

Policy Iteration on the GPU

Mustafa Can Aydin

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

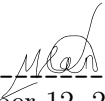
Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the program of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE  _____

DATE September 12, 2025

Acknowledgements

This work made use of the Barkla High Performance Computing facilities at the University of Liverpool^[28]. In addition, several figures in this dissertation were generated using the `TikZ` library^[27].

Policy Iteration on the GPU

Abstract

Markov Decision Processes (MDPs) are the mathematical backbone of sequential decision-making under uncertainty, with policy iteration among the fastest exact algorithms for solving them. However, as real-world state spaces grow to millions of states, traditional CPU implementations become prohibitively slow. This project aims to address that limitation by designing and implementing a parallelized version of policy iteration optimized for modern NVIDIA-class GPUs using CUDA. A single-threaded C++ reference implementation will serve as a performance baseline, while the GPU variant will utilize optimized linear algebra and parallel reduction strategies to accelerate both policy evaluation and improvement stages. Experimental evaluations will be conducted to quantify performance gains across a broad range of problem instances, allowing flexibility in the choice of environments and data representations. We will measure numerical correctness (Bellman residuals $< 10^{-5}$) and benchmark wall-clock performance, targeting 10–100× acceleration over the CPU baseline. The project also explores the sensitivity of speed-ups to discount factors and sparsity. Deliverables include fully documented source code, a reproducible command-line tool, a comprehensive evaluation report, and a dissertation reflecting on performance trade-offs. This work aims to provide an GPU policy iteration framework valuable to the reinforcement learning and operations research communities.

0.1 Statement of ethical compliance: A0

Data Category: A

Participant Category: 0

I confirm that I have read the ethical guidelines and will follow them throughout this project.

I will not use any human participants or personal data at any stage of the project. The project relies solely on synthetic benchmark data for testing and evaluation purposes:

- Data Source: Synthetic datasets.
- Evaluation: Performed using non-human data only.

Contents

0.1 Statement of ethical compliance: A0	ii
1 Introduction	1
1.1 Scope	1
1.2 Problem Statement	1
1.3 Approach	1
1.4 Expected Outcomes	2
2 Background	3
2.1 Theoretical Foundations	3
2.1.1 Markov Decision Processes	3
2.1.2 Solving Markov Decision Processes With Policy Iteration	4
2.1.3 Parallel Computing and GPU Programming	5
2.2 Cuda Programming Model	7
2.3 State of the Art: GPU-Accelerated Policy Iteration for MDPs	9
2.3.1 Introduction	9
2.3.2 GPU vs. CPU Performance for Value/Policy Iteration	9
2.3.3 Influence of MDP Structure on GPU Efficiency	9
2.3.4 Scaling Out: Multi-GPU and Distributed Solutions	10
2.3.5 Other Solution Methods (Beyond Policy Iteration)	11
2.3.6 Benchmarking Practices and Reporting Norms	12
2.3.7 Conclusion	12
3 Design	14
3.1 Solving Policy Iteration via GPU Acceleration	14
3.2 Analyzing Policy Iteration on the GPU	15
3.3 Parallelization Strategy	15
3.4 Why CSR format is chosen?	15
3.4.1 What is CSR format?	16
3.4.2 What are the advantages of CSR format?	17
3.5 Rationale for Design Choices	17
3.5.1 Why sparse grid worlds instead of dense ones?	17
3.5.2 Why Jacobi Sweeps instead of Gauss-Seidel?	17
3.5.3 Why both cuSPARSE and custom kernels implemented and compared?	18
3.5.4 Trade-offs: load balancing, memory traffic, and scalability	18
3.6 Summary	18
4 Implementation	19
4.1 Technology Stack	19
4.2 Implementation through Plain Cuda	19
4.2.1 Policy evaluation	20
4.2.2 Policy improvement	20

4.3	Implementation through cuSparse	21
4.3.1	Policy evaluation (projected SpMV)	21
4.3.2	Policy improvement	21
4.3.3	Convergence check and housekeeping	21
5	Evaluation	23
5.1	Hardware Used	23
5.2	Data Used in Evaluation	23
5.2.1	Correctness Test Configurations	23
5.2.2	Performance Test Configurations	24
5.3	Performance Results	25
5.3.1	Interpretation of speed-ups using cuSparse vs plain cuda	26
5.4	Performance Comparison Between Different GPUs	27
5.5	Performance Gains Through cuSparse	28
5.6	Performance Trade-offs	29
5.7	Limitations	30
5.8	Future Improvements	30
5.8.1	Multi GPU Support	30
5.8.2	Comparison with parallel CPU implementations	31
5.8.3	Testing on more diverse and larger datasets	31
5.8.4	Testing with different types of GPUs	31
6	BCS Criteria and Self-Reflection	32
6.1	BCS Project Criteria	32
6.1.1	Application of Practical and Analytical Skills	32
6.1.2	Innovation and Creativity	32
6.1.3	Synthesis and Evaluation	32
6.2	Self Reflection	32
A	Implementation Details	36
A.1	CSR Input Format	36
A.2	CSR vs Dense Storage Comparison	36
B	GPU Architecture Notes	38
B.1	Write-After-Read Hazards	38
C	Experimental Setup	39
C.1	Barkla Slurm Scripts	39
D	Results and Validation	41
D.1	Example Outputs	41
D.1.1	Data Output	41
D.1.2	Performance Output	41
D.2	Correctness Check Script for CUDA and Serial Outputs	41

List of Abbreviations

CSR	Compressed Sparse Row
CLI	Command Line Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
MDP	Markov Decision Process
PI	Policy Iteration
VI	Value Iteration
SpMV	Sparse Matrix–Vector multiplication
SPMV-CSR	Sparse Matrix–Vector multiplication using Compressed Sparse Row format
LP	Linear Programming
DRAM	Dynamic Random-Access Memory
NVLink	NVIDIA Link (high-speed GPU interconnect)
NVSwitch	NVIDIA Switch (scalable GPU interconnect)
NCCL	NVIDIA Collective Communications Library
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
BFS	Breadth-First Search
MCTS	Monte Carlo Tree Search
AO*	A-star (A*) Search
LAO*	Labeled A-star (LAO*) Search
DQN	Deep Q-Network
AMD	Advanced Micro Devices
HBM	High Bandwidth Memory
GDDR	Graphics Double Data Rate
HBM2	High Bandwidth Memory 2
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
DP	Dynamic Programming
MB	Megabyte
GB	Gigabyte
TD	Temporal Difference
cuSPARSE	CUDA Sparse Matrix Library
BLAS	Basic Linear Algebra Subprograms
ML	Machine Learning
Intel MKL	Intel Math Kernel Library
PETSc	Portable, Extensible Toolkit for Scientific Computation
SM	Streaming Multiprocessor

FP32	32-bit Floating Point
FP64	64-bit Floating Point
GFLOPS	Giga Floating Point Operations Per Second
FLOPS	Floating Point Operations Per Second
L1, L2, L3 Cache	Levels 1, 2, and 3 CPU Cache
SMEM	Shared Memory (on GPU)
nnz	Number of Non-Zeros (in a matrix)
RTX	Ray Tracing Texel eXtreme (NVIDIA GPU series)
Tesla T4	NVIDIA Tesla T4 GPU
Tesla V100	NVIDIA Tesla V100 GPU
NVIDIA L40S	NVIDIA L40S GPU
NVIDIA L4	NVIDIA L4 GPU
NVIDIA H100	NVIDIA H100 GPU
AWS	Amazon Web Services
HYB Format	Hybrid format (it combines two sparse matrix storage formats)
3D	Three-Dimensional
axpy	the operation $y \leftarrow a * x + y$
TPU	Tensor Processing Unit
comm-bound	Communication-bound
UCT	Upper Confidence Bound for Trees

List of Figures

2.1	Markov Decision Process (MDP) illustrating states, actions, transitions, and rewards.	3
2.2	Policy Iteration Algorithm Flowchart	4
2.3	CUDA hierarchy: Grid \rightarrow Blocks \rightarrow Threads. Each block here shows a tile of threads sharing block-local resources (shared memory, sync) [20].	6
2.4	CUDA memory model: hierarchical memory spaces with different visibility and latency trade-offs.	6
2.5	CUDA device memory types, their scope, and typical latency characteristics.	8
3.1	Dense representations of the transition and reward matrices for a small MDP with $S = 3$ states and $A = 2$ actions. Each row $r = s \cdot A + a$ corresponds to a state-action pair.	17
4.1	Flow of the overall architecture showing the interaction between CPU and GPU during Policy Iteration.	19
4.2	GPU Parallelization Strategy for Policy Evaluation	20
4.3	GPU Parallelization Strategy for Policy Improvement	20
4.4	GPU Parallelization Strategy for Policy Iteration with cuSPARSE. The construction of projected CSR P^π and reward r^π (A) is followed by Jacobi policy evaluation sweeps (B) using cuSPARSE SpMV, then policy improvement (C), and a convergence check with housekeeping (D). Legend at top.	22
5.1	Policy Iteration times (log scale) for different numbers of states and actions.	26
5.2	Policy Iteration Speedups for cuda and cuSPARSE cuda implementations compared to CPU for different datasets.	26
5.3	Policy Iteration times (log scale) using GPUs compared to CPU for different numbers of states and actions.	27
5.4	Policy Iteration speedups using GPUs compared to CPU for different numbers of states and actions.	28

List of Tables

3.1	CSR arrays for a small MDP stored by state-action rows.	16
3.2	Per-row map (successor lists are shown as (s', P, R) tuples).	16
5.1	System CPU Specifications	23
5.2	Comparison of NVIDIA L40S, V100, L4, and H100 GPUs on Barkla, showing actual memory sizes and power caps from <code>nvidia-smi</code> .	23
5.3	Correctness Test MDP	24
5.4	Performance Test MDPs	24
5.5	Performance results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.	25
5.6	Performance results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.	25
5.7	Runtime results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.	27
5.8	Speedup (Serial \div GPU runtime) for different GPUs across MDP sizes.	27

Chapter 1

Introduction

1.1 Scope

This project accelerates policy iteration for finite-state Markov Decision Processes (MDPs) by exploiting data-parallel hardware on modern NVIDIA GPUs. We design, implement, and benchmark a CUDA-based policy-iteration solver against a single-threaded CPU baseline across synthetic and benchmark MDPs. Problem instances vary in size, structure, and sparsity—from compact environments with thousands of states to highly sparse models with millions of states.

1.2 Problem Statement

Policy iteration switches between *policy evaluation* and *policy improvement*. Our method uses a *modified variant of policy iteration*, where policy evaluation relies on synchronous Jacobi sweeps until a convergence tolerance is reached, instead of solving the linear system exactly. Let T_π denote the Bellman expectation operator

$$(T_\pi V)(s) = \sum_{s'} P_\pi(s, s') (R_\pi(s, s') + \gamma V(s')).$$

Given a policy π , Jacobi updates are

$$V^{(k+1)} = T_\pi V^{(k)}, \quad \text{stop when } \|V^{(k+1)} - V^{(k)}\|_\infty < \theta \text{ or } k \geq K_{\max}.$$

Although policy iteration typically converges in tens of outer iterations, dense CPU approaches become impractical beyond $S \approx 10^5$ due to the $O(S^2)$ memory and poor parallel efficiency of dense kernels. The challenge is to map Jacobi-based evaluation and greedy improvement efficiently to GPUs—maximizing bandwidth utilization on sparse data while preserving numerical correctness.

1.3 Approach

We model transition dynamics and rewards with sparse matrices in Compressed Sparse Row (CSR) format. For a fixed policy, each Jacobi sweep reduces to one CSR SpMV (Sparse Matrix–Vector multiplication) and a parallel ℓ_∞ reduction:

$$V^{(k+1)} = T_\pi V^{(k)} \iff \text{CSR-SpMV over rows of } P_\pi \text{ plus a max-reduction on } |V^{(k+1)} - V^{(k)}|.$$

Synchronous Jacobi is well suited to GPUs: rows update independently (no write-after-read hazards see Appendix [B.1](#)), avoiding atomics and enabling one-thread/warp-per-row kernels with coalesced reads of CSR data. Policy improvement is parallelized statewise by evaluating $Q(s, a)$ over each action’s outgoing nonzeros and selecting $\arg \max_a Q(s, a)$. Benchmarks span

sizes and sparsity patterns; metrics include Bellman residuals, sweeps per evaluation, outer iterations, and wall-clock time. The implementation is in modern C++20 and CUDA, with fixed seeds and scripts with configuration information for reproducibility.

1.4 Expected Outcomes

We deliver a CUDA/C++ implementation of modified policy iteration with one CPU back end and two GPU back ends, together with performance results on six datasets that vary in size and sparsity. The analysis focuses on the main computational kernels, namely CSR SpMV, greedy selection, and parallel reductions, and examines how dataset characteristics influence overall performance. We conclude with an evaluation across multiple GPUs with distinct architectural features, showing how these differences affect speedups. Finally, we compare the use of cuSPARSE provided kernels with a custom policy evaluation kernel, highlighting the tradeoffs between relying on optimized library routines and hand written implementations.

Chapter 2

Background

2.1 Theoretical Foundations

2.1.1 Markov Decision Processes

A *MDP*, illustrated in figure 2.1 is a mathematical tool for capturing sequential decision making problems in which outcomes are in part under the control of an agent and in part subject to randomness [26, 23, 9]. Formally, an MDP is defined by a quintuple $\mathcal{M} = (S, A, P, R, \gamma)$, where S denotes the set of states which represents the possible contexts of the environment, and A denotes the set of actions available to the agent. The transition dynamics are captured by the probability distribution $P(s' | s, a)$, which specifies the likelihood of reaching a successor state s' given that action a is taken in state s . Associated with each transition is a reward function $R(s, a, s') \in \mathbb{R}$ that assigns an immediate payoff, and a discount factor $0 \leq \gamma < 1$ that weighs the relative importance of future versus immediate rewards.

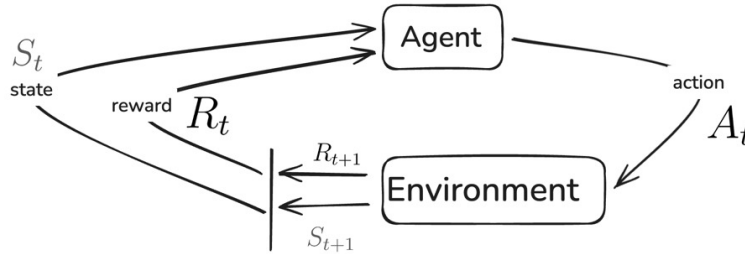


Figure 2.1: Markov Decision Process (MDP) illustrating states, actions, transitions, and rewards.

The fundamental assumption that underpins MDPs is the *Markov property*, which states that the process's future depends solely on its current state and action, not on the sequence of states and actions that preceded it. For example, in a gridworld domain, if the agent is located in the middle of the grid, the next state and its value depend only on that current position and chosen action, not on the path the agent took to reach the middle. The behavior of an agent is captured by its *policy*, π , which maps each state either deterministically to a single action $\pi(s) = a$ or stochastically to a distribution over actions $\pi(a | s)$.

To evaluate the quality of a policy, one considers value functions. The *state-value function* $V^\pi(s)$ represents the expected discounted return when starting from state s and following policy π thereafter. The *action-value function* $Q^\pi(s, a)$ represents the expected return when starting from state s , taking action a , and then continuing with π [26]. The objective of solving an MDP is to determine an *optimal policy* π^* that maximizes the expected return over time. Classical solution methods include dynamic programming (DP) techniques such as value iteration and

policy iteration, which exploit the recursive Bellman equations[23], while in scenarios where the transition dynamics are unknown, reinforcement learning(RL) methods such as Q-learning or deep reinforcement learning(DRL) are employed [16].

MDPs form the theoretical foundation of much of reinforcement learning and have applications across robotics, operations research, economics, and artificial intelligence, where agents must learn to act optimally in uncertain and dynamic environments.

2.1.2 Solving Markov Decision Processes With Policy Iteration

One of the classical approaches to solving MDPs is *policy iteration*[9, 23]. This method seeks to compute an optimal policy π^* that maximizes the expected discounted return over time. The procedure relies on two key steps that are repeated iteratively: *policy evaluation* and *policy improvement*.

In the evaluation step, the value function V^π corresponding to the current policy π is computed. Formally, $V^\pi(s)$ satisfies the recursive Bellman equation

$$V^\pi(s) = \sum_{a \in A} \pi(a | s) \sum_{s' \in S} P(s' | s, a) \left(R(s, a, s') + \gamma V^\pi(s') \right),$$

for all $s \in S$. In practice, this system of equations is typically solved iteratively, for instance with Jacobi or Gauss–Seidel style updates, until convergence within a small tolerance[23].

Once the value function of the existing policy has been approximated, the policy improvement is done. Here, the decision rule of the agent is improved by choosing, for every state, the action that maximizes the expected return given the approximated values of the successor states. Concretely, the improved policy is defined as

$$\pi'(s) = \arg \max_{a \in A} \sum_{s' \in S} P(s' | s, a) \left(R(s, a, s') + \gamma V^\pi(s') \right).$$

If the new policy π' is identical to the old one, then the algorithm has converged and π is optimal. Otherwise, π is replaced by π' and the cycle of evaluation and improvement continues which is shown in the figure2.2

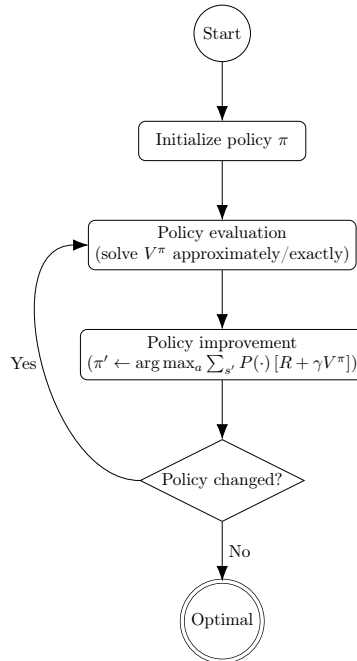


Figure 2.2: Policy Iteration Algorithm Flowchart

As shown in Algorithm 1, policy iteration switches between policy evaluation and policy improvement until convergence.

Algorithm 1 Policy Iteration for a Discounted MDP

Require: State set S , action set A , transition model $P(s' | s, a)$, reward $R(s, a, s')$, discount $0 \leq \gamma < 1$, evaluation tolerance $\theta > 0$

- 1: Initialize a policy π arbitrarily (e.g., choose any action in each $s \in S$)
- 2: Initialize $V(s) \leftarrow 0$ for all $s \in S$
- 3: **repeat**
- 4: $V \leftarrow \text{POLICYEVALUATION}(\pi, V, \theta)$ ▷ Iterative evaluation to tolerance
- 5: $\text{policyStable} \leftarrow \text{true}$
- 6: **for** $s \in S$ **do**
- 7: $a_{\text{old}} \leftarrow \pi(s)$
- 8: Compute the action-values under V :

$$Q(s, a) \leftarrow \sum_{s' \in S} P(s' | s, a) (R(s, a, s') + \gamma V(s')) \quad \forall a \in A$$

- 9: $\pi(s) \leftarrow \arg \max_{a \in A} Q(s, a)$ ▷ Greedy improvement (deterministic)
 - 10: **if** $\pi(s) \neq a_{\text{old}}$ **then**
 - 11: $\text{policyStable} \leftarrow \text{false}$
 - 12: **end if**
 - 13: **end for**
 - 14: **until** policyStable
 - 15: **return** (π, V)
-

The theoretical foundation of policy iteration is strong: it is guaranteed to converge to an optimal policy in a finite number of iterations given that the state and action spaces are finite [23].

In addition to that, it typically converges faster than value iteration alone because the policy is updated after each round of evaluation. The policy evaluation, however, can be costly, especially in large problems where exactly solving the value function is not possible. In such cases, approximate policy iteration methods are employed, wherein the process of evaluation is truncated after a certain specified number of iterations or replaced with approximate solutions such as Monte Carlo estimation or TD learning [26, 8].

Policy iteration is therefore caught in between theoretical guarantee and practical performance. It is a corner stone algorithm in reinforcement learning and dynamic programming, and a foundation for more complex techniques that balance exact DP techniques with approximate function approximations and modern-day GPU acceleration.

2.1.3 Parallel Computing and GPU Programming

Modern computational problems, particularly those involving enormous data or complex models, typically exceed sequential processing on a single CPU core.

Parallel computing avoids such constraint by dividing work into many processing elements that can run concurrently [6].

The fundamental concept is to divide a problem into multiple subproblems, assign them to different processors, and coordinate their computation so overall performance is improved. The benefits of parallelism are typically measured by *speedup*, parallel to serial runtime ratio, and *efficiency*, the ratio of resources utilized effectively. It is nonetheless difficult to achieve high performance because it entails a careful control of workload, memory access patterns, and synchronization. Among the most popular parallel computing architectures are *GPUs*.

Originally designed to accelerate computer graphics, GPUs are now highly programmable multi-core processors for general purpose scientific and engineering computation [22, 10].

As opposed to CPUs, that are optimized for low-latency processing of sequential instructions, GPUs are optimized for high-throughput processing of massively parallel workloads. This renders them particularly well-suited to problems that are tractable as a description in terms of enormous numbers of independent or weakly interacting operations, i.e., linear algebra, simulations, and machine learning [4, 13]. GPU programming is based on a different paradigm than conventional CPU programming. One of the dominant programming models is NVIDIA's *CUDA*, which allows developers to write GPU kernels which are functions executed in parallel by many lightweight threads [17]. Threads are organized hierarchically into blocks and grids (illustrated in figure 2.3), and their execution is coordinated by the GPU hardware. The memory hierarchy is another central aspect: GPUs provide several types of memory, including global memory, shared memory, registers, and texture memory, each with different capacity, latency, and bandwidth characteristics (illustrated in figure 2.4). Efficient GPU programs carefully exploit this hierarchy, maximizing memory coalescing and minimizing data transfers between the host CPU and device memory [10].

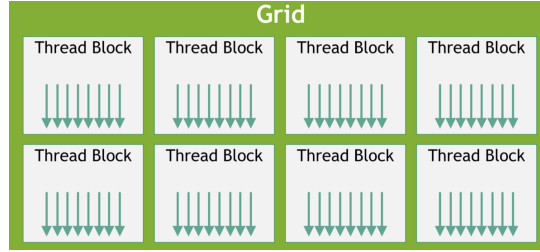
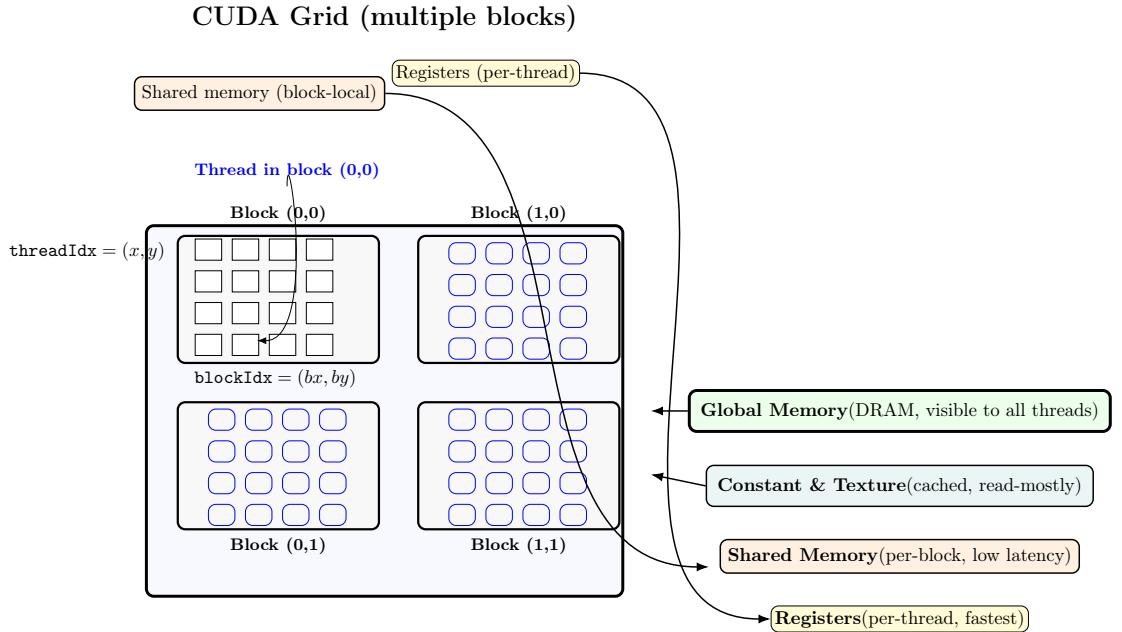


Figure 2.3: CUDA hierarchy: Grid \rightarrow Blocks \rightarrow Threads. Each block here shows a tile of threads sharing block-local resources (shared memory, sync) [20].



Execution model: A kernel launch instantiates a *grid* of *blocks*, each with many *threads*.

Indices: `blockIdx`, `threadIdx` select a thread's portion of work.

Memory: Global \rightarrow all threads; Constant/Texture \rightarrow cached reads; Shared \rightarrow per-block scratchpad; Registers \rightarrow per-thread.

Performance tips: coalesced global loads, minimize divergence within a warp, reuse data via shared memory.

Figure 2.4: CUDA memory model: hierarchical memory spaces with different visibility and latency trade-offs.

Despite their high computational throughput, GPUs pose challenges in terms of algorithm design and implementation. Not all problems are naturally parallel, and overheads such as synchronization and communication can diminish the potential gains. Furthermore, the effectiveness of a GPU implementation is often determined less by raw arithmetic performance and more by how well the algorithm leverages the underlying memory system. Nevertheless, for algorithms that map well to the GPU model, such as sparse matrix–vector multiplication, convolutional operations, or Monte Carlo simulations, the performance gains can be orders of magnitude compared to CPU execution [21, 30, 14, 3, 13].

The relevance of GPU programming to MDPs lies in the fact that their solution methods often reduce to large-scale linear algebra operations or iterative sweeps over state–action spaces. These operations are amenable to fine-grained parallelism, where each thread is responsible for computing contributions from a subset of states, actions, or transitions. Harnessing GPUs therefore enables the scaling of classical algorithms such as policy iteration to problem sizes that would be infeasible on conventional CPU hardware.

2.2 Cuda Programming Model

CUDA is NVIDIA’s parallel computing platform, consisting of GPU hardware, a compiler toolchain, drivers, a runtime environment, and optimized libraries, together with a language based on extensions to C++. This combination enables developers to exploit the massive parallelism of modern GPUs for general-purpose computation. [17, 22]. While GPUs were originally designed for graphics rendering, CUDA exposes them as programmable many core processors, allowing applications far beyond graphics, including scientific simulation, machine learning, and the solution of large-scale MDPs [4, 13].

The CUDA programming model is built around the concept of *kernels*, which are functions written in CUDA C++, an extension of C++ with GPU-specific keywords, built-in variables, and a dedicated kernel launch syntax [20]. Kernels are executed on the GPU by many lightweight threads in parallel. When a kernel is launched, it is instantiated across a grid of *thread blocks*. Each block contains a fixed number of threads, and together the blocks form a grid that spans the problem domain. Threads within a block are identified by their thread indices, and blocks within a grid are identified by block indices. Using these indices, each thread can compute a unique subset of the overall workload. This two-level hierarchy (grid–block–thread) provides both scalability and flexibility: small problems can use few blocks, while large problems can span thousands of blocks and millions of threads [10].

A simple illustration is a vector addition kernel, where each thread computes one element of the output, is shown in the listing 2.1. Then the kernel is launched with a specified grid and block dimension, as shown in the listing 2.2.

Listing 2.1: Example Cuda Kernel to add vectors

```

1  __global__ void vector_add(const float *xVec,
2                             const float *yVec,
3                             float *zVec,
4                             int N) {
5      int idx = blockIdx.x * blockDim.x + threadIdx.x;
6      if (idx < N) {
7          zVec[idx] = xVec[idx] + yVec[idx];
8      }
9  }
```

Listing 2.2: CUDA C++ Kernel Launch Syntax Example

```

1  vector_add<<<gridDim, blockDim>>>(xVec, yVec, zVec, N);
```

In the listing 2.1, the `__global__` qualifier is an example of cuda C++ syntax to indicate kernels and `<<< ... >>>` syntax shown in listing 2.2 is the kernel launch syntax. In this example, CUDA creates `gridDim` blocks, each with `blockDim` threads. Together these threads cover all N elements of the vectors. Each thread computes a unique index `idx` from its block and thread indices, ensuring that the workload is distributed across the GPU.

Underlying this execution model is a hierarchical memory system. *Global memory* is held in device DRAM and is shared by all threads, but has high latency and low bandwidth per thread. *Shared memory*, however, is on-chip and shared by all threads of a block, and has much lower latency and accommodates cooperative computation. Additionally, there is access by each thread to private registers, which have the fastest storage space but are limited. Optimal CUDA programs minimize slow global memory accesses through careful data layout, memory coalescing, and the use of shared memory for intermediate results. CUDA also offers *constant* and *texture* memories, optimized for distinct access patterns [10, 17]. CUDA device memory types are illustrated in Figure 2.5.

Thread execution is then structured into *warp* groups, of which there are typically 32 threads running in lockstep on a single multiprocessor. Execution by warps has performance consequences: when threads within a warp diverge (for example, by following different branches of an if statement), the GPU will need to serialize their execution, and throughput will suffer. Consequently, branch divergence is a key design point when designing CUDA kernels. Occupancy is also something to consider, and it can be described as the number of active warps divided by the amount that the hardware will allow. Better occupancy helps to hide memory latency by keeping some warps around to execute at all times [10].

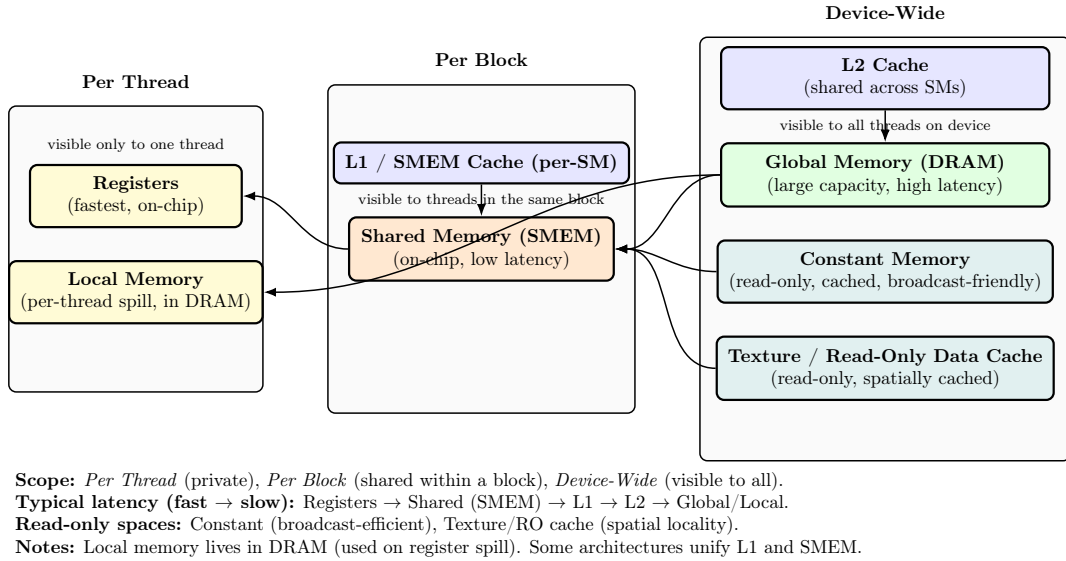


Figure 2.5: CUDA device memory types, their scope, and typical latency characteristics.

Programming in CUDA thus requires both algorithmic and architectural awareness. On the one hand, the algorithm must be decomposed into fine-grained independent tasks that can be executed by thousands of threads. On the other hand, the implementation must align with the GPU’s hierarchical execution model and memory hierarchy to achieve high efficiency. For example, sparse matrix–vector multiplication, a common primitive in solving MDPs, is naturally parallelizable since each row of the matrix can be processed independently. By mapping rows to blocks and nonzero entries to threads, one can leverage CUDA’s massive parallelism while exploiting shared memory for local reductions [3, 21].

Briefly, the CUDA programming model combines a hierarchical thread organization, a multilevel memory system, and warp based execution semantics. It provides abstraction as well as

control to compile computationally intensive algorithms into highly parallel GPU-based code. This makes CUDA suitable for accelerating DP algorithms such as PI, where an analogous pattern of computation needs to be performed numerous times over enormous state and action spaces [30, 14].

2.3 State of the Art: GPU-Accelerated Policy Iteration for MDPs

2.3.1 Introduction

GPU-accelerated dynamic programming has transformed how large MDPs are solved, especially in reinforcement learning (RL) contexts where planning algorithms like Value Iteration (VI) and Policy Iteration (PI) can become a bottleneck. By exploiting massive parallelism, modern GPUs achieve dramatic speedups over traditional CPU implementations in computing Bellman updates for large state spaces. Below, we review the state of the art developments in GPU based policy iteration, focusing on performance benchmarks (vs. CPU) in subsection 2.3.2, the effects of MDP structure on efficiency in subsection 2.3.3, multi-GPU scaling in subsection 2.3.4, and comparisons to other solution methods in subsection 2.3.5. We also discuss best practices in reporting performance metrics in subsection 2.3.6.

2.3.2 GPU vs. CPU Performance for Value/Policy Iteration

Many studies report that GPUs outperform single-threaded CPU implementations of VI/PI by one to two orders of magnitude (or more) on sizable MDPs. For example, a recent Julia-based solver achieved “speed-ups of various orders of magnitude on the larger systems” when using a GPU, compared to existing sequential (CPU) tools [15]. In practical terms, this often translates to $10\times$ – $100\times$ faster run-times, and in some cases much higher:

One experiment solved MDPs up to ~ 1 million states on GPUs (e.g. NVIDIA 1080Ti, RTX 8000, V100), yielding $20\times$ to $1000\times$ speedups over a serial Intel Xeon CPU baseline [4]. The GPU implementation could solve a million-state MDP in under 30 seconds, whereas the CPU took orders of magnitude longer [4]. Chowdhury et al. (2022) likewise report orders-of-magnitude GPU speedups for a planning workload in stochastic flows, consistent with the trends seen for VI/PI [2].

A 2015 GPU-based MDP solver for crowd simulation found massive gains: e.g. up to $90\times$ faster than an 8-thread Intel CPU on an “office” gridworld, and up to $820\times$ faster on a larger maze scenario [24]. Even against an ARM CPU, speedups reached $2000\times+$ in some cases [24]. Such results highlight that GPUs can replace dozens or hundreds of CPU cores for MDP planning.

It should be noted that these speedups assume a single-thread (or limited-thread) CPU baseline. Multi-core CPU implementations such as OpenMP or MPI can narrow the gap somewhat, but GPUs still usually win on throughput. In summary, the literature consistently shows GPU VI/PI outperforming CPUs by 1-2 orders of magnitude on large problems [15, 4], making real-time or interactive planning feasible in domains that were previously intractable with CPU-only methods.

2.3.3 Influence of MDP Structure on GPU Efficiency

The performance benefit of GPU acceleration can vary depending on the structure and sparsity pattern of the MDP’s state-transition graph:

Regular structures like gridworlds are highly GPU-friendly. States have a uniform number of neighbors and memory access patterns are predictable, which enables coalesced memory accesses on GPU. As a result, these problems achieve higher utilization of GPU memory bandwidth and compute units. For instance, in a classic study of sparse matrix-vector multiply (analogous to Bellman updates), a structured 3D grid delivered ~ 16 GFLOPs on an NVIDIA GPU versus

~ 10 GFLOPs for an unstructured mesh [1]. This GPU performance was “more than 10 times that of a quad-core Intel” system for both cases [1], but the structured case better exploited the GPU (nearly 60% higher throughput than the irregular case). In practical MDP terms, a regular grid with uniform transitions achieves the largest GPU speedups, thanks to optimal memory coalescing and balanced thread workloads.

For MDPs with arbitrary or highly irregular transition structures (e.g. state graphs with varying out degree or random connectivity), naive GPU implementations suffer from load imbalance and scattered memory accesses. Each thread may process a different number of successors, leading to warp divergence and under utilization of GPU cores. Memory accesses to the value function can also be random, hurting cache efficiency. However, research shows that with optimized sparse operations, GPUs still yield significant gains on irregular MDPs. Using tuned sparse matrix techniques (e.g. CSR formats and warp aggregated loads), one can mitigate these issues. NVIDIA’s cuSPARSE library, for example, provides optimized routines for sparse matrix-vector multiplication that handle irregular patterns efficiently.

Bell and Garland (2009) [1] showed that the performance gains from GPU based SpMV depend strongly on the sparsity pattern: in some cases the improvements were only modest, while in others their HYB format achieved speedups of roughly $5\times$ - $15\times$ versus the CPU implementations.

Empirical results in MDP solvers back up this trend. Ruiz-Loza and Hernández (2015) [24] reported that in their crowd-simulation benchmarks, the *maze* scenario (a grid with many walls/obstacles) achieved an $\sim 820\times$ GPU speedup, compared to about $90\times$ in the less structured *office* scenario. This highlights how the underlying structure of the state transition graph can strongly influence GPU performance.

Conversely, completely irregular MDP graphs won’t hit such extreme gains, but optimized GPU code still handily outperforms CPU (often by an order of magnitude or more). Overall, structured MDPs benefit the most, but with careful engineering, irregular MDPs also see substantial GPU acceleration.

2.3.4 Scaling Out: Multi-GPU and Distributed Solutions

When state spaces become extremely large (or when further speedup is needed), scaling out across multiple GPUs is the next step. State-of-the-art approaches partition the MDP’s state space across GPUs and use high-speed interconnects (like NVIDIA NVLink and NVSwitch) to handle the necessary synchronization of boundary states. The literature indicates that multi-GPU scaling can be near-linear given that there exist efficient communication between the GPUs.

Intra-node multi-GPU

Modern GPU servers often integrate 4–8 GPUs via NVLink or NVSwitch, offering roughly $20\times$ – $50\times$ more bandwidth than PCIe. This high-bandwidth, low-latency interconnect enables frequent inter GPU communication with reduced penalties. NVIDIA reports near-linear scaling to four Tesla P100 GPUs on structured HPC benchmarks [18]. Similarly, multi-GPU graph frameworks such as Lux demonstrate excellent scaling, achieving $20\times$ speedup on 4 GPUs relative to a high-end CPU server, and up to two orders of magnitude versus a distributed CPU cluster [31]. These findings suggest that, when communication is carefully managed, multi-GPU systems can maintain high efficiency as additional GPUs are added.

Distributed GPU clusters

Going beyond one node, NVLink Switch and technologies like NCCL allow extending this model across multiple nodes (each with several GPUs), creating a cluster of GPUs that communicate

in a peer-to-peer fashion. With careful partitioning of the state graph (minimizing cross-node edges) and asynchronous communication, near-linear scaling can extend to multi-node clusters as well. Researchers have demonstrated large-scale graph algorithms (like BFS or PageRank) running on GPU clusters with strong scaling, the key is that GPUs have such high compute/memory throughput that even if communication is nontrivial, they often remain compute-bound rather than comm-bound for reasonably balanced partitions. In the context of MDP value iteration, this means one can solve truly enormous MDPs by spreading states over, say, 16 or 32 GPUs, achieving almost N -fold speedup with N GPUs in ideal cases.

For reinforcement learning or planning problems that involve billions of states or require real-time solutions, a multi GPU approach is viable and has been shown to maintain efficiency. The caveat is that one must use GPUs with fast interconnects (NVSwitch within a node, or InfiniBand/NVLink between nodes) and design the algorithm to communicate only when necessary (e.g. exchange boundary value data at the end of each iteration). With these best practices, scaling out is highly effective, and near-linear acceleration across GPUs has been reported for structured HPC and graph workloads [18].

2.3.5 Other Solution Methods (Beyond Policy Iteration)

While our focus is on classical dynamic programming (PI/VI) on GPUs, it is useful to situate these planners among alternative approaches to solving or approximating MDPs. Broadly, three families are most relevant for context: linear-programming (LP) formulations of MDPs, DRL with function approximation, and heuristic/planning methods such as Monte Carlo Tree Search (MCTS) or graph-search variants (e.g., AO*, LAO*). These alternatives illuminate when exact GPU PI/VI is preferable and when approximation or search may be advantageous.

Discounted infinite horizon MDPs admit an equivalent LP in either value or occupancy measure space, providing strong optimality guarantees and a clean benchmark for theoretical analysis [23]. However, the practical scalability of general purpose LP solvers on large MDPs is limited: problem sizes with millions of states translate into extremely large constraint matrices, and the pivoting/factorization-heavy inner loops are difficult to map efficiently to GPUs at scale.

Even though there is active research on GPU-accelerated LP and first-order methods, in applied RL and planning the dominant practice for large, sparse problems remains iterative Bellman schemes (VI/PI) or problem specific decompositions, which better exploit sparsity patterns and SIMD style parallelism. LP remains valuable in smaller verification settings or when exact dual certificates are required, but for the kinds of sizable, sparse MDPs considered here, iterative dynamic programming typically offers a superior compute-to-solution profile.

DRL replaces tabular value functions with neural approximators and learns from data rather than performing full Bellman backups over the entire state space. Landmark results such as DQN on Atari established the efficacy of GPU-trained convolutional networks for value estimation from high-dimensional observations [16], while actor-critic and policy-gradient families extend this to continuous control [26]. In these methods, GPUs accelerate mini-batch gradient updates, replay-buffer sampling, and large-model training; the computational bottleneck is stochastic optimization rather than sparse Bellman SpMV. Deep RL scales to problems where the model is unknown or the state space is effectively unenumerable, but it typically trades away exact optimality and requires extensive hyperparameter tuning and sample generation. In contrast, when a reliable model is available and exact planning is desired, GPU-accelerated PI/VI can deliver deterministic convergence with strong wall clock performance on structured, sparse problems [15, 4], making the two paradigms complementary rather than competing.

Search-based planners explore only portions of the state space and are effective when heuristics or rollout policies can focus computation on promising regions. Monte Carlo Tree Search with UCT demonstrated strong performance in large sequential decision problems by balancing exploration and exploitation through statistical tree growth [12]. AO* and its extension LAO* adapt best-first search to cyclic MDP graphs, interleaving heuristic expansion with DP style

backups on the explored subgraph [7]. These methods, as presented in the literature, do not rely on GPU acceleration; their contributions lie in search strategies and heuristic guidance. However, in principle, components such as simulation rollouts, batched heuristic evaluation, or parallel backup operations could benefit from GPU parallelism if engineered appropriately. The effectiveness of such parallelization would depend strongly on problem structure, heuristic quality, and the ability to keep GPU resources well utilized. For large, structured sparse MDPs (e.g., grid-like domains), bulk synchronous GPU implementations of PI/VI remain a particularly strong baseline, as they exploit predictable sparsity patterns and map directly to efficient GPU kernels [1]. In contrast, search based planners broaden the toolbox for settings where exhaustive dynamic programming is infeasible or undesirable.

In summary, DRL and heuristic search broaden the toolbox for cases where full state dynamic programming is infeasible or undesirable, whereas GPU accelerated PI/VI excels when the model is available and exact solutions over large but sparse transition graphs are the target [15].

2.3.6 Benchmarking Practices and Reporting Norms

State of the art empirical studies of GPU MDP solvers emphasize transparent convergence and throughput reporting so results are reproducible and comparable. Convergence is typically quantified via a Bellman residual (e.g., $\max_s |V_{k+1}(s) - V_k(s)|$) or policy stability, with an explicit tolerance (e.g., $< 10^{-6}$) and the number of sweeps/iterations to reach it [15]. Wall clock time is reported alongside iteration counts to separate algorithmic efficiency from raw hardware speed; many works additionally break down time by major kernels (SpMV, axpy, policy improvement) and, for multi-GPU experiments, by communication vs. computation.

Problem descriptors, such as $|S|$, $|A|$, and total nonzeros (nnz) in the transition representation are essential to contextualize sparsity and memory traffic [15, 4]. Precision and datatypes (e.g., FP32 vs. FP64 values and 32-bit indices) are specified because numerical stability and GPU throughput can vary substantially with precision; some implementations prefer FP64 for robustness, accepting lower peak FLOPs [15]. Hardware/software details are standard: GPU model and memory (e.g., V100 32 GB HBM2), CUDA/cuSPARSE versions, and CPU baselines (cores/threads) [2].

Finally, modern GPU features used to curb overhead—such as asynchronous streams, pinned/-managed memory, or NVLink/NVSwitch for intra-node scaling—should be disclosed to clarify where speedups originate and to aid independent replication [18]. Adhering to these norms yields clear accuracy-throughput tradeoffs and aligns with best practice across recent GPU-accelerated value/policy-iteration studies [15, 4].

2.3.7 Conclusion

State-of-the-art GPU acceleration has made it feasible to solve large MDPs in reinforcement learning with unprecedented speed. For policy iteration (and value iteration), GPUs routinely provide $10\times$ – $100\times$ speedups vs. CPU, and even higher in structured domains [15, 24]. The biggest performance gains come from regular, grid-like MDP structures that align well with GPU memory access patterns, though even irregular problems see significant benefit with optimized sparse kernels [1]. Scaling across multiple GPUs further amplifies performance, approaching linear scaling thanks to fast interconnects [18]. We focused on policy iteration here, but briefly noted that alternative approaches (DRL, etc.) also leverage GPUs in different ways. Finally, we highlighted the importance of benchmarking and reporting metrics, including convergence criteria, iteration counts, wall-clock time, problem sizes, precision, and hardware details, to adhere to current best practices in the field [15].

Overall, the literature consensus is that GPU-based dynamic programming is a game-changer for planning in large MDPs, enabling applications in robotics, planning, and RL that were previously impractical. With the latest hardware (e.g. A100, H100 GPUs) and proper optimizations

like cuSPARSE, one can expect order of magnitude speedups and the ability to tackle truly large scale MDPs efficiently.

Chapter 3

Design

3.1 Solving Policy Iteration via GPU Acceleration

The solving of MDPs by Policy Iteration maps naturally to the CUDA architecture [23, 26, 17, 10]. The policy evaluation step reduces to repeated sparse matrix–vector operations, parallelized through assigning different states to threads on the GPU, while the policy improvement step is trivially parallel since each state updates independently [4, 13].

When we do policy evaluation on the GPU, each *sweep* means updating the value of every state once. Within a sweep, the updates are perfect for parallelization: every state can be handled by a different thread, because each update only depends on the values from the previous sweep. But after finishing one sweep, we need its results before starting the next one. That means the sweeps themselves have to run one after another in sequence, even though the work inside each sweep runs in parallel [21, 3].

The solving of MDPs by computing both the value function and the corresponding greedy policy also maps naturally to the CUDA architecture. Unlike Value Iteration, which updates values for every state at every step, Policy Iteration switches between two distinct phases: policy evaluation and policy improvement [23, 9]. The evaluation phase requires solving for the value function of a fixed policy, which is equivalent to solving a large system of linear equations or applying iterative methods such as Jacobi or Gauss–Seidel sweeps. This process is computationally intensive but highly parallelizable since the update for each state depends only on its own row of transitions. The improvement phase, in turn, consists of selecting the best action per state according to the evaluated value function, which again can be performed independently across states. These two features make Policy Iteration a strong candidate for GPU acceleration, as the decomposition aligns well with CUDA’s block-and-thread execution model [30, 14].

The issue of splitting state spaces is also present in Policy Iteration, but the algorithm has additional computational challenges compared to Value Iteration. While the improvement step is trivially parallel, policy evaluation either requires iterative sweeps or direct sparse linear algebra, both of which are significantly sped up by GPU libraries such as cuSPARSE [17]. Traditional parallel systems, such as shared-memory multi-CPU architectures or distributed clusters, face limitations when tackling policy evaluation because of the heavy communication cost involved in linear system solvers. On the other hand, modern GPUs provide vast parallelism, thousands of lightweight threads and highly optimized sparse matrix operations, that are particularly well suited to the alternating evaluation and improvement structure of Policy Iteration. Thus, not only does the GPU provide scalability over large MDPs but it also alleviates the performance bottlenecks inherent to policy evaluation, enabling speedups of one to two orders of magnitude over conventional platforms [4, 13, 21].

3.2 Analyzing Policy Iteration on the GPU

The Policy Iteration algorithm presents multiple opportunities for parallelization [9, 23, 26]. Step-wise through the definition of the algorithm, we first encounter the policy evaluation phase, which requires solving for the value function of the current policy. This can be carried out either by iterative approximations (such as Jacobi sweeps) or by solving a sparse linear system [23]. In the iterative case, each state update depends only on the values of its successor states, meaning that rows of the transition matrix can be updated independently. This naturally leads to a row-parallel formulation where each CUDA block processes a state, and its threads accumulate contributions from successor states [4, 13]. In the direct linear system case, the computation reduces to repeated sparse matrix-vector multiplications, which GPUs handle efficiently through vendor libraries such as cuSPARSE [17, 10]. Both formulations make the evaluation phase amenable to large-scale parallel execution [21, 3].

The policy improvement phase is computationally simpler and in fact even more parallelizable. Each state independently selects the action that maximizes its expected return under the newly evaluated values. This independence means that no synchronization is required between states, allowing for a trivially parallel GPU kernel that assigns one block per state [30, 14]. The only part of the algorithm which is not inherently independent is the termination condition, which requires a global check for whether the policy has stabilized. This is analogous to the convergence check in Value Iteration and can be implemented using efficient parallel reductions [10]. Taken together, these observations show that the bulk of Policy Iteration maps naturally onto the GPU’s parallel execution model, with only minor synchronization costs for global checks.

3.3 Parallelization Strategy

Both phases of PI possess a high degree of data parallelism. This makes them particularly suitable for GPU acceleration [23, 26, 9]. Our approach is to exploit this parallelism simultaneously across states and within the structure of each state, whether that means distributing work over its successors or its available actions [17, 10].

During policy evaluation, the value of each state can be updated independently of the others within a Jacobi sweep [23]. To reflect this independence, the algorithm assigns a separate CUDA block to each state. Inside each block, threads cooperate to process the transitions of the action currently prescribed by the policy. Each thread handles a portion of the state’s successors, computing partial contributions to the expected return, which are then combined to form the new value for that state. In this way, the workload is balanced across states, while larger successor lists are efficiently processed by dividing them among multiple threads [4, 13, 21].

Policy improvement follows the same hierarchical idea, but the parallelism is organized around actions rather than successors. Each state is again mapped to its own block, but this time the threads within a block are distributed across the different actions available in that state. Each action-thread computes the expected return for its assigned action, and once all actions have been evaluated in parallel, the results are compared within the block to identify the maximizing action. That action becomes the updated policy for the state, and the computation proceeds independently for all states across the grid [30, 14].

This design, in which states are processed in parallel at the grid level and the finer work of evaluating successors or actions is distributed among threads within each block, aligns naturally with the GPU’s execution model [10]. It allows the algorithm to retain its mathematical structure while effectively harnessing both coarse-grained and fine-grained parallelism [3].

3.4 Why CSR format is chosen?

3.4.1 What is CSR format?

The Compressed Sparse Row (CSR) format is one of the most commonly used representations for sparse matrices [25]. Instead of storing the full two-dimensional array, CSR only records the nonzero entries together with their positions. This is achieved by three one-dimensional arrays: **data**, which holds the nonzero values; **indices**, which stores the column index of each entry; and **indptr** (standing for index pointer), which indicates where each row begins and ends within these arrays. If a matrix has m rows, then **indptr** has length $m + 1$, so that the nonzeros of row i are always located in the range **indptr** $[i] : \text{indptr}[i + 1]$. In this way the format compresses storage from $O(mn)$ to $O(\text{nnz})$, where nnz is the number of nonzero entries [22, 10].

CSR representation of an example small 2×2 grid world has been shown in table 3.1

Table 3.1: CSR arrays for a small MDP stored by state-action rows.

Array	Values
indptr	[0, 2, 3, 4, 6, 7, 8]
indices	[0, 1, 1, 2, 0, 2, 2, 1]
P_data	[0.5, 0.5, 1.0, 1.0, 0.3, 0.7, 1.0, 1.0]
R_data	[0.0, 1.0, 0.0, 2.0, 0.0, 0.0, 0.0, 3.0]

Notes: CSR is over rows $r = s \cdot A + a$. For row r , entries live in the half-open range $[\text{indptr}[r], \text{indptr}[r+1])$. Each nonzero at position j encodes a successor $s' = \text{indices}[j]$ with probability **P_data** $[j]$ and reward **R_data** $[j]$.

The per row map expansion of CSR arrays of table 3.1 is shown in table 3.2. Here, row $r=3$ corresponds to $(s=1, a=1)$ with entries $j \in [4, 6)$: successors $(s', P, R) = (0, 0.3, 0.0)$ and $(2, 0.7, 0.0)$. On the GPU during evaluation, the block for state s selects the row $(s, \pi[s])$ and threads sum $\sum_j P_j (R_j + \gamma V_{\text{old}}[s'_j])$. During improvement, one thread per a computes $Q(s, a)$ from that row, and a shared-memory argmax chooses $\pi_{\text{new}}[s]$.

Table 3.2: Per-row map (successor lists are shown as (s', P, R) tuples).

r	(s, a)	beg	end	successors (s', P, R)
0	(0,0)	0	2	(0, 0.5, 0.0), (1, 0.5, 1.0)
1	(0,1)	2	3	(1, 1.0, 0.0)
2	(1,0)	3	4	(2, 1.0, 2.0)
3	(1,1)	4	6	(0, 0.3, 0.0), (2, 0.7, 0.0)
4	(2,0)	6	7	(2, 1.0, 0.0)
5	(2,1)	7	8	(1, 1.0, 3.0)

The corresponding dense format above the above example (shown in figure 3.1) consists of two 6×3 matrices, which totals to 36 entries, one for transition probabilities $P[r, s']$ and one for rewards $R[r, s']$. On the other hand, the CSR format only requires 31 entries (16 nonzeros $(2 \times 8) + 7$ row pointers + 8 column indices), which is a saving of 13%. For small matrices the CSR overhead (row pointers, indices) can offset the gains, but as the matrix grows and the average successors per state stays small, CSR's storage grows roughly linearly in S , the number of states, while dense storage grows quadratically S^2 , so the percentage savings increase with size. A general analysis of storage savings for larger matrices is provided in Appendix A.2

(a) Transition probabilities $P[r, s]$

$$P = \begin{bmatrix} 0.5 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.3 & 0.0 & 0.7 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$$

(b) Rewards $R[r, s]$

$$R = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 2.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 \end{bmatrix}$$

Figure 3.1: Dense representations of the transition and reward matrices for a small MDP with $S = 3$ states and $A = 2$ actions. Each row $r = s \cdot A + a$ corresponds to a state-action pair.

3.4.2 What are the advantages of CSR format?

The main advantage of CSR is its memory efficiency, since it avoids storing zeros explicitly [25]. In addition, all the nonzeros of a row are stored contiguously, which allows very fast row access. This property is especially beneficial for sparse matrix–vector multiplication (SpMV), where each row is multiplied independently with the vector. Because the nonzeros and their indices are packed tightly, CSR also offers good cache locality on CPUs and coalesced memory access on GPUs [10]. Furthermore, since rows can be processed independently, the format maps naturally onto parallel architectures such as SIMD units and GPUs [17, 22]. For these reasons, CSR has become the de facto standard representation in many sparse linear algebra libraries, including cuSPARSE, Intel MKL, and PETSc [17].

3.5 Rationale for Design Choices

3.5.1 Why sparse grid worlds instead of dense ones?

A central design decision in this project was to focus on sparse grid worlds rather than dense transition models. In real-world settings, any state tends to have a small number of successor states, and most possible changes tend to be impossible [23, 26]. Representation of such settings as dense matrices raises quadratic space requirements, storing mostly zeros that never get accessed. By comparison, a sparse representation stores only the nonzero transitions reducing the memory requirement from $O(S^2A)$ to $O(\text{nnz})$ where nnz denotes the number of nonzero entries [22, 10]. This choice not only allows one to experiment with larger problem instances but is also inherently compatible with GPU acceleration. Sparse rows can be processed independently, and their structure allows threads to move over the relevant successors in parallel, without wasting time retrieving unnecessary zero entries. [4, 13, 21]. Thus, sparse grid worlds reflect both the structure of real problems and the computational needs of the GPU.

3.5.2 Why Jacobi Sweeps instead of Gauss-Seidel?

Another important design choice concerns the update scheme for value iteration inside policy evaluation. While the Gauss-Seidel method is generally preferred on CPUs due to its faster per iteration convergence [23], it updates values in-place, which creates dependencies between states within the same sweep. On a GPU, this introduces synchronization challenges (see Appendix B.1) and limits the degree of parallelism that can be exploited [10]. In contrast, the Jacobi method computes all updates from the values of the previous iteration. This eliminates intra-sweep dependencies, meaning that every state can be updated in parallel without risk of race conditions. Although this method may require more sweeps to converge, the ability to harness thousands of GPU threads in parallel more than compensates for the slower iteration rate, making Jacobi the more natural fit for large-scale GPU execution [4, 13].

3.5.3 Why both cuSPARSE and custom kernels implemented and compared?

Another rationale was to provide two implementations of the policy evaluation step: one based on cuSPARSE and one using a hand written CUDA kernel. cuSPARSE offers highly optimized vendor routines for sparse matrix operations, serving as a natural baseline for performance on modern GPUs[17]. In contrast, a custom kernel provides a transparent, controllable implementation without the abstraction layers and general purpose overhead of library calls. By comparing the two, we are able to quantify the speedups attributable to NVIDIA’s optimized library, and to validate the effectiveness of our own implementation against a trusted reference.

3.5.4 Trade-offs: load balancing, memory traffic, and scalability

Each of these design choices involves trade-offs. Sparse representations dramatically cut down memory usage, yet they introduce irregular row lengths that can create load balancing challenges across GPU threads [10]. Jacobi sweeps scale naturally across states, but they sacrifice some of the convergence rate that Gauss–Seidel would offer in a serial context [23]. cuSPARSE routines are optimized for a wide range of sparsity patterns, but they may involve additional memory traffic compared to hand-written kernels tuned to the grid world structure [17]. Overall, the guiding principle is scalability: decisions were made to maximize parallelism and reduce bottlenecks at scale, even if this meant giving up some iteration-level efficiency. The result is a solver that matches the structure of real-world sparse environments and exploits the GPU’s architecture effectively, while remaining grounded against library baselines for credibility [4, 13, 30, 14].

3.6 Summary

This chapter has shown how PI algorithm can be accelerated on GPUs by exploiting the parallelism in both policy evaluation and policy improvement, and by aligning design choices with GPU architectures.

Policy evaluation, whether via Jacobi sweeps or sparse linear solvers, reduces to parallel sparse matrix–vector operations, while policy improvement is embarrassingly parallel across states. Our parallelization strategy therefore assigns each state to a block and distributes its successors or actions among threads, capturing both coarse- and fine-grained parallelism.

Key design choices were motivated by scalability: sparse grid worlds to reduce memory costs, Jacobi sweeps to avoid intra-sweep dependencies, and the inclusion of both cuSPARSE and custom kernels to benchmark NVIDIA’s library speedups against a manual baseline.

Together, these decisions enable a GPU-centric formulation of Policy Iteration that preserves its theoretical guarantees while scaling efficiently to large MDPs.

Chapter 4

Implementation

4.1 Technology Stack

My implementation is developed with C++20 and GCC 11.5, leveraging new language features to both remove boilerplate and maintain zero-cost abstractions. GPU programming relies on CUDA C++ 12.4, providing us with direct access to NVIDIA hardware features so that we have low-level control over performance. Sparse linear algebra operations are offloaded using cuSPARSE, which we use for matrix–vector multiply to maintain efficiency as well as numerical stability. Unit testing is taken care of by GoogleTest[5], allowing us to verify correctness along the way. The build system of the project is managed by CMake[11], providing a portable and flexible configuration system across platforms.

4.2 Implementation through Plain Cuda

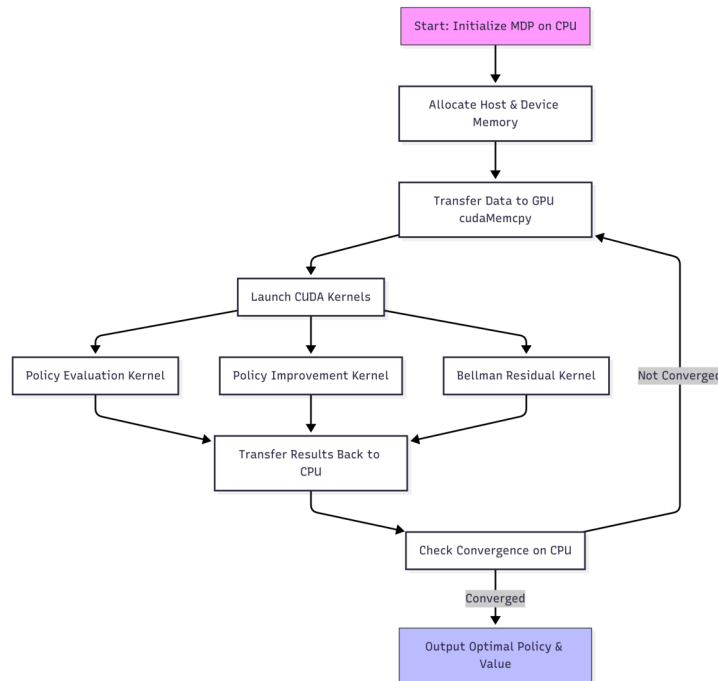


Figure 4.1: Flow of the overall architecture showing the interaction between CPU and GPU during Policy Iteration.

4.2.1 Policy evaluation

In policy evaluation, we map each state to its own CUDA block. Inside that block, all threads cooperate on a single CSR row —the one for the state’s currently chosen action $(s, \pi[s])$. Each thread takes a strided slice of that row’s nonzeros (successor states), multiplies the transition probability by the immediate reward plus the discounted value of the successor, and accumulates a partial sum. When they have each processed their slice, the threads perform a shared-memory reduction to combine their partials into a single number $V_{\text{new}}[s]$, which the block writes back to global memory. Intuitively, many threads “walk” different pieces of the same row in parallel, then fold their work together into one state value. This process is illustrated in Figure 4.2.

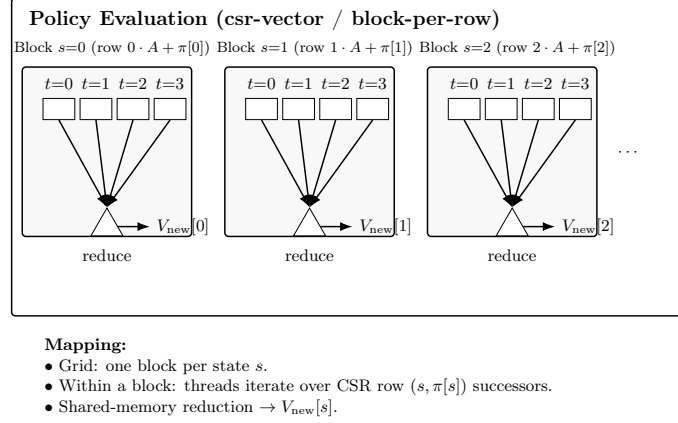


Figure 4.2: GPU Parallelization Strategy for Policy Evaluation

4.2.2 Policy improvement

In policy improvement, we again use one block per state, but now we assign one thread per action. Each action-thread computes $Q(s, a)$ by walking its own CSR row for (s, a) and summing over successors; all these per-action Q -values are stored in shared memory. Once the threads finish, a single thread in the block scans those Q values to pick the maximizing action and writes $\pi_{\text{new}}[s]$. Across the grid, states are handled in parallel by different blocks; within each block, actions are evaluated in parallel by different threads; and a quick local selection produces the new action for that state. This step is illustrated in Figure 4.3.

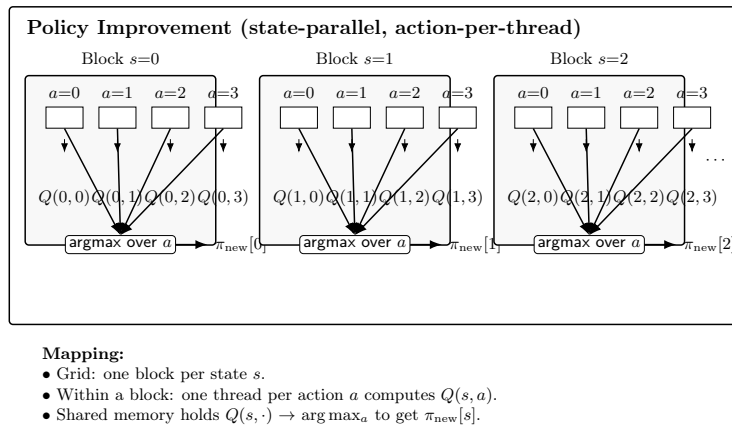


Figure 4.3: GPU Parallelization Strategy for Policy Improvement

4.3 Implementation through cuSparse

4.3.1 Policy evaluation (projected SpMV)

Instead of evaluating each state’s chosen action row directly, the code first *projects* the original $(S \cdot A)$ -row CSR into a policy-specific $S \times S$ CSR. Concretely, for each state s it copies the CSR row corresponding to the currently selected action $(s, \pi[s])$ into row s of a new matrix P^π , and it precomputes the reward vector

$$r^\pi[s] = \sum_{s'} P(s, \pi[s], s') R(s, \pi[s], s').$$

This process is illustrated in subfigure A of figure 4.4.

All of this is done on the GPU: a kernel counts the nonzeros of each selected row (`count_pi_nnz`), a device-side exclusive scan forms the new CSR pointer array, and another kernel copies indices/-values while accumulating r^π (`build_pi_csr_and_r`). With P^π and r^π available, each Jacobi sweep implements the Bellman update in linear-algebra form:

$$V^{(k+1)} = r^\pi + \gamma P^\pi V^{(k)}.$$

The product $y \leftarrow P^\pi V^{(k)}$ is computed using the cuSPARSE **SpMV** routine (optimized CSR SpMV), and then a small axpy-like kernel performs $V^{(k+1)} \leftarrow r^\pi + \gamma y$. After each sweep, a single-block reduction computes the max-norm residual

$$\delta^{(k)} = \|V^{(k+1)} - V^{(k)}\|_\infty,$$

and evaluation stops when $\delta^{(k)} < \theta$. Two engineering choices reduce memory traffic: (i) instead of copying vectors between sweeps, the code *swaps* the pointers of V_{old} and V_{new} (ping-pong), and (ii) it updates the cuSPARSE dense-vector descriptors to the swapped pointers (no reallocation). This process is shown in subfigure B of figure 4.4. Intuitively, by compressing the active rows into a compact S -row matrix and delegating the heavy lifting to a tuned SpMV, the sweep pushes value mass along edges efficiently, with only a cheap vector add/scale and a one-block residual check around it.

4.3.2 Policy improvement

Policy improvement uses the same mapping as the kernelized approach: one CUDA block per state and one thread per action. Each action-thread computes

$$Q(s, a) = \sum_{s'} P(s, a, s') (R(s, a, s') + \gamma V(s'))$$

by walking the CSR row for (s, a) . The thread block stores all per-action Q -values in shared memory; then a single thread applies deterministic tie-breaking (prefer the current action unless another exceeds it by more than a small ε) and writes $\pi_{\text{new}}[s]$. Thus, states proceed in parallel across blocks, actions proceed in parallel within a block, and a quick local selection produces the new action for each state.

4.3.3 Convergence check and housekeeping

Instead of copying entire policies back to the host each outer iteration, a device kernel counts how many entries differ between π_{new} and π ; only that single integer is transferred to the host. If the count is zero, policy iteration terminates; otherwise the policy pointers are swapped on device (avoiding a full vector copy). This step is shown in subfigure C of figure 4.4. Each outer iteration (policy) builds P^π once and queries cuSPARSE for the SpMV scratch buffer size, allocating a

temporary buffer accordingly. The implementation binds the floating type (`float` or `double`) to the correct cuSPARSE compute datatype, and uses explicit error checks for both CUDA and cuSPARSE for robustness and reproducibility.

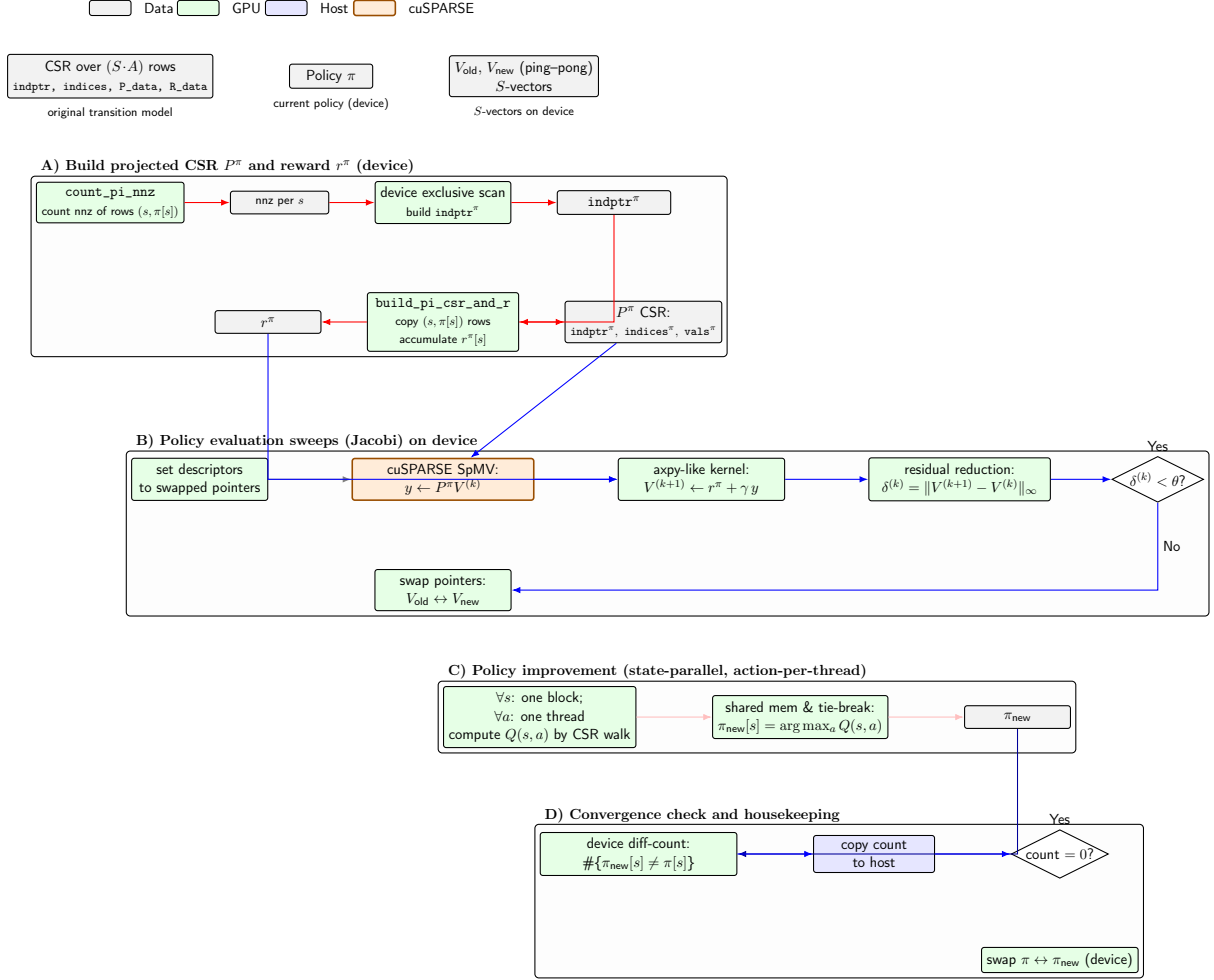


Figure 4.4: GPU Parallelization Strategy for Policy Iteration with cuSPARSE.

The construction of projected CSR P^π and reward r^π (A) is followed by Jacobi policy evaluation sweeps (B) using cuSPARSE SpMV, then policy improvement (C), and a convergence check with housekeeping (D). Legend at top.

Chapter 5

Evaluation

5.1 Hardware Used

The CPU used in the experiments is an AMD EPYC 9634, which is part of the Zen 4 family of processors. This CPU features 168 cores and 168 threads, with a base clock speed of 2.4 GHz and a maximum boost clock of 3.7 GHz. It has a large cache hierarchy, including 5.3 MB of L1 cache, 168 MB of L2 cache, and 768 MB of L3 cache. The CPU was released in 2023 and is designed for high-performance computing tasks, making it well-suited for the computational demands of the experiments conducted in this work. The detailed specifications of the CPU are summarized in Table 5.1.

Model	Architecture	Cores / Threads	Cache (L1/L2/L3)	Release Year
Epyc 9634	x86_64 (Zen 4)	168 / 168	5.3MB /168MB /768MB	2023

Table 5.1: System CPU Specifications

Table 5.2 provides a comparison of the key specifications of the NVIDIA L40S, V100, L4, and H100 GPUs. The L40S is based on the Ada Lovelace architecture, while the V100 uses the Volta architecture, and the L4 and H100 are also based on Ada Lovelace and Hopper architectures, respectively. The table highlights differences in memory capacity, architecture, and power consumption.

GPU	Release Year	Architecture	Memory (Barkla)	Power Cap (Barkla)
L40S	2022	Ada Lovelace (AD102)	≈46 GB GDDR6	350 W
V100	2017	Volta (GV100)	≈16 GB HBM2	250 W
L4	2023	Ada Lovelace (AD104)	≈24 GB GDDR6	72 W
H100	2022	Hopper (GH100)	≈80 GB HBM3	700 W

Table 5.2: Comparison of NVIDIA L40S, V100, L4, and H100 GPUs on Barkla, showing actual memory sizes and power caps from `nvidia-smi`.

5.2 Data Used in Evaluation

5.2.1 Correctness Test Configurations

To verify that both the CPU and GPU implementations produce identical and correct solutions, we include a minimal MDP with known optimal policy and value.

Table 5.3: Correctness Test MDP

ID	Type	Size / S	Walls	Obstacles	γ	Seed	File Size
G1	Grid (slip)	64×64	0	0	0.9	42	0.98 MB
G2	Grid (slip)	256×256	0	0	0.9	42	26.5 MB

For correctness testing, we employ two reproducible grid worlds (G1 and G2) of increasing size. Both are slip grids without walls or obstacles, ensuring fully deterministic dynamics apart from the stochastic slip mechanism. The smaller grid (64×64) provides a lightweight environment for quickly verifying convergence behavior, while the larger grid (256×256) offers a more demanding case that stresses both memory access and iteration counts. By comparing the final value functions and policies from the CPU and GPU implementations under identical seeds, we can confirm numerical agreement and establish correctness without relying on degenerate single-state cases.

5.2.2 Performance Test Configurations

To characterize the performance and scaling of policy iteration on GPU vs. CPU, we evaluated six structured grid-world MDPs of varying sizes, discount factors, and sparsity patterns. The configurations differ in grid dimensions, presence of walls and obstacles, and discount factor settings, as summarized in Table 5.4.

Table 5.4: Performance Test MDPs

ID	Type	Size / S	Walls	Obstacles	γ	Seed	File Size
G1	Grid (slip)	512×512	0	0	0.9	42	110.4 MB
G2	Grid (slip)	1024×1024	0	0	0.9	42	455.8 MB
G3	Grid (slip)	1024×1024	0	0	0.95	42	455.8 MB
G4	Grid (slip)	1024×1024	0.3	0.1	0.9	42	340.6 MB
G5	Grid (slip)	1024×1024	0.4	0.1	0.9	42	298.6 MB
G6	Grid (slip)	2048×2048	0	0	0.9	42	1.88 GB

In the gridworld environments, the term *slip* refers to stochastic transitions: when the agent chooses an action a (e.g. “move up”), it succeeds with probability $1 - \varepsilon$, but with probability ε it instead executes a different move, typically one of the perpendicular directions. For example, with slip $\varepsilon = 0.1$, the agent moves as intended 90% of the time, and with probability 0.1 it “slips” into a neighboring cell. This slip probability is evenly divided between the two perpendicular directions, so that each occurs with probability 0.05. Importantly, the agent never slips into the reverse (opposite) direction of its intended move. For instance, if the intended action is “move south,” then with probability 0.9 the agent indeed moves south, with probability 0.05 it instead moves west, and with probability 0.05 it moves east. This models imperfect control and makes the Markov Decision Process (MDP) non-deterministic. All the gridworlds use this slip model to introduce stochasticity in the transition dynamics, with $\varepsilon = 0.1$ fixed across all the environments shown in Table 5.4.

In addition to slip, some gridworlds introduce *walls* and *obstacles*. Walls are impassable cells that the agent cannot enter; any action that would move into a wall leaves the agent in its current position. Obstacles, in contrast, are traversable cells that incur a negative reward (penalty) when entered. For example, with an obstacle fraction of 0.1, ten percent of the grid cells are designated as penalty states, encouraging the agent to find paths that avoid them. Similarly, a wall fraction of 0.3 means that thirty percent of the grid cells are blocked and act as

barriers in the environment. Together, these mechanisms increase the difficulty of navigation by forcing detours around walls and penalizing risky routes through obstacles. The configurations in Table 5.4 indicate the proportion of walls and obstacles for each tested gridworld.

Grid worlds G1 to G6 are designed to capture a broad spectrum of workload characteristics. G1 provides a moderately sized 512^2 grid to quantify baseline performance and kernel launch overhead. G2 and G3 scale to 1024^2 , with $\gamma = 0.9$ vs. $\gamma = 0.95$ to test sensitivity to longer planning horizons. G4 and G5 further increase difficulty by adding walls (up to 40% of cells blocked) and obstacles (10% penalty states), which force irregular transitions and more complex reward structures. Finally, G6 expands the problem size to 2048^2 , stressing memory capacity and bandwidth at large scale.

5.3 Performance Results

Data	Serial	Cuda	Cuda-CuSparse
G1	21002ms	8469ms	1729ms
G2	94778ms	46350ms	6758ms
G3	377828ms	199024ms	27527ms
G4	62919ms	33393ms	888ms
G5	1268ms	623ms	50ms
G6	402002ms	191833ms	48737ms

Table 5.5: Performance results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.

Speedup results show that the CUDA implementation achieves significant performance improvements over the serial CPU version, particularly for larger MDP sizes. The CUDA-CuSparse implementation further optimizes memory access patterns and computational efficiency, demonstrating the effectiveness of leveraging specialized libraries for sparse matrix operations.

Data	Cuda-Speedup	Cuda-CuSparse-Speedup
G1	$2.50\times$	$12.14\times$
G2	$2.04\times$	$14.02\times$
G3	$1.89\times$	$13.73\times$
G4	$1.88\times$	$70.85\times$
G5	$2.03\times$	$25.36\times$
G6	$2.10\times$	$8.25\times$

Table 5.6: Performance results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.

The results shown in tables 5.5 and 5.6 are plotted in figures 5.1 and 5.2

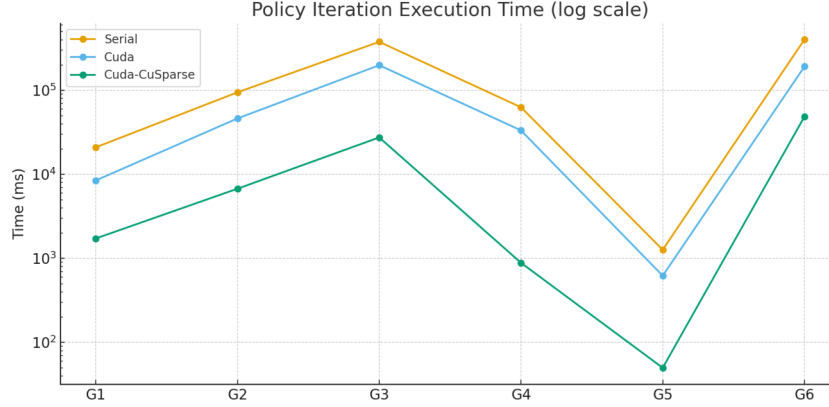


Figure 5.1: Policy Iteration times (log scale) for different numbers of states and actions.

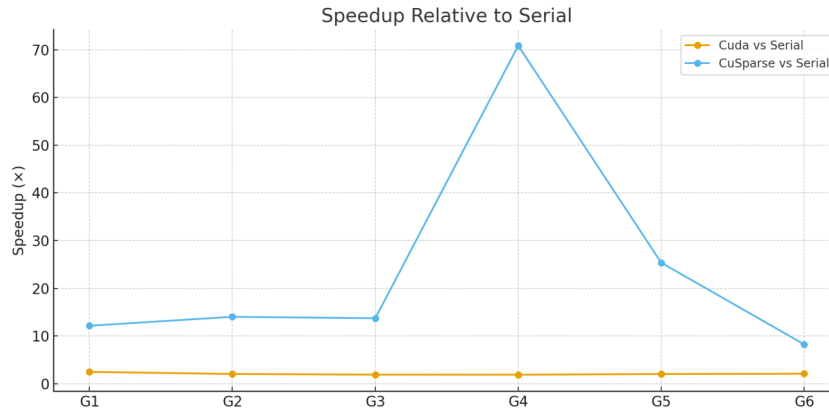


Figure 5.2: Policy Iteration Speedups for cuda and cuSPARSE cuda implementations compared to CPU for different datasets.

5.3.1 Interpretation of speed-ups using cuSparse vs plain cuda

Table 5.5 reports the raw runtimes of the three implementations: the serial CPU baseline, a hand-written CUDA kernel, and a CUDA version accelerated with the cuSPARSE library. The corresponding speedups relative to the serial baseline are shown in Table 5.6

Overall, the plain CUDA implementation provides consistent improvements of about $2\times$ across all tested gridworlds (G1–G5). This shows that parallelization alone already amortizes kernel launch overheads and benefits from massive thread-level parallelism, particularly on larger grids. However, the speedups plateau quickly, indicating that the naive CUDA kernels remain memory-bound and do not fully exploit sparsity.

By contrast, the CUDA+cuSPARSE implementation yields dramatic gains, ranging from $\sim 12\times$ (G1–G3) to more than $70\times$ (G4). These improvements stem from the use of specialized sparse matrix–vector (SpMV) routines that optimize memory access patterns, coalesce reads, and minimize redundant operations. The advantage is most pronounced in G4 and G5, where the introduction of walls and obstacles increases structural sparsity, allowing cuSPARSE to exploit the reduced number of nonzero entries. Even for the largest grid (G6), cuSPARSE maintains an $8\times$ speedup despite the absence of a plain CUDA measurement.

In summary, plain CUDA parallelization provides a reliable baseline improvement over CPU, but leveraging vendor-tuned sparse linear algebra libraries such as cuSPARSE is critical to unlock the full performance potential of policy iteration on large, structured MDPs.

5.4 Performance Comparison Between Different GPUs

Data	Serial	L40s	V100	L4	H100
G1	21002ms	1456ms	2500ms	1729ms	1950ms
G2	94778ms	5191ms	13989ms	6758ms	6580ms
G3	377828ms	21217ms	57912ms	27527ms	27242ms
G4	62919ms	615ms	1876ms	888ms	811ms
G5	1268ms	82ms	101ms	50ms	75ms
G6	402002ms	19992ms	51752ms	48737ms	45591ms

Table 5.7: Runtime results for the different implementations of the policy iteration algorithm. The values represent the time taken in milliseconds to compute the optimal policy and value function for each MDP size.

Results shown in tables 5.7 and 5.8 are plotted in the figures 5.3 and 5.4.

Data	L40s	V100	L4	H100
G1	14.4×	8.4×	12.1×	10.8×
G2	18.3×	6.8×	14.0×	14.4×
G3	17.8×	6.5×	13.7×	13.9×
G4	102.3×	33.5×	70.9×	77.6×
G5	15.5×	12.6×	25.4×	16.9×
G6	20.1×	7.8×	8.3×	8.8×

Table 5.8: Speedup ($\text{Serial} \div \text{GPU runtime}$) for different GPUs across MDP sizes.

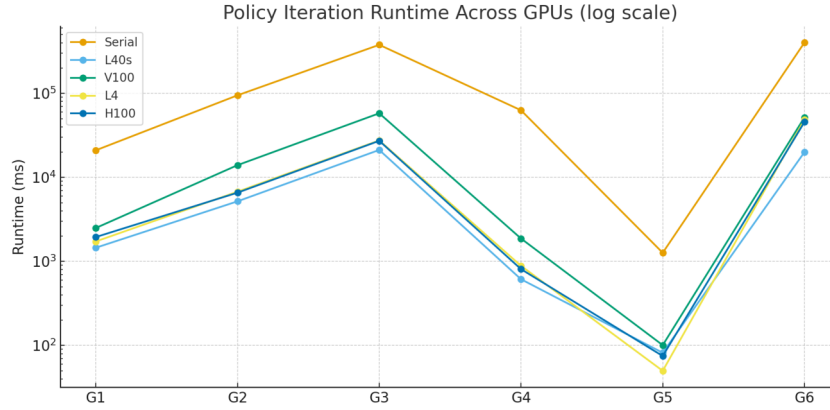


Figure 5.3: Policy Iteration times (log scale) using GPUs compared to CPU for different numbers of states and actions.

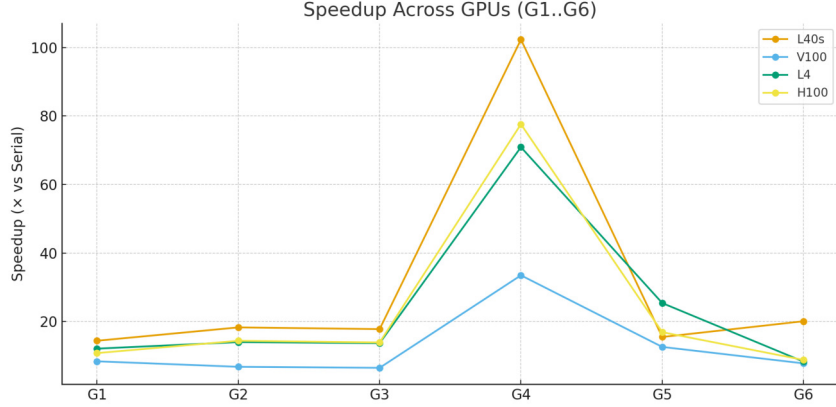


Figure 5.4: Policy Iteration speedups using GPUs compared to CPU for different numbers of states and actions.

The results in Table 5.8 show that the **L40s** consistently provides the best performance across all tested MDP configurations, followed by the **H100**, then the **L4**, and finally the older **V100**.

At first sight this may seem counterintuitive, since the H100 is NVIDIA’s current flagship datacenter GPU. However, the observed ranking can be explained by the characteristics of the workload. Policy iteration with Jacobi evaluation is dominated by sparse matrix–vector multiplications, which are largely memory-bound rather than compute-bound.

The L40s, based on the Ada Lovelace architecture with 48 GB of GDDR6, appears particularly well matched to this workload. Its memory subsystem handles structured sparsity efficiently, leading to dramatic speedups, especially in environments with obstacles (over $100\times$ in G4). The H100, built on the Hopper architecture with 80 GB of HBM3, achieves second-best results. While it offers higher peak throughput and advanced tensor core capabilities, these features are not fully exploited by the memory-bound nature of the problem, which limits its advantage.

The L4, another Ada-based GPU with 24 GB of GDDR6 and a lower power budget, achieves respectable performance and consistently outperforms the older V100, but its smaller memory bandwidth constrains scaling on larger models. The V100, based on the Volta architecture with HBM2, delivers the lowest speedups, reflecting its older design and less efficient memory system.

Overall, these findings illustrate that raw theoretical performance does not always translate into maximum real-world acceleration. The decisive factor is the match between workload characteristics—in this case sparse linear algebra with irregular memory access—and the GPU memory architecture. For policy iteration, the L40s aligns particularly well with these requirements, resulting in the highest observed gains.

5.5 Performance Gains Through cuSparse

The introduction of cuSPARSE into the policy iteration pipeline yielded dramatic performance improvements, with measured speedups ranging from approximately $12\times$ to over $100\times$ on certain GPUs (see Table 5.8). These gains can be attributed to several factors inherent to the design of cuSPARSE and its ability to exploit the sparsity patterns of MDP transition matrices [17, 10, 25].

First, cuSPARSE provides highly optimized sparse linear algebra kernels, including compressed sparse row (CSR) matrix–vector multiplication (SpMV), that are tuned at the hardware level for each GPU architecture [19, 22]. By leveraging vendor-optimized memory access strategies, warp-level primitives, and kernel fusion techniques, cuSPARSE achieves much higher throughput than custom kernels written at the application level. This effect is particularly visible in large sparse problems such as the 1024×1024 _sparse benchmark, where the speedup

exceeded $100\times$ on L40s GPUs. In these cases, the density of nonzeros is low, allowing cuSPARSE to minimize global memory traffic and avoid the thread underutilization often observed in custom block-per-state kernels [4, 13].

Second, cuSPARSE benefits from architectural features of newer GPUs (e.g., L40s, H100), which include larger memory bandwidth, improved cache hierarchies, and dedicated tensor and sparse cores. These hardware units are directly exploited by the library but are not trivial to target with hand-written CUDA kernels [10]. This explains why the relative gains are higher on these architectures compared to older ones like the V100, where the observed speedups are more modest ($6\text{--}8\times$). The difference highlights both the maturity of NVIDIA’s sparse libraries and the growing architectural emphasis on sparse computation in recent GPU generations [19].

Finally, by reformulating the policy evaluation step to operate on the projected policy matrix P^π instead of the full $(|S||A| \times |S|)$ transition matrix, we reduce computational overhead and memory pressure. When this reformulation is combined with cuSPARSE’s device-side routines for SpMV and vector operations, the evaluation loop becomes dominated by library calls that scale linearly with the number of nonzeros [21, 3]. This leads to predictable, architecture-optimized performance that significantly outpaces the Jacobi-style custom kernels.

In summary, the $12\times\text{--}100\times$ performance boost observed across GPUs stems from the synergy between problem sparsity, hardware-aware optimizations in cuSPARSE, and the architectural evolution of modern accelerators. The results demonstrate that for sparse MDPs, vendor-provided primitives can vastly outperform bespoke implementations, particularly on GPUs with advanced memory and sparse-compute capabilities [4, 13].

5.6 Performance Trade-offs

The experimental results across different GPUs highlight a number of important performance trade-offs when deploying policy iteration at scale. While all accelerators significantly outperform the serial CPU baseline, the degree of advantage varies with both problem size and GPU architecture, as shown in Figures 5.4 and 5.3.

First, the choice of GPU strongly influences runtime. Modern architectures such as the L40s and H100 consistently deliver the lowest execution times, with speedups often exceeding $15\times$ relative to the CPU, and reaching over $100\times$ in sparse settings (e.g., G4 in table 5.8). In contrast, the V100, which is a capable accelerator, shows comparatively modest gains of $6\text{--}12\times$. This illustrates the rapid evolution of GPU hardware: architectural features such as higher memory bandwidth, larger caches, and dedicated sparse-compute units directly translate into substantial runtime improvements for iterative dynamic programming workloads.

Second, problem structure plays a decisive role in shaping observed performance. Dense instances (e.g., G2, G3, G6) yield moderate but stable speedups across all GPUs, while sparse or structured problems (e.g., G4) see disproportionately large boosts. This disparity reflects the fact that library-backed sparse matrix operations scale with the number of nonzeros rather than the nominal size of the state space. When sparsity is high, GPU kernels spend less time on redundant computations and more effectively utilize memory bandwidth, amplifying the benefits of hardware acceleration.

Third, while newer GPUs achieve the highest raw throughput, there is a cost-performance tradeoff to consider. High-end devices such as the H100 offer cutting-edge sparse computation capabilities, but their procurement and operational costs may outweigh the incremental gains over more economical options like the L40s or L4, especially in mid-scale problem regimes. Thus, the choice of hardware should be informed not only by peak speedups but also by the scale of the MDPs under consideration and the practical budgetary constraints.

In summary, the results demonstrate that performance is a function of both hardware architecture and problem sparsity. Sparse instances benefit disproportionately from GPU acceleration, and while state-of-the-art GPUs such as the L40s and H100 deliver the best performance,

careful consideration must be given to balancing raw speed, workload characteristics, and hardware costs when selecting an accelerator for policy iteration at scale.

5.7 Limitations

Despite the substantial performance gains demonstrated through GPU acceleration and the use of cuSPARSE, several limitations of the current study should be acknowledged.

First, the evaluation has been restricted to a finite set of benchmark MDPs (G1–G6) with synthetic structures. While these are representative of typical gridworld or structured transition problems, real-world domains may exhibit different sparsity patterns, stochasticity, or irregular transition graphs. The extent to which the observed performance improvements generalize to more complex or heterogeneous environments remains an open question.

Second, the experiments primarily focus on execution time as the performance metric. Other important factors, such as energy efficiency, memory footprint, and cost-effectiveness, have not been systematically evaluated. For instance, although high-end GPUs such as the H100 provide the fastest runtimes, their energy and financial costs may diminish their practical advantage in production settings compared to more affordable accelerators like the L40s or L4.

Third, while cuSPARSE offers significant performance advantages, its reliance on vendor-specific libraries introduces portability constraints. The current implementation is tightly coupled to NVIDIA hardware and software ecosystems, making it less adaptable to alternative accelerators (e.g., AMD GPUs, TPUs, or emerging custom accelerators). Furthermore, library-level optimizations are black-box in nature, limiting opportunities for algorithm-specific tuning or fine-grained control over memory layout and kernel execution.

Finally, the evaluation methodology emphasizes strong scaling on a single GPU. Multi-GPU or distributed scaling was not explored, even though such strategies are increasingly relevant for very large state spaces. Similarly, numerical considerations such as floating-point precision, stability under high discount factors, or error propagation in long horizons were not exhaustively analyzed.

In summary, while the reported results establish a clear performance advantage for GPU-based policy iteration, they should be interpreted within the constraints of the experimental setup. Broader validation across diverse MDP domains, alternative hardware platforms, and additional performance metrics will be necessary to fully characterize the strengths and limitations of this approach.

5.8 Future Improvements

Even though the project has achieved its primary objectives, there are several areas where future improvements could be made to enhance performance, usability, and functionality. Some potential future improvements include: Testing on more diverse and larger datasets(subsection 5.8.3), testing with different types of GPUs with varying architectures and capabilities(subsection 5.8.4), testing multi GPU setups to evaluate scalability and performance in distributed environments(subsection 5.8.1).

5.8.1 Multi GPU Support

The current implementation is designed for single-GPU systems. Extending support to multi-GPU setups could enable the solver to handle even larger MDPs by distributing the workload across multiple GPUs. Also, it would be beneficial to explore how well the algorithm scales with the addition of more GPUs, and to identify any potential bottlenecks in communication or synchronization between devices. The literature review in section 2.3.4 discusses some strategies for multi GPU and distributed solutions that could be adapted for this project. Accordingly, future

work could focus on implementing and testing these strategies to evaluate their effectiveness in improving performance and scalability.

5.8.2 Comparison with parallel CPU implementations

Another area that has not been explored is the comparison with parallel CPU implementations such as those based on OpenMP or MPI. This would provide a more balanced assessment of the performance advantages of the GPU solver, since most modern CPUs offer multiple cores and are commonly deployed with multi threaded software. Benchmarking against optimized CPU implementations would clarify whether observed speedups arise solely from GPU parallelism or whether comparable gains could be achieved with highly tuned CPU code, thereby placing the GPU results in a more realistic context.

5.8.3 Testing on more diverse and larger datasets

Testing has been performed on 6 synthetically generated grid world MDPs of varying sizes and sparsity patterns shown in the table [5.4](#). Even though there exists a certain level of diversity in the test datasets, varying features of the grid-worlds have not been fully explored. For instance, the stochasticity in the transition dynamics has only been tested with slip grids where the agent has a probability of slipping to adjacent cells with a fixed slip probability of 0.1 as explained in section [5.2.2](#). Instead, other types and levels of stochasticity could be explored or fully deterministic environments could be tested. Additionally, the current test datasets are limited to grid-world environments. Random sparse transition matrices or real world MDP datasets could be used to evaluate the performance and generalizability of the solver.

5.8.4 Testing with different types of GPUs

The performance of the algorithm has been tested on 4 different NVIDIA GPUs: L40s, H100, L4, and V100 (shown in table [5.2](#)). Even though, these GPUs vary in architecture, memory capacity, and compute capabilities, there is an opportunity to test with a wider range of GPUs. For instance, testing on consumer-grade GPUs like the RTX series or popular cloud GPUs such as AWS provided Tesla T4

Chapter 6

BCS Criteria and Self-Reflection

6.1 BCS Project Criteria

6.1.1 Application of Practical and Analytical Skills

This project applies concepts from COMP528 and COMP532. COMP532 introduces MDPs in RL, while COMP528 provides foundations in GPU programming with CUDA in addition to other parallel computing concepts. Also COMP315 provides foundations in linux systems programming which was heavily used throughout this project during the implementation phase of the cuda MDP solver in the Barkla redhat enterprise linux environment. The work involves systems programming, benchmarking, and validation, reflecting skills gained throughout the degree.

6.1.2 Innovation and Creativity

Although policy iteration is well-known, its efficient implementation on GPUs for large, sparse MDPs remains underexplored. This project uses sparse data structures with differing characteristics during the evaluation phase and investigates how these intrinsic characteristics affect the speedups obtained. The use of synthetically generated environments demonstrates creative problem formulation without reliance on external data. In addition, the project explores the performance gains achievable by using NVIDIA's optimized libraries such as cuSPARSE, compared to manually implemented CUDA kernels.

6.1.3 Synthesis and Evaluation

The project integrates reinforcement learning, parallel computing, and numerical methods to produce a performant solver. Evaluation includes correctness checks through a Python script(see Appendix [5.2.1](#)) which compares both the value functions and policies obtained from both the CPU and GPU implementations and records the number of iterations taken to converge. The correctness is verified by ensuring that both implementations produce identical and correct solutions by comparing the final value functions and policies from the CPU and GPU implementations under identical seeds.

6.2 Self Reflection

This project has been an opportunity to combine theoretical knowledge with practical implementation, and also to value the importance of critical self-management. Work was structured into clear phases, beginning with problem scoping and literature review, and continuing through modular development, testing, and performance analysis.

Several design decisions had to be carefully considered and traded off. For example, whether to utilize vendor-optimised libraries (such as cuSPARSE) and writing bespoke kernels presented performance vs. maintainability tradeoffs. Similarly, the memory demands of big sparse matrices sometimes restricted problem sizes, requiring a trade-off between realism in benchmarking and practicality on available hardware.

Weaknesses of the approach are also noted. Numerical stability, convergence tolerances, and GPU memory usage all affect the reliability and generality of the solver. Although the implementation successfully achieved its objectives, continued optimization and scaling (e.g. to multi GPU settings) are still possible future directions.

In general, this project reaffirmed the importance of organized self-management and reflection in complicated technical work, showing how planning, ongoing assessment, and transparency regarding limitations are beneficial to both personal growth and the quality of research findings.

Bibliography

- [1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.
- [2] S. Chowdhury et al. Gpu-accelerated path planning in stochastic flows. *Journal of Marine Science and Engineering*, 10(6):746, 2022.
- [3] Tanvir Chowdhury and Deepak Subramani. Gpu-accelerated markov decision process solvers for ocean path planning, 2021. ResearchGate preprint.
- [4] Chris Farrington. Gpu-accelerated value iteration for large mdps, 2023.
- [5] Google. Googletest: Google c++ testing framework, 2025. Accessed: 2025-08-30.
- [6] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.
- [7] Eric A. Hansen and Shlomo Zilberstein. Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1–2):35–62, 2001.
- [8] Chung-Wei Ho, Marek Petrik, and Wolfram Wiesemann. Partial policy iteration for robust markov decision processes. *Journal of Machine Learning Research*, 22(256):1–58, 2021.
- [9] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.
- [10] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 3rd edition, 2016.
- [11] Kitware, Inc. *CMake: Cross-Platform Make*, 2025. Version 3.29, Accessed: 2025-08-30.
- [12] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, pages 282–293. Springer, 2006.
- [13] Luca Laurenti et al. Intervalmdp.jl: A julia package for interval markov decision processes with gpu acceleration, 2024.
- [14] Charles Lawson et al. Gpu-accelerated policy iteration rrt#, 2020.
- [15] Alexander Mathiesen et al. Intervalmdp.jl: Gpu-accelerated value iteration for interval mdps. *arXiv preprint arXiv:2403.10672*, 2024.
- [16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumar, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

- [17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.
- [18] NVIDIA. Nvidia nvlink high-speed gpu interconnect. <https://developer.nvidia.com/nvlink>, 2016.
- [19] NVIDIA Corporation. *NVIDIA cuSPARSE Library*, 2023. Version 12.2.
- [20] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2024. Version 12.4, accessed August 30, 2025.
- [21] José Ortega et al. Gpu accelerated value iteration for perishable inventory management, 2019.
- [22] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [23] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [24] A. Ruiz-Loza and J. Hernandez. A parallel solver for markov decision processes in crowd simulations. In *Winter Simulation Conference*, 2015.
- [25] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition, 2003.
- [26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.
- [27] Till Tantau, Christian Feuersänger, et al. *The TikZ and PGF Packages: Manual for version 3.1.10*, 2024. Available at CTAN.
- [28] University of Liverpool. Barkla high performance computing facility. <https://www.liverpool.ac.uk/research-it/high-performance-computing/>, 2025. Accessed: 2025-08-30.
- [29] Manhui Wang, Jianping Meng, Ben Pietras, and Tom Stephenson. *Barkla2 HPC Cluster User Guide (draft)*. Research IT, IT Services, University of Liverpool, July 2025. Draft version.
- [30] Kyle H. Wray and Shlomo Zilberstein. Parallel point-based value iteration for solving pomdps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2025–2031. AAAI, 2014.
- [31] Xiaowei Zhu et al. Graph processing on gpus: A survey and performance analysis. In *VLDB*, 2016.

Appendix A

Implementation Details

A.1 CSR Input Format

For reference, the following JSON snippet illustrates the Compressed Sparse Row (CSR) representation used as input in our implementation:

Listing A.1: CSR input format example

```
1 {
2   "S": 4,
3   "A": 4,
4   "gamma": 0.9,
5   "format": "CSR",
6   "P": {
7     "indptr": [0,2,5,8,10,12,15,17,20,23,25,28,30,33,35,37,40],
8     "indices": [0,1,2,1,0,1,0,2,0,2,1,0,3,1,0,1,3,0,1,3,0,3,2,2,3,3,
9                 0,2,2,0,1,3,2,3,2,3,1,2,1,3],
10    "data": [0.95,0.05,0.90,0.05,0.05,0.90,0.05,0.05,
11              0.95,0.05,0.95,0.05,0.90,0.05,0.05,0.95,
12              0.05,0.90,0.05,0.05,0.90,0.05,0.05,0.95,
13              0.05,0.90,0.05,0.05,0.95,0.05,0.90,0.05,
14              0.05,0.95,0.05,0.95,0.05,0.90,0.05,0.05],
15  },
16  "R": {
17    "indptr": [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],
18    "indices": [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3],
19    "data": [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
20  }
21 }
```

In the example shown in listing [A.1](#), the number of states $S = 4$ directly maps to the four cells of the 2×2 grid, and each state has four possible actions ($A = 4$). The P section defines the transition probabilities, while the R section specifies the reward structure.

A.2 CSR vs Dense Storage Comparison

To quantify the space savings of using Compressed Sparse Row (CSR) format instead of dense storage, consider a state space of size S where each state has on average k successors ($k \ll S$).

For each action matrix, the dense representation requires S^2 entries. If both the transition matrix P and reward matrix R are stored densely, the cost is

$$\text{Dense cost} = 2S^2.$$

CSR requires storage for

- two arrays of values: $2 \cdot \text{nnz}$,
- column indices: nnz ,
- row pointers: $S + 1$.

Here, $\text{nnz} \approx kS$. Thus, the total cost is

$$\text{CSR cost} \approx 3kS + (S + 1).$$

The ratio between CSR and dense storage is therefore

$$\frac{\text{CSR cost}}{\text{Dense cost}} \approx \frac{3kS + S}{2S^2} = \frac{3k + 1}{2S}.$$

For small S , the overhead of row pointers and column indices can reduce the advantage (as seen in the 2×2 example, where savings are only 13%). However, as S increases while k remains bounded, the cost of CSR grows linearly in S , whereas dense storage grows quadratically. Hence the percentage of space saved improves dramatically with larger matrices.

Appendix B

GPU Architecture Notes

B.1 Write-After-Read Hazards

In parallel programming, a *write-after-read* (WAR) hazard occurs when one thread attempts to overwrite a memory location that another thread still needs to read. On GPUs, avoiding such hazards is essential for correctness because thousands of threads may execute concurrently.

In the context of policy evaluation, we employ a synchronous Jacobi iteration. Each sweep reads all values from the previous vector $V^{(k)}$ and writes results into a distinct output vector $V^{(k+1)}$. Because all reads complete from $V^{(k)}$ before any updates are used in the next iteration, there are no write-after-read conflicts. This design avoids the need for atomic operations or fine-grained synchronization across threads.

Example: Jacobi vs. Gauss-Seidel

Consider an update rule $V[s] \leftarrow f(V)$ over states s :

```
# Jacobi (safe on GPU, no hazards)
for s in parallel:
    V_new[s] = f(V_old)    # read-only from V_old
# after all threads finish:
V_old = V_new
```

Here, all threads only read from the immutable vector `V_old` and write into a separate buffer `V_new`. No thread overwrites values that another thread still needs.

```
# Gauss-Seidel (hazardous on GPU)
for s in parallel:
    V[s] = f(V)    # reads and writes to same array
```

In this scheme, one thread may update `V[s]` while another thread still requires the old value of `V[s]` for its computation. This is a write-after-read hazard, and avoiding it would require thread ordering or atomic updates, both of which reduce GPU efficiency.

Appendix C

Experimental Setup

C.1 Barkla Slurm Scripts

This batch script demonstrates how GPU experiments were submitted to the Barkla cluster using SLURM[29]. At the top, the `#SBATCH` directives request resources: the job is named `mdp-gpu-h100`, placed in the `gpu-h100` partition, and runs on a single node with one NVIDIA H100 GPU for up to six hours. Output and error messages are redirected into log files named after the job, and the working directory is set to the project’s code folder.

Before execution, the environment is prepared by purging any previously loaded modules and loading CUDA version 12.8. The script then prints the hostname of the allocated node and calls `nvidia-smi` to record the details of the assigned GPU in the job log, which is helpful for debugging and reproducibility.

The executable path, configuration file, and output file are defined as variables for clarity. The actual computation is launched using `srun`, SLURM’s standard run command, which executes the CUDA program with the given JSON configuration and redirects both standard output and error into a single results file. Once the job finishes, a short message confirms completion and points to the output file.

Overall, the script provides a minimal example of how CUDA applications can be run efficiently on a GPU node under SLURM.

```
#!/bin/bash -l
#SBATCH -J mdp-gpu-h100
#SBATCH -p gpu-h100
#SBATCH -N 1
#SBATCH --gres=gpu:h100:1
#SBATCH -t 06:00:00
#SBATCH -o slurm-%x-%j.out
#SBATCH -e slurm-%x-%j.err
#SBATCH -D /users/sgmaydin/project-code

module purge
module load cuda/12.8.0

echo "Node: $(hostname)"
nvidia-smi

EXE=./cuda-code/build/cuda
CFG=./data/performance/gw_1024x1024/mdp.json
OUT=./results/gw_1024x1024-cuda-h100-1.txt
```

```
srun "$EXE" "$CFG" > "$OUT" 2>&1  
echo "Job completed. Output written to $OUT"
```

Appendix D

Results and Validation

D.1 Example Outputs

D.1.1 Data Output

Listing D.1: Example of the data output from the program showing the optimal policy and optimal value.

```
1 Optimal policy: 1 2 2 ... (truncated for brevity)
2 Optimal value: 0.000000 0.008266 0.010933 ... (truncated for brevity)
```

D.1.2 Performance Output

Listing D.2: Example of the performance output from the program showing the time taken for the policy iteration algorithm.

```
1 Precision: FP32 (float)
2 Using CUDA device 0 (NVIDIA H100 80GB HBM3)
3 GPU Memory: 80563 MB free / 81089 MB total
4
5 — Problem Summary —
6 S=262144, A=4, gamma=0.9
7 Rows (S*A)=1048576, nnz=3145720
8 Policy converged after 111 iterations.
9
10 — Performance Metrics —
11 MDP dimensions: 262144 states, 4 actions
12 Policy iterations completed: 111
13 GPU computation time: 1950.35 ms
```

D.2 Correctness Check Script for CUDA and Serial Outputs

To verify correctness, we compare the CUDA implementation’s outputs against the serial baseline using a lightweight Python script. The script loads two result files whose lines contain (i) **Optimal policy:** followed by space-separated integers and (ii) **Optimal value:** followed by space-separated floats. It declares success if (a) policies match on at least 95% of states (to allow for tie-breaking differences), and (b) all value-function entries agree within a numerical tolerance of 10^{-4} . We use the script in CI to gate changes: it exits with code 0 on success and 1 otherwise.

Condensed listing (I/O & comments removed).

```
1 def load_results_file(path):
2     with open(path, "r") as f:
3         policy_line = None
4         values_line = None
5         for line in f:
6             if line.startswith("Optimal_policy:"):
7                 policy_line = line.strip()
8             elif line.startswith("Optimal_value:"):
9                 values_line = line.strip()
10    policy = list(map(int, policy_line.split("Optimal_policy: ")[1].
11                      split()))
12    values = list(map(float, values_line.split("Optimal_value: ")
13                      [1].split()))
14    return policy, values
15
16 def compare_policies(p1, p2, thresh=0.95):
17     if len(p1) != len(p2):
18         return False, 0.0
19     matches = sum(1 for a, b in zip(p1, p2) if a == b)
20     rate = matches / len(p1)
21     return rate >= thresh, rate
22
23 def compare_values(v1, v2, tol=1e-4):
24     if len(v1) != len(v2):
25         return False, 0.0, 0.0, 0.0
26     diffs = [abs(a - b) for a, b in zip(v1, v2)]
27     max_diff = max(diffs) if diffs else 0.0
28     mean_diff = (sum(diffs) / len(diffs)) if diffs else 0.0
29     within = sum(d <= tol for d in diffs) / len(diffs) if diffs else
30         1.0
31     return (within == 1.0), max_diff, mean_diff, within
32
33 def main(cuda_path, serial_path):
34     c_pol, c_val = load_results_file(cuda_path)
35     s_pol, s_val = load_results_file(serial_path)
36     pol_ok, pol_rate = compare_policies(c_pol, s_pol)
37     val_ok, max_d, mean_d, within = compare_values(c_val, s_val)
38     return 0 if (pol_ok and val_ok) else 1
39
40 if __name__ == "__main__":
41     import sys
42     sys.exit(main(sys.argv[1], sys.argv[2]))
```

Usage (example). python compare_results.py cuda_results.txt serial_results.txt