Gas-Efficient Vector Processing for On-Chain Index Order Execution

Sonia Kolasinska, IndexMaker Labs

December 29, 2025

Abstract

The execution of financial primitives involving baskets of underlying assets poses a significant computational challenge in constrained smart contract environments. This paper formalizes a novel, high-dimensional algorithm, implemented on a VM^2 environment, which is a Virtual Machine deployed as a smart contract (Stylus/WASM). This VM^2 is optimized for vector processing using its **Vector Intermediate Language (VIL)**, designed to execute complex index order primitives (Buy, Sell, Rebalance) and dynamically update the index quote and margin parameters. The complete system operation integrates six key steps: (1) dynamically deriving the index quote (Price, Slope, Capacity), (2) updating the Vendor's net supply and market delta, (3) solving a vector-based quadratic pricing equation, (4) determining a systemic margin capacity limit, (5) performing complex market demand rebalancing, and (6) updating final net exposure (Delta). Furthermore, we detail the maintenance VIL routines essential for robust, continuous operation, including market data integration, margin limit updates, and asset list expansion. A crucial component of this low-cost execution is the selection of the VIL's Join-operation implementation, which is formally justified by comparative gas cost analysis, ensuring $\mathbf{O}(\mathbf{N_I} + \mathbf{N_M})$ complexity and cost predictability. We demonstrate that this approach successfully manages a flexible Index structure (e.g., a 50-asset index) against a 150-asset Market, utilizing the VIL's labels manipulation to handle various index sizes (e.g., top 20, top 100). This capability is crucial to quantifying the operational cost and providing a blueprint for efficient, high-fidelity vector arithmetic on a blockchain.

1 Introduction to the Problem

Decentralized finance protocols increasingly rely on complex, multi-asset products. An index, composed of N_I assets (e.g., $N_I \in \{20, 50, 100\}$), must interact with a broader market environment of N_M assets (where $N_M = 150$). The specific case where $N_I = 50$ is used throughout this analysis as an illustrative example of an index being a portion of the total market. The core difficulty lies in processing high-dimensional vectors (up to N_M components) entirely on-chain, where computation is constrained and expensive. Traditional Ethereum Virtual Machine (EVM) operations struggle with loop-based, element-wise vector arithmetic, leading to prohibitive gas costs for products requiring continuous rebalancing.

This challenge is overcome by deploying a separate, custom VM^2 environment, which is the **Vector IL VM** running inside the Arbitrum WASM VM (implemented as a smart contract compiled via Stylus). The VM^2's core advantage lies in its specialized **Vector Instruction Set (VIS)**, providing native, low-level vector operations (e.g., $\mathbf{A} \oplus \mathbf{B}$). Critically, the VIS includes **Labels Manipulation** instructions ('JFLT', 'JUPD', 'JADD'). These commands are essential for correctly aligning and propagating data between the Index domain (e.g., 50 components) and the Market domain (150 components) without compromising state integrity.

It is critical to note that the internal execution complexity of these Label Manipulation instructions is linear, specifically $O(N_I + N_M)$ where N_I is the index size and N_M is the market size. However, this contained $O(N_I + N_M)$ complexity within the highly optimized VIL VM represents a transformative gas saving compared to an EVM implementation. The alternative's gas cost is dominated by the necessity of performing high-cost storage operations (SLOAD & SSTORE) on the underlying blockchain, which makes it prohibitively expensive. By performing this alignment in a single VIL operation, the overall transaction maintains a flat, low gas cost profile. The goal is to define an algorithm that is computationally efficient and mathematically sound, ensuring the transaction satisfies pricing, collateral, and systemic (Margin) constraints simultaneously.

2 Decentralized Architecture and Thematic Roles (DeIndex)

The system is formally known as the **Decentralised Index Maker (DeIndex)**. Its design is composed of several smart contract components and an off-chain coordinator, each assigned a thematic role to simplify the system's conceptual framework.

Table 1: Thematic Roles and Technical Functions of the DeIndex Architecture

Thematic Role	**Technical Function**	**Summary**
Daxos (Hero)	Main Contract	The main contract that orchestrates all transactions following the business logic.
DeVIL (Machine)	Vector Math	An on-chain computation machine that can perform operations on vectors. It is VIL VM in WASM VM (VM^2), which executes Vector Intermediate Language (VIL) code.
Vault (Safe)	Index Token (ITP)	The tokenized derivative (the index primitive) being bought, sold, or rebalanced.
Vendor (Merchant)	Authorized Provider	The off-chain service responsible for providing supply and pricing to the Market.
Market (Place)	Supply & Demand	Stores all critical system vectors: Supply, Demand, Delta, Price, Slope, and Margin.

Throughout the paper, references to the **Vendor** refer to the entity responsible for coordinating asset transfers (as detailed in Section 4) and committing new market data (Section 5), while the **Market** refers to the immutable on-chain state vectors themselves.

3 Quote Determination via Vector Processing

Before any order execution can take place, the core pricing and capacity parameters of the index must be derived from the underlying market data. This process is handled by the VIL function `update_quote` and is critical for ensuring the index price accurately reflects the sum of its components and that the system capacity respects underlying asset liquidity. The Index Quote comprises the Price (P), the Slope (S), and the Market Capacity (C).

3.1 Index Price Determination (P)

The Index Price (P) is defined as the weighted sum of the current filtered market prices ($\mathbf{P}_i$) of the index's constituent assets. This calculation demonstrates the fundamental use of the VIL's JFLT (Join-Filter) instruction to align the 150-component market prices with the 50-component index weights ($\mathbf{W}$) before performing the summation.

$$P = \sum_i \mathbf{W}_i \cdot \mathbf{P}_i$$

3.2 Index Slope Determination (S)

The Index Slope (S) governs the convexity of the index's pricing function, ensuring that larger orders incur a higher marginal cost (similar to a bonding curve). This design ensures the aggregate index cost function follows the quadratic form:

$$C = P \cdot Q + S \cdot Q^2$$

where C is the total cost for an Index quantity Q. The total cost must equal the sum of the individual assets' costs. The quadratic price impact term for a single asset i is $\text{Cost}_{\text{impact},i} = \mathbf{S}_i \cdot \mathbf{A}_i^2$, where $\mathbf{A}_i$ is the executed quantity of asset i. Since $\mathbf{A}_i = Q \cdot \mathbf{W}_i$ (the executed asset quantity is proportional to the Index quantity and the asset's weight), we substitute this into the total quadratic impact cost:

$$\text{Cost}_{\text{impact}} = \sum_i \mathbf{S}_i \cdot \mathbf{A}_i^2 = \sum_i \mathbf{S}_i \cdot (Q \cdot \mathbf{W}_i)^2 = Q^2 \cdot \sum_i (\mathbf{S}_i \cdot \mathbf{W}_i^2)$$

By comparing this derived term ($Q^2 \cdot \sum_i (\mathbf{S}_i \cdot \mathbf{W}_i^2)$) to the aggregate quadratic term ($S \cdot Q^2$), the Index Slope S is formally defined as the weighted sum of the underlying assets' market slopes ($\mathbf{S}_i$), utilizing the square of the index weights ($\mathbf{W}_i^2$):

$$S = \sum_i \mathbf{S}_i \cdot \mathbf{W}_i^2$$

3.3 Index Market Capacity Determination (C)

The Market Capacity (C) sets the absolute upper bound on the index quantity that can be traded, based on the liquidity constraints of the underlying assets. This is determined by the most restrictive asset, requiring a component-wise division followed by a vector-wide minimum extraction. This value is used as the Cap input in the order execution logic (Section 6.3).

$$C = \min_i \left(\frac{\mathbf{L}_i}{\mathbf{W}_i} \right)$$

where $\mathbf{L}_i$ is the market liquidity of asset i. The VIL efficiently computes this using DIV (component-wise) and VMIN (vector-wide minimum).

3.4 VIL implementation of Quote determination

Load asset weights, names, and makret asset names:

```
LDV         weights_id
STR         _AssetWeights
LDL         index_asset_names_id
LDL         market_asset_names_id
```

Compute:

$$P = \sum_i \mathbf{W}_i \cdot \mathbf{P}_i$$

```
LDV         asset_prices_id
JFLT        1   2
LDR         _AssetWeights
SWAP        1
MUL         1
VSUM
STR         _Price
```

Compute:

$$S = \sum_i \mathbf{S}_i \cdot \mathbf{W}_i^2$$

```
MUL         0
LDV         asset_slopes_id
JFLT        2   3
MUL         1
VSUM
STR         _Slope
POPN        1
```

Compute:

$$C = \min_i \left(\frac{\mathbf{L}_i}{\mathbf{W}_i} \right)$$

```
LDR         _AssetWeights
LDV         asset_liquidity_id
JFLT        2   3
DIV         1
```

```
VMIN
STR         _Capacity
POPN        3
```

Store results P, S, and C:

```
LDM         _Capacity
LDM         _Price
LDM         _Slope
PKV         3
STV         quote_id
```

The above code snippet is included for demonstration purposes. For most up to date version of the code please refer to project repository.

4 The Vendor's Role and Market State Update

The Vendor, which indirectly invokes the VIL execution, is responsible for maintaining the system's overall health and managing the inventory of underlying assets. This involves two critical steps: updating the **Supply** vectors ($\mathbf{S}_L$, $\mathbf{S}_S$) based on external asset transfers, and calculating the resultant **Net Exposure (Delta)** vectors (Δ_L, Δ_S). This is handled by the `update_supply` VIL function.

The Vendor's primary role (see Figure 4) is satisfying demand for assets by providing adequate supply. This effectively means maintaining the Δ_L and Δ_S vectors below a certain operational threshold, MinOrderSize. Specifically, the Vendor is considered dormant if $\max(\Delta_L, \Delta_S) < \text{MinOrderSize}$. Conversely, if $\Delta_L > \text{MinOrderSize}$, the Vendor starts selling assets to reduce the long exposure (Δ_L). If $\Delta_S > \text{MinOrderSize}$, the Vendor starts buying new assets to cover the short exposure (Δ_S). The Vendor's **active objective** is to always maintain $\max(\Delta_L, \Delta_S) < \text{MinOrderSize}$ through these proactive buy/sell operations.

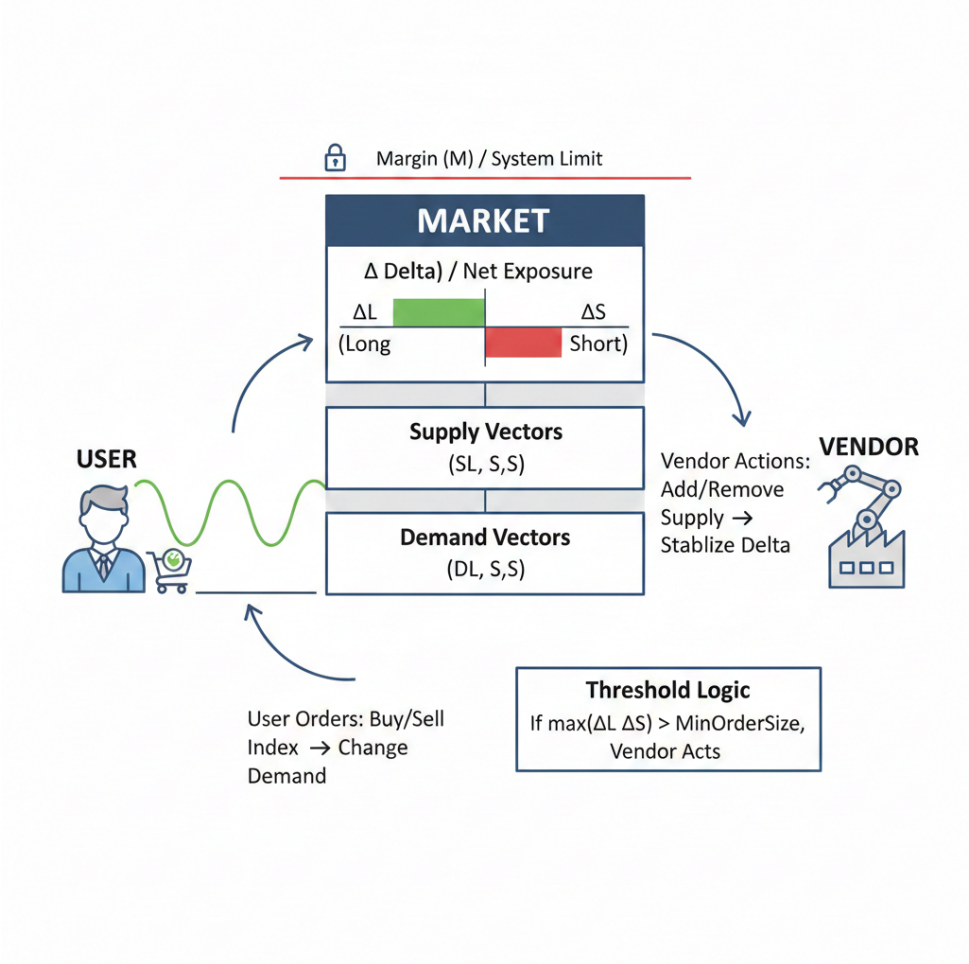

Figure 1: User vs. Vendor

4.1 Supply Update via External Transfers

The Vendor manages the available supply of each underlying asset. These supplies are represented by $\mathbf{S}_L$ (Long Supply, assets available for shorting) and $\mathbf{S}_S$ (Short Supply, assets available for selling long). The supply vectors are updated with external transfers ($\mathbf{A}_L$, $\mathbf{A}_S$) from the broader market using the VIL's JUPD (Join-Update) instruction. This is a vector-aware replacement for simple storage updates.

$$\mathbf{S}_{L,new} = \mathbf{S}_L \text{ join\_update } \mathbf{A}_L$$

$$\mathbf{S}_{S,new} = \mathbf{S}_S \text{ join\_update } \mathbf{A}_S$$

where $\mathbf{A}_L$ and $\mathbf{A}_S$ are the vectors of asset quantities received by the Vendor, and join_update performs the update only on assets present in the index's domain, effectively applying the VIL JUPD instruction for correct vector alignment.

4.2 Net Exposure (Delta) Calculation

The net exposure, or Delta, represents the system's current net position on each asset after accounting for all market activities (Supply and Demand). It is calculated by normalizing the total available assets $(\mathbf{T}_L, \mathbf{T}_S)$ using the VIL's SSB (Saturating Subtraction) instruction, which ensures that an asset's net position is either entirely long or entirely short (i.e., Δ_L and Δ_S are mutually exclusive, element-wise).

The total Long and Short asset vectors $(\mathbf{T}_L, \mathbf{T}_S)$ are first computed:

$$\mathbf{T}_L = \mathbf{S}_{L,new} \oplus \mathbf{D}_S$$

$$\mathbf{T}_S = \mathbf{S}_{S,new} \oplus \mathbf{D}_L$$

where $\mathbf{D}_S$ and $\mathbf{D}_L$ are the current market Short and Long Demand vectors, respectively.

The final net exposure vectors are derived by:

$$\Delta_{L,new} = \mathbf{T}_L \ominus \mathbf{T}_S \quad \text{(using SSB in VIL)}$$

$$\Delta_{S,new} = \mathbf{T}_S \ominus \mathbf{T}_L \quad \text{(using SSB in VIL)}$$

The SSB operation is critical here, as it ensures that for any asset i, if $T_{L,i} > T_{S,i}$, then $\Delta_{S,i} = 0$, and vice versa, maintaining the integrity of the margin system.

4.3 VIL implementation of Supply and Delta update

Load asset names:

```
LDL         asset_names_id
LDL         market_asset_names_id
```

Compute:

$$\mathbf{S}_{S,new} = \mathbf{S}_S \text{ join\_update } \mathbf{A}_S$$

```
LDV         asset_quantities_short_id
LDV         supply_short_id
JUPD        1   2   3
STR         _SupplyShort
POPN        1
```

Compute:

$$\mathbf{S}_{L,new} = \mathbf{S}_L \text{ join\_update } \mathbf{A}_L$$

```
LDV         asset_quantities_long_id
LDV         supply_long_id
JUPD        1   2   3
STR         _SupplyLong
POPN        1
```

Compute:

$$\mathbf{T}_L = \mathbf{S}_{L,new} \oplus \mathbf{D}_S$$

```
LDR         _SupplyLong
LDV         demand_short_id
ADD         1
SWAP        1
POPN        1
```

Compute:

$$\mathbf{T}_S = \mathbf{S}_{S,new} \oplus \mathbf{D}_L$$

```
LDR         _SupplyShort
LDV         demand_long_id
ADD         1
SWAP        1
POPN        1
```

Compute:

$$\Delta_{S,new} = \mathbf{T}_S \ominus \mathbf{T}_L$$
$$\Delta_{L,new} = \mathbf{T}_L \ominus \mathbf{T}_S$$

```
LDD         0
SSB         2
STR         _DeltaShort
SWAP        1
SSB         1
STR         _DeltaLong
POPN        3
```

Store S_S, S_L

```
LDM         _SupplyLong
LDM         _SupplyShort
STV         supply_short_id
STV         supply_long_id
```

Store Δ_S, Δ_L:

```
LDM         _DeltaLong
LDM         _DeltaShort
STV         delta_short_id
STV         delta_long_id
```

The above code snippet is included for demonstration purposes. For most up to date version of the code please refer to project repository.

5 Market Data Integrity and Maintenance Routines

For the system to maintain a continuous, accurate, and solvent state, several maintenance routines must be executed periodically. These operations are also implemented as specialized VIL routines to ensure vector integrity and computational efficiency. They primarily rely on the VIL's 'JUPD' (Join-Update) instruction to merge new data vectors with existing market vectors ($\mathbf{V}_{\text{old}}$) based on the asset label space, which is critical for the $N_M = 150$ assets.

5.1 Data Feed Integration (`update_market_data`)

The `update_market_data` routine is executed by the **Vendor (Authorized Provider)**. The Vendor is responsible for the computation of the core market vectors ($\mathbf{P}, \mathbf{S}, \mathbf{L}$) from raw, off-chain order book data before injection. This process ensures that the on-chain state accurately reflects real-time liquidity and price impact, thereby maintaining system solvency.

For each asset, the Vendor processes the top K levels (e.g., $K = 5$) of the order book (Bids and Asks) to derive the three key vector components ($\mathbf{P}_i, \mathbf{S}_i, \mathbf{L}_i$):

1. **Liquidity (L):** The available liquidity, $\mathbf{L}_i$, is defined as the total sum of quote-side depth within the scanned K levels on both the bid and ask sides. Let Q_{Bk} be the quantity at bid level k and Q_{Ak} be the quantity at ask level k.

$$\mathbf{L}_i = \sum_{k=1}^{K} Q_{Bk} + \sum_{k=1}^{K} Q_{Ak}$$

2. **Price (P):** The asset's spot price, $\mathbf{P}_i$, is defined as the **Micro-Price** (Mid-Price weighted by the best-level liquidity), which reduces volatility and bias toward the side with less depth. Let P_{B1} and P_{A1} be the best bid and ask prices, and Q_{B1} and Q_{A1} be the corresponding best quantities.

$$\mathbf{P}_i = \frac{P_{A1} \cdot Q_{B1} + P_{B1} \cdot Q_{A1}}{Q_{B1} + Q_{A1}}$$

3. **Slope (S):** The market slope, $\mathbf{S}_i$, quantifies the price impact across the $2K$ levels. It is calculated as the total price range spanned by the K^{th} levels divided by the total liquidity $\mathbf{L}_i$ within that range. Let P_{AK} be the price of the K^{th} ask level and P_{BK} be the price of the K^{th} bid level.

$$\mathbf{S}_i = \frac{P_{AK} - P_{BK}}{\mathbf{L}_i}$$

These calculated vectors are then injected into the Market state using the VIL's JUPD instruction to update only the specified assets:

- **Prices (P)**: The current price of each asset.
- **Slopes (S)**: The current market slope/depth for each asset.
- **Liquidity (L)**: The available liquidity for each asset used to calculate capacity.

The routine uses the JUPD instruction to apply the new values only for the assets explicitly included in the input vector, leaving all other assets in the 150-component vector unchanged.

$$\mathbf{P}_{\text{new}} = \mathbf{P}_{\text{old}} \text{ join\_update } \mathbf{P}_{\text{input}}$$

Similarly for $\mathbf{S}_{\text{new}}$ and $\mathbf{L}_{\text{new}}$.

5.2 Margin Headroom Update (`update_margin`)

The maximum allowed net exposure vector ($\mathbf{M}$), which defines the systemic limit (Margin Headroom), must also be dynamically adjustable. The `update_margin` routine allows the Vendor or governance mechanism to inject new margin limits, also using the JUPD instruction for selective updates.

$$\mathbf{M}_{\text{new}} = \mathbf{M}_{\text{old}} \text{ join\_update } \mathbf{M}_{\text{input}}$$

5.3 Market Asset Expansion (`update_assets`)

One of the most powerful VIL operations is the atomic expansion of the Market Asset Names (N_M) list. The `update_assets` routine handles the addition of new tokens to the market's domain (e.g., increasing N_M from 150 to 155). This is achieved by extending the master list of asset names and subsequently extending all associated vectors ($\mathbf{P}, \mathbf{S}, \mathbf{L}, \mathbf{S}_L, \mathbf{S}_S, \mathbf{D}_L, \mathbf{D}_S, \Delta_L, \Delta_S, \mathbf{M}$) to the new dimension. For any newly added asset, its corresponding vector components are initialized to zero using a ZEROS vector and the JUPD instruction, thus maintaining a safe initial state with no exposure, liquidity, or demand.

5.4 VIL implementation of Market Data update

Update Prices, Slopes, and Liquidity:

```
LDL         asset_names_id
LDL         market_asset_names_id
```

Compute:

$$\mathbf{P}_{\text{new}} = \mathbf{P}_{\text{old}} \text{ join\_update } \mathbf{P}_{\text{input}}$$

```
LDV         asset_prices_id
LDV         market_asset_prices_id
JUPD        1   2   3
STR         _Prices
POPN        1
```

Compute:

$$\mathbf{S}_{\text{new}} = \mathbf{S}_{\text{old}} \text{ join\_update } \mathbf{S}_{\text{input}}$$

```
LDV         asset_slopes_id
LDV         market_asset_slopes_id
JUPD        1   2   3
STR         _Slopes
POPN        1
```

Compute:

$$\mathbf{L}_{\text{new}} = \mathbf{L}_{\text{old}} \text{ join\_update } \mathbf{L}_{\text{input}}$$

```
LDV         asset_liquidity_id
LDV         market_asset_liquidity_id
JUPD        1   2   3
STR         _Liquidity
POPN        1
```

Store asset Prices, Slopes, and Liquidity:

```
LDM         _Prices
LDM         _Slopes
LDM         _Liquidity
STV         market_asset_liquidity_id
STV         market_asset_slopes_id
STV         market_asset_prices_id
```

The above code snippet is included for demonstration purposes. For most up to date version of the code please refer to project repository.

6 Mathematical Framework for Index Acquisition

The execution algorithm is an atomic, six-step process. Variables denoted in bold ($\mathbf{V}$) are vectors, while italic variables (s) are scalars. Operations $\oplus, \ominus, \odot, \oslash$ represent vector addition, saturating subtraction, component-wise multiplication, and component-wise division, respectively. $\min(\mathbf{V})$ denotes the minimum value across all vector components. **Note: When a scalar (s) is used in a vector operation (e.g., $\min(\mathbf{V}, s)$ or $s \odot \mathbf{W}$), it implies a component-wise operation where the scalar is broadcast to a vector of matching dimension.**

6.1 Step 1: Collateral Update and Single-Step Limit (*MaxOrderSize*)

The user's effective collateral (C_{eff}) is calculated based on initial funds (C_{old}) and external transfers (C_{add}, C_{rem}). The final usable collateral is immediately capped by the system-defined maximum order size (MaxOrderSize).

Role of MaxOrderSize: This scalar limit serves as an execution fragmentation tool. It dictates the maximum dollar value of an index quantity that can be executed in a single atomic VIL call. For large orders, this fragmentation into smaller steps ensures that the order progresses optimally along the quadratic price curve, benefiting both small orders (by preventing a single large order from dominating the price impact) and the overall execution quality of large orders (by achieving a better average price across steps).

The total current collateral pool (C_{new}) is:

$$C_{\text{new}} = (C_{\text{old}} + C_{add}) - C_{rem}$$

The effective collateral used for the pricing calculation (C_{eff}) is:

$$C_{\text{eff}} = \min(\text{MaxOrderSize}, C_{\text{new}})$$

The market provides the scalar Index Price Slope (S) and Price (P), which define the marginal cost function for the total index quantity Q. These are the values computed in Section 3.

6.2 Step 2: Maximum Quantity Determination via Quadratic Solver

The maximum possible Index Quantity based on collateral ($Q_{\text{max, pricing}}$) is determined by solving the Index Price Function, which links the effective cost (C_{eff}) to the quantity (Q): $C_{\text{eff}} = P \cdot Q + S \cdot Q^2$.

This is reformulated into a standard scalar quadratic equation:

$$S \cdot Q^2 + P \cdot Q - C_{\text{eff}} = 0$$

The maximum quantity is found by taking the positive root of the solution:

- **For a Buy Order**:
$$Q_{\text{buy, max, pricing}} = \frac{\sqrt{P^2 + 4 \cdot S \cdot C_{\text{eff}}} - P}{2 \cdot S}$$
- **For a Sell Order**:
$$Q_{\text{sell, max, pricing}} = \frac{P - \sqrt{P^2 - 4 \cdot S \cdot C_{\text{eff}}}}{2 \cdot S}$$

6.3 Step 3: Margin-Based Capacity Limit (CL) with Progressive Constraint

To prevent systemic limits breaches, the transaction must be capped by the market's remaining margin capacity ($\mathbf{M}$) relative to the current long (Δ_L) and short (Δ_S) exposures. The **Final Capacity Limit (CL)** for the entire index purchase is defined by the minimum available capacity across all assets, **constrained progressively by the AssetContributionFractions vector ($\mathbf{F}$)**.

Role of F: This vector $\mathbf{F}$ is derived from a batch of pending index orders. It ensures that the available margin capacity of bottleneck assets is split proportionally among the batch of orders, guaranteeing that the orders can progress and preventing any single large order from blocking the entire batch.

$$CL = \min\left(\mathbf{F} \odot \frac{\mathbf{L}}{\mathbf{W}}\right)$$

1. **Margin Limit Vector ($\mathbf{L}$):** The available capacity ($\mathbf{L}$) is the **sum** of the system's available resources: the **Inventory** that can be closed (Δ_L or Δ_S) plus the **Remaining Margin Headroom**, capped by the equivalent vector for the scalar Market Capacity ($C \odot \mathbf{W}$).

   - **For a Buy Order** (either reducing Δ_L or increasing Δ_S):

     $$\mathbf{L} = \Delta_L \oplus \min(\mathbf{M} \ominus \Delta_S, C \odot \mathbf{W})$$

   - **For a Sell Order** (either reducing Δ_S or increasing Δ_L):

     $$\mathbf{L} = \Delta_S \oplus \min(\mathbf{M} \ominus \Delta_L, C \odot \mathbf{W})$$

   Note: $\mathbf{M}$ represents the maximum allowed net exposure, which serves as a **hard, systemic upper limit** such that $\text{Abs}(\Delta_L \oplus \Delta_S) \leq \mathbf{M}$ must always be true. This Margin ($\mathbf{M}$) is typically equivalent to the total Vendor-managed reserves. The Market Capacity C used here is derived from the `update_quote` function (Section 3.3). Since Δ_L and Δ_S are mutually exclusive (an asset cannot be net long and net short simultaneously), the additive vector operation ($\oplus$) correctly aggregates the two distinct sources of capacity.

2. **Asset Capacity Limit Vector ($\mathbf{CL_{vec}}$):** This vector is aligned via Label Manipulation ('JFLT') to the Index Asset Weights ($\mathbf{W}$) and determines the maximum *Index* quantity each asset can sustain.

3. **Final Capacity Limit (CL):** The system capacity is the minimum across all assets after the asset-level capacity ($\mathbf{L} \oslash \mathbf{W}$) is constrained by the 'AssetContributionFractions' vector ($\mathbf{F}$). This ensures that the current iterative order consumes no more than its allocated fraction of the total remaining margin.

$$CL = \min\left(\mathbf{F} \odot (\mathbf{L} \oslash \mathbf{W})\right)$$

6.4 Step 4: Final Quantity Execution and Asset Calculation

The executed Index Quantity (Q_{final}) is capped by the maximum allowed by pricing ($Q_{\text{max, pricing}}$) and the Margin Capacity (CL).

$$Q_{\text{final}} = \min(Q_{\text{max, pricing}}, CL)$$

The resulting executed asset quantities ($\mathbf{A}$) are calculated by distributing Q_{final} according to the Index Weights ($\mathbf{W}$).

$$\mathbf{A} = Q_{\text{final}} \odot \mathbf{W}$$

6.5 Step 5: Market Demand Rebalancing

This is next step, where the 50-component $\mathbf{A}$ vector updates the 150-component market demand vectors $(\mathbf{D}_S, \mathbf{D}_L)$.

1. **New Short Demand ($\mathbf{D}_{S,new}$):** The executed assets reduce the standing short demand, saturating at zero.

$$\mathbf{D}_{S,new} = \mathbf{D}_S \ominus \mathbf{A}$$

2. **Residual Quantities ($\mathbf{R_A}$):** Any asset quantities that could not be matched by existing short demand become residual.

$$\mathbf{R}_\mathrm{A} = \mathbf{A} \ominus \mathbf{D}_S$$

3. **New Long Demand ($\mathbf{D}_{L,new}$):** The residual quantities are added to the long demand.

$$\mathbf{D}_{L,new} = \mathbf{D}_L \oplus \mathbf{R}_\mathrm{A}$$

6.6 Step 6: Delta Update and Financial Commit

The final net exposures (Δ_L, Δ_S) are updated based on the new demand/supply state ($\mathbf{D}_S$, $\mathbf{D}_L$, $\mathbf{S}_L$, $\mathbf{S}_S$).

$$\mathbf{T}_L = \mathbf{S}_L \oplus \mathbf{D}_{S,new} \quad ; \quad \mathbf{T}_S = \mathbf{S}_S \oplus \mathbf{D}_{L,new}$$

$$\Delta_{L,new} = \mathbf{T}_L \ominus \mathbf{T}_S \quad ; \quad \Delta_{S,new} = \mathbf{T}_S \ominus \mathbf{T}_L$$

The order's spent collateral is calculated:

$$C_\text{spent} = Q_\text{final} \cdot (S \cdot Q_\text{final} + P)$$

All market and order state vectors are then committed.

6.7 VIL implementation of Index Order execution

Load asset weights:

```
LDV         asset_weights_id
STR         _Weights
```

Load Index order:

```
LDV         order_id
UNPK
STR         _Minted
STR         _Spent
```

Compute:

$$C_\text{new} = (C_\text{old} + C_{add}) - C_{rem}$$

```
IMMS        collateral_added
ADD         1
IMMS        collateral_removed
SWAP        1
SUB         1
STR         _Collateral
POPN        2
```

Store updated order:

```
LDR         _Collateral
LDR         _Spent
LDR         _Minted
PKV         3
STV         order_id
```

Load Index quote:

```
LDV         index_quote_id
UNPK
SWAP        2
STR         _Capacity
STR         _Price
STR         _Slope
```

Cap $MaxOrderSize$:

$$C_{\text{eff}} = \min(\text{MaxOrderSize}, C_{\text{new}})$$

and Solve Quadratic:

$$S \cdot Q^2 + P \cdot Q - C_{\text{eff}} = 0$$

```
IMMS        max_order_size
LDR         _Slope
LDR         _Price
LDR         _Collateral
MIN         3
B           solve_quadratic_id  3  1  4
STR         _IndexQuantity
POPN        1
```

Compute:

$$\mathbf{L} = \Delta_L \oplus \min(\mathbf{M} \ominus \Delta_S, C \odot \mathbf{W})$$

and then:

$$CL = \min\left(\mathbf{F} \odot \frac{\mathbf{L}}{\mathbf{W}}\right)$$

```
LDL         asset_names_id
LDL         market_asset_names_id
LDV         asset_contribution_fractions_id
LDV         margin_id
LDV         delta_long_id
JFLT        3   4
LDV         delta_short_id
```

```
SWAP        2
SSB         2
JFLT        4   5
LDR         _Weights
LDM         _Capacity
LDD         1
MUL         1
MIN         3
ADD         4
DIV         2
MUL         6
VMIN
SWAP        6
POPN        6
SWAP        2
STR         _AssetNames
STR         _MarketAssetNames
```

Cap Index Quantity with Capacity:

$$Q_{\text{final}} = \min(Q_{\text{max, pricing}}, CL)$$

```
LDR         _IndexQuantity
MIN         1
STR         _CappedIndexQuantity
POPN        1
```

Generate individual Asset Orders (compute asset quantities):

$$\mathbf{A} = Q_{\text{final}} \odot \mathbf{W}$$

```
LDR         _CappedIndexQuantity
LDM         _Weights
MUL         1

STR         _AssetQuantities
POPN        1
```

Load asset names *(for Index order and for market)*:

```
LDM         _AssetNames
LDM         _MarketAssetNames
```

Compute:

$$\mathbf{D}_{S,new} = \mathbf{D}_S \ominus \mathbf{A}$$

and then:

$$\mathbf{R}_{\text{A}} = \mathbf{A} \ominus \mathbf{D}_S$$

```
LDV         demand_short_id
LDR         _AssetQuantities
LDD         1
JFLT        3   4
LDD         0
```

```
SSB         2
SWAP        3
JUPD        3   4   5
SWAP        3
POPN        1
```

Compute:

$$\mathbf{D}_{L,new} = \mathbf{D}_L \oplus \mathbf{R}_{\mathrm{A}}$$

```
SWAP        1
SSB         1
LDV         demand_long_id
JADD        1   4   5
SWAP        2
POPN        2
STR         _DemandLong
STR         _DemandShort
```

Compute:

$$\mathbf{T}_L = \mathbf{S}_L \oplus \mathbf{D}_{S,new} \quad ; \quad \mathbf{T}_S = \mathbf{S}_S \oplus \mathbf{D}_{L,new}$$

```
LDV         supply_long_id
LDR         _DemandShort
ADD         1
SWAP        1
POPN        1
LDV         supply_short_id
LDR         _DemandLong
ADD         1
SWAP        1
POPN        1
```

Compute:

$$\Delta_{L,new} = \mathbf{T}_L \ominus \mathbf{T}_S \quad ; \quad \Delta_{S,new} = \mathbf{T}_S \ominus \mathbf{T}_L$$

```
LDD         0
SSB         2
STR         _DeltaShort
SWAP        1
SSB         1
STR         _DeltaLong
POPN        3
```

Compute:

$$C_{\text{spent}} = Q_{\text{final}} \cdot (S \cdot Q_{\text{final}} + P)$$

```
LDR         _CappedIndexQuantity
LDM         _Slope
MUL         1
LDM         _Price
ADD         1
SWAP        1
POPN        1
MUL         1
```

Compute Index Order total remaining Collateral:

$$C_{\text{remain}} = C_{\text{new}} - C_{\text{spent}}$$

```
LDM         _Collateral
SSB         1
SWAP        1
```

Compute Index order total spent Collateral:

$$C_{\text{total\_new}} = C_{\text{total\_old}} + C_{\text{spent}}$$

```
LDM         _Spent
ADD         1
SWAP        1
POPN        1
SWAP        1
SWAP        2
```

Store updated Index order:

```
PKV         3
STV         order_id
```

Store executed Index quantity and remaining quantity:

```
LDM         _CappedIndexQuantity
LDM         _IndexQuantity
SUB         1
PKV         2
STV         executed_index_quantities_id
```

The above code snippet is included for demonstration purposes. For most up to date version of the code please refer to project repository.

7 Performance Analysis and Efficiency

The efficiency of this vectorized execution, even with the complex $50 \to 150$ vector alignment, is quantified by the observed gas cost.

7.1 Observed Transaction Cost

The analyzed transaction required **3,461,868 gas** to execute, based on the provided transaction receipt. This cost is incurred for the following operations:

- Solving the scalar quadratic equation.
- Performing exactly 11 vector arithmetic operations ($\oplus, \ominus, \odot, \oslash$) on vectors up to 150 components.
- Executing exactly 4 Label Manipulation instructions ('JFLT', 'JUPD', 'JADD') to align and commit data between the Index and Market domains.

Table 2: Transaction Cost and Scale Summary (Gas Price: 100 Gwei)

Metric	**Value**	**Cost Equivalency (at 100 Gwei)**
Total Gas Used	$\mathbf{3,461,868}$	Actual on-chain consumption
Asset Scale	$N_I = 50$ vs. $N_M = 150$	High dimensional processing
ETH Cost	$3,461,868 \cdot 100 \cdot 10^{-9}$ ETH	**0.0003461868** ETH
USD Cost	0.0003461868 ETH $\times$ \$3,600/ETH*	$\approx$ **\$1.25** USD (or **124.63** cents)

*Assuming an ETH price of \$3,600 for illustrative purposes.

7.2 VIL Performance Analysis: Justification for Low Gas Cost

The core of the low-cost execution is the efficient implementation of the VIL's **Join operations**: 'JFLT' (Join-Filter), 'JUPD' (Join-Update), and 'JADD' (Join-Add). These operations are essential for aligning the asset index vector with the asset market vector. The paper, "Aligning Market and Index Vectors" (2025), provided a comparative analysis of four implementation strategies, summarized in Table 3, which empirically justifies the mandated choice.

Comparative Analysis of VIL Implementations

The challenge involves synchronizing two sorted vectors: a large Market Vector (150 components) and a smaller, sparse Input Vector (e.g., 50 components, potentially non-contiguous).

Table 3: Comparative Empirical Gas Costs (Test Sets A and B)

Version	**Implementation Method**	**Gas Cost (Test A)**	**(Test B - Mixed Data)**
(a)	Linear scan, filtering **in-place**	3,380,000	**3,993,165**
(b)	Linear scan, filtering and **appending**	**3,380,000**	3,990,747
(c)	Pure Binary Search and Append	3,970,000	3,991,967
(d)	**Hybrid Linear/Logarithmic**	3,420,000	**3,990,453**

Justification for the Hybrid Solution (d)

The analysis proved that deterministic, predictable performance is non-negotiable for a financial primitive operating on-chain:

1. **Deterministic Safety:** Candidates (b), (c), and (d) are safe.
2. **Stability Validation:** Solution (a) incurred the highest gas cost ($3,993,165$) under mixed-data conditions, proving its cost is volatile and unacceptable.
3. **Comparative Efficiency:** The three deterministic solutions performed closely: Solution (d) at $3,990,453$ gas, Solution (b) at $3,990,747$ gas, and Solution (c) at $3,991,967$ gas. This confirms that the hybrid approach (**d**) is the most gas-efficient under these mixed-data conditions, validating the strategic decision to prioritize its adaptive logic.

The implementation of the VIL's core instructions with the **Hybrid Solution (d)** appeared most optimal, enabling the complex, 6-step Index Buy Execution algorithm to maintain an extremely low, predictable overall transaction cost of $\approx \$\mathbf{1.25}$. This choice fundamentally guarantees that vector alignment overhead will not compromise the system's economic viability.

Rust implementation of adaptive Join-Filter operation

Here we want to demonstrate our adaptive $Join-Filter$ algorithm using snippet from our Rust codebase:

```
let mut result = Vec::with_capacity(labels_b.data.len());
let mut i = 0;
for j in 0..labels_b.data.len() {
    if i < labels_a.data.len() {
        if labels_a.data[i] != labels_b.data[j] {
            i += labels_a.data[i..]
                .binary_search(&labels_b.data[j])
                .map_err(|_| ErrorCode::MathUnderflow)?;
        }
        result.push(v1.data[i]);
        i += 1;
    }
}
v1.data = result;
```

The above code snippet is included for demonstration purposes. For most up to date version of the code please refer to project repository.

8 Conclusion

The Vectorized Index Buy Order Execution Algorithm provides a mathematically sound and highly performant mechanism for executing high-dimensional financial derivatives on-chain. The formalism highlights the balancing act between quote determination (Section 3), Vendor state management (Section 4), data integrity (Section 5), pricing constraints ($Q_{\text{max, pricing}}$), systemic stability (CL), and market dynamics ($\mathbf{D}_{S,new}, \mathbf{D}_{L,new}$). By offloading vector operations and data alignment to the optimized VIS layer, underpinned by the empirically validated **Hybrid Solution (d)** for Label Manipulation, the system achieves a transaction cost of $\approx$ \$**1.25** for a complex 50/150 asset trade, confirming the viability of vector-based financial primitives in decentralized ecosystems.

8.1 Project Location

The complete project can be found at: `github.com/IndexMaker/deindex`