VaR Calculation

Financial Risk Analysis

Parallel Computing with CUDA

Dataset

KraKen Open, Close, High, Low, Volume, Trades

- 390 cryptocurrency trading pairs
- 465,147 total price records

Risk Metrics Computation

- **Daily Returns**

  Return[t] = (Price[t] - Price[t-1]) / Price[t-1]
- **Statistical Measures**

  Mean: Average daily return

  Standard Deviation: Daily volatility

  Annualized Volatility: $\sigma \times \sqrt{365}$ (365 trading days for crypto)
- **Value at Risk (VaR)**

  Sorting daily returns

  Find Percentile

Sequential Algorithm

- **Daily Returns** O(n)
  Return[t] = (Price[t] - Price[t-1]) / Price[t-1]
- **Statistical Measures** O(n)
  Mean: Average daily return
  Standard Deviation: Daily volatility
  Annualized Volatility: σ × √365 (365 trading days for crypto)
- **Value at Risk (VaR)** O(n log n)
  Sorting daily returns
  Find Percentile

Sequential Algorithm

Daily Return, Mean, Variance

```
float *returns = (float*)malloc(num_returns * sizeof(float));
float sum = 0.0f;
float sum_sq = 0.0f;

for (int j = 0; j < num_returns; j++) {
    returns[j] = (stocks[i].prices[j + 1] - stocks[i].prices[j]) / stocks[i].prices[j];
    sum += returns[j];
    sum_sq += returns[j] * returns[j];
}

float mean = sum / num_returns;
float variance = (sum_sq - sum * sum / num_returns) / (num_returns - 1.0f);
float std_dev = sqrtf(variance);

results[i].mean = mean;
results[i].std_dev = std_dev;
results[i].volatility = std_dev * sqrtf(TRADING_DAYS_PER_YEAR);
```

Parallel Algorithm

Daily Return

```
__global__ void calculate_kernel(float *prices, int *days_per_stock, float *means,
                                 float *variances, float *all_returns, int max_days) {
    int stock_idx = blockIdx.x;
    int tid = threadIdx.x;
    int num_days = days_per_stock[stock_idx];
    int num_returns = num_days - 1;

    if (num_returns <= 0) {
        if (tid == 0) {
            means[stock_idx] = 0.0f;
            variances[stock_idx] = 0.0f;
        }
        return;
    }

    __shared__ float s_sum[BLOCK_SIZE];
    __shared__ float s_sum_sq[BLOCK_SIZE];
```

```
    for (int i = tid; i < num_returns; i += blockDim.x) {
        float price_curr = prices[stock_idx * max_days + i + 1];
        float price_prev = prices[stock_idx * max_days + i];
        float return_val = (price_curr - price_prev) / price_prev;

        all_returns[stock_idx * max_days + i] = return_val;
        local_sum += return_val;
        local_sum_sq += return_val * return_val;
    }

    s_sum[tid] = local_sum;
    s_sum_sq[tid] = local_sum_sq;
    __syncthreads();
```

Parallel Algorithm

Mean, Variance (Reduction Tree)

```
    s_sum[tid] = local_sum;
    s_sum_sq[tid] = local_sum_sq;
    __syncthreads();

    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            s_sum[tid] += s_sum[tid + stride];
            s_sum_sq[tid] += s_sum_sq[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0) {
        float sum = s_sum[0];
        float sum_sq = s_sum_sq[0];
        float n = (float)num_returns;

        float mean = sum / n;
        float variance = (sum_sq - sum * sum / n) / (n - 1.0f);

        means[stock_idx] = mean;
        variances[stock_idx] = variance;
    }
```

Sequential Algorithm

Merge Sort

```cpp
__global__ void batch_merge_sort_kernel(float *data, float *temp, int *num_returns_per_stock, int max_days) {
    // Kernel initialization
    int stock_idx = blockIdx.x;
    int tid = threadIdx.x;
    int block_size = blockDim.x;

    int num_returns = num_returns_per_stock[stock_idx];
    if (num_returns <= 1) return;

    // Memory pointers for this crypto
    float *input_data = data + stock_idx * max_days;
    float *output_data = temp + stock_idx * max_days;

    // Iterative merge sort
    for (int width = 1; width < num_returns; width *= 2) {
        // Calculate number of merges needed
        int num_merges = (num_returns + width * 2 - 1) / (width * 2);

        // Distribute work to threads
        for (int merge_idx = tid; merge_idx < num_merges; merge_idx += block_size) {
            // Calculate merge boundaries
            int left = merge_idx * width * 2;
            if (left >= num_returns) break;

            int mid = min(left + width, num_returns);
            int right = min(left + width * 2, num_returns);
```

Parallel Algorithm

Merge Sort

```
__global__ void batch_merge_sort_kernel(float *data, float *temp, int *num_returns_per_stock, int max_days) {
    // Kernel initialization
    int stock_idx = blockIdx.x;
    int tid = threadIdx.x;
    int block_size = blockDim.x;

    int num_returns = num_returns_per_stock[stock_idx];
    if (num_returns <= 1) return;

    // Memory pointers for this crypto
    float *input_data = data + stock_idx * max_days;
    float *output_data = temp + stock_idx * max_days;

    // Iterative merge sort
    for (int width = 1; width < num_returns; width *= 2) {
        // Calculate number of merges needed
        int num_merges = (num_returns + width * 2 - 1) / (width * 2);

        // Distribute work to threads
        for (int merge_idx = tid; merge_idx < num_merges; merge_idx += block_size) {
            // Calculate merge boundaries
            int left = merge_idx * width * 2;
            if (left >= num_returns) break;

            int mid = min(left + width, num_returns);
            int right = min(left + width * 2, num_returns);
            // Perform parallel merge
            if (mid < right) {
                int i = left, j = mid, k = left;

                while (i < mid && j < right) {
                    if (input_data[i] <= input_data[j]) {
                        output_data[k++] = input_data[i++];
                    } else {
                        output_data[k++] = input_data[j++];
                    }
                }

                while (i < mid) {
                    output_data[k++] = input_data[i++];
                }

                while (j < right) {
                    output_data[k++] = input_data[j++];
                }
            }
        }

        // Synchronize all threads
        __syncthreads();

        // Buffer swapping for next iteration
        if (width < num_returns / 2) {
            for (int idx = tid; idx < num_returns; idx += block_size) {
                input_data[idx] = output_data[idx];
            }
            __syncthreads();
        }
    }

    // Final copy to original data array
    if ((num_returns & (num_returns - 1)) != 0) {
        for (int idx = tid; idx < num_returns; idx += block_size) {
            data[stock_idx * max_days + idx] = output_data[idx];
        }
    }
}
```

Parallel Algorithm

- **Daily Returns** O(n/p)
  Return[t] = (Price[t] - Price[t-1]) / Price[t-1]
- **Statistical Measures** O((n/p) + log p)
  Mean: Average daily return
  Standard Deviation: Daily volatility
  Annualized Volatility: σ × √365 (365 trading days for crypto)
- **Value at Risk (VaR)** O((n log n) / p)
  Sorting daily returns
  Find Percentile # Not implemented in Parallel

Performance

```
PS C:\Junior\Parallel\CUDA_Project> ./sequential crypto_prices.csv
Sequential:
Dataset: 390 stocks, 465146 total days
Time: 0.0530 seconds
Top 5 stocks:
1INCHEUR 0.052612 1.005147 -0.080686 -0.176094
1INCHUSD 0.054225 1.035966 -0.081220 -0.180389
AAVEBTC_1440 0.049316 0.942190 -0.063483 -0.133326
AAVEETH 0.037449 0.715454 -0.048569 -0.103661
AAVEEUR 0.060701 1.159693 -0.088975 -0.184377
```

Sequential : 0.053 s

```
PS C:\Junior\Parallel\CUDA_Project> ./parallel crypto_prices.csv
Parallel:
Dataset: 390 stocks, 465146 total days
Time: 0.0040 seconds

Top 5 stocks:
1INCHEUR 0.052612 1.005147 -0.080686 -0.176094
1INCHUSD 0.054225 1.035966 -0.081220 -0.180389
AAVEBTC_1440 0.049316 0.942190 -0.063483 -0.133326
AAVEETH 0.037449 0.715454 -0.048569 -0.103661
AAVEEUR 0.060701 1.159692 -0.088975 -0.184377
```

Parallel : 0.004 s

Thank You

Made by: Puwit Anmahapong 6610405999