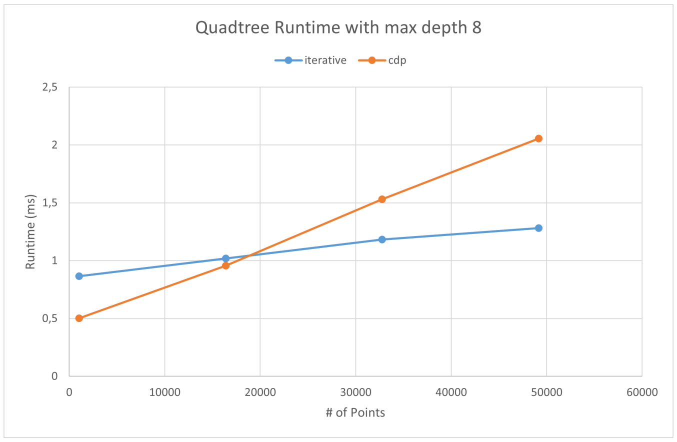

Fig. 5. Quadtree construction number of points vs. runtime (ms) with a fixed depth. (lower is better)

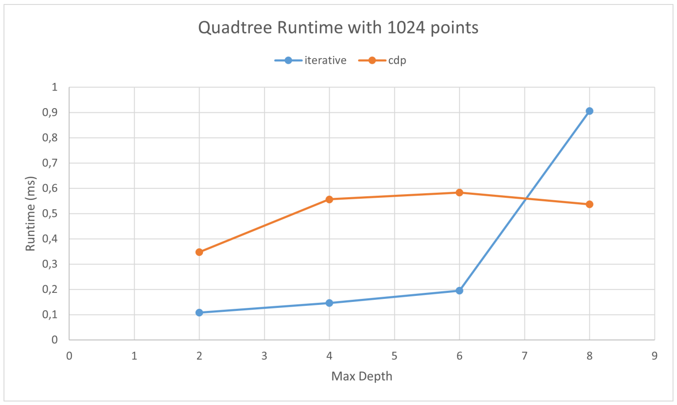

Fig. 6. Quadtree construction max depth vs. runtime (ms) with a fixed number of points. (lower is better)

in two forms: (i) an iterative series of host-side CUDA kernel launches (without CDP) and (ii) a recursive CDP version with device-side launches (cf. Section III-C). We vary both the number of input points and the maximum tree depth.

Scaling with number of points. For $N \in \{1024, 16384, 32768, 49152\}$ points with a fixed maximum depth of 8, the average runtimes (ms), as shown in Fig. 5, are:

- $N = 1024$: iterative 0.866, CDP $0.501 \Rightarrow$ CDP $1.73\times$ faster.
- $N = 16384$: iterative 1.018, CDP $0.957 \Rightarrow$ CDP $1.06\times$ faster.
- $N = 32768$: iterative 1.184, CDP $1.531 \Rightarrow$ CDP $0.77\times$ iterative.
- $N = 49152$: iterative 1.282, CDP $2.055 \Rightarrow$ CDP $0.62\times$ iterative.

A linear interpolation suggests a crossover near $\sim$ 18800 points: below this, CDP wins; above it, the iterative variant is faster.

Effect of maximum depth. With fixed input distribution of 1024 points and varying maximum depth $d_{\max} \in \{2, 4, 6, 8\}$, as shown in Fig. 6:

- $d_{\max} = 2$: iterative 0.109, CDP $0.348 \Rightarrow$ CDP $0.31\times$ iterative.
- $d_{\max} = 4$: iterative 0.147, CDP $0.556 \Rightarrow$ CDP $0.26\times$ iterative.
- $d_{\max} = 6$: iterative 0.195, CDP $0.583 \Rightarrow$ CDP $0.34\times$ iterative.
- $d_{\max} = 8$: iterative 0.906, CDP $0.537 \Rightarrow$ CDP $1.69\times$ iterative.

Interpolating between $d_{\max} = 6$ and 8 places the break-even at approximately $d_{\max} \approx 7$.

Discussion. This benchmark keeps the algorithmic logic constant and reveals pure orchestration costs: for shallow trees or many points per node, the CDP version pays significant fixed costs (nested launch latency, device runtime bookkeeping, `-rdc` effects). As the tree becomes deeper and the work per node shrinks, recursive sub-kernels expose additional parallelism and better resource utilization, reversing the outcome.

The main advantage of the CDP version is that, if necessary, it can terminate the computation earlier than the iterative version. When the CDP encounters a leaf, it does not call another sub-kernel, while the iterative version must scan all levels of the tree. This coincides with cases where the CDP version is better, i.e. when the tree is less prone to saturation. The only possible improvement for the iterative version is to stop the computation earlier only if there are only leaves at the nth level; otherwise, it must continue until this condition is met or the maximum depth is reached.

V. Conclusions

In this work we evaluated the performance of CUDA Dynamic Parallelism (CDP) across three different benchmarks: Mandelbrot set computation with adaptive mesh refinement, the N-Body simulation (Direct-Sum vs. Barnes–Hut), and quadtree construction with recursive and iterative implementations. The chosen applications allowed us to analyze both scenarios where CDP enables algorithmic improvements (Mandelbrot with Mariani–Silver, N-Body with Barnes–Hut) and cases where the algorithmic logic remains unchanged (quadtree), thereby isolating the effect of CDP on orchestration overhead.

The results confirm that CDP is not a universal performance booster but a tool that must be applied selectively:

- In the Mandelbrot benchmark, CDP only becomes advantageous for very large images ($\gtrsim$ 12k $\times$ 12k), where adaptive refinement amortizes kernel launch overheads. For smaller inputs, the standard flat implementation is preferable.
- In the N-Body simulation, the Barnes–Hut algorithm implemented with CDP provides orders-of-magnitude speedups compared to the Direct-Sum method. Here, CDP is essential to enable a scalable recursive implementation in a convenient way.

- In the quadtree benchmark, where both CDP and iterative versions implement the same algorithm, CDP reveals its intrinsic overhead. The iterative version outperforms CDP in shallow trees or with many points per node, while CDP shows benefits only at greater recursion depths, when the ability to stop computation early reduces redundant work.

From these findings we conclude that CDP is most effective when algorithmic structure naturally maps to recursive parallelism, particularly in problems with adaptive or irregular workloads. Conversely, when the problem can be expressed iteratively with predictable parallelism, the additional overhead of device-side launches can outweigh potential benefits.

We believe that the right approach is to consider CDP as a *specialized tool*: indispensable for certain hierarchical algorithms, but unnecessary or even detrimental for regular computations. Although our study focused on three representative benchmarks, the insights can be generalized to other applications in scientific computing and data structures. Future work could expand this evaluation to more complex domains such as graph processing.

REFERENCES

[1] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 51–60.

[2] Ł. Jarząbek and P. Czarnul, "Performance evaluation of unified memory and dynamic parallelism for selected parallel cuda applications," *The Journal of Supercomputing*, vol. 73, pp. 5378–5401, 2017.

[3] NVIDIA Developer Blog. (2014) Introduction to cuda dynamic parallelism. Accessed: 2025-07-02. [Online]. Available: https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

[4] H. Wu. (2024) Optimizing n-body simulation with barnes-hut algorithm and cuda. Accessed: 2025-07-02. [Online]. Available: https://medium.com/@hsinhungw/optimizing-n-body-simulation-with-barnes-hut-algorithm-and-cuda-c76e78228c28