A *SIMD* Solution for the Quadratic Assignment Problem with GPU Acceleration

Abhilash Chaparala
Computer Science Dept.
Texas State University
601 University Drive
San Marcos, TX 78666-4684
a_c219@txstate.edu

Clara Novoa
Ingram School of Engineering
Texas State University
601 University Drive
San Marcos, TX 78666-4684
cn17@txstate.edu

Apan Qasem
Computer Science Dept.
Texas State University
601 University Drive
San Marcos, TX 78666-4684
apan@txstate.edu

ABSTRACT

In the Quadratic Assignment Problem (QAP), n units (usually departments , machines, or electronic components) must be assigned to n locations given the distance between the locations and the flow between the units. The goal is to find the assignment that minimizes the sum of the products of distance traveled and flow between units. The QAP is a combinatorial problem difficult to solve to optimality even for problems where n is relatively small (e.g., $n = 30$). In this paper, we solve the QAP problem using a parallel algorithm that employs a *2-opt* heuristic and leverages the compute capabilities of current GPUs. The algorithm is implemented on the Stampede cluster hosted by the Texas Advanced Computing Center (TACC) at the University of Texas at Austin and on a GPU-equipped server housed at Texas State University. We enhance our implementation by fine tuning the occupancy levels and by exploiting inter-thread data locality through improved shared memory allocation. On a series of experiments on the well-known QAPLIB data sets, our algorithm, on average, outperforms an OpenMP implementation by a factor of 16.31 and a Tabu search based GPU implementation by a factor of 58.61. Given the wide applicability of QAP, the algorithm we propose has very good potential to accelerate the discovery in scholarly research in Engineering, particularly in the fields of Operations Research and design of electronic devices.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; I.2.8 [**Artificial Intelligence**]: Search—*Heuristic Methods*

General Terms

Algorithms, Experimentation, Performance

XSEDE '14, July 13 - 18 2014, Atlanta, GA, USA

http://dx.doi.org/10.1145/2616498.2616521

Keywords

2-Opt heuristic, Random search, Quadratic Assignment Problem, Parallel computing, GPU computing

1. INTRODUCTION

The Quadratic Assignment Problem (QAP) is an NP-hard combinatorial optimization problem [18, 22, 31]. The problem is to assign n units to n locations to minimize the total cost measured as the sum of the products of flows between units and distances between locations [25]. Flow and distance matrices are assumed to be known. QAP has applications [4, 9] being the most common one the design of facility layouts in manufacturing systems. In such application, the QAP finds an optimal allocation of n facilities (departments, machines, or workstations) to sites to minimize the total layout costs [8]. Other applications of QAP include the placement of modules on board positions so that the total wire length to connect them is minimized (i.e. computer backboard wiring [23]), campus planning [12], hospital layout [13], ergonomic design [1] (i.e., keyboard and control panel design [21]), scheduling problems [14], placement of electronic components [20], and memory layout optimization in signal processors [30].

The QAP has attracted many researches not only because of its practical and theoretical importance but also due to its complexity. In general, instances with $n > 30$ are difficult to solve by exact methods in reasonable time [1, 18]. This fact has motivated the use of the problem as a benchmark to test solution methodologies. Metaheuristic approaches have been applied most often than heuristics [2]. QAPLIB serves as the most prominent source for problem instances and known solutions [6]. Besides QAPLIB, others in the the scientific community have facilitated research work by posting instances and optimal or best known solutions. An increased interest in parallel computing to solve the QAP was evidenced in the 1990's [18]. This trend began with the development of CPU parallel implementations such as the ones in [7] and [24]. In the last few years there has been a shift to finding solutions using GPUs [3, 11, 19, 26, 27, 31].

GPU's are powerful accelerators, require less energy than other computing devices, and are widely available and relatively cheap. These GPU characteristics and the fact that GPU computing has been identified as a very promising direction in the field of Operations Research, motivated the authors of this paper to solve the QAP using the GPU.

In this paper, we present a single instruction multiple data *2-Opt* algorithm for the QAP implemented on the GPU. The

computational framework used is the TACC Stampede cluster at the University of Texas at Austin. TACC is one of the XSEDE partner institutions. The computational study in this paper consists of three parts. First, we investigate the performance and accuracy of the proposed GPU implementation. The performance is assessed by comparing our implementation to a parallel one under OpenMP and to the tabu search GPU implementation described in [31]. Next, we perform an occupancy study varying threads per block and total number threads for QAP's ranging from 30 -100 units. In the third part, we study the effect of using shared memory to further reduce the computational time.

The paper is organized as follows. Section 2 presents formulations for the quadratic assignment problem and a brief description of *2-opt* heuristic search for QAP in a serial environment. Section 3 provides an overview of GPU computing, a description of the *2-Opt* search implementation in GPU, and a literature review focused on work solving QAP using GPU. Section 4 presents the computational experiments and the numerical results. Section 5 summarizes the conclusions and discusses future research.

2. QUADRATIC ASSIGNMENT PROBLEM FORMULATION

2.1 Koopmans-Beckman QAP Formulation

Koopmans and Beckmann provide the following formulation for the QAP [16]: let F and D be two given $n \times n$ matrices that represent flows between units and distances between locations such that $F = [f_{kl}]$ and $D = [d_{ij}]$. Consider the set of positive integers $1, 2, ..., n$ and let Π_n be the set of all permutations of $1, 2, ..., n$. The QAP can be defined as finding a permutation $\pi^* \in \Pi_n$ such that the sum of the products below is minimized.

$$z_\pi = \sum_{i=1}^{n}\sum_{j=1}^{n} f_{\pi_i\pi_j} \cdot d_{ij} \tag{1}$$

2.2 Quadratic 0-1 Formulation

Koopmans and Beckmann also stated a quadratic 0-1 integer programming formulation [16]. In this formulation, $X = [x_{mk}]$ represents an $n \times n$ matrix of decision variables. x_{ki} takes the value of 1 if unit k is assigned to location i and 0 otherwise. The solution to this formulation finds the values of all variables x_{ki} and x_{lj} that minimize the sum of the products of flows and distances (Eq. 2) and satisfy the constraints expressed in Eqs. 3-5. Constraints in (3) assign each unit k to a single location i. Constraints in (4) assign only one unit to each location. Constraints in (5) force the decision variables to take binary values

$$min \quad z = \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} f_{kl}d_{ij}x_{ki}x_{lj} \tag{2}$$

s.t

$$\sum_{i=1}^{n} x_{ki} = 1, \qquad k = 1, 2, ..., n \tag{3}$$

$$\sum_{k=1}^{n} x_{ki} = 1, \qquad i = 1, 2, ..., n \tag{4}$$

$$x_{ki} \in \{0, 1\} \qquad k = 1, 2, ..., n \quad i = 1, 2, ..., n \tag{5}$$

The information encoded in a permutation π in Π_n has a one-to-one correspondence to the information stored in the $n \times n$ matrix $X = [x_{ki}]$ since x_{ki} equals to 1 if $\pi_i = k$. It means the quadratic 0-1 and the Koopmans-Beckman formulations are equivalent. In the remainder of the paper we will call Eq. (2) (or alternatively Eq. (1)) the objective function. Its value (or cost) permits to assess and rank different problem solutions.

2.3 Serial 2-Opt Search for QAP

As part of this study we wanted to develop a relatively simple heuristic and concentrate our efforts on learning about the GPU capabilities and how to exploit them. We selected the *2-Opt* heuristic as the search strategy to implement. This heuristic was originally proposed by Croes for the traveling salesman problem [10]. In a serial environment, the *2-Opt* heuristic for QAP starts with an initial random permutation array π (i.e. a feasible solution in the language of quadratic programming). The cost of the random permutation is computed as in Eq. (1) (or as in Eq. (2)) . The permutation array and its costs are stored in the *best-solution* array and the *best-cost-so-far* variable, respectively. To get a single neighborhood solution, two positions i and j in π are randomly selected and a pairwise exchange of their content is performed. Burkard and Rendl [5] provide a formula to compute the change (i.e. *delta*) in the objective function value after a pair-wise exchange (i.e. a swap). The advantage of this formula is that it can be evaluated in $O(n)$ operations for all potential $O(n^2)$ swaps. In contrast, the computation of cost using Eq. (1) requires $O(n^2)$ operations. Following is the formula in [5] for the case in which both flows and distances are asymmetric.

$$\begin{aligned}\Delta_{ij} &= (d_{ji} - d_{ij})(f_{\pi_i\pi_j} - f_{\pi_j\pi_i}) \\ &+ \sum_{k\in n\setminus\{i,j\}} ((d_{jk} - d_{ik})(f_{\pi_i\pi_k} - f_{\pi_j\pi_k}) \\ &+ (d_{kj} - d_{ki})(f_{\pi_k\pi_i} - f_{\pi_k\pi_j}))\end{aligned} \tag{6}$$

Once the costs of all neighbor solutions obtained by pairwise exchange is computed, it is checked if the *best-solution* array and the *best-cost-so-far* variable need to be updated. If a neighboring solution has a cost lower than the one in *best-cost-so-far*, an update occurs. To start a new iteration, *best-solution* replaces the initial random permutation. The process of neighborhood generation and evaluation is repeated for a predetermined number of iterations. When a pre-determined number of iterations is reached the *best-solution* array and the *best-cost-so-far* value are output as the problem solution. It can be the optimal solution if n is small or more likely a suboptimal solution if n is large.

3. GPU COMPUTING

Microprocessors based on a single central processing unit (CPU) drove rapid performance increase and cost reductions in computer applications. This drive slowed due to the energy-consumption and the heat-dissipation that limited the clock frequency and the level of productive activities performed in each clock period within a single CPU [15]. Semiconductor industry then settled on two main trajectories for designing microprocessors, multi-core and many-core. Multi-core aimed to maximize the speed of sequential

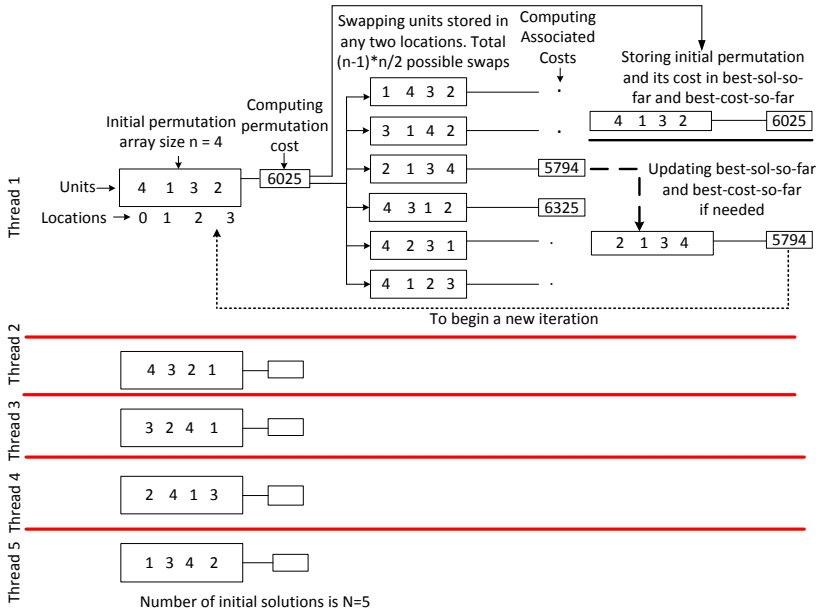

Figure 1: 2-Opt Search in GPU

programs while the many-core trajectory focuses more on enhancing the execution throughput of parallel applications.

In the past, graphics processing units (GPU) were special-purpose hardwired application accelerators, suitable only for conventional graphics applications. Modern GPUs are fully programmable, autonomous parallel floating point processors which may simultaneously execute the same program instruction on different data. Nvidia, the leading manufacturer of GPUs, released CUDA, a parallel computing platform and programming model that provides a C programming language interface to program the GPU hardware. CUDA enables dramatic increases in computing performance by harnessing the power of the GPUs.

One appealing characteristic of the GPU is that it efficiently launches many threads and executes them in parallel to enable computational throughput across large amounts of data. Each thread runs the same program named a kernel. Threads are grouped into thread blocks and all threads in a thread block may cooperate to solve a sub problem. A block has a dimensionality of one, two or three. A grid is a set of blocks which are completely independent. A grid has dimensionality of one or two. A warp is a group of threads within a block that are launched together and execute together. Warp size is typically 32 threads on current generations of GPUs. Shared memory can be accessed by all threads within a block but not across blocks. Luong et al. describe several factors that affect the performance of GPU-based QAP applications [19]. These include efficient distribution of data processing between CPU and GPU, the level of required communication and synchronization among threads, the optimization of data transfer between the different parts of the memory hierarchy, and the capacity constraints of these memories.

3.1 The 2-Opt Search in GPU

This section describes the algorithm implemented in this work to heuristically solve the QAP. Using specific seed values, a set of N initial random permutations of size n is generated and stored in a matrix of size $N \times n$. Each permutation is assigned to a single GPU thread which computes its cost using Eq. (1) described in Section 2.1. In each thread, the permutation is also copied in an array named *best-solution-so-far*. The associated permutation cost is copied to a variable named *best-cost-so-far*. Fig. 1 illustrates the case in which five random permutations (i.e. N=5) of size n=4 are assigned to five threads. Next, each thread independently performs all *2-Opt* interchanges to produce a neighborhood of $n \times (n-1)/2$ new permutations. This neighborhood is explored by computing the associated costs of the permutations using Eq. (6) from Section 2.3. The systematic way in which the six *2-Opt* interchanges are done for a permutation of size four is illustrated in Fig. 2. This figure is similar to the one in [2]

To evaluate a single new permutation the flow and distance matrices are needed. These matrices are maintained in global memory to be accessible by all threads. As mentioned later in Section 4.5, the time to evaluate the costs of the *2-Opt* neighborhood is reduced if copies of these matrices are kept in the kernel shared memory.

After computing the costs of the *2-Opt* neighborhood of an original random permutation, it is possible that *best-cost-so-far* and *best-solution-so-far* need to be updated. This is done by selecting the lowest cost of the new permutations and comparing it to the value currently stored in *best-cost-so-far*. If the cost of a new permutation is lower than *best-cost-so-far* then this single new permutation is stored in *best-solution-so-far* and its cost is stored in *best-cost-so-far*. Also the new permutation with the lowest cost replaces the original random permutation. This sets the start another iteration. The total number of iterations the algorithm is repeated is set as a function of the problem size n. After the total number of iterations is reached, each thread returns its *best-*

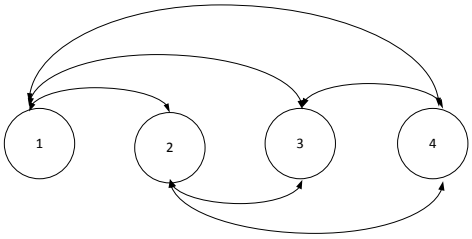

Figure 2: 2-Opt Swaps for a Permutation of Size 4

solution-so-far and *best-cost-so-far* values. The process of comparing the results returned by the threads and finding the single permutation with the minimum cost is done in the CPU. Once the solution and its cost is identified and output to a file the *2-opt* algorithm terminates.

3.2 Previous Work on Solving QAP using GPU

Luong *et al.* proposed a GPU based iterated tabu search for solving the quadratic 3-D assignment problem, an extension of the QAP and of the axial 3-D assignment problem [19]. The motivation for using the GPU was to enlarge the neighborhood structure without incurring an expensive computational process. The quadratic 3-D problem searches for a double permutation which minimizes a cost function. Each new solution is based on the exchange of two positions in either the first permutation or the second one. Thus, the size of the neighborhood is $(n \times (n-1)/2)^2$. The authors concluded that GPU permitted to exploit parallelism in the neighborhood search and improve the quality of the solutions obtained. The authors propose as further research a multi-GPU idea. It is partitioning the neighborhood set and assign each partition to a single GPU.

Zhu *et al.* proposed a single-instruction multiple data tabu search (SIMD-TS) for the QAP using a single GPU [31] on a personal computer. The parallelization consisted of running 6144 simultaneous independent tabu searches (6144 threads, 32 blocks, 192 threads per block) on 128 processors. Texture memory (a fast read-only memory) was used to store the distance and flow matrices. To assure each thread searches a different but promising area the authors implemented diversification and intensification operations every m iterations. The authors demonstrated the implemented algorithm was fast and effective. They used instances of different sizes ($30 < n < 90$) from QAPLIB and the worst performance gap was 0.73%. The authors also stated that the cache size (8k) of the texture memory affected the experimental performance. Since their tabu search implementation only had short-term memory, as a future research they proposed to develop long-term memory.

Czapinski proposed an effective parallel multistart tabu search (PMTS) for the QAP on the CUDA platform [11]. The technique consisted of diversifying an initial solution, running multiple tabu searches on each diversified solution, and re-starting the search with the best solutions after a certain number of iterations. The set of tabu searches to run in different threads results from systematic swaps of an initial solution. It permits the author to conclude that each thread can save just two rows or two columns of the flow and distance matrix for symmetric matrices. It avoids keeping the whole matrices in shared memory. In the non-symmetric case, two rows and two columns of the matrices are needed. To get a full-benefit of coalescing transposed copies of the matrices are stored. The proposed search also benefits from communication between parallel tabu search instances which is achieved by passing the best obtained solutions to the CPU, examining them and choosing new configurations in the CPU, and re-starting the parallel tabu search in the GPU. From initial experiments the author agreed with [31] that 192 threads per block was the best choice. Instances of size 50-70 ran faster in GPU when compared to a six-core MPI implementation.

Other work related to solving QAP with genetic algorithms using GPU is the one in [26]. Also [27] proposes a fast QAP solver which implements Ant Colony Optimization (ACO) and Tabu Seach on GPU.

4. EXPERIMENTAL RESULTS

4.1 GPU Platforms

The computational experiments were executed on the Stampede cluster on the TACC system. Stampede is a 10 PFLOPS Dell Linux Cluster based on 6,400+ Dell Zeus PowerEdge server nodes, each outfitted with 2 Intel Xeon 8-Core 64-bit E5 processors (2.7 GHz) and an Intel Xeon Phi Co-processor (1.1.GHz). Each node runs Centos 6.3 (2.6 32x86_64 Linux kernel) . The nodes are managed with batch services through SLURM 2.4. Stampede has 128 compute nodes outfitted with a single Nvidia K20 GPU on each node with 5GB of on-board GDDR5 memory. Each K20 GPU has 2496 CUDA cores distributed over 13 streaming multiprocessors (SM's). Each SM can hold a maximum of 2048 thread contexts, which amounts to 26624 (13*2048) threads that can simultaneously be active on the GPU. The clock speed for each core is 0.706 GHz, L1 cache size is 64 KB/SM and L2 cache size is 768 KB (shared).

For all experiments, the serial CPU and the OpenMP variants of the code were compiled with GCC Version 4.4.7. The CUDA code was compiled with nvcc using CUDA version 5.5. The `sbatch` script was used to submit jobs to the cluster and to specify the node configuration. For our experiments we ran four jobs simultaneously by assigning each job to a different Stampede node. This significantly expedited our evaluation process.

In addition to the Stampede cluster we also ran several experiments on a local server with a six-core Intel processor based on the Sandybridge architecture. The server is equipped with a Tesla K20c NVIDIA GPU. This GPU also has a total of 2496 cores grouped into 13 stream multiprocessors (SM's). The amount of shared memory available per block is 48K. On this platform, the CPU code was compiled with GCC Version 4.6.3 and the CUDA code was compiled with nvcc with CUDA version 5.5 The server runs Ubuntu 12.04 as its operating system.

4.2 Benchmark Data Sets

The instances we used to test our *2-opt* algorithm come from QAPLIB, a library of published test problems for the QAP described in [6]. We selected 2 instances from [17], 7 from [24], and 8 from [25] to do a direct comparison with the work reported in [31] (Table 4 page 1044). The *Lipa* instances come from problem generators described in [17]. These generators provide asymmetric instances (i.e. non-

Table 1: Computational results for problems from the QAPLIB

	Problems	Best known cost	Cost	Gap	Time
1	tai30a	1,818,146	1,838,184	1.10%	3.84
2	tai30b	637,117,113	637,117,113	0.00%	3.78
3	tai35a	2,422,002	2,464,946	1.77%	7.03
4	tai35b	283,315,445	283,349,722	0.01%	6.90
5	tai40a	3,139,370	3,187,882	1.55%	11.83
6	tai40b	637,250,948	637,349,459	0.02%	11.68
7	tai50a	4,938,796	5,026,692	1.78%	29.40
8	tai50b	458,821,517	459,528,298	0.15%	29.17
9	tai60a	7,205,962	7,386,338	2.50%	62.15
10	tai60b	608,215,054	609,612,341	0.23%	61.19
11	tai64c	1,855,928	1,855,928	0.00%	81.13
12	tai80a	13,515,450	13,833,332	2.35%	202.11
13	tai80b	818,415,043	822,630,127	0.52%	199.20
14	tai100a	21,054,656	21,550,036	2.35%	501.65
15	tai100b	1,185,996,137	1,196,603,999	0.89%	493.62
16	lipa70a	169,755	171,068	0.77%	117.08
17	lipa90a	360,630	362,948	0.64%	327.19

symmetric flow and/or distance matrices) with known optimal solutions. The instances named *Taixxa* are uniformly generated and proposed in [24]. The other problems were introduced in [25]. Problems instances labeled as *Taixxb* are asymmetric and randomly generated. Instances *Taixxc* occur in the generation of grey patterns. We ran the *2-Opt* algorithm in each problem instance eight times and computed the average values as well as the standard deviation, minimum, maximum, and coefficient of variation (standard deviation/average). All coefficients of variation are low. Consequently, in the tables and figures in the next sub-sections we report the average values found for each instance.

4.3 Performance and Accuracy

Table 1 reports the performance and accuracy of the GPU accelerated 2-opt algorithm (referred to as *2-opt* in the rest of the section). Numbers are reported for the 17 different instances selected from the QAPLIB data sets. For each instance, the number of initial random solutions was $N = 6144$. The total number of threads, blocks and threads per block were set to 6144, 24, and 256, respectively. The best known cost for a particular instance is derived from previously published results [6]. We measure performance in terms of total execution time (seconds) of the GPU kernel. Accuracy is the percent difference between the best known cost and the cost discovered by our algorithm.

We observe that *2-opt* yields good accuracy across different problem instances. For several instances the computed cost is within 0.1% of the best known value while the average accuracy over all instances is 1%. The execution times reported in Table 1 show that *2-opt* is able to attain this high level of accuracy within a reasonable amount of time even for large data sets. The longest running time is 8 minutes 21 seconds for *tail100a*. As a quick comparison, a sequential implementation on a high-end CPU takes more than 2 days to compute the solution for this same instance.

To better evaluate our algorithm, we compare its performance and accuracy with two other parallel implementations

1. *OpenMP:* an OpenMP-based CPU implementation of *2-opt*

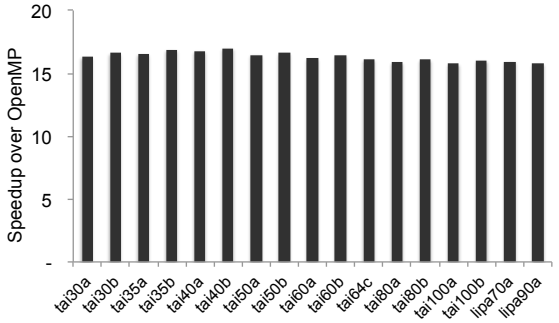

Figure 3: *2-opt* and *OpenMP* performance comparison

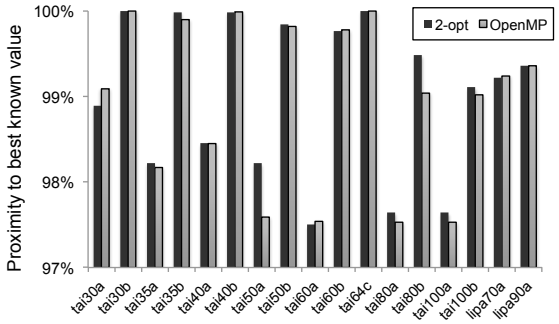

Figure 4: *2-opt* and *OpenMP* accuracy comparison

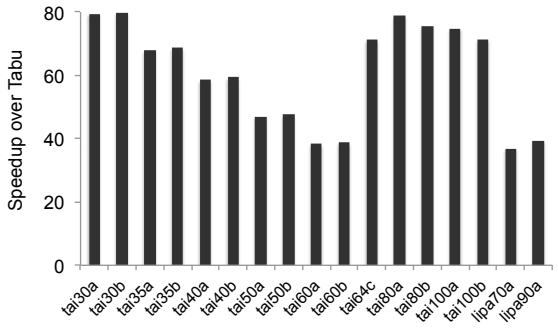

Figure 5: *2-opt* and *Tabu* performance comparison

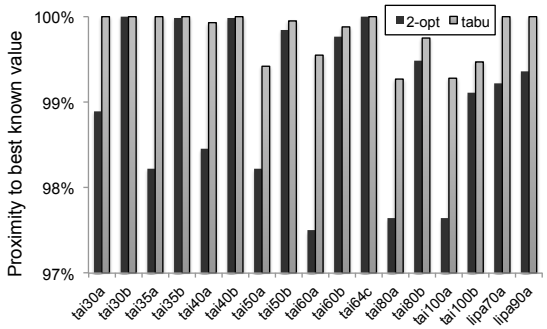

Figure 6: *2-opt* and *Tabu* accuracy comparison

2. *Tabu:* a GPU implementation of tabu search for QAP performed by Zhu *et al.* [31].

Figs. 3 and 4 show the performance and accuracy comparisons between *2-opt* and *OpenMP*. *2-opt* achieves at least a factor of 16 speedup over *OpenMP* on all problem sizes. We attribute this performance gains mainly to the additional computation power available on the GPU. *OpenMP* was implemented using 16 threads which proved to be optimal for the compute node configurations on the Stampede cluster. On the other hand, *2-opt* was designed to make use of *all* available SMs on the target GPU allowing it achieve more parallelism on different problem instances. In terms of accuracy, there is no clear advantage for either *OpenMP* or *2-opt*. On some instances *2-opt* performs significantly better while on others *OpenMP* yields a better solution. This is an expected result as both versions employ a random heuristic for searching.

Fig. 5 reports the speedup obtained by *2-opt* over *Tabu*. *Tabu* numbers were obtained from previously published results in [31]. Then the comparison is not totally straightforward since the computing resorces used in the two implementations were different. This particular implementation of *Tabu* uses 6144 threads and was executed on an Nvidia GeForce 8800 GPU. We observe that *2-opt* yields impressive speedups over *Tabu*. For most *Taillard* instances, *2-opt* improves performance by a factor of 40 or more. Overall, performance gains on *Taillard* sets are higher than those for *Lipa* data sets. But even for *Lipa*, *2-opt* achieves a 38-fold speedup on average. Some of the performance improvements can be attributed to the more powerful GPU used for our experiments. However, most of the benefits come from increased parallelism, fewer synchronizations and an efficient cost calculation formula.

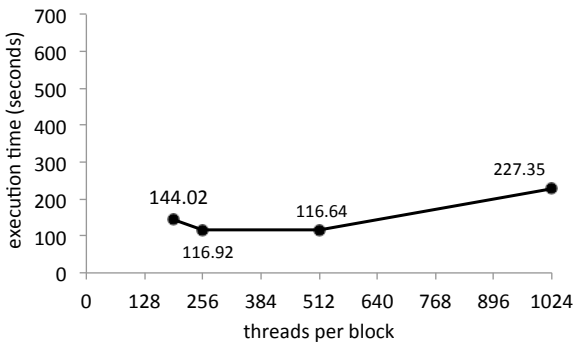

Figure 7: Lipa70 performance for varying thread configurations

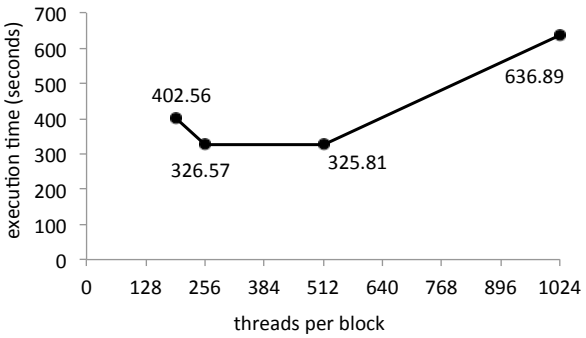

Figure 8: Lipa90 performance for varying thread configurations

Although *2-opt* yields significantly higher performance it lags behind *Tabu* in terms of accuracy, as depicted in Fig. 6. On average *Tabu* delivers results that are 0.77% closer to the best known solution. The improved accuracy can be attributed to the more sophisticated search heuristic and the CPU-based double-precision cost recalculation algorithm used in *Tabu*. It should be noted, however that the difference in accuracy is fairly small and in several instances *2-opt* produces similar accuracy as *Tabu*. Among the different instances, *2-opt* performed poorly on data sets that were uniformly generated (i.e., data sets with a specific pattern). Since *2-opt* uses a random heuristic it is not too surprising that it was not able to exploit the patterns in the data set to its advantage. Nevertheless, the accuracy on uniform data sets is an important consideration and we intend to address this issue in future work (see Section 5).

4.4 Thread Block Configuration

Determining the right thread hierarchy is an important consideration for any GPU implementation. We ran a series of experiments to find a suitable thread configuration for *2-opt*. We parameterized the algorithm and executed the code with different thread and block parameters to vary the number of active warps per SM and attain different levels of occupancy. Figs. 7-10 present selected results from these experiments. These experiments reveal that the best

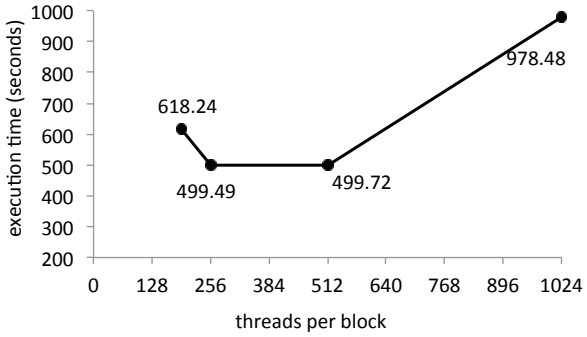

Figure 9: Taillard100a performance for varying thread configurations

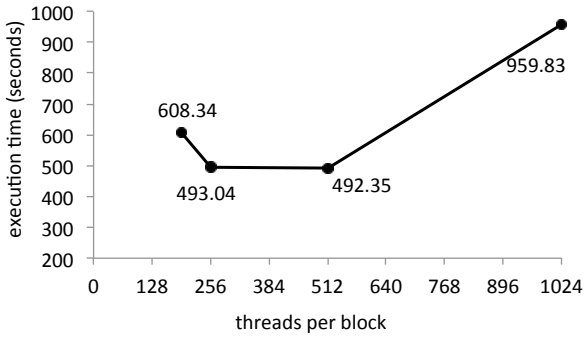

Figure 10: Taillard100b performance for varying thread configurations

performance for *2-opt* is not necessarily achieved at maximum *threads per block*, in spite of the fewer synchronization events occurring on those implementations. For both *Lipa* and *Taillard* data sets, the highest performance is achieved at 256 threads per block. We attribute this performance gain to better register usage and shared memory utilization. These results corroborate results from earlier studies on GPU occupancy and data locality [28, 29].

4.5 Shared Memory

To optimize memory access, two key data structures, *flow* and *distance*, were allocated to shared memory. Fig. 11 shows performance results for the implementation of *2-opt* with shared memory allocation. We notice that the shared memory implementation provides yet more performance improvements over the highly efficient non-shared version of *2-opt*. These gains stem from two different sources. First, allocation into shared memory replaces many of the global memory accesses with accesses to shared memory with lower latencies. Second, because each thread in a block accesses the data structures in every iteration, the shared memory allocation helps exploit the abundant inter-thread data locality exhibited by these threads.

5. CONCLUSIONS AND FUTURE WORK

This paper presented an efficient parallel solution to QAP, an important problem in the domain of Operations Research (OR). Experimental results show that the described algorithm can provide impressive speedups over a comparable GPU implementation while incurring only a small penalty in terms of accuracy. In many contexts, the sacrifice in accuracy for improved and scalable performance is considered to be a worthwhile trade-off. Thus, by providing a fast solution to QAP, the proposed *2-Opt* algorithm has the potential to accelerate discovery in OR. Practical OR problems that may be impacted are in the contexts of facility layout optimization ergonomics, health care optimization and scheduling. The OR community researching on exact solutions to the QAP may benefit also from this *2-Opt* accelerated heuristic. Fast and reasonably accurate heuristics are useful to establish bounds at the initial stages of exact solution approaches such as the branch-and-cut and branch-and-bound methods. Besides, the OR community may hybridize the accelerated *2-Opt* heuristic with other heuristic or meta-heuristics to investigate possible accuracy gains. Also, the proposed *2-Opt* algorithm may be used by electronic industries working in the layout of electronic devices in computer backboards and memories in signal processors.

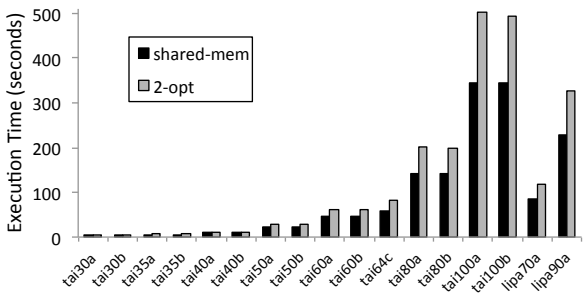

Figure 11: *2-opt* performance with shared memory allocation

The experimental results revealed a weakness in our approach in finding suitable costs for uniformly generated data sets (i.e. *Taixxa* instances). We will address this issue by restarting the GPU portion of the algorithm multiple times. Fifty percent of the threads will get a diversified solution of the last permutation stored and the remaining fifty percent will re-start with the *best-so-far* solution. Also we plan on incorporating a more sophisticated heuristic in our search such as tabu with long term memory and/or other enhanced features. We will investigate additional avenues of extracting parallelism and exploiting locality including parallelization of the swap operations and division of the solution set into disjoint search spaces. The accessibility to the Stampede cluster reduced significantly the time to complete the experimentation phase. The on-line documentation from TACC and the suggestions from its staff members were clear and appropriate to accomplish the objectives. These facts should motivate more OR practitioners to use a computational cyberinfrastructure similar to the Stampede cluster.

6. ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing high performance computing resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu. We especially

thank Yaakoub El Khamra, Paul Navarti and Lars Koesterke for the support they provided on working on CUDA in Stampede. The third author also acknowledges support from the National Science Foundation through awards CNS-1253292 and CNS-1305302.

7. REFERENCES

[1] K. Anstreicher, N. Brixius, J. P. Goux, and L. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming Series B*, 91:563–588, 2002.

[2] M. Bashiri and H. Karimi. Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons. *Journal of Industrial Engineering International*, 8(6):1–9, 2012.

[3] V. Boyer and D. El Baz. Recent advances on GPU computing in operations research. In *2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW)*, pages 1778–1787. IEEE, May 2013.

[4] R. Burkard. Quadratic assignment problems. *European Journal of Operational Research*, 15(3):283–289, 1984.

[5] R. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984.

[6] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB-A quadratic assignment problem library. *European Journal of Operational Research*, 55(1):115–119, 1991.

[7] J. Chakrapani and J. Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Operations Research*, 41(4):327–341, 1993.

[8] W. C. Chiang and P. Kouvelis. An improved tabu search heuristic for solving facility layout design problems. *International Journal of Production Research*, 34(9):2565–2585, 1996.

[9] C. Commander. A survey of the quadratic assignment problem, with applications. *Morehead Electronic Journal of Applicable Mathematics*, 4:1–15, 2005.

[10] G. Croes. A method for solving traveling salesman problems. *Operations Research*, 6:791–812, 1958.

[11] M. Czapinski. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73:1461–1468, 2013.

[12] J. Dickey and J. Hopkins. Campus building arrangement using topaz. *Transportation Research*, 6:59–68, 1972.

[13] A. Elshafei. Hospital layout as a quadratic assignment problem. *Operations Research Quarterly*, 28:167–179, 1977.

[14] A. Geoffrion and G. Graves. Scheduling parallel production lines with changeover costs: Practical applications of a quadratic assignment/lp approach. *Operations Research*, 24:595–610, 1957.

[15] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors*. Morgan Kaufmann, Burlington, Massachussetts, 2010.

[16] T. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 15:53–76, 1957.

[17] Y. Li and P. Pardalos. Generating quadratic assignment test problems with known optimal permutations. *Computational Optimization and Applications*, 1:163–184, 1992.

[18] E. M. Loiola, N. de Abreu, P. Boaventura Netto, P. Hahn, and T. Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, 2007.

[19] T. Luong, L. Loukil, N. Melab, and E. Talbi. A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem. In *2010 IEE/ACS international Conference on Computer Systems and Applications (AICCSSA)*, pages 1–8. AICCSA, May 2010.

[20] G. Miranda, H. Luna, G. Mateus, and R. Ferreira. A performance guarantee heuristic for electronic components placement problems including thermal effects. *Computers and Operations Research*, 32:2937–2957, 2005.

[21] M. Pollatschek, N. Greshoni, and Y. Raddday. Optimization of the typewriter keyboard by simulation. *Angewandte Informatik*, 17:438–439, 1976.

[22] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of Association for Computing Machinery*, 23(3):555–565, 1976.

[23] L. Steinberg. The blackboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.

[24] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(3-4):443–455, 1991.

[25] E. D. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2):87–105, 1995.

[26] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with GPU computation: A case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2523–2530. ACM, July 2009.

[27] S. Tsutsui and N. Fujimoto. Fast QAP solver with ACO and taboo search on GPU using move-cost adjusted thread assignment. In *Genetic and Evolutionary Computation Conference*, pages 1–2, 2011.

[28] S. Unkule, C. Shaltz, and A. Qasem. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *Proc. Int'l. Conf. on Compiler Construction (CC12)*, pages 21–40, 2012.

[29] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[30] B. Wess and T. Zeitlhofer. On the phase coupling problem between data memory layout generation and address pointer assignment. *Lecture Notes in Computer Science*, 3199:152–166, 2004.

[31] W. Zhu, J. Curry, and A. Marquez. SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. *International Journal of Production Research*, 48(4):1035–1047, 2010.