""" DARKBOT™ Benchmark Script

This script runs comprehensive benchmarks on the DARKBOT™ system to measure performance across various operations and dimensions.

import torch import numpy as np import time import json import argparse import os import pandas as pd from pathlib import Path from datetime import datetime import matplotlib.pyplot as plt from darkbot import DarkBot, DarkBotConfig from darkbot.lattice import construct_e8_lattice from darkbot.tensor_ops import (calculate_resonance, resonant_product, hierarchical_resonant_product)

def run_resonance_benchmark(iterations, dimensions, batch_sizes, device): """Run benchmarks for resonance calculation""" results = []

```
for dim in dimensions:
    dim_results = {}

    # Generate test fields
    field1 = torch.randn(dim, dtype=torch.complex64, device=device)
    field2 = torch.randn(dim, dtype=torch.complex64, device=device)

    # Standard resonance
    start_time = time.time()
    for _ in range(iterations):
        _ = calculate_resonance(field1, field2, device=device)
        if device.type == "cuda":
            torch.cuda.synchronize()
    end_time = time.time()
    dim_results["standard"] = ((end_time - start_time) / iterations) * 1000  # ms

    # Classical calculation (dot product)
    start_time = time.time()
    for _ in range(iterations):
        _ = torch.abs(torch.sum(field1 * torch.conj(field2))) / (
            torch.sqrt(torch.sum(torch.abs(field1)**2)) *
            torch.sqrt(torch.sum(torch.abs(field2)**2))
        )
        if device.type == "cuda":
            torch.cuda.synchronize()
    end_time = time.time()
    dim_results["classical"] = ((end_time - start_time) / iterations) * 1000  # ms

    # Batch resonance (various batch sizes)
    batch_results = {}
    for batch_size in batch_sizes:
        # Create batch tensors
        batch1 = torch.stack([field1] * batch_size)
        batch2 = torch.stack([field2] * batch_size)

        start_time = time.time()
        for _ in range(iterations):
            _ = torch.zeros((batch_size, batch_size), device=device)
            for i in range(batch_size):
                for j in range(batch_size):
                    _ = calculate_resonance(batch1[i], batch2[j], device=device)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        individual_time = ((end_time - start_time) / iterations) * 1000  # ms
```

```
        # Vectorized batch
        from darkbot.tensor_ops import calculate_batch_resonance
        start_time = time.time()
        for _ in range(iterations):
            _ = calculate_batch_resonance(batch1, batch2, device=device)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        vectorized_time = ((end_time - start_time) / iterations) * 1000  # ms

        batch_results[batch_size] = {
            "individual": individual_time,
            "vectorized": vectorized_time,
            "speedup": individual_time / vectorized_time
        }

    dim_results["batch"] = batch_results
    results.append({
        "dimensions": dim,
        "results": dim_results
    })

return results
```

def run_one_draw_benchmark(iterations, dimensions, pattern_counts, device): """Run benchmarks for One Draw search""" results = []

```python
# Initialize DarkBot with different dimensions
for dim in dimensions:
    config = DarkBotConfig()
    config.field.dimensions = dim
    darkbot = DarkBot(config)

    dim_results = {}

    for pattern_count in pattern_counts:
        # Create pattern library
        patterns = [
            torch.randn(dim, dtype=torch.complex64, device=device)
            for _ in range(pattern_count)
        ]

        # Create query
        query = torch.randn(dim, dtype=torch.complex64, device=device)

        # One Draw search
        start_time = time.time()
        for _ in range(iterations):
            _ = darkbot.one_draw_search(query, patterns)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        one_draw_time = ((end_time - start_time) / iterations) * 1000  # ms

        # Classical search (O(N))
        start_time = time.time()
        for _ in range(iterations):
            distances = [torch.norm(query - p) for p in patterns]
            _ = np.argmin(distances)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        classical_time = ((end_time - start_time) / iterations) * 1000  # ms

        # Binary search simulation (O(log N))
        # This is a simplified simulation of binary search
        start_time = time.time()
        for _ in range(iterations):
            # Calculate first distance
            mid = pattern_count // 2
            dist = torch.norm(query - patterns[mid])

            # Simulate binary search steps
```

```
            steps = int(np.log2(pattern_count))
            for _ in range(steps):
                # Just do some computation equivalent to binary search steps
                dist += torch.norm(query - patterns[np.random.randint(0, pattern_count)])

            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        binary_time = ((end_time - start_time) / iterations) * 1000  # ms

        # Grover simulation (O(√N))
        # This is a simplified simulation of Grover search
        start_time = time.time()
        for _ in range(iterations):
            # Simulate sqrt(N) steps
            steps = int(np.sqrt(pattern_count))
            dist = 0
            for _ in range(steps):
                idx = np.random.randint(0, pattern_count)
                dist += torch.norm(query - patterns[idx])

            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        grover_time = ((end_time - start_time) / iterations) * 1000  # ms

        dim_results[pattern_count] = {
            "one_draw": one_draw_time,
            "classical": classical_time,
            "binary": binary_time,
            "grover": grover_time,
            "speedup_vs_classical": classical_time / one_draw_time,
            "speedup_vs_binary": binary_time / one_draw_time,
            "speedup_vs_grover": grover_time / one_draw_time
        }

    results.append({
        "dimensions": dim,
        "results": dim_results
    })

return results
```

def run_e8_benchmark(iterations, device): """Run benchmarks for E8 lattice construction""" results = {}

```
# Standard implementation
start_time = time.time()
for _ in range(iterations):
    _ = construct_e8_lattice(device=device)
    if device.type == "cuda":
        torch.cuda.synchronize()
end_time = time.time()
results["standard"] = ((end_time - start_time) / iterations) * 1000  # ms

# With pre-computation and caching
cached_e8 = construct_e8_lattice(device=device)

start_time = time.time()
for _ in range(iterations):
    # Simulate retrieval from cache
    _ = cached_e8.clone()
    if device.type == "cuda":
        torch.cuda.synchronize()
end_time = time.time()
results["cached"] = ((end_time - start_time) / iterations) * 1000  # ms

return results
```

def run_resonant_product_benchmark(iterations, dimensions, device): """Run benchmarks for resonant product operations""" results = []

```
for dim in dimensions:
    dim_results = {}

    # Generate test fields
    field1 = torch.randn(dim, dtype=torch.complex64, device=device)
    field2 = torch.randn(dim, dtype=torch.complex64, device=device)

    # Standard resonant product
    start_time = time.time()
    for _ in range(iterations):
        _ = resonant_product(field1, field2, device=device)
        if device.type == "cuda":
            torch.cuda.synchronize()
    end_time = time.time()
    dim_results["standard"] = ((end_time - start_time) / iterations) * 1000  # ms

    # Only for larger dimensions, test hierarchical approach
    if dim >= 1024:
        # Hierarchical resonant product
        start_time = time.time()
        for _ in range(iterations):
            _ = hierarchical_resonant_product(field1, field2, device=device)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        dim_results["hierarchical"] = ((end_time - start_time) / iterations) * 1000  # ms
        dim_results["speedup"] = dim_results["standard"] / dim_results["hierarchical"]

    results.append({
        "dimensions": dim,
        "results": dim_results
    })

return results
```

def run_full_processing_benchmark(iterations, dimensions, batch_sizes, device): """Run benchmarks for the full processing cycle""" results = []

```
for dim in dimensions:
    config = DarkBotConfig()
    config.field.dimensions = dim
    darkbot = DarkBot(config)

    dim_results = {}

    # Single item processing
    input_data = torch.randn(dim, dtype=torch.complex64, device=device)

    start_time = time.time()
    for _ in range(iterations):
        _ = darkbot._process_quantum_core(input_data)
        if device.type == "cuda":
            torch.cuda.synchronize()
    end_time = time.time()
    dim_results["single"] = ((end_time - start_time) / iterations) * 1000  # ms

    # Batch processing
    batch_results = {}
    for batch_size in batch_sizes:
        # Skip large batch sizes for large dimensions to avoid OOM
        if dim * batch_size > 2**20:  # Arbitrary limit to avoid OOM
            continue

        batch_input = torch.randn(batch_size, dim, dtype=torch.complex64, device=device)

        start_time = time.time()
        for _ in range(max(1, iterations // 5)):  # Fewer iterations for batch as it's
slower
            _ = darkbot._process_batch_core(batch_input)
            if device.type == "cuda":
                torch.cuda.synchronize()
        end_time = time.time()
        batch_time = ((end_time - start_time) / max(1, iterations // 5)) * 1000  # ms

        # Calculate scaling efficiency
        single_time_total = dim_results["single"] * batch_size
        scaling_efficiency = (single_time_total / batch_time) * 100

        batch_results[batch_size] = {
            "time_ms": batch_time,
            "per_item_ms": batch_time / batch_size,
            "scaling_efficiency": scaling_efficiency
        }
```

```
        dim_results["batch"] = batch_results
        results.append({
            "dimensions": dim,
            "results": dim_results
        })

    return results
```

def create_visualizations(benchmark_results, output_dir): """Create visualizations from benchmark results""" os.makedirs(output_dir, exist_ok=True)

```
# Resonance benchmark visualization
if "resonance" in benchmark_results:
    resonance_results = benchmark_results["resonance"]

    # Extract data
    dimensions = [r["dimensions"] for r in resonance_results]
    standard_times = [r["results"]["standard"] for r in resonance_results]
    classical_times = [r["results"]["classical"] for r in resonance_results]

    # Create plot
    plt.figure(figsize=(10, 6))
    plt.plot(dimensions, standard_times, 'o-', label='DARKBOT™ Resonance')
    plt.plot(dimensions, classical_times, 's-', label='Classical Calculation')
    plt.title('Resonance Calculation Performance')
    plt.xlabel('Dimensions')
    plt.ylabel('Time (ms)')
    plt.xscale('log2')
    plt.yscale('log2')
    plt.grid(True, which='both', linestyle='--', alpha=0.7)
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, 'resonance_performance.png'))
    plt.close()

    # Batch speedup visualization
    if len(resonance_results) > 0 and "batch" in resonance_results[0]["results"]:
        # Use the largest dimension for batch comparison
        largest_dim = max(dimensions)
        largest_idx = dimensions.index(largest_dim)
        batch_results = resonance_results[largest_idx]["results"]["batch"]

        batch_sizes = list(batch_results.keys())
        speedups = [batch_results[bs]["speedup"] for bs in batch_sizes]

        plt.figure(figsize=(10, 6))
        plt.bar(range(len(batch_sizes)), speedups)
        plt.xticks(range(len(batch_sizes)), batch_sizes)
        plt.title(f'Batch Resonance Speedup (Dimensions={largest_dim})')
        plt.xlabel('Batch Size')
        plt.ylabel('Speedup Factor (Individual / Vectorized)')
        plt.grid(True, which='both', linestyle='--', alpha=0.7, axis='y')
        plt.tight_layout()
        plt.savefig(os.path.join(output_dir, 'batch_resonance_speedup.png'))
        plt.close()

# One Draw benchmark visualization
```

```
if "one_draw" in benchmark_results:
    one_draw_results = benchmark_results["one_draw"]

    # Use a middle dimension for comparison
    if len(one_draw_results) > 0:
        mid_idx = len(one_draw_results) // 2
        mid_dim_results = one_draw_results[mid_idx]
        dim = mid_dim_results["dimensions"]
        results = mid_dim_results["results"]

        pattern_counts = sorted(list(map(int, results.keys())))
        one_draw_times = [results[pc]["one_draw"] for pc in pattern_counts]
        classical_times = [results[pc]["classical"] for pc in pattern_counts]
        binary_times = [results[pc]["binary"] for pc in pattern_counts]
        grover_times = [results[pc]["grover"] for pc in pattern_counts]

        plt.figure(figsize=(12, 7))
        plt.plot(pattern_counts, one_draw_times, 'o-', label='DARKBOT™ One Draw')
        plt.plot(pattern_counts, classical_times, 's-', label='Classical O(N)')
        plt.plot(pattern_counts, binary_times, '^-', label='Binary O(log N)')
        plt.plot(pattern_counts, grover_times, 'd-', label='Grover O(√N)')
        plt.title(f'Search Algorithm Performance (Dimensions={dim})')
        plt.xlabel('Pattern Count')
        plt.ylabel('Time (ms)')
        plt.xscale('log2')
        plt.yscale('log2')
        plt.grid(True, which='both', linestyle='--', alpha=0.7)
        plt.legend()
        plt.tight_layout()
        plt.savefig(os.path.join(output_dir, 'search_performance.png'))
        plt.close()

# Full processing benchmark visualization
if "full_processing" in benchmark_results:
    processing_results = benchmark_results["full_processing"]

    # Extract data
    dimensions = [r["dimensions"] for r in processing_results]
    single_times = [r["results"]["single"] for r in processing_results]

    # Create plot
    plt.figure(figsize=(10, 6))
    plt.plot(dimensions, single_times, 'o-', label='Processing Time')
    plt.title('DARKBOT™ Processing Performance')
    plt.xlabel('Dimensions')
    plt.ylabel('Time (ms)')
    plt.xscale('log2')
```

```
    plt.yscale('log2')
    plt.grid(True, which='both', linestyle='--', alpha=0.7)
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, 'processing_performance.png'))
    plt.close()

    # Batch scaling efficiency
    if len(processing_results) > 0 and "batch" in processing_results[0]["results"]:
        # Use a middle dimension for batch comparison
        mid_idx = len(processing_results) // 2
        mid_dim_results = processing_results[mid_idx]
        dim = mid_dim_results["dimensions"]
        batch_results = mid_dim_results["results"]["batch"]

        if batch_results:  # Check if not empty
            batch_sizes = sorted(list(map(int, batch_results.keys())))
            efficiencies = [batch_results[bs]["scaling_efficiency"] for bs in batch_sizes]

            plt.figure(figsize=(10, 6))
            plt.bar(range(len(batch_sizes)), efficiencies)
            plt.xticks(range(len(batch_sizes)), batch_sizes)
            plt.title(f'Batch Processing Scaling Efficiency (Dimensions={dim})')
            plt.xlabel('Batch Size')
            plt.ylabel('Scaling Efficiency (%)')
            plt.ylim([0, 110])  # Leave room for 100% mark
            plt.axhline(y=100, color='r', linestyle='--', label='Perfect Scaling')
            plt.grid(True, which='both', linestyle='--', alpha=0.7, axis='y')
            plt.legend()
            plt.tight_layout()
            plt.savefig(os.path.join(output_dir, 'batch_scaling_efficiency.png'))
            plt.close()
```

def save_results(benchmark_results, output_dir): """Save benchmark results to JSON and CSV files""" os.makedirs(output_dir, exist_ok=True) timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

```python
# Save full results as JSON
json_path = os.path.join(output_dir, f"darkbot_benchmark_{timestamp}.json")
with open(json_path, 'w') as f:
    json.dump(benchmark_results, f, indent=2, default=str)

# Create summary CSV for resonance benchmarks
if "resonance" in benchmark_results:
    resonance_results = benchmark_results["resonance"]
    resonance_df = pd.DataFrame([
        {
            "Dimensions": r["dimensions"],
            "DARKBOT™ Resonance (ms)": r["results"]["standard"],
            "Classical Calculation (ms)": r["results"]["classical"],
            "Speedup": r["results"]["classical"] / r["results"]["standard"]
        }
        for r in resonance_results
    ])
    resonance_csv_path = os.path.join(output_dir, f"resonance_benchmark_{timestamp}.csv")
    resonance_df.to_csv(resonance_csv_path, index=False)

# Create summary CSV for One Draw benchmarks
if "one_draw" in benchmark_results:
    one_draw_dfs = []
    for result in benchmark_results["one_draw"]:
        dim = result["dimensions"]
        dim_df = pd.DataFrame([
            {
                "Dimensions": dim,
                "Pattern Count": int(pc),
                "One Draw (ms)": data["one_draw"],
                "Classical O(N) (ms)": data["classical"],
                "Binary O(log N) (ms)": data["binary"],
                "Grover O(√N) (ms)": data["grover"],
                "Speedup vs Classical": data["speedup_vs_classical"],
                "Speedup vs Binary": data["speedup_vs_binary"],
                "Speedup vs Grover": data["speedup_vs_grover"]
            }
            for pc, data in result["results"].items()
        ])
        one_draw_dfs.append(dim_df)

    if one_draw_dfs:
        one_draw_df = pd.concat(one_draw_dfs)
        one_draw_csv_path = os.path.join(output_dir, f"one_draw_benchmark_{timestamp}.csv")
        one_draw_df.to_csv(one_draw_csv_path, index=False)
```

```
    # Create summary CSV for full processing benchmarks
    if "full_processing" in benchmark_results:
        processing_results = benchmark_results["full_processing"]
        processing_df = pd.DataFrame([
            {
                "Dimensions": r["dimensions"],
                "Processing Time (ms)": r["results"]["single"]
            }
            for r in processing_results
        ])
        processing_csv_path = os.path.join(output_dir, f"processing_benchmark_{timestamp}.csv")
        processing_df.to_csv(processing_csv_path, index=False)

    print(f"Benchmark results saved to {output_dir}")
    return json_path
```

def main(): """Main benchmark function""" parser = argparse.ArgumentParser(description='Run DARKBOT™ benchmarks') parser.add_argument('--iterations', type=int, default=100, help='Number of iterations for each test') parser.add_argument('--dimensions', type=str, default='64,128,256,512,1024,2048', help='Comma-separated list of dimensions to test') parser.add_argument('--batch-sizes', type=str, default='2,4,8,16,32', help='Comma-separated list of batch sizes to test') parser.add_argument('--pattern-counts', type=str, default='16,32,64,128,256,512,1024,2048', help='Comma-separated list of pattern counts to test for search') parser.add_argument('--output-dir', type=str, default='benchmark_results', help='Directory to save benchmark results') parser.add_argument('--with-gpu', action='store_true', help='Force GPU usage if available') parser.add_argument('--ci-mode', action='store_true', help='Run in CI mode (reduced benchmarks)') args = parser.parse_args()

```python
# Parse dimensions and batch sizes
dimensions = list(map(int, args.dimensions.split(',')))
batch_sizes = list(map(int, args.batch_sizes.split(',')))
pattern_counts = list(map(int, args.pattern_counts.split(',')))

# If in CI mode, reduce test sizes
if args.ci_mode:
    print("Running in CI mode with reduced benchmarks")
    dimensions = dimensions[:2]  # Just the first two dimensions
    batch_sizes = batch_sizes[:2]  # Just the first two batch sizes
    pattern_counts = pattern_counts[:3]  # Just the first three pattern counts
    args.iterations = min(args.iterations, 10)  # Limit iterations

# Set device
if args.with_gpu and torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Running DARKBOT™ benchmarks on {device}")
print(f"Iterations: {args.iterations}")
print(f"Dimensions: {dimensions}")
print(f"Batch sizes: {batch_sizes}")
print(f"Pattern counts: {pattern_counts}")

# Initialize benchmark results dictionary
benchmark_results = {
    "metadata": {
        "timestamp": datetime.now().isoformat(),
        "device": str(device),
        "cuda_available": torch.cuda.is_available(),
        "iterations": args.iterations,
        "dimensions": dimensions,
        "batch_sizes": batch_sizes,
        "pattern_counts": pattern_counts
    }
}

# Run resonance benchmarks
print("\nRunning resonance benchmarks...")
benchmark_results["resonance"] = run_resonance_benchmark(
    args.iterations, dimensions, batch_sizes, device
)

# Run One Draw benchmarks
print("Running One Draw search benchmarks...")
```

```
    benchmark_results["one_draw"] = run_one_draw_benchmark(
        args.iterations, dimensions[:2], pattern_counts, device
    )

    # Run E8 lattice benchmarks
    print("Running E8 lattice benchmarks...")
    benchmark_results["e8_lattice"] = run_e8_benchmark(
        args.iterations, device
    )

    # Run resonant product benchmarks
    print("Running resonant product benchmarks...")
    benchmark_results["resonant_product"] = run_resonant_product_benchmark(
        args.iterations, dimensions, device
    )

    # Run full processing benchmarks
    print("Running full processing benchmarks...")
    benchmark_results["full_processing"] = run_full_processing_benchmark(
        max(1, args.iterations // 5),  # Fewer iterations for full processing as it's slower
        dimensions,
        batch_sizes,
        device
    )

    # Save results
    output_dir = args.output_dir
    os.makedirs(output_dir, exist_ok=True)

    # Create subdirectory with timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    result_dir = os.path.join(output_dir, f"benchmark_{timestamp}")
    os.makedirs(result_dir, exist_ok=True)

    # Save results and create visualizations
    print("\nSaving results and creating visualizations...")
    json_path = save_results(benchmark_results, result_dir)
    create_visualizations(benchmark_results, result_dir)

    print(f"\nBenchmark complete. Results saved to {json_path}")
    print(f"Visualizations saved to {result_dir}")
```

if **name** == "**main**": main()