Stock Selection With Parallel Bounded Knapsack Algorithm

Jonathan Adotey
adotej@rpi.edu

Fatih Orhan
orhanf@rpi.edu

Simon Sandrew
sandrs@rpi.edu

Zhiqi Wang
wangz56@rpi.edu

***Abstract*—This paper proposes a parallel knapsack algorithm for simulating stock market investments in a multi-GPU computation environment. The proposed implementation distributes simulated investors among multiple GPUs, such that each GPU computes the performance of a subset of investors and their corresponding investing profile, including strategy and aggressiveness. NVIDIA CUDA is utilized for parallel processing and the MPI interface is employed to manage data exchange and collection. The proposed approach is evaluated on the AiMOS supercomputer, which has 268 compute nodes, each with 6 NVIDIA V100 GPU's and 2 IBM Power 9 processors at 3.15 GHz, each with 20 cores.**

I. Introduction

The stock market is a complex system that has been studied for centuries by economists, mathematicians, and computer scientists[1]. With the advent of high-performance computing, researchers can simulate various stock market scenarios to test investment strategies, assess risk, and predict market behavior[2]. In this paper, we present a parallel knapsack algorithm for stock market investment simulation, which utilizes NVIDIA CUDA GPUs and MPI (message passing interface) on the AiMOS supercomputer system.

Our goal is to evaluate six different stock evaluation strategies and eight aggressiveness levels [3] to determine which combination is the most profitable. The stock evaluation strategies include expected value, maximum potential value, minimum value, most likely values, an average of the two most likely values, and randomly sampled from the stock's probability distribution.

Our simulation is done in rounds, where investors evaluate stocks in their market and select which stocks to invest in for that round. Then, the stocks' prices go up or down, and investors either gain or lose money. If an investor runs out of money, they are out of the game. We have divided the nodes into three groups, where the A group (6 ranks per node) computes the knapsack values, the B group (25 ranks per node) handles the stock data set, and the C group (1 rank per node) handles miscellaneous tasks such as file output.

In this paper, we will present the design, implementation, and evaluation of our parallel knapsack algorithm for stock market investment simulation. We will also record metrics on investment strategy performance, operation timings, and I/O timings. Our research aims to contribute to the field of high-performance computing for financial simulations and provide insights for investors and economists on profitable investment strategies.

II. Parallel Knapsack Algorithm

There are a variety of situations where an optimal selection of items is desirable. Items have weights and values as the regular Knapsack problem defined in the literature [4]. There is a constraint on the total weight we can include in our selection. Our goal is to compute a selection that maximizes the total value while not exceeding the constraint on the total weight. We choose choice vectors C where c_i is zero or one. Our capacity constraint is W.

Maximize:

$$\sum_{i}^{n} c_i v_i$$

Such that:

$$\sum_{i}^{n} c_i w_i \leq W$$

Algorithm 1 0/1 Knapsack Algorithm

Require: $v[n]$ item value; $w[n]$ item weights; W capacity
Initialize $dp[n+1][W+1]$ with 0
for $i = 1$ to n **do**
  for $j = 1$ to W **do**
    $prev \leftarrow dp[i-1][j]$
    if $c - w[i-1] \geq 0$ **then**
      $new \leftarrow dp[i-1][j - w[i-1]] + v[i-1]$
      $dp[i][j] \leftarrow \max(prev, new)$
    else
      $dp[i][j] \leftarrow prev$
    end if
  end for
end for
return $dp[n][W]$

However, the 0/1 knapsack algorithm applies if we can choose zero or one of each item. For our simulation, each stock can be chosen a certain number of times. Therefore we need the bounded knapsack problem, which is a variation of knapsack [4]. One approach is to simply duplicate items. However, this can quickly eat up space, especially if an item can be chosen thousands of times. Instead, we can "compress" those duplicates into one column, and iterate until we have either reached the maximum quantity or the maximum weight. We need to compute a quantity vector Q where q_i is the number of times that item i is included in our solution. We

also are given the limit vector L where l_i is the maximum number of times that item i can be included in any solution.

Maximize:

$$\sum_{i}^{n} q_i v_i$$

Such that:

$$\sum_{i}^{n} q_i w_i \leq W$$

And:

$$\forall_i q_i \leq l_i$$

Algorithm 2 Bounded Knapsack Algorithm

Require: $v[n]$ values; $w[n]$ weights; $L[n]$ limits; W capacity
Initialize $dp[n+1][W+1]$ with 0
for $i = 1$ to n **do**
  for $j = 1$ to W **do**
    $dp[i][j] \leftarrow dp[i-1][j]$
    for $q = 1$ to $L[i-1]$ **do**
      if $q \times w[i-1] > j$ **then**
        continue
      end if
      $new \leftarrow dp[i-1][j - q \times w[i-1]] + q \times v[i-1]$
      $dp[i][j] \leftarrow \max(dp[i][j], new)$
    end for
  end for
end for
return $dp[n][W]$

The overall time complexity of the 0/1 knapsack algorithm is $O(nW)$, where n is the number of items and W is the given weight constraint. The overall time complexity of the bounded knapsack algorithm is $O(nWq)$ the new variable q is the maximum value of l_i. The space complexity for both algorithms is $O(nW)$. The shown algorithms only compute the maximum value that can be obtained, but they do not tell us which items are selected. To retrieve the selection, we simply backtrack from the final answer.

For 0/1 Knapsack:

Algorithm 3 Backtrack Algorithm

Require: $dp[n+1][W+1]$ table; $w[n]$ item weights; W capacity
$j \leftarrow W$
$i \leftarrow n$
$c \leftarrow [0, 0, ..., 0]$ {selection vector}
while $i > 0$ **do**
  if $dp[i-1][j] = dp[i][j]$ **then**
    $c[i-1] \leftarrow 0$
  else
    $c[i-1] \leftarrow 1$
    $j \leftarrow j - w[i-1]$
  end if
  $i \leftarrow i - 1$
end while
return c

For bounded knapsack, we backtrack by finding a value of q that can take us to a previous cell that would give us the current answer. Any value of q can work, so we simply take the largest one. The time complexity of this step would be $O(nq)$, however this can be problematic if our q value is very large.

Algorithm 4 Backtrack Algorithm with Bounded Knapsack

Require: $dp[n+1][W+1]$ table; $w[n]$ weights; $L[n]$ limits; W capacity
$j \leftarrow W$
$i \leftarrow n$
$c \leftarrow [0, 0, ..., 0]$ {selection vector}
while $i > 0$ **do**
  $q \leftarrow L[i-1]$
  while $q \geq 0$ **do**
    if $dp[i-1][j - q \times w[i-1]] = dp[i][j]$ **then**
      $c[i-1] \leftarrow q$
      $j \leftarrow j - q \times w[i-1]$
      break
    end if
    $q \leftarrow q - 1$
  end while
  $i \leftarrow i - 1$
end while
return c

For 0/1 knapsack, time complexity of backtracking is $O(n)$, and for the bounded knapsack it is $O(nq)$. However, for bounded knapsack, we can store pointers that tell us where the previous maximum solution was in the table. We can think of these as weighted edges. For each cell in the table, we store a value in table P which tells us what the previous c value was in the previous row. We also store a value in table S which tells us how many items we selected. This would require 3 times as much space, but it would lower the time complexity of the backtracking step to $O(n)$. The time complexity of the full algorithm would still be $O(nWq)$ This is what we chose

for our simulation, because we had sufficient space, and we tried minimizing run time.

Algorithm 5 Bounded Knapsack Algorithm

Require: $v[n]$ values; $w[n]$ weights; $L[n]$ limits; W capacity
Initialize $dp[n+1][W+1]$ with 0
Initialize $P[n+1][W+1]$ with 0
Initialize $S[n+1][W+1]$ with 0
for $i = 1$ to n **do**
  for $j = 1$ to W **do**
    $dp[i][j] \leftarrow dp[i-1][j]$
    $P[i][j] \leftarrow c$
    $S[i][j] \leftarrow 0$
    for $q = 1$ to $L[i-1]$ **do**
      if $q \times w[i-1] > c$ **then**
        continue
      end if
      $new \leftarrow dp[i-1][j - q \times w[i-1]] + q \times v[i-1]$
      if $new > dp[i][j]$ **then**
        $dp[i][j] \leftarrow new$
        $P[i][j] \leftarrow j - q \times w[i-1]$
        $S[i][j] \leftarrow q$
      end if
    end for
  end for
end for
$j \leftarrow W$
$i \leftarrow n$
$r \leftarrow dp[n][W]$
$c \leftarrow [0, 0, ..., 0]$ {selections (0 to $L[i]$)}
while $i > 0$ **and** $r > 0$ **do**
  $c[i-1] \leftarrow S[i][j]$
  $j \leftarrow P[i][j]$
  $r \leftarrow dp[i-1][j]$
  $i \leftarrow i - 1$
end while
return c

There is a possible space optimization with the shown algorithms. Since every computation only uses values from the previous row, we only need to store the previous row. This would bring the space complexity down to $O(W)$. However, this complicates the process of retrieving the selected items in the bounded knapsack algorithm, so we decided to disregard this optimization.

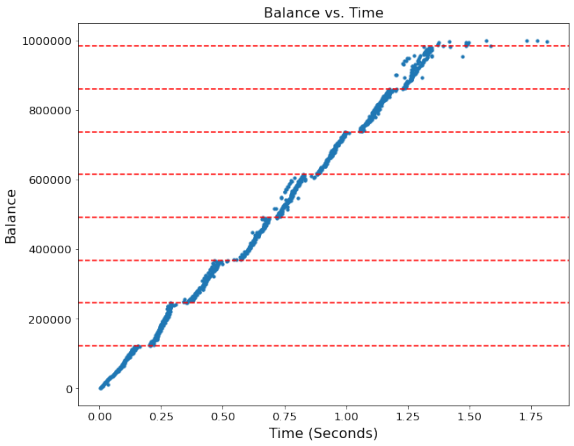
However, with parallel processing, the time complexity can be reduced. According to Amdahl's Law, the speedup from performing part of an operation in parallel is limited by the fraction of time that the part of the operation is actually used[5]. Additionally, parts of computation can only be accurately done in parallel when their operations can be performed independently. If their operations are not independent, then we run into a number of issues such as race conditions and deadlock, so it is best that they remain in serial.

For the bounded knapsack algorithm, each column only depends on the value from the previous row. However, they do not depend on values from the same row. Therefore each cell in a row can be computed independently. However, the backtracking cannot be done in parallel, because the current value of w depends on prior selections. Our original time complexity was $O(nWq)$, however we can split this into two parts, computation and backtracking, giving us $O(nWq + n)$. If variable p is the number of processors, then running the bounded knapsack algorithm in parallel would give us an overall time complexity of $O(n\frac{W}{p}q + n)$, since W can be done in parallel.

Algorithm 6 Bounded Knapsack Algorithm with Input from a Processor Index

Require: $v[n]$ values; $w[n]$ weights; $L[n]$ limits; W capacity; processor index p
Initialize $dp[n+1][W+1]$ with 0
Initialize $P[n+1][W+1]$ with 0
Initialize $S[n+1][W+1]$ with 0
for $i = 1$ to n **do**
  for j in processor index p **do**
    $dp[i][j] \leftarrow dp[i-1][j]$
    $P[i][j] \leftarrow j$
    $S[i][j] \leftarrow 0$
    for $q = 1$ to $L[i-1]$ **do**
      if $q \times w[i-1] > j$ **then**
        continue
      end if
      $new \leftarrow dp[i-1][j - q \times w[i-1]] + q \times v[i-1]$
      if $new > dp[i][j]$ **then**
        $dp[i][j] \leftarrow new$
        $P[i][j] \leftarrow j - q \times w[i-1]$
        $S[i][j] \leftarrow q$
      end if
    end for
  end for
end for
$j \leftarrow W$
$i \leftarrow n$
$r \leftarrow dp[n][W]$
$c \leftarrow [0, 0, ..., 0]$ {selection vector}
while $i > 0$ **and** $r > 0$ **do**
  $c[i-1] \leftarrow S[i][j]$
  $j \leftarrow P[i][j]$
  $r \leftarrow dp[i-1][j]$
  $i \leftarrow i - 1$
end while
return c

We run the parallel bounded knapsack algorithm in CUDA on the NVIDIA Volta V100 GPUs. We use the cooperative groups library to give us exactly 122,880 synchronized threads. In terms of our time complexity, this is our p variable.



For a given set of items size 1000 a roughly linear relationship between the weight constraint and the run time. As expected, there were noticeable increases in time correlated to the balance of the run. These increases occur at points in which the balance exceeded a multiple of our processor count p, which in the following runs above $p = 122,880$. These increases are due to the additional overhead required in reallocating processors to operate on the next sections of the table.

III. Stock Simulation

In our simulation, we use the parallel bounded knapsack algorithm to solve a stock selection problem. Given a group of stocks and an initial balance, we aim to invest in a subset of the stocks that will generate the most profit after a fixed period of time. Given prior data on a stock, there are many ways to predict the future state. However, since the stock behavior can be volatile[6], it is best to model its future behavior with a probability distribution.

For simplicity, we use a probability density function for each stock that gets regenerated after each period of time. The model of a PDF is as such: in the set of outcomes, each outcome v_i can occur at probability p_i. There are many ways to evaluate a probability density function. The most commonly used method is the expected value formula.

$$f(x) = \sum_{i}^{n} v_i p_i$$

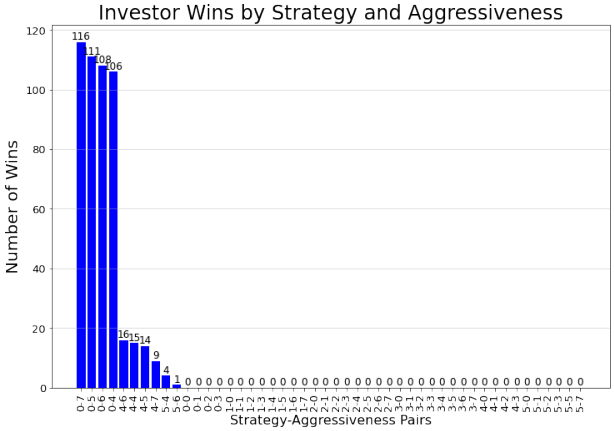
. For the following simulation, we employed the following evaluation methods: expected value, maximum value, minimum value, most likely value, average of the two most likely values, and random sample. We also have 8 aggressiveness levels, each corresponding to a percentage of an investor's balance that they will invest in a single round. The goal is to observe which combination of stock evaluation method and aggressiveness level is most effective at generating profit. We simulate 100 independent markets, each market with 48 investors. We also run 5 total simulations.

IV. Methodology

We use the CUDA and MPI frameworks to utilize the resources of the AiMOS supercomputer system. Each node on AiMOS has 32 MPI ranks. We have 3 groups of MPI ranks. Group C is 25 ranks per compute node, and it handles the state of all stocks in the simulation. Group W has 1 rank per compute node and handles the state of investors. Group G is used for running the parallel bounded knapsack algorithm discussed above.

Initially, we load the stock data from a file, into group C. Then, group W generates the investor profiles. Each round has 3 steps. Step one involves sending data to Group G. Group C sends stock data, and Group W sends investor data. Since a higher balance correlates with longer run times, we applied a greedy load balancing algorithm to more evenly divide the work between ranks in group G by weighing investors based on their current balances. Step 2 is where the ranks in group G run the knapsack algorithm. This simulates the investors observing the market and making optimal choices based on their evaluation methods. Step 3 is where we elapse time. Group C sends the updated stock prices to Group G, where investor gains and losses are computed. Lastly, group G sends the updated investor balances back to Group W.

V. Results and Discussion



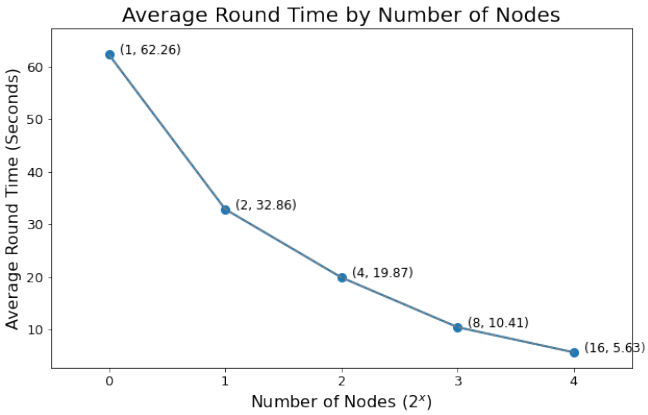
On the x-axis, each (a, b) pair corresponds to a combination of evaluation strategy and aggressiveness level. For a, 0, 1, 2, 3, 4, and 5 correspond to expected value, maximum value, minimum value, most likely value, average of two most likely values, and random sample respectively. For b, values 0, 1, 2, 3, 4, 5, 6, and 7 correspond to the aggressiveness levels.

The investors that used the expected value strategy won most often. Among them, increasing aggressiveness improved their profit. In second place was the average of the two most likely values, and in third place was random sampling. This result implies that the expected value theorem is the most effective way of consistently evaluating probability density functions.

VI. Performance Analysis

A. Strong Scaling Analysis

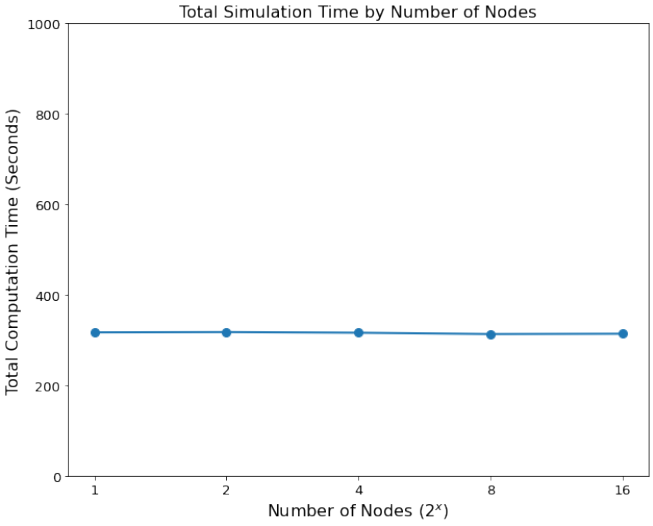
For our runs, we measured the strong scaling performance as follows:



We observed an average of a 45% decrease in time as we increased the number of nodes by a factor of 2. This aligns with our speculation, as doubling the number of nodes will cut the workload of a given node exactly in half. Factoring in the synchronization and communication overhead, the decrease fell below 50%.

B. Weak Scaling Analysis

For our runs, we measured the weak scaling analysis as follows:

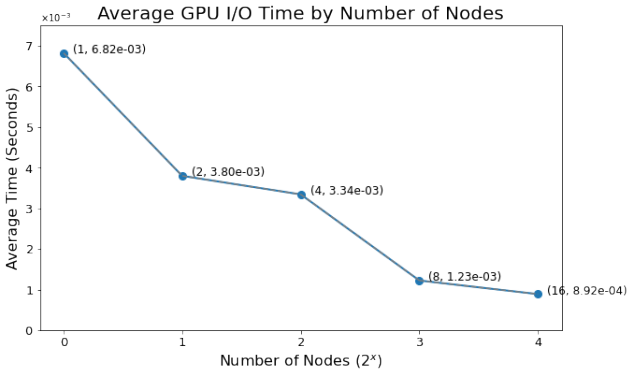


We observe an average change in $-1.77e-03$. This is an extremely small change, which is as expected. If we increase the size of our input and increase the number of nodes, the variance in our data is minimized.

C. GPU I/O Analysis

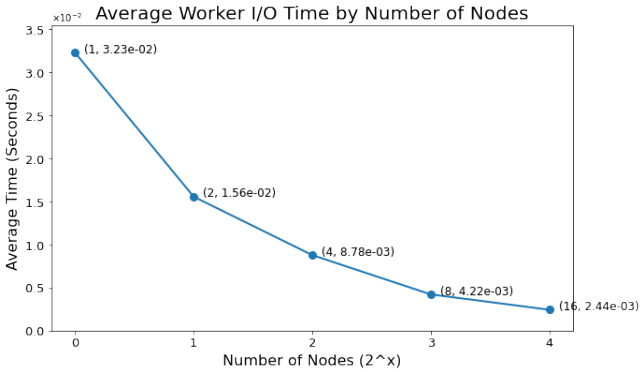
We observed how I/O Performance for group G is affected by the scale of the simulation

As expected, the I/O time observed an average decrease of 36.73 as the number of nodes doubled.



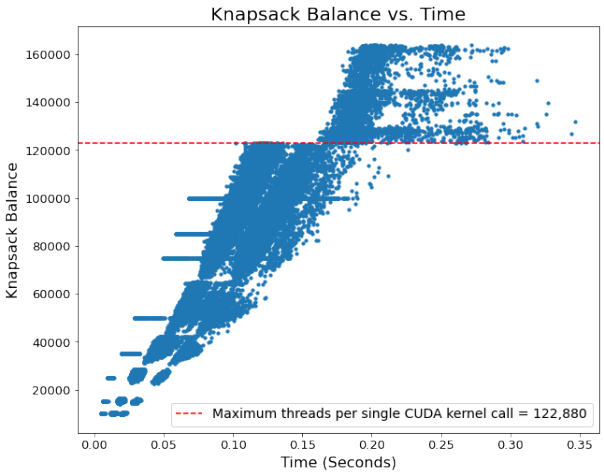
D. Worker I/O Analysis

We observed how I/O Performance for group W is affected by the scale of the simulation



The I/O time observed an average decrease of 47.40 as the number of nodes doubled.

VII. Knapsack & Kernel Analysis



As expected, there was a time jump at $y = 122880$, which was due to the fact that an additional kernel call was needed, since the balance exceeded our processor count. This graph also shows the fact that we had a strategy that opted to not invest in a single stock at any point in the simulation. This is shown by the horizontal lines within the data set.

VIII. Related Work

The stock market has been studied extensively by economists, mathematicians, and computer scientists. Various

simulation and optimization techniques have been employed to analyze stock market behavior and develop profitable investment strategies.

One common approach is Monte Carlo simulation, which involves generating random market scenarios to estimate the probability of different outcomes. For instance, F. Cong et al. [7] utilized Monte Carlo simulation to evaluate diverse portfolio optimization strategies.

Other researchers have leveraged optimization algorithms, such as genetic algorithms optimization, to optimize investment portfolios. For instance, Richard J. Bauer [8] highlighted how the speed, power, and flexibility of Genetic Algorithms (GAs) could assist in consistently devising winning investment strategies.

Parallel computing has also been utilized to speed up financial simulations. Xiang [9] employed FPGA-based supercomputers, GPUs, and CPUs to parallelize a Quasi-Monte Carlo Financial Simulation and compare their performance.

As a classic algorithm that can be scaled up easily, Knapsack was also parallelized with multi-GPU by Jiri [10]. In his work, he introduced a novel approach to solving knapsack with an island-based genetic algorithm to exploit multi-GPU's performance. Each warp handles one individual calculation, which is similar to our implementation. In that paper, Knapsack was solved as a generic algorithm. Another approach to Knapsack is dynamic programming. In the work [11] by Boyer et al., their team utilized parallel programming with CUDA with a compression technique to reduce memory usage. It's worth noting that this work was published in 2012 with a GTX 260 and an Intel Xeon 3.0 GHz, achieving a speedup of 26 times.

However, to the best of our knowledge, no prior work has employed a parallel knapsack algorithm for stock market investment simulation. Our research aims to fill this gap and provide insights into the effectiveness of this approach for evaluating investment strategies.

IX. Conclusion

In this paper, we proposed a parallel knapsack algorithm for simulating stock market investments in a multi-GPU computation environment. We distributed simulated investors among multiple GPUs using the NVIDIA CUDA parallel processing technology and employed the MPI interface to manage data exchange and collection. Our approach evaluated six different stock evaluation strategies and eight levels of aggressiveness to determine the most profitable investment strategy. We conducted our simulation in rounds, where investors evaluated stocks and selected which ones to invest in for each round, and their prices either went up or down. We presented the design, implementation, and evaluation of our algorithm, including metrics on investment strategy performance, operation timings, and I/O timings. In our paper, we discovered that the most successful investing strategy was expected value and high aggressiveness, which won 118 of the 500 total rounds. Additionally, we noticed a jump in time corresponding to the size of the grid. The proposed parallel knapsack algorithm for stock market investment simulation can be extended to other domains, such as portfolio optimization and risk analysis.

References

[1] D. Sornette, *Why stock markets crash: critical events in complex financial systems*. Princeton university press, 2009.

[2] J. Greenwood and B. Jovanovic, "The information-technology revolution and the stock market," *American Economic Review*, vol. 89, no. 2, pp. 116–122, 1999.

[3] W. J. Ferrier and H. Lee, "Strategic aggressiveness, variation, and surprise: How the sequential pattern of competitive rivalry influences stock market returns," *Journal of Managerial Issues*, pp. 162–180, 2002.

[4] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[6] R. Wood and J. L. Zaichkowsky, "Attitudes and trading behavior of stock market investors: A segmentation approach," *The Journal of Behavioral Finance*, vol. 5, no. 3, pp. 170–179, 2004.

[7] F. Cong and C. Oosterlee, "Multi-period mean–variance portfolio optimization based on monte-carlo simulation," *Journal of Economic Dynamics and Control*, vol. 64, pp. 23–38, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165188916000026

[8] R. Bauer, *Genetic Algorithms and Investment Strategies*, ser. Wiley Finance. Wiley, 1994. [Online]. Available: https://books.google.com/books?id=4pbxEIDyQzUC

[9] X. Tian and K. Benkrid, "High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, nov 2010. [Online]. Available: https://doi.org/10.1145/1862648.1862656

[10] J. Jaros, "Multi-gpu island-based genetic algorithm for solving the knapsack problem," in *2012 IEEE Congress on Evolutionary Computation*, 2012, pp. 1–8.

[11] V. Boyer, D. El Baz, and M. Elkihel, "Solving knapsack problems on gpu," *Computers & Operations Research*, vol. 39, no. 1, pp. 42–47, 2012, special Issue on Knapsack Problems and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054811000876