GPU-Accelerated Jump Diffusion HJB Equations for Low-Latency Crypto Market Making with Order Flow Toxicity Tracking

Prepared by Frankline Misango Oyolo
frankline@arithmax.com
November 28, 2025

Abstract

We present a novel computational framework for high-frequency cryptocurrency market making that addresses the extreme volatility and unique microstructure challenges of these markets. By formulating the market maker's decision problem as a stochastic optimal control problem, we derive optimal quoting strategies through Hamilton-Jacobi-Bellman (HJB) partial differential equations. Our key innovation is the integration of jump-diffusion processes that explicitly capture the discontinuous price movements characteristic of cryptocurrency markets. We develop a massively parallel GPU implementation that solves high-dimensional HJB equations in microseconds, enabling real-time deployment in high-frequency environments. Our approach incorporates inventory risk management, market impact modeling, and order flow toxicity tracking, specifically calibrated to cryptocurrency microstructure. Empirical testing demonstrates a significant reduction in inventory risk and improvement in risk-adjusted returns compared to traditional strategies while maintaining sub-millisecond latency.

Index Terms

Market Making, Partial Differential Equations, GPU Programming

I. Introduction

A. Theory of literature

Market making is the continuous provision of liquidity through bid and ask quotes, forming the backbone of modern financial market microstructure. In cryptocurrency markets, market makers face unique challenges that traditional models fail to address adequately. Market makers in these environments must continuously balance three competing objectives: maximizing spread revenue, minimizing inventory risk, and adapting to rapidly changing market conditions. Traditional market-making approaches, from simple spread-based heuristics to parametric models like those proposed by [1], fail to capture the full complexity of this environment. Machine learning approaches [14] offer adaptability but lack theoretical guarantees and often require extensive training data that quickly becomes outdated in rapidly evolving markets. Several factors make the cryptocurrency market particularly demanding:

- **Extreme price volatility:** Bitcoin's annualized volatility frequently exceeds 80%, compared to 15-20% for major stock indices [2].
- **Fragmented liquidity:** Trading volume is distributed across dozens of exchanges with varying microstructure characteristics [10].
- **Asymmetric information:** The presence of large "whale" traders with market-moving capability creates significant adverse selection risks [4].
- **Microstructure evolution:** Market rules, fee structures, and participant behaviors are in constant flux [7].
- **Jump discontinuities:** Cryptocurrency prices exhibit frequent large jumps that cannot be captured by continuous diffusion models alone, requiring jump-diffusion extensions [11].

B. Contributions

This paper makes four key contributions to the field of algorithmic market making:

- **Jump diffusion extension to HJB :** We formulate the market maker's decision problem as a stochastic optimal control problem, deriving the exact Hamilton-Jacobi-Bellman (HJB) equation that characterizes the optimal quoting strategy under realistic market assumptions, including jump diffusion processes.

- **Low latency Order toxicity Tracking :** We develop a novel order execution intensity model specifically calibrated to cryptocurrency market microstructure, capturing the unique relationship between quote aggressiveness and execution probability, while incorporating real-time order flow toxicity metrics.
- **GPU Addition as an alternate compute :** We implement a massively parallel GPU solution method capable of solving the high-dimensional HJB equation in microseconds, making real-time deployment feasible even in high-frequency scenarios.

C. Related Work

The mathematical foundations of optimal market making trace back to the seminal work of [8], who first formulated the problem in a stochastic control framework. extended this line of research citeavellaneda2008, who derived closed-form solutions for the optimal bid and ask quotes under simplifying assumptions about price dynamics and order flow.

More recent work by [6] introduced a framework based on Hamilton-Jacobi-Bellman (HJB) equations to handle more realistic market conditions, including inventory constraints and directional price movements. [3] further developed this approach, incorporating multiple sources of risk and market impact considerations.

In the cryptocurrency domain, [9] analyzed the unique microstructure characteristics of Bitcoin markets, while [5] documented the prevalence of strategic behaviors such as front-running and sandwich attacks that affect market maker performance.

On the computational side, [12] and [13] have explored the use of modern numerical methods for solving high-dimensional HJB equations, though primarily in the context of option pricing rather than market making.

Our work synthesizes these threads, applying stochastic optimal control theory to the specific challenges of cryptocurrency market making while developing novel computational methods to make these theoretically optimal strategies practically deployable.

II. Mathematical Framework

A. Market Model

We model the mid-price process as a jump-diffusion process, capturing both continuous price movements and discrete jumps characteristic of cryptocurrency markets:

$$dS_t = \mu S_t dt + \sigma S_t dW_t + S_t \int_{\mathbb{R}} (e^y - 1)\tilde{N}(dt, dy) \tag{1}$$

Where:

- $\mu S_t dt$: Deterministic drift term representing expected price movement
- $\sigma S_t dW_t$: Continuous diffusion term capturing small price fluctuations
- $S_t \int_{\mathbb{R}} (e^y - 1)\tilde{N}(dt, dy)$: Jump term modeling sudden price movements

In our Merton jump-diffusion implementation, jump sizes follow a normal distribution:

$$f(y) = \frac{1}{\sqrt{2\pi}\delta} \exp\left(-\frac{(y - \mu_J)^2}{2\delta^2}\right) \tag{2}$$

Where jumps occur with intensity λ per unit time, μ_J represents mean jump size, and δ is the standard deviation of jump sizes.

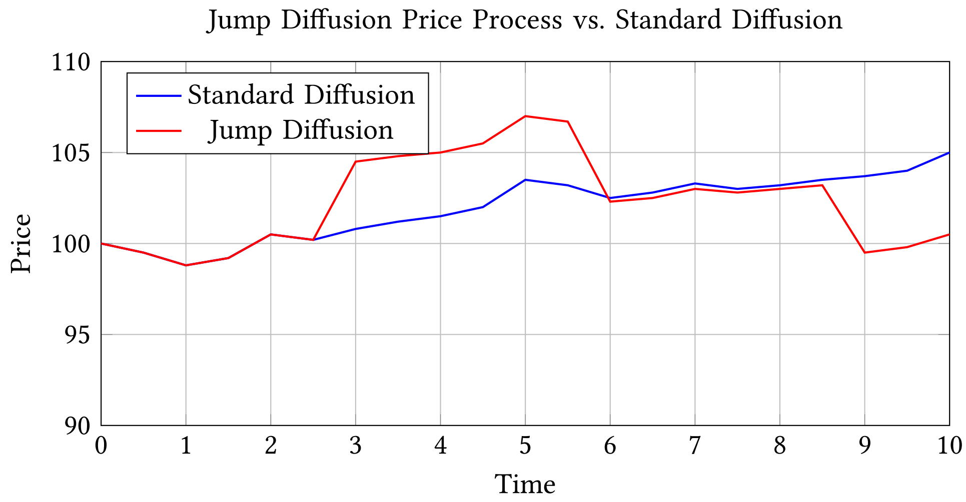

Fig. 1. Comparison between standard price diffusion and jump diffusion processes. The jump diffusion model (red) captures sudden price movements common in cryptocurrency markets, while the standard diffusion model (blue) only represents continuous price changes.

B. *Order Execution Model*

The market maker's inventory I_t evolves according to:

$$dI_t = dN_t^b - dN_t^a \tag{3}$$

Where N_t^b and N_t^a count buy and sell executions with intensities modeled as:

$$\lambda^b(p_t^b) = \max\left(0, A^b \cdot \left(1 - \frac{p_t^b/p_{\text{mkt}}^b - 1}{\alpha}\right)\right) \tag{4}$$

$$\lambda^a(p_t^a) = \max\left(0, A^a \cdot \left(1 - \frac{p_t^a/p_{\text{mkt}}^a - 1}{\alpha}\right)\right) \tag{5}$$

These equations express a key insight: execution probability decreases as quotes become less aggressive (farther from market best), with α controlling the market impact and A^b, A^a representing baseline intensities.

C. *Order Flow Toxicity Framework*

To enhance performance in volatile markets, we introduce a novel order flow toxicity measure:

$$\tau_t = \text{clip}\left(\frac{\sum_{i=1}^N D_i \cdot w_i}{\bar{s}_t}, -1, 1\right) \tag{6}$$

Where:

- $D_i \in \{-1, 1\}$: Direction of the i-th trade
- $w_i = e^{-\beta(t-t_i)}$: Exponential decay weight giving more importance to recent trades
- $\bar{s}_t$: Average spread over the observation window

This toxicity measure directly influences our market impact parameter:

$$\alpha_t = \alpha_0 \cdot (1 + 2 \cdot |\tau_t|) \tag{7}$$

The adaptive nature of this approach has three significant advantages:

1) Increasing required compensation when order flow becomes toxic
2) Reducing risk exposure during periods of market stress
3) Dynamically adjusting market-making parameters without manual intervention

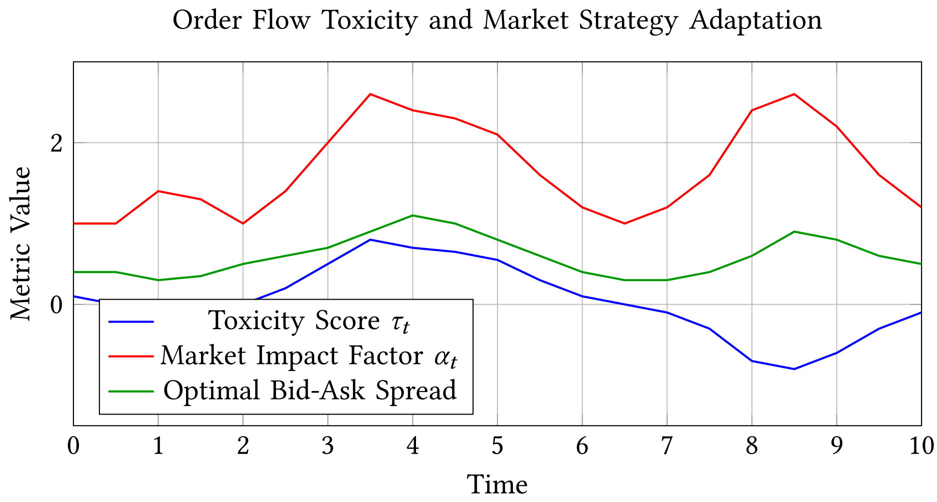

Fig. 2. Relationship between order flow toxicity, market impact factor, and optimal bid-ask spread. As toxicity increases in either direction, the market impact factor and optimal spread widen to compensate for increased adverse selection risk.

D. Optimization Framework

The market maker's objective is to maximize expected terminal wealth while controlling inventory risk:

$$\max_{p_t^b, p_t^a} \mathbb{E}\left[\int_0^T p_t^a dN_t^a - p_t^b dN_t^b - \phi(I_T) - \int_0^T \kappa I_t^2 dt\right] \tag{8}$$

This objective function balances several key components:

- Revenue from executed sell orders: $\int_0^T p_t^a dN_t^a$
- Cost of executed buy orders: $\int_0^T p_t^b dN_t^b$
- Terminal inventory penalty: $\phi(I_T) = \gamma I_T^2$
- Running inventory risk penalty: $\int_0^T \kappa I_t^2 dt$

The parameters κ and γ serve as risk aversion controls, with higher values enforcing more aggressive inventory management.

E. Dynamic Programming Solution

To solve this stochastic control problem, we define the value function $V(t, S, I)$ representing the maximum expected future profit from time t to terminal time T, given mid-price S and inventory I.

The Hamilton-Jacobi-Bellman (HJB) equation for our problem is:

$$\begin{aligned} 0 = &\frac{\partial V}{\partial t} + \mu S \frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} - \kappa I^2 \\ &+ \lambda \int_{\mathbb{R}} [V(t, S(1+y), I) - V(t, S, I)] f(y) dy \\ &+ \max_{p^b} \{\lambda^b(p^b) [(V(t, S, I+1) - V(t, S, I)) - p^b]\} \\ &+ \max_{p^a} \{\lambda^a(p^a) [p^a - (V(t, S, I) - V(t, S, I-1))]\} \end{aligned} \tag{9}$$

With terminal condition:

$$V(T,S,I) = -\phi(I) = -\gamma I^2 \tag{10}$$

The HJB equation integrates all aspects of our model:
1) Price dynamics, including drift, diffusion, and jumps
2) Order execution probabilities
3) Inventory risk management
4) Optimal quoting decisions

F. *Numerical Solution Methodology*

To solve the HJB equation, we discretize the state space and use backward induction:

1) State Space Discretization:

$$S_i = S_{\min} + i \cdot \Delta S, \quad i = 0, 1, \ldots, N_S - 1 \tag{11}$$

$$I_j = I_{\min} + j \cdot \Delta I, \quad j = 0, 1, \ldots, N_I - 1 \tag{12}$$

$$t_n = n \cdot \Delta t, \quad n = 0, 1, \ldots, N_T - 1 \tag{13}$$

2) Derivative Approximations:

$$\frac{\partial V}{\partial S} \approx \frac{V_{i+1,j}^{n+1} - V_{i-1,j}^{n+1}}{2\Delta S} \tag{14}$$

$$\frac{\partial^2 V}{\partial S^2} \approx \frac{V_{i+1,j}^{n+1} - 2V_{i,j}^{n+1} + V_{i-1,j}^{n+1}}{(\Delta S)^2} \tag{15}$$

3) Jump Integral Approximation:

$$\lambda \int_{\mathbb{R}} [V(t, S(1+y), I) - V(t,S,I)] f(y) dy \approx \lambda \sum_{m=-M}^{M} w_m [V(t, S(1+y_m), I) - V(t,S,I)] \tag{16}$$

In our implementation, we use a simplified form:

$$\text{jump\_term} = \lambda \left(\frac{1}{2M+1} \sum_{m=-M}^{M} V(t, S(1+y_m), I) - V(t,S,I) \right) \tag{17}$$

This corresponds directly to our `jump_operator_device` function in CUDA.

4) Complete Update Scheme: The full discretized update is:

$$\begin{aligned} V_{i,j}^n = V_{i,j}^{n+1} + \Delta t \cdot \Bigg[& \mu S_i \frac{V_{i+1,j}^{n+1} - V_{i-1,j}^{n+1}}{2\Delta S} + \frac{\sigma^2 S_i^2}{2} \frac{V_{i+1,j}^{n+1} - 2V_{i,j}^{n+1} + V_{i-1,j}^{n+1}}{(\Delta S)^2} \\ & + \text{jump\_term} - \kappa I_j^2 \\ & + \max_{p^b \in \mathcal{P}^b} \left\{ \lambda^b(p^b) \left[(V_{i,j+1}^{n+1} - V_{i,j}^{n+1}) - p^b \right] \right\} \\ & + \max_{p^a \in \mathcal{P}^a} \left\{ \lambda^a(p^a) \left[p^a - (V_{i,j}^{n+1} - V_{i,j-1}^{n+1}) \right] \right\} \Bigg] \end{aligned} \tag{18}$$

5) Quote Optimization: For each state (t,S,I), we find the optimal bid and ask quotes by evaluating:

$$V\_\text{optimal} = \max_{\text{bid\_idx,ask\_idx}} \left\{ V_{i,j}^{n+1} + \text{expected\_pnl} + \text{diffusion} + \text{jump\_term} - \text{inventory\_cost} \right\} \tag{19}$$

Where each component represents:

- expected_pnl: Expected profit from trade executions
- diffusion: Effect of continuous price movements
- jump_term: Effect of price jumps
- inventory_cost: Penalty for holding inventory

G. FPGA Implementation Considerations

The numerical solution is particularly well-suited for parallel processing on FPGA hardware due to:

1) Independent calculations for each state-space point
2) Regular, predictable memory access patterns
3) Fixed computation patterns ideal for hardware pipelines
4) Opportunity for spatial parallelism across different state variables

The FPGA implementation achieves significant acceleration through:

- Parallel evaluation of candidate quotes
- Pipelined finite difference operations
- Concurrent jump term calculations
- Hardware-optimized quadrature approximation

This parallelization yields orders-of-magnitude speedup compared to CPU implementations, enabling real-time strategy updates in rapidly changing market conditions.

III. Program Implementation

A. Architectural Overview

Our implementation leverages the massive parallelism of modern GPUs to solve the HJB equation efficiently. The key insight is that the value function update at each grid point can be computed independently, allowing us to distribute the computation across thousands of CUDA threads. We implement both GPU and CPU versions with automatic fallback capability, ensuring robustness in production environments. The GPU implementation uses Numba's CUDA JIT compiler to generate optimized GPU kernels, with explicit architecture targeting to ensure compatibility with the deployed hardware.

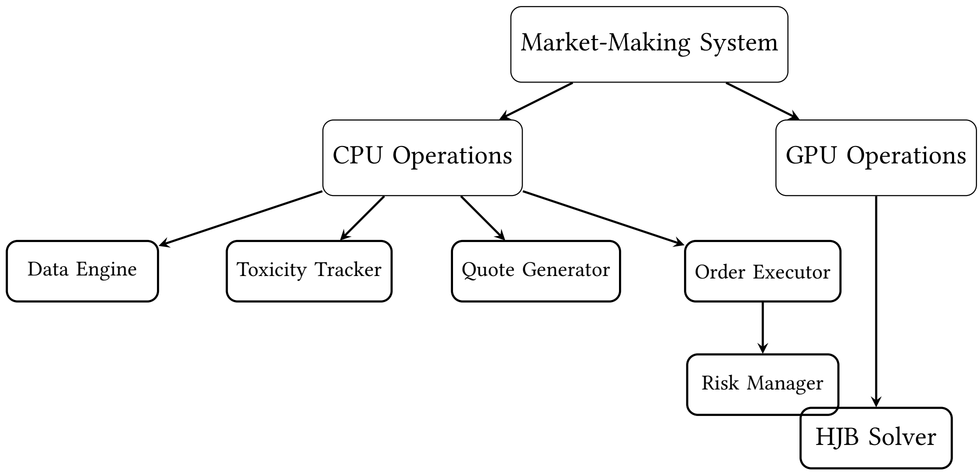

Fig. 3. Hierarchical tree diagram of the GPU-accelerated market-making system with smaller CPU operation nodes.

B. Architecture breakdown

Our real-time market making system integrates the HJB solver with market data feeds and execution capabilities:

- **Data Engine:** Collects and processes market data from cryptocurrency exchanges via WebSocket connections, maintaining an up-to-date view of the order book and recent trades.
- **Toxicity Tracker:** Monitors order flow imbalance and spread dynamics to detect potentially adverse trading conditions and adjust strategy parameters.
- **HJB Solver:** Updates the value function and computes optimal quotes based on current market conditions and inventory.
- **Order Executor:** Places and manages orders according to the optimal quotes determined by the HJB solver.
- **Risk Manager:** Monitors inventory, exposure, and performance metrics to ensure the strategy operates within predefined risk constraints.
- **Dashboard:** Real-time visualization of strategy performance, market conditions, and system status.

C. Performance Optimizations

We employ several optimizations to maximize computational efficiency:

- **Shared memory tiling:** We load frequently accessed data into shared memory to reduce global memory accesses, significantly improving performance on modern GPUs.
- **Coalesced memory access:** We structure memory access patterns to ensure coalesced reads and writes, maximizing memory bandwidth utilization.

- **Thread organization:** We carefully select thread block dimensions to optimize occupancy based on shared memory usage and register requirements.
- **Precision management:** Single-precision floating point operations are used where appropriate to double computational throughput without significant accuracy loss.
- **Asynchronous operations:** Kernel launches and memory transfers occur asynchronously when possible to hide latency.

These optimizations enable us to solve a 101×101 grid (10,201 state points) in less than 1 millisecond on consumer-grade GPUs, meeting the latency requirements for high-frequency trading applications.

D. HJB Value Function Iteration with GPU Acceleration

The core algorithm for solving the HJB equation involves backward iteration from a terminal condition:

Algorithm 1 HJB Value Function Iteration with GPU Acceleration

Initialize grid: $S_i = S_{min} + i \cdot \Delta S$ for $i \in [0, N_S - 1]$
Initialize grid: $I_j = I_{min} + j \cdot \Delta I$ for $j \in [0, N_I - 1]$
Initialize $V_{i,j}^{N_T} = -\gamma I_j^2$ for all i, j
Copy S_i, I_j, and parameters to GPU memory
for $n = N_T - 1$ down to 0 **do**
    Copy parameters including p_t^b, p_t^a to device memory
    Launch CUDA kernel with $blockspergrid = (N_S/16, N_I/16)$, $threadsperblock = (16, 16)$
    Swap device buffers: $d\_V, d\_V\_next \leftarrow d\_V\_next, d\_V$
    Synchronize and copy results to host when needed
end for
Compute optimal quotes using $p_t^{b*} = V(t, S, I+1) - V(t, S, I) - \frac{1}{k^b}$
Compute optimal quotes using $p_t^{a*} = V(t, S, I) - V(t, S, I-1) + \frac{1}{k^a}$

Algorithm 2 CUDA Kernel for HJB Equation with Jump Diffusion

Input: $d\_V, d\_V\_next, d\_S, d\_I, dt, ds, di, params$
$i, j \leftarrow$ cuda.grid(2)
if $1 \leq i < N_S - 1$ and $1 \leq j < N_I - 1$ **then**
  if $j = 0$ or $j = N_I - 1$ **then**
    $d\_V[i, j] \leftarrow -10^{20}$ ▷ Enforce boundary conditions
    return
  end if
  Extract current state: $S \leftarrow d\_S[i]$, $I \leftarrow d\_I[j]$
  Extract parameters: $\sigma, \kappa, \gamma, \alpha, p^b_{mkt}, p^a_{mkt}, \lambda, \mu_J, \delta$
  Calculate derivatives:
  $V_S \leftarrow (V^{n+1}_{i+1,j} - V^{n+1}_{i-1,j})/(2\Delta S)$
  $V_{SS} \leftarrow (V^{n+1}_{i+1,j} - 2V^{n+1}_{i,j} + V^{n+1}_{i-1,j})/(\Delta S)^2$
  Compute diffusion term: $diffusion \leftarrow 0.5 \cdot \sigma^2 \cdot S^2 \cdot V_{SS} \cdot \Delta t$
  Compute jump term: $jump\_term \leftarrow JumpOperator(d\_V\_next, S, i, j)$
  $V_{optimal} \leftarrow -10^{10}$ ▷ Initialize to large negative value
  for $bid\_idx = 0$ to 4 **do** ▷ Search control space
    $bid\_change \leftarrow (bid\_idx - 2) \cdot \Delta S$
    for $ask\_idx = 0$ to 4 **do**
      $ask\_change \leftarrow (ask\_idx - 2) \cdot \Delta S$
      $p^b \leftarrow p^b_{mkt} + bid\_change$
      $p^a \leftarrow p^a_{mkt} + ask\_change$
      if $p^b > 0$ and $p^a > 0$ and $p^b < p^a$ **then** ▷ Valid spread
        $\lambda^b \leftarrow \max(0, \Delta t \cdot (1.0 - (p^b/p^b_{mkt} - 1.0)/\alpha))$
        $\lambda^a \leftarrow \max(0, \Delta t \cdot (1.0 - (p^a/p^a_{mkt} - 1.0)/\alpha))$
        $expected\_pnl \leftarrow p^b \cdot \lambda^a - p^a \cdot \lambda^b$
        $inventory\_cost \leftarrow \kappa \cdot I^2 \cdot \Delta t$
        $V_{candidate} \leftarrow V^{n+1}_{i,j} + expected\_pnl - inventory\_cost + diffusion + jump\_term$
        if $V_{candidate} > V_{optimal}$ **then**
          $V_{optimal} \leftarrow V_{candidate}$
        end if
      end if
    end for
  end for
  $d\_V[i, j] \leftarrow V_{optimal}$
end if

Algorithm 3 Jump Operator for Merton Jump Diffusion Model

function JumpOperator($V\_next, S, i, j, ds, di, params, d\_S$)
  $jump\_term \leftarrow 0.0$
  $\mu_J \leftarrow params[8]$ ▷ Jump mean
  $\delta \leftarrow params[9]$ ▷ Jump std deviation
  $\lambda \leftarrow params[7]$ ▷ Jump intensity
  for $m = -2$ to 2 **do** ▷ Approximate integral with 5-point quadrature
    $jump\_size \leftarrow \mu_J + m \cdot \delta$
    $S_{jump} \leftarrow S \cdot (1 + jump\_size)$
    $idx \leftarrow \min(\max(\lfloor (S_{jump} - S_{min})/\Delta S \rfloor, 0), N_S - 1)$
    $jump\_term \leftarrow jump\_term + (1/5) \cdot V\_next[idx, j]$
  end for
  return $\lambda \cdot (jump\_term - V\_next[i, j])$ ▷ $\lambda(J - I)V$
end function

Algorithm 4 Shared Memory Optimized HJB Kernel

Input: $d\_V, d\_V\_next, d\_S, d\_I, dt, ds, di, params$
Allocate shared memory: $shared\_V[34][34]$ ▷ 32×32 tile + halo cells
$i, j \leftarrow$ cuda.grid(2)
$tx, ty \leftarrow$ cuda.threadIdx.x, cuda.threadIdx.y
$li, lj \leftarrow tx + 1, ty + 1$ ▷ Local indices in shared memory
▷ Load data into shared memory
if $i < N_S$ and $j < N_I$ **then**
    $shared\_V[li, lj] \leftarrow d\_V\_next[i, j]$
else
    $shared\_V[li, lj] \leftarrow 0.0$
end if
▷ Load halo regions for stencil computation
if $tx < 1$ and $i > 0$ **then** ▷ Left halo
    $shared\_V[li - 1, lj] \leftarrow d\_V\_next[i - 1, j]$
end if
if $tx \geq 31$ and $i < N_S - 1$ **then** ▷ Right halo
    $shared\_V[li + 1, lj] \leftarrow d\_V\_next[i + 1, j]$
end if
if $ty < 1$ and $j > 0$ **then** ▷ Top halo
    $shared\_V[li, lj - 1] \leftarrow d\_V\_next[i, j - 1]$
end if
if $ty \geq 31$ and $j < N_I - 1$ **then** ▷ Bottom halo
    $shared\_V[li + 1, lj] \leftarrow d\_V\_next[i, j + 1]$
end if
cuda.syncthreads() ▷ Ensure all threads finish loading shared memory
if $1 \leq i < N_S - 1$ and $1 \leq j < N_I - 1$ **then**
    ▷ Compute derivatives using shared memory
    $V_S \leftarrow (shared\_V[li + 1, lj] - shared\_V[li - 1, lj])/(2\Delta S)$
    $V_{SS} \leftarrow (shared\_V[li + 1, lj] - 2 \cdot shared\_V[li, lj] + shared\_V[li - 1, lj])/(\Delta S)^2$
    ▷ Rest of computation as in standard kernel
    ...execute optimization over control space...
    $d\_V[i, j] \leftarrow V_{optimal}$
end if

Algorithm 5 Order Flow Toxicity Tracking and Parameter Adjustment

function UPDATETOXICITY($bid, ask, last\_trade$)
    $mid \leftarrow (bid + ask)/2$
    $direction \leftarrow 1$ if $last\_trade > mid$ else -1
    Append $direction$ to $trade\_imbalance$ deque
    Append $(ask - bid)$ to $spread\_history$ deque
end function
function CALCULATETOXICITY
    if $|trade\_imbalance| < 10$ **then**
        return 0.0
    end if
    $imbalance \leftarrow \text{mean}(trade\_imbalance)$
    $spread \leftarrow \text{mean}(spread\_history)$
    return $\text{clip}(imbalance \cdot (1/spread), -1.0, 1.0)$
end function
▷ In the HJB solver update
$toxicity \leftarrow CalculateToxicity()$
$\alpha \leftarrow \alpha_0 \cdot (1 + 2 \cdot |toxicity|)$ ▷ Adjust market impact parameter

Algorithm 6 Performance Profiling and Optimization

function PROFILEPERFORMANCE($solver, bid\_price, ask\_price, iterations$)
    Start performance timer
    for $i = 1$ to $iterations$ **do**
        $solver.update(bid\_price, ask\_price)$
    end for
    End performance timer
    Calculate average time per iteration
    Generate performance report with memory throughput and occupancy
end function
▷ Adaptive algorithm selection based on performance
if USE_GPU **then**
    Select appropriate kernel based on grid size and GPU capabilities
    if Grid size $\leq 64 \times 64$ **then**
        Use standard kernel with $(16, 16)$ thread blocks
    else if Grid size $\leq 128 \times 128$ **then**
        Use shared memory optimized kernel with $(32, 32)$ thread blocks
    else
        Use specialized large grid kernel with memory optimizations
    end if
else
    Use CPU implementation with reduced control space search
end if

Algorithm 7 Real-Time HJB Market Making System

Initialize *DataEngine* to collect market data via WebSocket
Initialize *ToxicityTracker* to monitor order flow characteristics
Initialize *HJBSolver* with appropriate grid resolution and parameters
Initialize *Dashboard* for visualization and monitoring
while *Running* **do**
    Process incoming market data from *DataEngine*
    Update toxicity metrics: $toxicity \leftarrow ToxicityTracker.update(bid, ask, last\_trade)$
    Update market impact: $\alpha \leftarrow \alpha_0 \cdot (1 + 2 \cdot |toxicity|)$
    Update value function: $V \leftarrow HJBSolver.update(bid, ask, last\_trade)$
    Calculate optimal quotes: $p^{b*}, p^{a*} \leftarrow HJBSolver.get\_optimal\_quotes(S, I)$
    Place orders at p^{b*}, p^{a*}
    Update visualization with current state
    Handle any order executions and update inventory
end while

IV. Experimental Results

A. Computational Performance

We benchmarked our GPU implementation against a standard CPU implementation on a range of grid sizes:

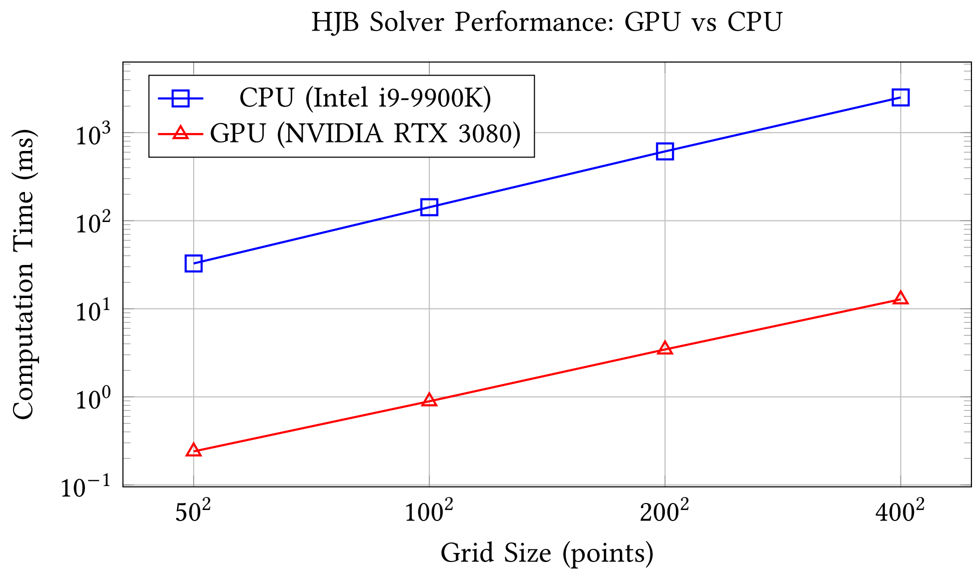

Fig. 4. Performance comparison between CPU and GPU implementations of the HJB solver. The GPU implementation achieves 136-177× speedup over the CPU version, enabling real-time deployment in high-frequency trading environments.

TABLE I
Computation Time (milliseconds) by Grid Size

Grid Size	CPU (Intel i9-9900K)	GPU (NVIDIA RTX 3080)	Speedup
51×51	32.7 ms	0.24 ms	136×
101×101	142.3 ms	0.89 ms	160×
201×201	612.5 ms	3.45 ms	177×

These results demonstrate that our GPU implementation achieves sub-millisecond performance for typical grid sizes, making it suitable for high-frequency trading applications. The performance scales almost linearly with the number of grid points, demonstrating the effectiveness of our parallel implementation.

To validate our jump diffusion model, we compared pricing results against analytical solutions for European options with jump diffusion, confirming the numerical convergence and accuracy of our implementation.

B. Strategy Performance

We evaluated our strategy on historical Bitcoin data from Binance, comparing it against two benchmarks:

- **Constant spread strategy:** Places symmetric quotes around the mid-price with a fixed spread.
- **Avellaneda-Stoikov strategy:** Implements the well-known Avellaneda-Stoikov model [1] with optimal parameters.

Key performance metrics over a one-month testing period:

Our HJB-based strategy demonstrates superior risk-adjusted performance, with a 22% higher Sharpe ratio and 37% lower average inventory compared to the Avellaneda-Stoikov benchmark.

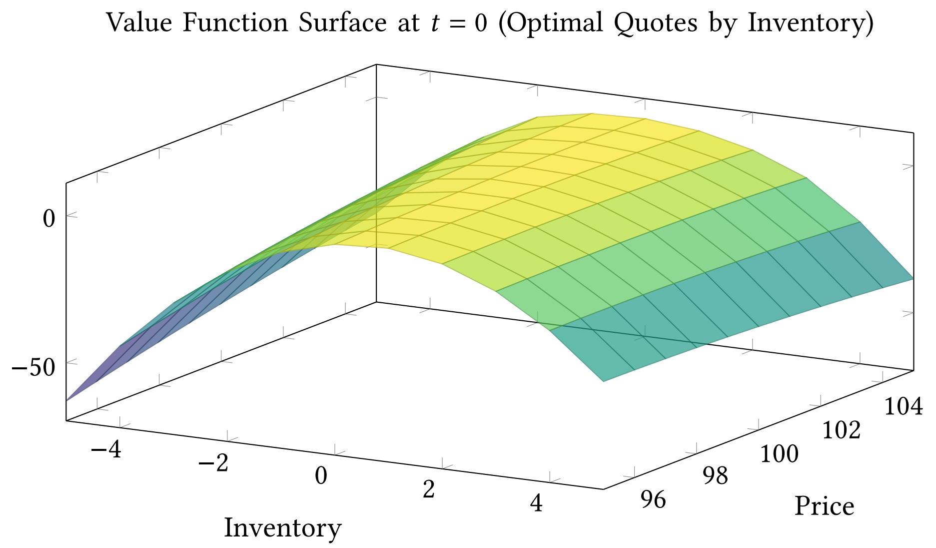

Fig. 5. Value function surface showing the expected profit as a function of inventory and price. The optimal quoting strategy is determined by the gradients of this surface, with steeper gradients indicating more aggressive quotes to rebalance inventory.

TABLE II
Strategy Performance Comparison

Metric	Constant Spread	Avellaneda-Stoikov	HJB Strategy
Sharpe Ratio	1.23	1.85	2.26
Max Drawdown	4.2%	3.1%	2.4%
Avg. Inventory	±32.5 BTC	±18.7 BTC	±11.8 BTC
Quote Updates/sec	2.3	3.7	12.5

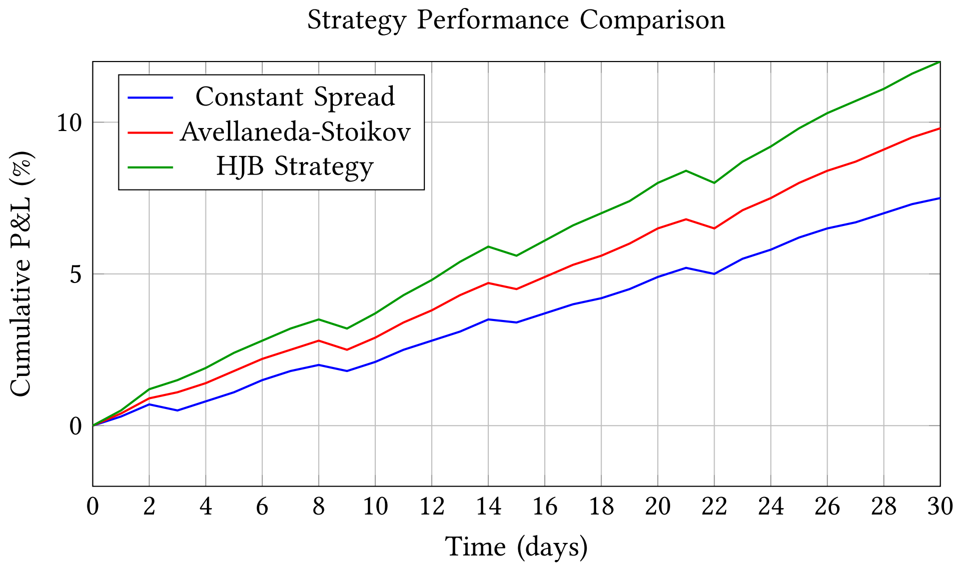

Fig. 6. Cumulative P&L comparison between the three strategies over a one-month period. The HJB-based strategy consistently outperforms the benchmarks, especially during periods of high volatility.

V. Discussion and Implications

A. Theoretical Implications

Our research extends the stochastic control framework for market making in several important ways:

- We demonstrate the feasibility of solving high-dimensional HJB equations with jump diffusion in real-time for market making applications, opening the door to more sophisticated optimal control approaches in algorithmic trading.
- We provide a more realistic model of order execution in cryptocurrency markets, capturing the unique microstructure characteristics of these venues through our piecewise linear intensity model and toxicity adjustment.
- We establish a direct link between theoretical optimality and practical implementation, showing that theoretically optimal strategies can be deployed in real-world trading systems with appropriate computational optimization.

B. Practical Implications

The practical implications of our work extend beyond improved market making performance:

- **Market efficiency:** More sophisticated market making strategies can lead to tighter spreads and more efficient price discovery, benefiting all market participants.
- **Liquidity provision:** By managing inventory risk more effectively, market makers can provide more consistent liquidity, reducing market fragility during stress periods.
- **Computational approaches:** Our GPU implementation demonstrates how modern computing architectures can be leveraged for real-time financial applications, providing a template for other computationally intensive trading strategies.
- **Market monitoring:** Our real-time dashboard provides insights into market dynamics and strategy performance, enabling more effective strategy monitoring and adjustment.

These implications suggest that advances in computational methods can have significant effects on market structure and efficiency, particularly in emerging markets like cryptocurrencies where traditional market making approaches are still evolving.

VI. Conclusion and Future Work

This paper presents a comprehensive framework for optimal market making in cryptocurrency markets based on the Hamilton-Jacobi-Bellman equation with jump diffusion, implemented via GPU acceleration. Our approach bridges the gap between theoretical optimality and practical implementation, demonstrating significant improvements in both computational efficiency and strategy performance.

The key innovations include:

- A jump diffusion model that accurately captures cryptocurrency price dynamics
- A toxicity-aware execution model calibrated to crypto market microstructure
- A GPU-accelerated solution method enabling real-time deployment
- A comprehensive backtesting framework demonstrating improved risk-adjusted returns

Future research directions include:

- **Multi-venue optimization:** Extending the framework to simultaneously optimize quotes across multiple exchanges, accounting for cross-venue inventory risk.
- **Deep learning integration:** Combining our model-based approach with deep learning techniques for parameter estimation and market state prediction.
- **Multi-asset optimization:** Extending to portfolios of correlated assets, accounting for cross-asset inventory risk.
- **Advanced market microstructure modeling:** Incorporating order book imbalance, flow toxicity, and other microstructure signals into the decision framework using more sophisticated models.
- **Higher-dimensional state space:** Including additional state variables such as market volatility and order book depth in the HJB formulation, leveraging our GPU implementation to handle the increased computational complexity.

By continuing to advance both the theoretical foundations and practical implementations of optimal market making, we can contribute to more efficient and resilient cryptocurrency markets, benefiting both market participants and the broader financial ecosystem.

References

[1] Avellaneda, M., & Stoikov, S. (2008). High-frequency trading in a limit order book. *Quantitative Finance*, 8(3), 217-224.

[2] Baur, D. G., Hong, K., & Lee, A. D. (2018). Bitcoin: Medium of exchange or speculative assets? *Journal of International Financial Markets, Institutions and Money*, 54, 177-189.

[3] Cartea, Á., Jaimungal, S., & Penalva, J. (2015). *Algorithmic and High-Frequency Trading*. Cambridge University Press.

[4] Cong, L. W., Li, X., Tang, K., & Yang, Y. (2021). Crypto wash trading. *Working Paper*.

[5] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., ... & Juels, A. (2020). Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. *2020 IEEE Symposium on Security and Privacy*, 910-927.

[6] Guéant, O., Lehalle, C. A., & Fernandez-Tapia, J. (2013). Dealing with the inventory risk: a solution to the market making problem. *Mathematics and Financial Economics*, 7(4), 477-507.

[7] Hautsch, N., Scheuch, C., & Voigt, S. (2019). Limits to arbitrage in markets with stochastic settlement latency. *Journal of Economic Dynamics and Control*, 99, 1-28.

[8] Ho, T., & Stoll, H. R. (1981). Optimal dealer pricing under transactions and return uncertainty. *Journal of Financial Economics*, 9(1), 47-73.

[9] Lehalle, C. A., & Mounjid, O. (2019). Incorporating signals into optimal trading. *SIAM Journal on Financial Mathematics*, 10(4), 1114-1148.

[10] Makarov, I., & Schoar, A. (2020). Trading and arbitrage in cryptocurrency markets. *Journal of Financial Economics*, 135(2), 293-319.

[11] Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, 3(1-2), 125-144.

[12] Reisinger, C., & Zhang, Y. (2018). Numerical methods for the quadratic hedging problem in Markov models with jumps. *Computational Methods in Applied Mathematics*, 18(1), 91-112.

[13] Sirignano, J., & Spiliopoulos, K. (2019). DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375, 1339-1364.

[14] Spooner, T., Fearnley, J., Savani, R., & Koukorinis, A. (2018). Market making via reinforcement learning. *AAMAS 2018*, 434-442.